# Selected Homework Solutions – Unit 1

CMPSC 465

## Exercise 2.1-3

Here, we work with the linear search, specified as follows:

LINEAR-SEARCH($A$, $v$)
Input: $A = \langle a_1, a, \ldots, a_n \rangle$; and a value $v$.
Output: index $i$ if there exists an $i$ in $1..n$ s.t. $v = A[i]$; NIL, otherwise.

We can write pseudocode as follows:

```
LINEAR-SEARCH(A, v)
    i = 1
    while i ≤ A.length and A[i] ≠ v          // check elements of array until end or we find key
    {
        i = i + 1
    }

    if i == A.length + 1                     // case that we searched to end of array, didn't find key
        return NIL
    else                                     // case that we found key
        return i
```

Here is a loop invariant for the loop above:

At the start of the $i$th iteration of the **while** loop, $A[1..i\text{-}1]$ doesn't contain value $v$

Now we use the loop invariant to do a proof of correctness:

**Initialization:**

Before the first iteration of the loop, $i = 1$. The subarray $A[1..i\text{-}1]$ is empty, so the loop invariant vacuously holds.

**Maintenance:**

For $i \in \mathbf{Z}$ s.t. $1 \leq i \leq A$.length, consider iteration $i$. By the loop invariant, at the start of iteration $i$, $A[1..i\text{-}1]$ doesn't contain $v$. The loop body is only executed when $A[i]$ is not $v$ and we have not exceeded $A.length$. So, when the $i$th iteration ends, $A[1..i]$ will not contain value $v$. Put differently, at the start of the $(i+1)^{st}$ iteration, $A[1..i\text{-}1]$ will once again not contain value $v$.

**Termination:**

There are two possible ways the loop terminates:

- If there exists an index $i$ such that $A[i] == v$, then the **while** loop will terminate at the end of the $i$th iteration. The loop invariant says $A[1..i\text{-}1]$ doesn't contain $v$, which is true. And, in this case, $i$ will not reach $A.length + 1$, so the algorithm returns $i$ s.t. $A[i] = v$, which is correct.
- Otherwise, the loop terminates when $i = n + 1$ (where $n = A.length$), which implies $n = i - 1$. By the loop invariant, A[1..$i$-1] is the entire array A[1..$n$], and it doesn't contain value $v$, so NIL will correctly be returned by the algorithm.

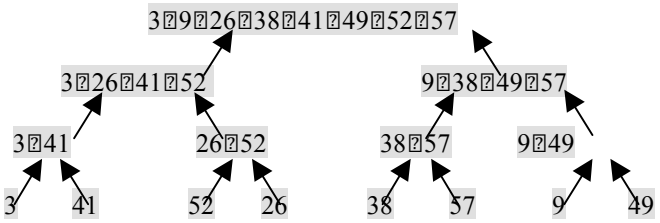
---

**Note:** Remember a few things from intro programming and from Epp:

- Remember to think about which kind of loop to use for a problem. We don't know how many iterations the linear search loop will run until it's done, so we should use an indeterminate loop structure. (If we do, the proof is cleaner.)
- As noted in Epp, the only way to get out of a loop should be by having the loop test fail (or, in the **for** case, the counter reach the end). Don't return or break out of a loop; proving the maintenance step becomes very tricky if you do.

## Exercise 2.3-1

The figure below illustrates the operations of the procedure bottom-up of the merge sort on the array $A = \{3, 41, 52, 26, 38, 57, 9, 49\}$:

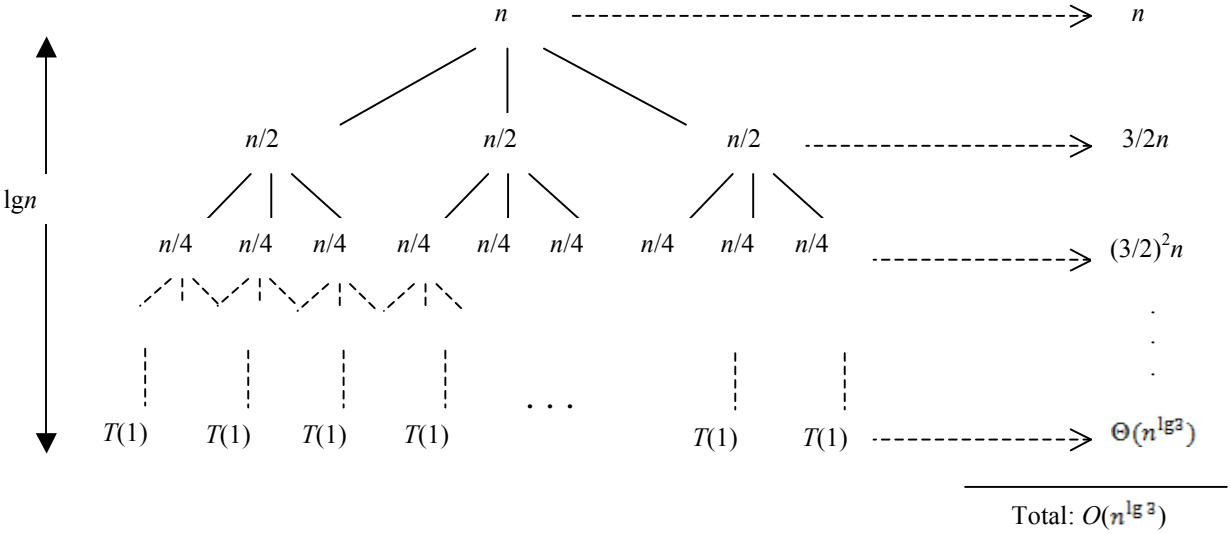



The algorithm consists of merging pairs of 1-item sequence to form sorted sequences of length 2, merging pairs of sequences of length 2 to form sorted sequences of length 4, and so on, until two sequences of length $n/2$ are merged to form the final sorted sequence of length $n$.

## Exercise 4.4-1

The recurrence is $T(n) = 3T(\lfloor n/2 \rfloor) + n$. We use a recurrence tree to determine the asymptotic upper bound on this recurrence.

Because we know that floors and ceilings usually do not matter when solving recurrences, we create a recurrence tree for the recurrence $T(n) = 3T(n/2) + n$. For convenience, we assume that $n$ is an exact power of 2 so that all subproblem sizes are integers.




Because subproblem sizes decrease by a factor of 2 each time when go down one level, we eventually must reach a boundary condition $T(1)$. To determine the depth of the tree, we find that the subproblem size for a node at depth $i$ is $n/2^i$. Thus, the subproblem size hits $n = 1$ when $n/2^i = 1$ or, equivalently, when $i = \lg n$. Thus, the tree has $\lg n + 1$ levels (at depth 0, 1, 2, 3, … , $\lg n$).

Next we determine the cost at each level of the tree. Each level has 3 times more nodes than the level above, and so the number of nodes at depth $i$ is $3^i$. Because subproblem sizes reduce by a factor of 2 for each level when go down from the root, each node at depth $i$, for $i = 0, 1, 2, 3, \ldots, \lg n - 1$, has a cost of $n/2^i$. Multiplying, we see that the total cost over all nodes at depth $i$, for $i = 0, 1, 2, 3, \ldots, \lg n - 1$, is $3^i * n/2^i = (3/2)^i n$. The bottom level, at depth $\lg n$, has $3^{\lg n} = n^{\lg 3}$ nodes, each contributing cost $T(1)$, for a total cost of $n^{\lg 3} T(1)$, which is $\Theta(n^{\lg 3})$, since we assume that $T(1)$ is a constant.

Now we add up the costs over all levels to determine the cost for the entire tree:

$T(n) = n + \frac{3}{2}n + (\frac{3}{2})^2 n + (\frac{3}{2})^3 n + \cdots + (\frac{3}{2})^{\lg n - 1} n + \Theta(n^{\lg 3})$

$= \sum_{k=0}^{\lg n - 1} (\frac{3}{2})^k n + \Theta(n^{\lg 3})$ by using Sigma to sum all the elements except the last one

$= \frac{(\frac{3}{2})^{\lg n} - 1}{\frac{3}{2} - 1} n + \Theta(n^{\lg 3})$ by geometric series sum formula

$= 2n^{\lg 3} - 2n + \Theta(n^{\lg 3})$ by evaluating the fraction

$= O(n^{\lg 3})$

Thus, we have derived a guess of $T(n) = O(n^{\lg 3})$ for our original recurrence $T(n) = 3T(\lfloor n/2 \rfloor) + n$. Now we can use the inductive proof to verify that our guess is correct.

**To prove:**

For all integers $n$ s.t. $n \geq 1$, the property $P(n)$:

The closed form $T(n) \leq dn^{\lg 3} - cn$, for some constants $d$ and $c$ s.t. $d > 0$ and $c > 0$,
matches the recurrence $T(n) = 3T(\lfloor n/2 \rfloor) + n$.

**Proof**:

We will reason with strong induction.

Since we're only proving a bound and not an exact running time, we don't need to worry about a base case.

**Inductive Step**:

Let $k \in Z$ s.t. $k \geq 1$ and assume $\forall i \in Z$ s.t. $1 \leq i \leq k$, $P(i)$ is true. i.e. $T(i) \leq di^{\lg 3} - cn$ matches the recurrence.
[inductive hypothesis]

Consider $T(k+1)$:

$T(k+1) = 3T(\lfloor (k+1)/2 \rfloor) + k + 1$ by using the recurrence definition (as $k \geq 1$ implies $k + 1 \geq 2$, so we are in the recursive case)

$\leq 3d(\frac{k+1}{2})^{\lg 3} - \frac{3}{2}c(k+1) + k + 1$ by subs. from inductive hypothesis, $\lfloor (k+1)/2 \rfloor \leq (k+1)/2$, and $d \geq c + 1$

$\leq \frac{3}{2^{\lg 3}} d(k+1)^{\lg 3} - \frac{3}{2}c(k+1) + k + 1$ by laws of exp.

$\leq d(k+1)^{\lg 3} - \frac{3}{2}c(k+1) + k + 1$ by laws of exp. and log.

$\leq d(k+1)^{\lg 3} - c(k+1)$ as long as $c \geq 2$, $\frac{3}{2}c(k+1) - (k+1) \geq c(k+1)$

So $P(k+1)$ is true.

So, by the principle of strong mathematical induction, $P(n)$ is true for all integers $n$ s.t. $n \geq 1$, and constant $d = 3$, $c = 2$.

## Exercise 4.5-1

a) Use the master method to give tight asymptotic bounds for the recurrence $T(n) = 2T(n/4) + 1$.

**Solution:**
For this recurrence, we have $a = 2$, $b = 4$, $f(n) = 1$, and thus we have that $n^{\log_b a} = n^{\log_4 2}$. Since $f(n) = 1 = O(n^{\log_4 2 - \epsilon})$, where $\epsilon = 0.2$, we can apply case 1 of the master theorem and conclude that the solution is $T(n) = \Theta(n^{\log_4 2}) = \Theta(n^{\frac{1}{2}}) = \Theta(\sqrt{n})$.

b) Use the master method to give tight asymptotic bounds for the recurrence $T(n) = 2T(n/4) + \sqrt{n}$.

**Solution:**
For this recurrence, we have $a = 2$, $b = 4$, $f(n) = \sqrt{n}$, and thus we have that $n^{\log_b a} = n^{\log_4 2} = n^{\frac{1}{2}} = \sqrt{n}$. Since $f(n) = \Theta(\sqrt{n})$, we can apply case 2 of the master theorem and conclude that the solution is $T(n) = \Theta(\sqrt{n} \lg n)$.

c) Use the master method to give tight asymptotic bounds for the recurrence $T(n) = 2T(n/4) + n$.

**Solution:**
For this recurrence, we have $a = 2$, $b = 4$, $f(n) = n$, and thus we have that $n^{\log_b a} = n^{\log_4 2}$. Since $f(n) = \Omega(n^{\log_4 2 + \epsilon})$ , where $\epsilon = 0.2$, we can apply case 3 of the master theorem if we can show that the regularity condition holds for $f(n)$.

To show the regularity condition, we need to prove that $af(n/b) \le cf(n)$ for some constant $c < 1$ and all sufficiently large $n$. If we can prove this, we can conclude that $T(n) = \Theta(f(n))$ by case 3 of the master theorem.

**Proof of the regularity condition:**
$af(n/b) = 2(n/4) = n/2 \le cf(n)$ for $c = 0.7$, and $n \ge 2$.

So, we can conclude that the solution is $T(n) = \Theta(f(n)) = \Theta(n)$.

d) Use the master method to give tight asymptotic bounds for the recurrence $T(n) = 2T(n/4) + n^2$.

**Solution:**
For this recurrence, we have $a = 2$, $b = 4$, $f(n) = n^2$, and thus we have that $n^{\log_b a} = n^{\log_4 2}$. Since $f(n) = \Omega(n^{\log_4 2 + \epsilon})$ , where $\epsilon = 1$, we can apply case 3 of the master theorem if we can show that the regularity condition holds for $f(n)$.

**Proof of the regularity condition:**
$af(n/b) = 2(n/4)^2 = (1/8)\, n^2 \le c\, f(n)$ for $c = 0.5$, and $n \ge 4$.

So, we can conclude that the solution is $T(n) = \Theta(n^2)$.

# Selected Homework Solutions – Unit 2

CMPSC 465

## Exercise 6.1-1

**Problem:** What are the minimum and maximum numbers of elements in a heap of height $h$?

Since a heap is an almost-complete binary tree (complete at all levels except possibly the lowest), it has at most $1+2+2^2+2^3+\ldots+2^h=2^{h+1}-1$ elements (if it is complete) and at least $2^h-1+1=2^h$ elements (if the lowest level has just 1 element and the other levels are complete).

## Exercise 6.1-3

**Problem:** Show that in any subtree of a max-heap, the root of the subtree contains the largest value occurring anywhere in that subtree.

**To prove:**

For any subtree rooted at node $k$ of a max-heap $A[1, 2, \ldots, n]$, the property $P(k)$:

The node $k$ of the subtree rooted at $k$ contains the largest value occurring anywhere in that subtree.

**Proof**:

**Base Case:**

When $k \in \left[\lfloor n/2 \rfloor + 1, n\right]$, $k$ is a leaf node of a max-heap since $\lfloor n/2 \rfloor$ is the index of the last parent, and the subtree rooted at $k$ just contains one node. Thus, node $k$ contains the largest value in that subtree.

**Inductive Step**:

Let $k \in \left[1, \lfloor n/2 \rfloor\right]$ s.t. $k$ is an internal node of a max-heap, and assume $\forall i \in Z$ s.t. $k < i \leq n$, $P(i)$ is true. i.e.

The node $i$ of the subtree rooted at $i$ contains the largest value occurring anywhere in that subtree.

[inductive hypothesis]

Now let us consider node $k$:

1. $k$'s left child $2k$ and right child $2k + 1$ contain the largest value of $k$'s left and right subtree, respectively. (by the inductive hypothesis that for $k < i \leq n$, $P(i)$ is true)
2. $k$'s value is larger than its left child $2k$ and right child $2k + 1$. (by the max-heap property)

So, we can conclude that node $k$ contains the largest value in the subtree rooted at $k$.

Thus, by the principle of strong mathematical induction, $P(k)$ is true for all nodes in a max-heap.

## Exercise 6.1-4

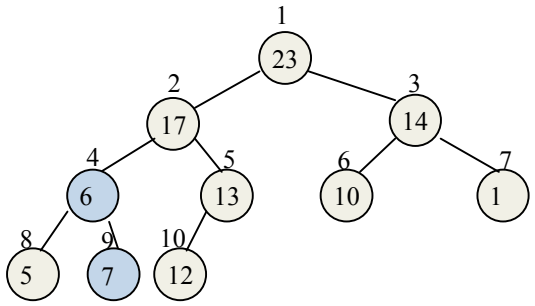
**Problem:** Where in a max-heap might the smallest element reside, assuming that all elements are distinct?

The smallest element can only be one of leaf nodes. If not, it will have its own subtree and is larger than any element on that subtree, which contradicts the fact that it is the smallest element.

## Exercise 6.1-6

**Problem:** Is the array with values (23, 17, 14, 6, 13, 10, 1, 5, 7, 12) a max-heap?

Consider this illustration of the heap:




So, this array isn't a max-heap. As shown in the figure above, the value of node 9 is greater than that of its parent node 4.

## Exercise 6.4-2

We argue the correctness of HEAPSORT using the following loop invariant:

> At the start of each iteration of the **for** loop of lines 2-5, the subarray $A[1..i]$ is a max-heap containing the $i$ smallest elements of $A[1..n]$, and the subarray $A[i+1..n]$ contains the $n-i$ lagest elements of $A[1..n]$, sorted.

**Proof:**

We need to show that this invariant is true prior to the first step, that each iteration of the loop maintains the invariant. It provides the property to show correctness of HEAPSORT.

**Initialization**:

Prior to the first iteration of the loop, $i = n$. The subarray $A[1..i]$ is a max-heap due to BUILD-MAX-HEAP. The subarray $A[i + 1..n]$ is empty, hence the claim about it being sorted is vacuously true.

**Maintenance**:

For each iteration, By the loop invariant, $A[1..i]$ is a max-heap containing the $i$ smallest elements of $A[1..n]$, so the $A[1]$ contains the $(n-i+1)$st largest element. The execution of line 3 exchanges $A[1]$ with $A[i]$, so it makes the subarray $A[i..n]$ contain the $(n-i+1)$ largest elements of $A[1..n]$, sorted. The heap size of $A$ is decreased in to heap-size$(A) = i - 1$. Now the subarray $A[1..i-1]$ is not a max-heap, since the root node violates the map-heap property. But the children of the root maintain the max-heap property, so we can restore the max-heap property by calling MAX-HEAPIFY$(A, 1)$, which leaves a max-heap in $A[1..i-1]$. Consequently, the loop invariant is reestablished for the next iteration.

**Termination**:

At termination, $i = 1$. By the loop invariant, the subarray $A[2.. n]$ contains the $n - 1$ largest elements of $A[1..n]$, sorted, and thus, $A[1]$ contains the smallest element of $A[1..n]$. $A[1..n]$ is a sorted array.

## Exercise 6.4-3

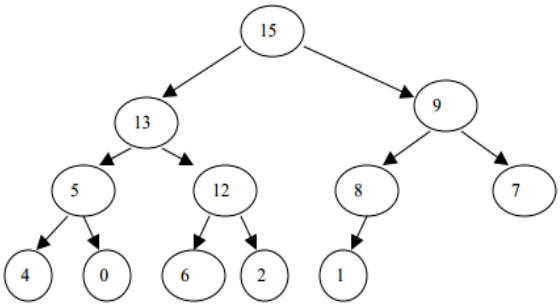
**Problem:** What is the running time of HEAPSORT on an array $A$ of length $n$ that is already sorted in increasing order? What about decreasing order?

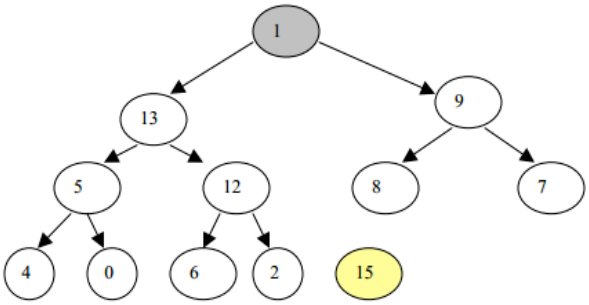
The running time of HEAPSORT on an array of length that is already sorted in increasing order is $\Theta(n \lg n)$, because even though it is already sorted, it will be transformed back into a heap and sorted.

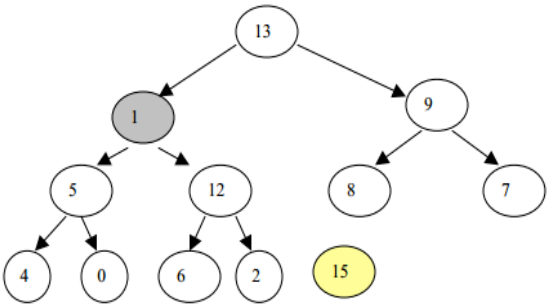
The running time of HEAPSORT on an array of length that is sorted in decreasing order will be $\Theta(n \lg n)$. This occurs because even though the heap will be built in linear time, every time the element is removed and HEAPIFY is called, it could cover the full height of the tree.

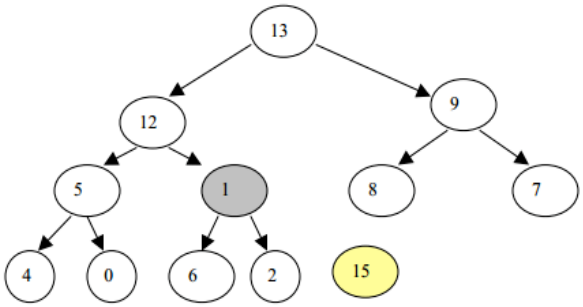
## Exercise 6.5-1

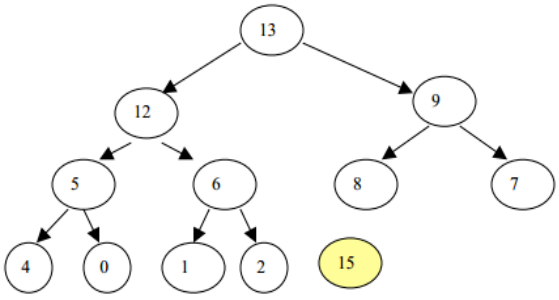
**Problem:** Illustrate the operation of HEAP-EXTRACT-MAX on the heap $A$ = <15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1>.



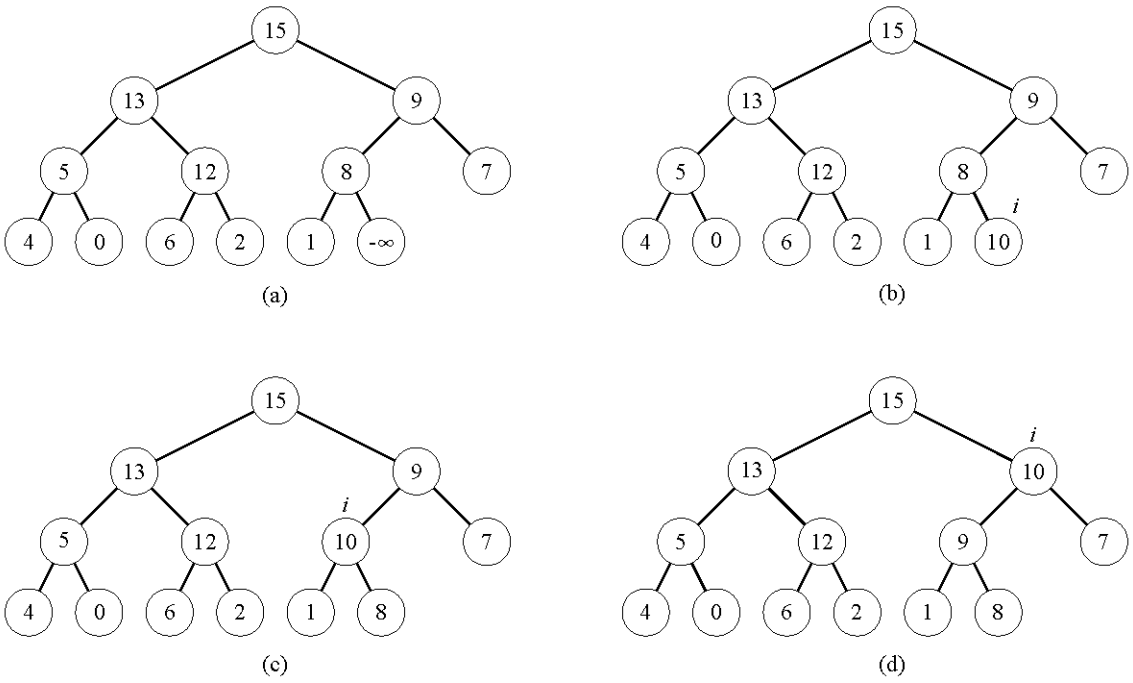



## Exercise 6.5-2

**Problem:** Illustrate the operation of MAX-HEAP-INSERT (*A*, 10) on the heap *A* = (15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1).




## Exercise 7.1-3

**Problem:** Give a brief argument that the running time of PARTITION on a subarray of size n is $\Theta(n)$.

Since each iteration of the for loop involves a constant number of operations and there are *n* iterations total, the running time is $\Theta(n)$.

## Exercise 7.2-1

Use the substitution method to prove that the recurrence $T(n) = T(n\text{-}1) + \Theta(n)$ has the solution $T(n) = \Theta(n^2)$, as claimed at the beginning of Section 7.2.

**To prove:**

For all integers $n$ s.t. $n \geq 1$, the property $P(n)$:

The closed form $T(n) \leq cn^2$ and $T(n) \geq dn^2$, for some constants $c$ and $d$, s.t. $c > 0$ and $d > 0$, matches the recurrence $T(n) = T(n–1) + \Theta(n)$.

**Proof**:

We will reason with strong induction.

**Base case**:

Let $n = 1$. $T(1) = T(0) + \Theta(1) = \Theta(1) \leq c1^2$ , if $c$ is large enough. $T(1) = T(0) + \Theta(1) = \Theta(1) \geq d1^2$, if $d$ is small enough. So $P(1)$ holds.

**Inductive Step**:

Let $k \in Z$ s.t. $k \geq 1$ and assume $\forall i \in Z$ s.t. $1 \leq i \leq k–1$, $P(i)$ is true. i.e.

$T(i) \leq ci^2$ and $T(i) \geq di^2$ matches the recurrence. [inductive hypothesis]

Consider $T(k)$ to find an upper bound:

| | |
|---|---|
| $T(k) = T(k–1) + \Theta(k)$. | by using the recurrence definition |
| $\leq c(k–1)^2 + c_1k$ | by subs. from inductive hypothesis, where $c_1$ is some constant |
| $= c(k^2 – 2k + 1) + c_1k$ | since $(k–1)^2 = k^2 – 2k + 1$ |
| $= ck^2 – (2c–c_1)k + c$ | by combining like terms |
| $\leq ck^2$ | as long as $c > c_1$ |

Consider $T(k)$ to find a lower bound:

| | |
|---|---|
| $T(k) = T(k–1) + \Theta(k)$. | by using the recurrence definition |
| $\geq d(k–1)^2 + c_1k$ | by subs. from inductive hypothesis, where $c_1$ is some constant |
| $= d(k^2 – 2k + 1) + c_1k$ | since $(k–1)^2 = k^2 – 2k + 1$ |
| $= dk^2 + (c_1–2d)k + d$ | by combining like terms |
| $\geq dk^2$ | as long as $c_1–2d \geq 0$ or $0 < d \leq c_1/2$ |

So $P(k)$ is true.

So, by the principle of strong mathematical induction, $P(n)$ is true for all integers $n$ s.t. $n \geq 1$, and constants $c > c_1$, $0 < d \leq c_1/2$.

Since $T(n) = \Omega(n^2)$ and $T(n) = O(n^2)$, we obtain that $T(n) = \Theta(n^2)$.

## Exercise 7.2-4

**Problem:** Banks often record transactions on an account in order of the times of the transaction, but many people like to receive their bank statements with checks listed in order by check number. People usually write checks in order by check number, and merchants usually cash them with reasonable dispatch. The problem of converting time-of-transaction ordering is therefore the problem of sorting almost-sorted input. Argue that the procedure INSERTION-SORT would tend to beat the procedure QUICKSORT on this problem.

INSERTION-SORT's running time on perfectly-sorted input runs in $\Theta(n)$ time. So, it takes almost $\Theta(n)$ running time to sort an almost-sorted input with INSERTION-SORT. However, QUICKSORT requires almost $\Theta(n^2)$ running time, recalling that it takes $\Theta(n^2)$ time to sort perfectly-sorted input. This is because when we pick the last element as the pivot, it is usually the biggest one, and it will produce one subproblem with close to $n - 1$ elements and one with 0 elements. Since the cost of PARTITION procedure of QUICKSORT is $\Theta(n)$, the recurrence running time of QUICKSORT is $T(n) = T(n - 1) + \Theta(n)$. In another problem, we, use the substitution method to prove that the recurrence $T(n) = T(n - 1) + \Theta(n)$ has the solution $T(n) = \Theta(n^2)$. So we use INSERTION-SORT rather than QUICKSORT in this situation when the input is almost sorted.

## Exercise 8.4-2

**Problem:** Explain why the worst-case running time for bucket sort is $\Theta(n^2)$. What simple change to the algorithm preserve its linear average-case running time and makes its worst-case running time $O(n \lg n)$?

The worst case for bucket sort occurs when the all inputs fall into a single bucket, for example. Since we use INSERTION-SORT for sorting buckets and INSERTION-SORT has a worst case of $\Theta(n^2)$, the worst case run time for bucket sort is $\Theta(n^2)$.

By using an algorithm like MERGE-SORT with worst case run time time of $O(n \lg n)$ instead of INSERTION-SORT for sorting buckets, we can ensure that the worst case of bucket sort is $O(n \lg n)$ without affecting the average case running time.

## Exercise 8.4-3

**Problem:** Let $X$ be a random variable that is equal to the number of heads in two flips of a fair coin. What is $E[X^2]$? What is $E^2[X]$?

$$
\begin{aligned}
E[X^2] &= 1^2 * P(\text{head in one flip}) + 0^2 * P(\text{tail in one flip}) \\
&= 1 * 1/2 + 0 * 1/2 \\
&= 1/2
\end{aligned}
$$

$$
\begin{aligned}
E^2[X] &= E[X] * E[X] && \text{as the two flips are independent} \\
&= 1/2 * 1/2 && \text{as } E[X] = 1/2 \\
&= 1/4
\end{aligned}
$$

# Homework Solutions – Unit 3, Chapter 11

CMPSC 465 Spring 2013

## Exercise 11.1-1

Suppose that a dynamic set $S$ is represented by a direct-address table $T$ of length $m$. Describe a procedure that finds the maximum element of $S$. What is the worst-case performance of your procedure?

**Solution**:

We can do a linear search to find the maximum element in $S$ as follows:

Pre-condition: table $T$ is not empty; $m \in Z^+$, $m \geq 1$.
Post-condition: FCTVAL == maximum value of dynamic set stored in T.

```
FindMax (T, m)
{
    max = −∞

    for i = 1 to m
    {
        if T[i] != NIL && max < T[i]
            max = T[i]
    }
    return max
}
```

In the worst-case searching the entire table is needed. Thus the procedure must take $O(m)$ time.

## Exercise 11.2-1

Suppose we use a hash function h to hash n distinct keys into an array T of length m. Assuming simple uniform hashing, what is the expected number of collisions? More precisely, what is the expected cardinality of $\{\{k, l\}: k \neq l \text{ and } h(k) = h(l)\}$?

**Solution**:

For each pair of keys $k$, $l$, where $k \neq l$, define the indicator random variable $X_{lk} = I\{h(k) = h(l)\}$. Since we assume simple uniform hashing, $\Pr\{X_{lk} = 1\} = \Pr\{h(k) = h(l)\} = 1/m$, and so $E[X_{lk}] = 1/m$.

Now define the random variable $Y$ to be the total number of collisions, so that $Y = \sum_{k \neq l} X_{kl}$. The expected number of collisions is

$$
\begin{aligned}
E[Y] &= E\left[\sum_{k \neq l} X_{kl}\right] && \text{since } Y = \sum_{k \neq l} X_{kl} \\
&= \sum_{k \neq l} E[X_{kl}] && \text{by linearity of expectation} \\
&= \sum_{k \neq l} \frac{1}{m} && \text{since } E[X_{kl}] = \frac{1}{m} \\
&= \binom{n}{2} \frac{1}{m} && \text{by choosing } k \text{ and } l \text{ out of } n \text{ keys} \\
&= \frac{n(n-1)}{2} \frac{1}{m} && \text{by } \binom{n}{r} = \frac{n!}{r!(n-r)!}, \binom{n}{2} = \frac{n!}{2!(n-2)!} = \frac{n(n-1)}{2} \\
&= \frac{n(n-1)}{2m}
\end{aligned}
$$

## Exercise 11.2-2

Demonstrate what happens when we insert the keys 5, 28, 19, 15, 20, 33, 12, 17, 10 into a hash table with collisions resolved by chaining. Let the table have 9 slots, and let the hash function be $h(k) = k \bmod 9$.

**Solution**: omitted.

## Exercise 11.2-3

Professor Marley hypothesizes that he can obtain substantial performance gains by modifying the chaining scheme to keep each list in sorted order. How does the professor's modification affect the running time for successful searches, unsuccessful searches, insertions, and deletions?

**Solution:**

- Successful searches: $\Theta(1 + \alpha)$, which is identical to the original running time. The element we search for is equally likely to be any of the elements in the hash table, and the proof of the running time for successful searches is similar to what we did in the lecture.
- Unsuccessful searches: 1/2 of the original running time, but still $\Theta(1 + \alpha)$, if we simply assume that the probability that one element's value falls between two consecutive elements in the hash slot is uniformly distributed. This is because the value of the element we search for is equally likely to fall between any consecutive elements in the hash slot, and once we find a larger value, we can stop searching. Thus, the running time for unsuccessful searches is a half of the original running time. Its proof is similar to what we did in the lecture.
- Insertions: $\Theta(1 + \alpha)$, compared to the original running time of $\Theta(1)$. This is because we need to find the right location instead of the head to insert the element so that the list remains sorted. The operation of insertions is similar to the operation of unsuccessful searches in this case.
- Deletions: $\Theta(1 + \alpha)$, same as successful searches.

## Exercise 11.3-3

Consider a version of the division method in which $h(k) = k \bmod m$, where $m = 2^p - 1$ and $k$ is a character string interpreted in radix $2^p$. Show that if we can derive string $x$ from string $y$ by permuting its characters, then $x$ and $y$ hash to the same value. Give an example of an application in which this property would be undesirable in a hash function.

**Solution:**
First, we observe that we can generate any permutation by a sequence of interchanges of pairs of characters. One can prove this property formally, but informally, consider that both heapsort and quicksort work by interchanging pairs of elements and that they have to be able to produce any permutation of their input array. Thus, it suffices to show that if string $x$ can be derived from string $y$ by interchanging a single pair of characters, then $x$ and $y$ hash to the same value.

Let $x_i$ be the $i$th character in $x$, and similarly for $y_i$. We can interpret $x$ in radix $2^p$ as $\sum_{i=0}^{n-1} x_i 2^{ip}$, and interpret $y$ as $\sum_{i=0}^{n-1} y_i 2^{ip}$. So

$h(x) = (\sum_{i=0}^{n-1} x_i 2^{ip}) \bmod (2^p - 1)$. Similarly, $h(y) = (\sum_{i=0}^{n-1} y_i 2^{ip}) \bmod (2^p - 1)$.

Suppose that $x$ and $y$ are identical strings of $n$ characters except that the characters in positions $a$ and $b$ are interchanged:

$x_a = y_b$ and $y_a = x_b$. (1)

Without loss of generality, let $a > b$. We have:

$$h(x) - h(y) = (\sum_{i=0}^{n-1} x_i 2^{ip}) \bmod (2^p - 1) - (\sum_{i=0}^{n-1} y_i 2^{ip}) \bmod (2^p - 1) \quad (2)$$

Since $0 \le h(x), h(y) < 2^p - 1$, we have that $-(2^p - 1) < h(x) - h(y) < 2^p - 1$. If we show that $(h(x) - h(y)) \bmod (2^p - 1) = 0$, then $h(x) = h(y)$. To prove $(h(x) - h(y)) \bmod (2^p - 1) = 0$, we have:

$$(h(x) - h(y)) \bmod (2^p - 1) = ((\sum_{i=0}^{n-1} x_i 2^{ip}) \bmod (2^p - 1) - (\sum_{i=0}^{n-1} y_i 2^{ip}) \bmod (2^p - 1)) \bmod (2^p - 1)$$ by (2)

$$= (\sum_{i=0}^{n-1} x_i 2^{ip} - \sum_{i=0}^{n-1} y_i 2^{ip}) \bmod (2^p - 1)$$ by relation in footnote[1]

$$= ((x_a 2^{ap} + x_b 2^{bp}) - (y_a 2^{ap} + y_b 2^{bp})) \bmod (2^p - 1)$$ as $x$ and $y$ are identical strings of $n$ characters except that chars. in positions $a$ and $b$ are interchanged

$$= ((x_a 2^{ap} + x_b 2^{bp}) - (x_b 2^{ap} + x_a 2^{bp})) \bmod (2^p - 1)$$ as $x_a = y_b$, $x_b = y_a$ see (1)

$$= ((x_a - x_b)2^{ap} + (x_b - x_a)2^{bp}) \bmod (2^p - 1)$$ by combining like terms

$$= ((x_a - x_b)2^{ap} - (x_a - x_b)2^{bp}) \bmod (2^p - 1)$$ as $(x_b - x_a) = -(x_a - x_b)$

$$= ((x_a - x_b)(2^{ap} - 2^{bp})) \bmod (2^p - 1)$$ by factoring out $(x_a - x_b)$

$$= ((x_a - x_b)(2^{ap}(2^{bp}/2^{bp}) - 2^{bp})) \bmod (2^p - 1)$$ by multiplication by $2^{bp}/2^{bp} = 1$

$$= ((x_a - x_b)2^{bp}(2^{(a-b)p} - 1)) \bmod (2^p - 1)$$ by factoring out $2^{bp}$

$$= ((x_a - x_b)2^{bp}(\sum_{i=0}^{a-b-1} 2^{ip})(2^p - 1)) \bmod (2^p - 1)$$ by substituting $[2^{(a-b)p} - 1]$[2]

$$= 0$$ since one factor is $2^p - 1$

---

[1] Consider the congruence relation: $(m_1 \text{ o } m_2) \bmod n = ((m_1 \bmod n) \text{ o } (m_2 \bmod n)) \bmod n$, where o is +, –, or *

[2] Consider the equation $\sum_{i=0}^{a-b-1} 2^{ip} = \frac{2^{(a-b)p} - 1}{2^p - 1}$ (geometric series) and multiplying both sides by $2^p - 1$ to get

$2^{(a-b)p} - 1 = (\sum_{i=0}^{a-b-1} 2^{ip})(2^p - 1)]$

Because we deduced earlier that $(h(x) - h(y)) \bmod (2p - 1) = ((x_a - x_b)2^{bp}(2^{(a-b)p} - 1)) \bmod (2^p - 1)$ and have shown here that $((x_a - x_b)2^{bp}(2^{(a-b)p} - 1)) \bmod (2^p - 1) = 0$, we can conclude $(h(x) - h(y)) \bmod (2^p - 1) = 0$, and thus $h(x) = h(y)$. So we have proven that if we can derive string $x$ from string $y$ by permuting its characters, then $x$ and $y$ hash to the same value.

Examples of applications:
A dictionary which contains words expressed by ASCII code can be one of such example when each character of the dictionary is interpreted in radix $2^8 = 256$ and $m = 255$. The dictionary, for instance, might have words "STOP," "TOPS," "SPOT," "POTS," all of which are hashed into the same slot.

## Exercise 11.3-4

Consider a hash table of size $m = 1000$ and a corresponding hash function $h(k) = \lfloor m(kA \bmod 1) \rfloor$ for $A = \dfrac{\sqrt{5}-1}{2}$. Compute the locations to which the keys 61, 62, 63, 64, and 65 are mapped.

**Solution**:

$$
\begin{aligned}
h(61) &= \left\lfloor 1000\left(61\left(\frac{\sqrt{5}-1}{2}\right) \bmod 1\right)\right\rfloor \\
&= \lfloor 1000(37.700 \bmod 1) \rfloor \\
&= \lfloor 1000 \times 0.700 \rfloor \\
&= 700
\end{aligned}
$$

$$
\begin{aligned}
h(62) &= \left\lfloor 1000\left(62\left(\frac{\sqrt{5}-1}{2}\right) \bmod 1\right)\right\rfloor \\
&= \lfloor 1000 \times (38.318 \bmod 1) \rfloor \\
&= \lfloor 1000 \times 0.318 \rfloor \\
&= 318
\end{aligned}
$$

$$
\begin{aligned}
h(63) &= \left\lfloor 1000\left(63\left(\frac{\sqrt{5}-1}{2}\right) \bmod 1\right)\right\rfloor \\
&= \lfloor 1000 \times (38.936 \bmod 1) \rfloor \\
&= \lfloor 1000 \times 0.936 \rfloor \\
&= 936
\end{aligned}
$$

$$
\begin{aligned}
h(64) &= \left\lfloor 1000\left(64\left(\frac{\sqrt{5}-1}{2}\right) \bmod 1\right)\right\rfloor \\
&= \lfloor 1000 \times (39.554 \bmod 1) \rfloor \\
&= \lfloor 1000 \times 0.554 \rfloor \\
&= 554
\end{aligned}
$$

$$
\begin{aligned}
h(65) &= \left\lfloor 1000\left(65\left(\frac{\sqrt{5}-1}{2}\right) \bmod 1\right)\right\rfloor \\
&= \lfloor 1000 \times (40.172 \bmod 1) \rfloor \\
&= \lfloor 1000 \times 0.172 \rfloor \\
&= 172
\end{aligned}
$$

## Exercise 11.4-1

Consider inserting the keys 10, 22, 31, 4, 15, 28, 17, 88, 59 into a hash table of length $m = 11$ using open addressing with the auxiliary hash function $h'(k) = k$. Illustrate the result of inserting these keys using linear probing, using quadratic probing with $c_1 = 1$ and $c_2 = 3$, and using double hashing with $h'(k) = k$ and $h_2'(k) = 1 + (k \bmod (m - 1))$.

**Solution:**

**Linear Probing**
With linear probing, we use the hash function $h(k, i) = (h'(k) + i) \bmod m = (k + i) \bmod m$. Consider hashing each of the following keys:

1) Hashing 10:
   $h(10, 0) = (10 + 0) \bmod 11 = 10$. Thus we have $T[10] = 10$.
2) Hashing 22:
   $h(22, 0) = (22 + 0) \bmod 11 = 0$. Thus we have $T[0] = 22$.
3) Hashing 31:
   $h(31, 0) = (31 + 0) \bmod 11 = 9$. Thus we have $T[9] = 31$.
4) Hashing 4:
   $h(4, 0) = (4 + 0) \bmod 11 = 4$. Thus we have $T[4] = 4$.
5) Hashing 15:
   $h(15, 0) = (15 + 0) \bmod 11 = 4$, collision!
   $h(15, 1) = (15 + 1) \bmod 11 = 5$. Thus we have $T[5] = 15$.
6) Hashing 28:
   $h(28, 0) = (28 + 0) \bmod 11 = 6$. Thus we have $T[6] = 28$.
7) Hashing 17:
   $h(17, 0) = (17 + 0) \bmod 11 = 6$, collision!
   $h(17, 1) = (17 + 1) \bmod 11 = 7$. Thus we have $T[7] = 17$.
8) Hashing 88:
   $h(88, 0) = (88 + 0) \bmod 11 = 0$, collision!
   $h(88, 1) = (88 + 1) \bmod 11 = 1$. Thus we have $T[1] = 88$.
9) Hashing 59:
   $h(59, 0) = (59 + 0) \bmod 11 = 4$, collision!
   $h(59, 1) = (59 + 1) \bmod 11 = 5$, collision!
   $h(59, 2) = (59 + 2) \bmod 11 = 6$, collision!
   $h(59, 3) = (59 + 3) \bmod 11 = 7$, collision!
   $h(59, 4) = (59 + 4) \bmod 11 = 8$. Thus we have $T[8] = 59$.

The final hash table is shown as:

| **key** | 22 | 88 | | | 4 | 15 | 28 | 17 | 59 | 31 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **index** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

**Quadratic Probing**
With quadratic probing, and $c_1 = 1$, $c_2 = 3$, we use the hash function $h(k, i) = (h'(k) + i + 3i^2) \bmod m = (k + i + 3i^2) \bmod m$. Consider hashing each of the following keys:

1) Hashing 10:
   $h(10, 0) = (10 + 0 + 0) \bmod 11 = 10$. Thus we have $T[10] = 10$.
2) Hashing 22:
   $h(22, 0) = (22 + 0 + 0) \bmod 11 = 0$. Thus we have $T[0] = 22$.
3) Hashing 31:
   $h(31, 0) = (31 + 0 + 0) \bmod 11 = 9$. Thus we have $T[9] = 31$.
4) Hashing 4:
   $h(4, 0) = (4 + 0 + 0) \bmod 11 = 4$. Thus we have $T[4] = 4$.
5) Hashing 15:
   $h(15, 0) = (15 + 0 + 0) \bmod 11 = 4$, collision!
   $h(15, 1) = (15 + 1 + 3) \bmod 11 = 8$. Thus we have $T[8] = 15$.
6) Hashing 28:
   $h(28, 0) = (28 + 0 + 0) \bmod 11 = 6$. Thus we have $T[6] = 28$.
7) Hashing 17:

$h(17, 0) = (17 + 0 + 0) \bmod 11 = 6$, collision!
$h(17, 1) = (17 + 1 + 3) \bmod 11 = 10$, collision!
$h(17, 2) = (17 + 2 + 12) \bmod 11 = 9$, collision!
$h(17, 3) = (17 + 3 + 27) \bmod 11 = 3$. Thus we have $T[3] = 17$.

8) Hashing 88:
$h(88, 0) = (88 + 0 + 0) \bmod 11 = 0$, collision!
$h(88, 1) = (88 + 1 + 3) \bmod 11 = 4$, collision!
$h(88, 2) = (88 + 2 + 12) \bmod 11 = 3$, collision!
$h(88, 3) = (88 + 3 + 27) \bmod 11 = 8$, collision!
$h(88, 4) = (88 + 4 + 48) \bmod 11 = 8$, collision!
$h(88, 5) = (88 + 5 + 75) \bmod 11 = 3$, collision!
$h(88, 6) = (88 + 6 + 108) \bmod 11 = 4$, collision!
$h(88, 7) = (88 + 7 + 147) \bmod 11 = 0$, collision!
$h(88, 8) = (88 + 8 + 192) \bmod 11 = 2$. Thus we have $T[2] = 88$.

9) Hashing 59:
$h(59, 0) = (59 + 0 + 0) \bmod 11 = 4$, collision!
$h(59, 1) = (59 + 1 + 3) \bmod 11 = 8$, collision!
$h(59, 2) = (59 + 2 + 12) \bmod 11 = 7$. Thus we have $T[7] = 59$.

The final hash table is shown as:

| **key** | 22 | | 88 | 17 | 4 | | 28 | 59 | 15 | 31 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **index** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

**Doubling Hashing**

With double hashing, we use the hash function:

$h(k, i) = (h'(k) + ih_2'(k)) \bmod m = (k + i(1 + (k \bmod (m - 1)))) \bmod m$.

Consider hashing each of the following keys:

1) Hashing 10:
$h(10, 0) = (10 + 0) \bmod 11 = 10$. Thus we have $T[10] = 10$.

2) Hashing 22:
$h(22, 0) = (22 + 0) \bmod 11 = 0$. Thus we have $T[0] = 22$.

3) Hashing 31:
$h(31, 0) = (31 + 0) \bmod 11 = 9$. Thus we have $T[9] = 31$.

4) Hashing 4:
$h(4, 0) = (4 + 0) \bmod 11 = 4$. Thus we have $T[4] = 4$.

5) Hashing 15:
$h(15, 0) = (15 + 0) \bmod 11 = 4$, collision!
$h(15, 1) = (15 + 1 * h_2'(15)) \bmod 11 = 10$, collision!
$h(15, 2) = (15 + 2 * h_2'(15)) \bmod 11 = 5$. Thus we have $T[5] = 15$.

6) Hashing 28:
$h(28, 0) = (28 + 0) \bmod 11 = 6$. Thus we have $T[6] = 28$.

7) Hashing 17:
$h(17, 0) = (17 + 0) \bmod 11 = 6$, collision!
$h(17, 1) = (17 + 1 * h_2'(17)) \bmod 11 = 3$. Thus we have $T[3] = 17$.

8) Hashing 88:
$h(88, 0) = (88 + 0) \bmod 11 = 0$, collision!
$h(88, 1) = (88 + 1 * h_2'(88)) \bmod 11 = 9$, collision!
$h(88, 2) = (88 + 2 * h_2'(88)) \bmod 11 = 7$. Thus we have $T[7] = 88$.

9) Hashing 59:
$h(59, 0) = (59 + 0) \bmod 11 = 4$, collision!
$h(59, 1) = (59 + 1 * h_2'(59)) \bmod 11 = 3$, collision!
$h(59, 2) = (59 + 2 * h_2'(59)) \bmod 11 = 2$. Thus we have $T[2] = 59$.

The final hash table is shown as:

| **key** | 22 | | 59 | 17 | 4 | 15 | 28 | 88 | | 31 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **index** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

# Homework Solutions – Unit 3: Chapter 13

CMPSC 465

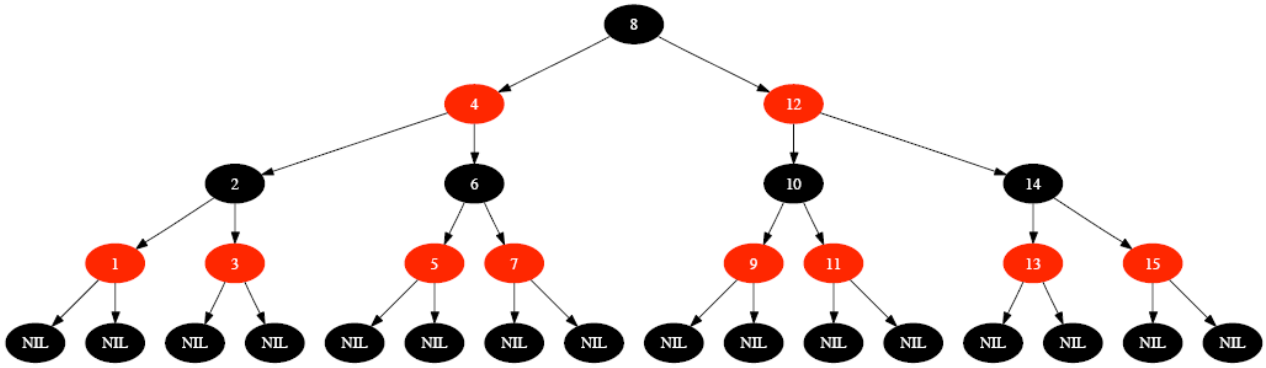
Disclaimer: This is a draft of solutions that has been prepared by the TAs and the instructor makes no guarantees that solutions presented here contain the level of detail that would be expected on an exam. Any errors or explanations you find unclear should be reported to either of the TAs for corrections first.

## Exercise 13.1-1

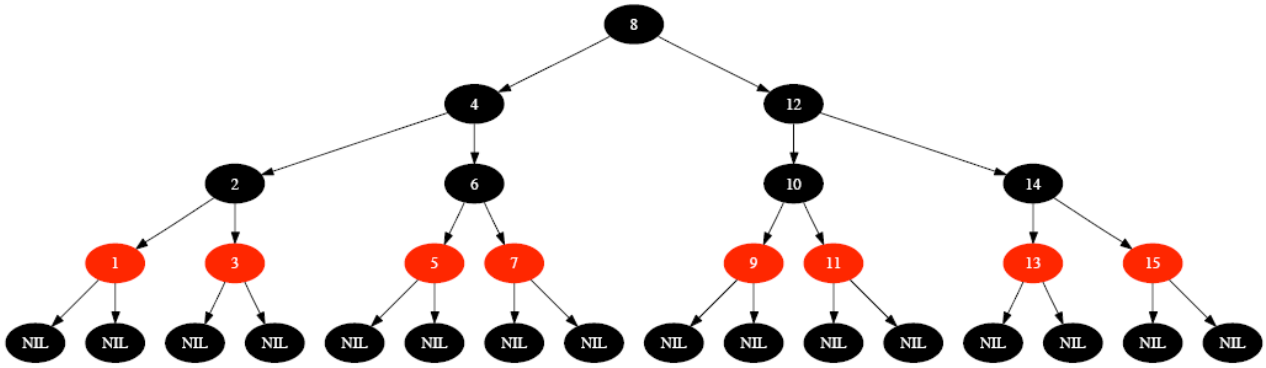
In the style of Figure 13.1(a), draw the complete binary search tree of height 3 on the keys {1, 2, …, 15}. Add the NIL leaves and color the nodes in three different ways such that the black-heights of the resulting red-black trees are 2, 3, and 4.

**Solution**:

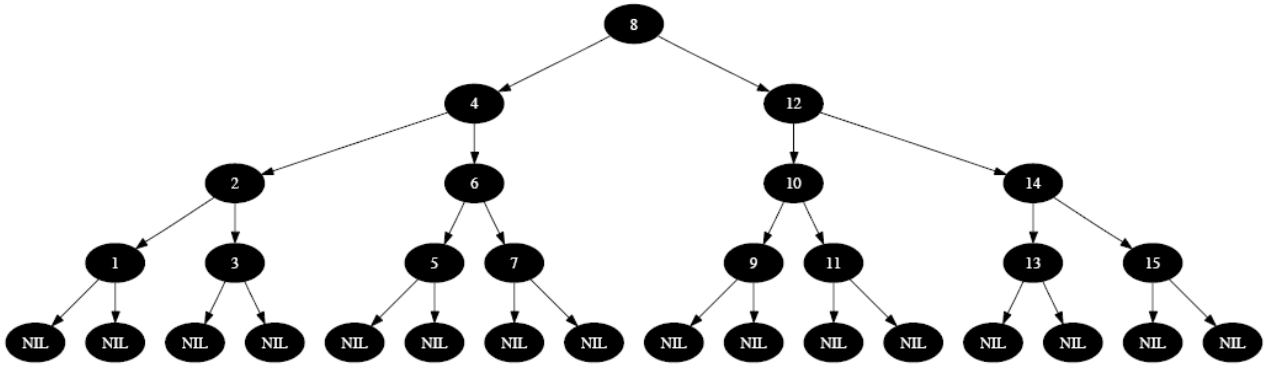
Black-height 2:




Black-height 3:

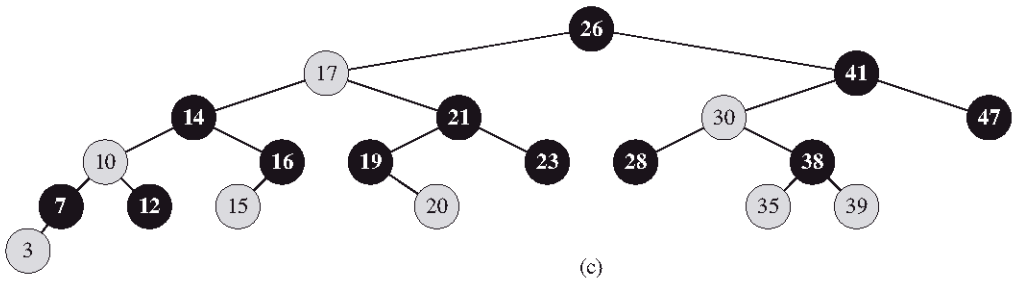



Black-height 4:



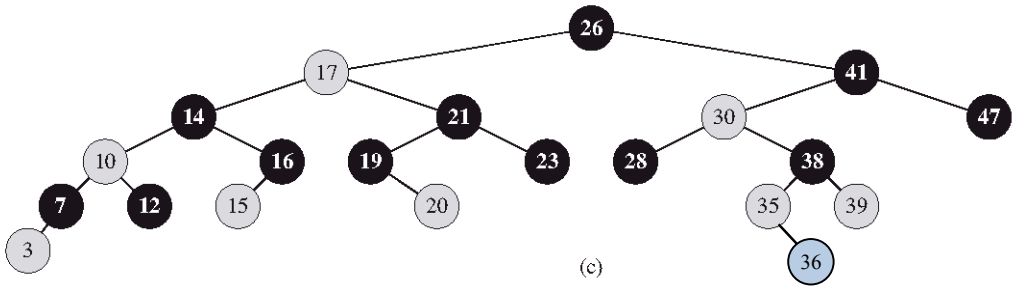
## Exercise 13.1-2

Draw the red-black tree that results after TREE-INSERT is called on the tree shown in the figure below with key 36. If the inserted node is colored red, is the resulting tree a red-black tree? What if it is colored black?




**Solution**:

If the node with key 36 is inserted and colored red, the red-black tree becomes:

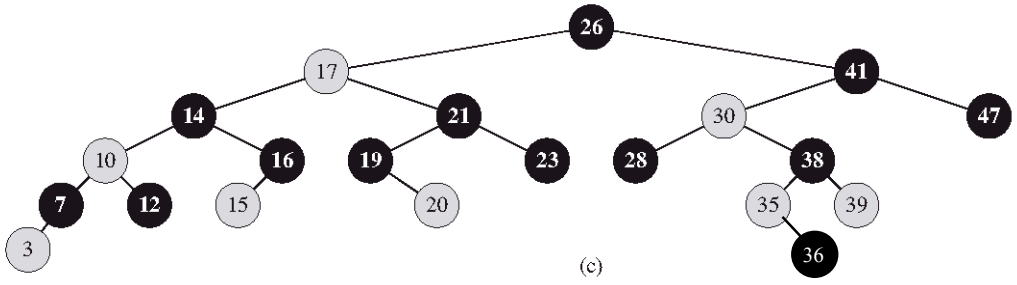



We can see that it violates following red-black tree property:

A red node in the red-black tree cannot have a red node as its child.

So the resulting tree is not a red-black tree.

If the node with key 36 is inserted and colored black, the red-black tree becomes:




We can see that it violates following red-black tree property:

For each node, all paths from the node to descendent leaves contain the same number of black nodes (e.g. consider node with key 30).

So the resulting tree is not a red-black tree either.

## Exercise 13.1-5

Show that the longest simple path from a node $x$ in a red-black tree to a descendant leaf has length at most twice that of the shortest simple path from node $x$ to a descendant leaf.

**Proof**:

In the longest path, at least every other node is black. In the shortest path, at most every node is black. Since the two paths contain equal numbers of black nodes, the length of the longest path is at most twice the length of the shortest path.

We can say this more precisely, as follows:

Since every path contains bh($x$) black nodes, even the shortest path from $x$ to a descendant leaf has length at least bh($x$). By definition, the longest path from $x$ to a descendant leaf has length height($x$). Since the longest path has bh($x$) black nodes and at least half the nodes on the longest path are black (by property 4 in CLRS), bh($x$) ≥ height($x$)/2, so

length of longest path = height($x$) ≤ 2×bh($x$) ≤ twice length of shortest path.

## Exercise 13.2-1

Write pseudocode for RIGHT-ROTATE.

**Solution**:

The pseudocode for RIGHT-ROTATE is shown below:

```
RIGHT-ROTATE (T, x)
    y = x.left                  //set y
    x.left = y.right            // turn y's right subtree into x's left subtree
    if y.right ≠ NIL
        y.right.p = x
    y.p = x:p                   // link x's parent to y
    if x.p == NIL
        T.root = y
    else if x == x.p.right
        x.p.right = y
    else
        x.p.left = y
    y.right = x                 //put x on y's right
    x.p = y
```

## Exercise 13.2-4

Show that any arbitrary $n$-node binary search tree can be transformed into any other arbitrary $n$-node binary search tree using $O(n)$ rotations. (Hint: First show that at most $n-1$ right rotations suffice to transform the tree into a right-going chain.)

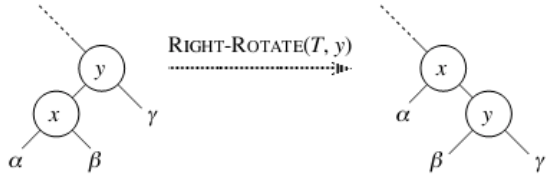
**Solution:**

Since the exercise asks about binary search trees rather than the more specific red-black trees, we assume here that leaves are full-fledged nodes, and we ignore the sentinels.

Taking the book's hint, we start by showing that with at most $n-1$ right rotations, we can convert any binary search tree into one that is just a right-going chain.

The idea is: Let us define the ***right spine*** as the root and all descendants of the root that are reachable by following only right pointers from the root. A binary search tree that is just a right-going chain has all $n$ nodes in the right spine.

As long as the tree is not just a right spine, repeatedly find some node $y$ on the right spine that has a non-leaf left child $x$ and then perform a right rotation on $y$:




(In the above figure, note that any of α, β, and γ can be an empty subtree.)

Observe that this right rotation adds $x$ to the right spine, and no other nodes leave the right spine. Thus, this right rotation increases the number of nodes in the right spine by 1. Any binary search tree starts out with at least one node — the root — in the right spine. Moreover, if there are any nodes not on the right spine, then at least one such node has a parent on the right spine. Thus, at most $n-1$ right rotations are needed to put all nodes in the right spine, so that the tree consists of a single right-going chain.

If we knew the sequence of right rotations that transforms an arbitrary binary search tree $T$ to a single right-going chain $T'$, then we could perform this sequence in reverse — turning each right rotation into its inverse left rotation — to transform $T'$ back into $T$.

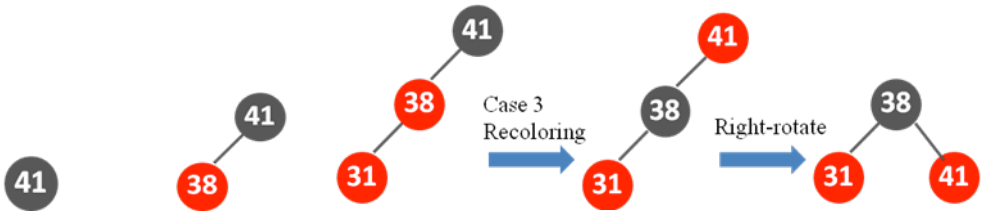
Therefore, here is how we can transform any binary search tree $T_1$ into any other binary search tree $T_2$. Let $T'$ be the unique right-going chain consisting of the nodes of $T_1$ (which is the same as the nodes of $T_2$). Let $r = \langle r_1, r_2, ..., r_k \rangle$ be a sequence of right rotations that transforms $T_1$ to $T'$, and let $r' = \langle r'_1, r'_2, ..., r'_{k'} \rangle$ be a sequence of right rotations that transforms $T_2$ to $T'$. We know that there exist sequences $r$ and $r'$ with $k, k' \le n-1$. For each right rotation $r'_i$, let $l'_i$ be the corresponding inverse left rotation. Then the sequence $\langle r_1, r_2, ..., r_k, l'_{k'}, l'_{k'-1}, \ldots, l'_2, l'_1 \rangle$ transforms $T_1$ to $T_2$ in at most $2n-2$ rotations.

## Exercise 13.3-2

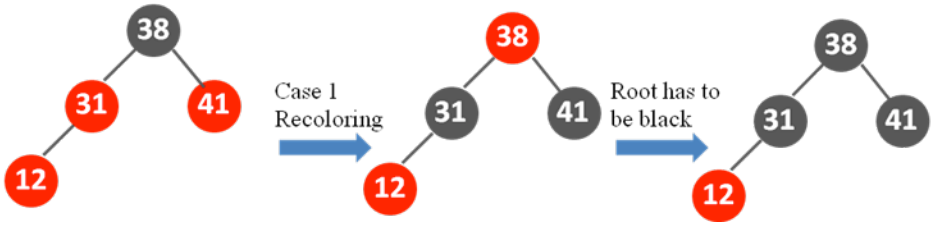
Show the red-black trees that result after successively inserting the keys 41, 38, 31, 12, 19, 8 into an initially empty red-black tree.

**Solution**:

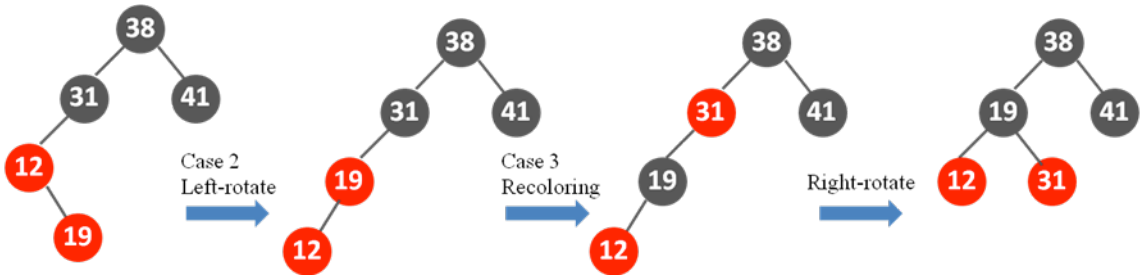
Insert 41 Insert 38 Insert 31



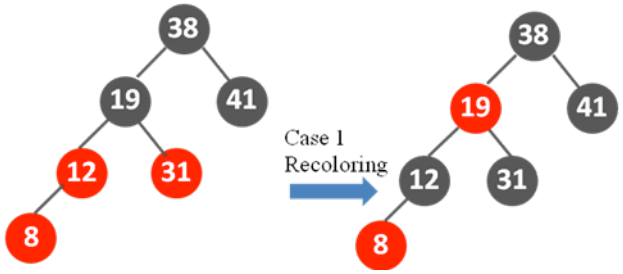

Insert 12




Insert 19




Insert 8



## Exercise 13.3-3

Suppose that the black-height of each of the subtrees $\alpha, \beta, \gamma, \delta, \varepsilon$ in Figures 13.5 and 13.6 is $k$. Label each node in each figure with its black-height to verify that the indicated transformation preserves property 5.

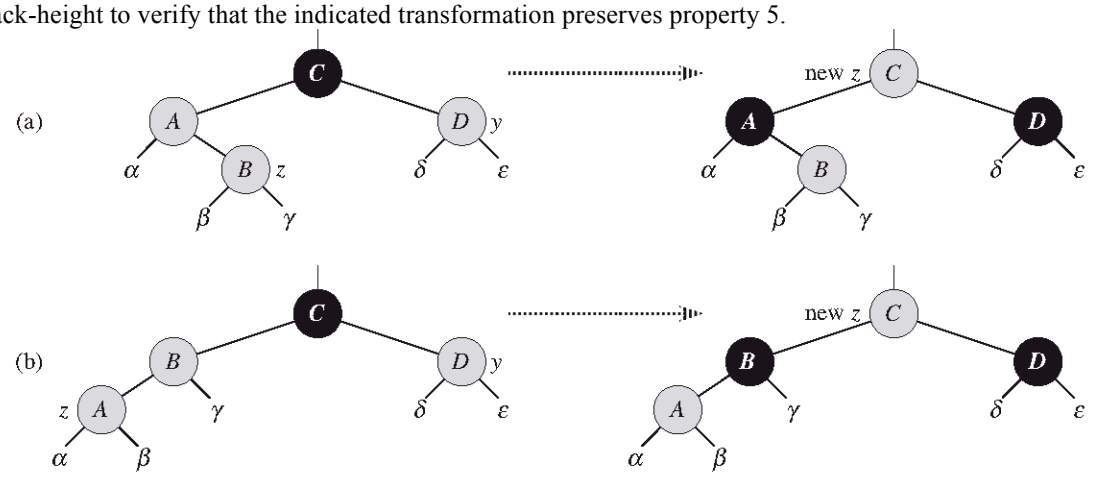



Figure 13.5

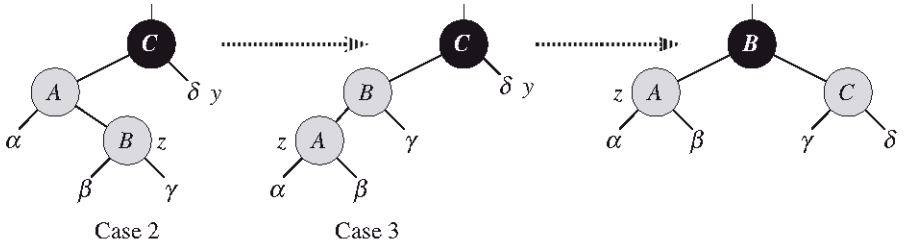
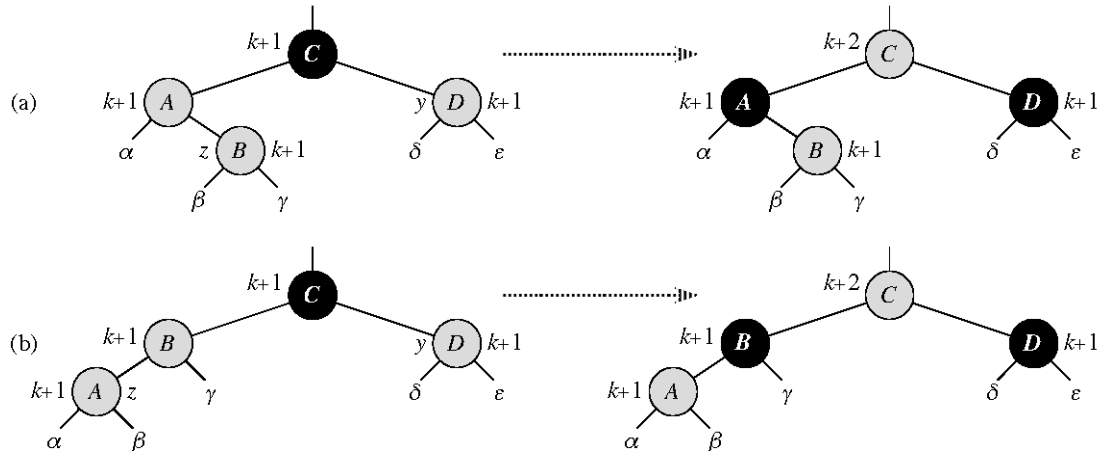


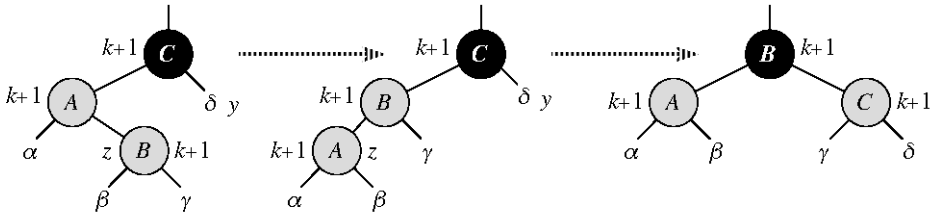

Figure 13.6

**Solution**:

In Figure 13.5, nodes $A$, $B$, and $D$ have black-height $k + 1$ in all cases, because each of their subtrees has black-height $k$ and a black root. Node $C$ has black-height $k + 1$ on the left (because its red children have black-height $k + 1$) and black-height $k + 2$ on the right (because its black children have black-height $k + 1$).




In Figure 13.6, nodes $A$, $B$, and $C$ have black-height $k + 1$ in all cases. At left and in the middle, each of $A$'s and $B$'s subtrees has black-height $k$ and a black root, while C has one such subtree and a red child with black-height $k + 1$. At the right, each of $A$'s and $C$'s subtrees has black-height $k$ and a black root, while $B$'s red children each have black-height $k + 1$.




Property 5 is preserved by the transformations. We have shown above that the black-height is well-defined within the subtrees pictured, so property 5 is preserved within those subtrees. Property 5 is preserved for the tree containing the subtrees pictured, because every path through these subtrees to a leaf contributes $k + 2$ black nodes.

# Homework Solutions – Unit 3: Chapter 18

CMPSC 465

Disclaimer: This is a draft of solutions that has been prepared by the TAs and the instructor makes no guarantees that solutions presented here contain the level of detail that would be expected on an exam. Any errors or explanations you find unclear should be reported to either of the TAs for corrections first.

## Exercise 18.1-1

Why don't we allow a minimum degree of $t = 1$?

**Solution**:

According to the definition, minimum degree $t$ means every node other than the root must have at least $t - 1$ keys, and every internal node other than the root thus has at least $t$ children. So, when $t = 1$, it means every node other than the root must have at least $t - 1 = 0$ key, and every internal node other than the root thus has at least $t = 1$ child.

Thus, we can see that the minimum case doesn't exist, because no node exists with 0 key, and no node exists with only 1 child in a B-tree.

## Exercise 18.1-2

For what values of $t$ is the tree of Figure 18.1 a legal B-tree?

**Solution**:

According to property 5 of B-tree, every node other than the root must have at least $t-1$keys and may contain at most $2t-1$ keys. In Figure 18.1, the number of keys of each node (except the root) is either 2 or 3. So to make it a legal B-tree, we need to guarantee that

$t - 1 \leq 2$ and $2t - 1 \geq 3$,

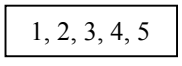
which yields $2 \leq t \leq 3$. So $t$ can be 2 or 3.

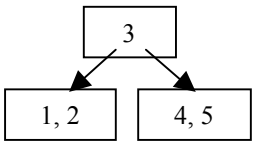
## Exercise 18.1-3

Show all legal B-trees of minimum degree 3 that represent {1, 2, 3, 4, 5}.

**Solution**:

We know that every node except the root must have at least $t - 1 = 2$ keys, and at most $2t - 1 = 5$ keys. Also remember that the leaves stay in the same depth. Thus, there are 2 possible legal B-trees:



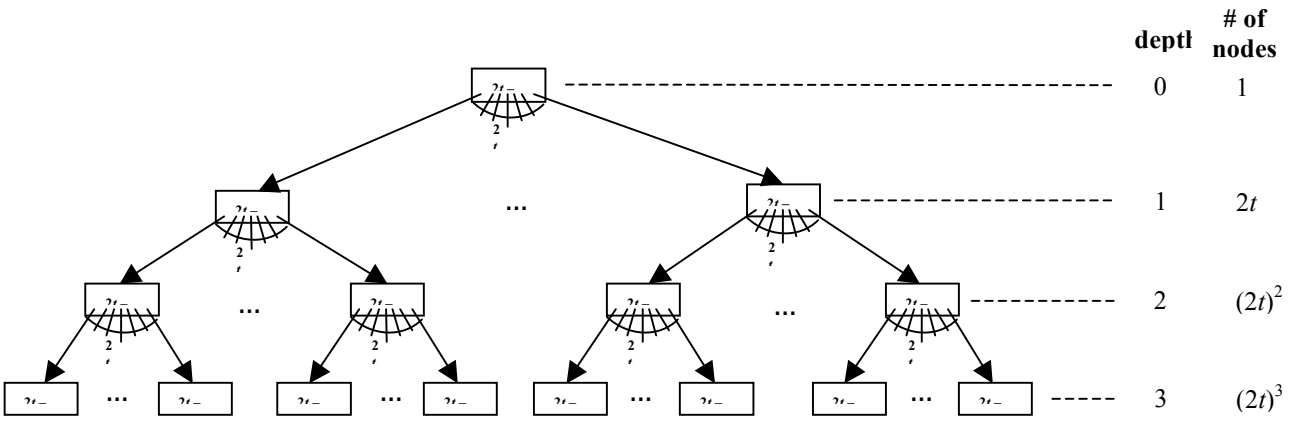



## Exercise 18.1-4

As a function of the minimum degree $t$, what is the maximum number of keys that can be stored in a B-tree of height $h$?

**Solution**:

A B-tree with maximum number of keys is shown below:




We can see that each node contains $2t - 1$ keys, and at depth $k$, the tree at most has $(2t)^k$ nodes. The total nodes is therefore the sum of $(2t)^0$, $(2t)^1$, $(2t)^2$, $(2t)^3$, …, $(2t)^h$. Let MaxKeyNum($t$, $h$) as a function that returns the maximum number of keys in a B-tree of height $h$ and the minimum degree $t$. We can get that:

$$\begin{aligned}
\text{MaxKeyNum}(t, h) &= (2t-1)[(2t)^0 + (2t)^1 + (2t)^2 + (2t)^3 + \ldots + (2t)^h] && \text{as [keys per node] * [total \# of nodes]} \\
&= (2t-1)\sum_{i=0}^{h}(2t)^i && \text{by using Sigma to sum up total \# of nodes} \\
&= (2t-1)\frac{(2t)^{h+1}-1}{2t-1} && \text{by the summation formula of geometric series} \\
&= (2t)^{h+1}-1
\end{aligned}$$

## Exercise 18.1-5

Describe the data structure that would result if each black node in a red-black tree were to absorb its red children, incorporating their children with its own.

**Solution**:

After absorbing each red node into its black parent, each black node may contain 1, 2 (1 red child), or 3 (2 red children) keys, and all leaves of the resulting tree have the same depth, according to property 5 of red-black tree (For each node, all paths from the node to descendant leaves contain the same number of black nodes). Therefore, a red-black tree will become a B-tree with minimum degree $t = 2$, i.e., a 2-3-4 tree.

## Exercise 18.2-3

Explain how to find the minimum key stored in a B-tree and how to find the predecessor of a given key stored in a B-tree.

**Solution**:

Finding the minimum in a B-tree is quite similar to finding a minimum in a binary search tree. We need to find the left most leaf for the given root, and return the first key.

```
B-TREE-FIND-MIN(x)
//PRE: x is a node on the B-tree T. The top level call is B-TREE-FIND-MIN(T.root).
//POST: FCTVAL is the minimum key stored in the subtree rooted at x.
{
    if x == NIL                          //T is empty
    {
        return NIL
    }
    else if x.leaf                       //x is leaf
    {
        return x.key_1                   //return the minimum key of x
    }
    else
    {
        DISK-READ(x.c_1)
        return B-TREE-FIND-MIN(x.c_1)
    }
}
```

Finding the predecessor of a given key $x.key_i$ is according to the following rules:

- If $x$ is not a leaf, return the maximum key in the $i$-th child of $x$, which is also the maximum key of the subtree rooted at $x.c_i$
- If $x$ is a leaf and $i > 1$, return the $(i–1)$st key of x, i.e., $x.key_{i–1}$
- Otherwise, look for the last node $y$ (from the bottom up) and $j > 0$, such that $x.key_i$ is the leftmost key in $y.c_j$; if j = 1, return NIL since $x.key_i$ is the minimum key in the tree; otherwise we return $y.key_{j–1}$.

```
B-TREE-FIND-PREDECESSOR(x, i)
//PRE: x is a node on the B-tree T. i is the index of the key.
//POST: FCTVAL is the predecessor of x.key_i.
{
    if ! x.leaf
    {
        DISK-READ(x.c_i)
        return B-TREE-FIND-MAX(x.c_i)
    }
    else if i > 1                        //x is a leaf and i > 1
        return x.key_{i–1}
    else                                 //x is a leaf and i = 1
    {
        z = x

        while (1)
        {
            if z.p == NIL                //z is root
                return NIL               // z.key_i is the minimum key in T; no predecessor

            y = z.p
            j = 1
            DISK-READ(y.c_1)
```

```
            while (y.c_j != x)
            {
                j = j + 1
                DISK-READ(y.c_j)
            }

            if j == 1
                z = y
            else
                return y.key_{j-1}
        }
    }
}
```

```
B-TREE-FIND-MAX(x)
//PRE: x is a node on the B-tree T. The top level call is B-TREE-FIND-MAX(T.root).
//POST: FCTVAL is the maximum key stored in the subtree rooted at x.
{
    if x == NIL                                  //T is empty
    {
        return NIL
    }
    else if x.leaf                               //x is leaf
    {
        return (x, x.n)                          //return the maximum key of x
    }
    else
    {
        DISK-READ(x.c_{x.n+1})
        return B-TREE-FIND-MAX(x.c_{x.n+1})
    }
}
```

## Exercise 18.2-6

Suppose that we were to implement B-TREE-SEARCH to use binary search rather than linear search within each node. Show that this change makes the CPU time required $O(\lg n)$, independently of how $t$ might be chosen as a function of $n$.

**Solution**:

As in the TREE-SEARCH procedure for binary search trees, the nodes encountered during the recursion form a simple path downward from the root of the tree. Thus, the B-TREE-SEARCH procedure needs $O(h) = O(\log_t n)$ CPU time to search along the path, where $h$ is the height of the B-tree and $n$ is the number of keys in the B-tree, and we know that $h \leq \log_t \frac{n+1}{2}$. Since the number of keys in each node is less than $2t - 1$, a binary search within each node is $O(\lg t)$. So the total time is:

$$O(\lg t * \log_t n) = O(\lg t * \frac{\lg n}{\lg t}) \quad \text{by changing the base of the logarithm}$$
$$= O(\lg n)$$

Thus, the CPU time required is $O(\lg n)$.

# Homework Solutions – Unit 4: Chapter 22

CMPSC 465

Disclaimer: This is a draft of solutions that has been prepared by the TAs and the instructor makes no guarantees that solutions presented here contain the level of detail that would be expected on an exam. Any errors or explanations you find unclear should be reported to either of the TAs for corrections first.

## Exercise 22.1-1

Given an adjacency-list representation of a directed graph, how long does it take to compute the out-degree of every vertex? How long does it take to compute the in-degrees?

**Solution**:

Given an adjacency-list representation $Adj$ of a directed graph, the out-degree of a vertex $u$ is equal to the length of $Adj[u]$, and the sum of the lengths of all the adjacency lists in $Adj$ is $|E|$. Thus the time to compute the out-degree of every vertex is $\Theta(|V| + |E|)$.
The in-degree of a vertex $u$ is equal to the number of times it appears in all the lists in $Adj$. If we search all the lists for each vertex, the time to compute the in-degree of every vertex is $\Theta(|V||E|)$.

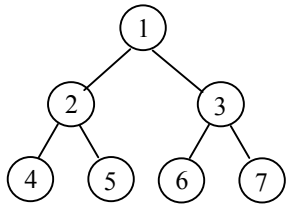
(Alternatively, we can allocate an array $T$ of size $|V|$ and initialize its entries to zero. Then we only need to scan the lists in $Adj$ once, incrementing $T[u]$ when we see $u$ in the lists. The values in $T$ will be the in-degrees of every vertex. This can be done in $\Theta(|V| + |E|)$ time with $\Theta(|V|)$ additional storage.)

## Exercise 22.1-2

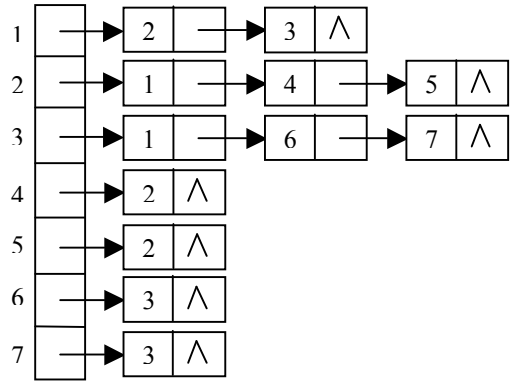
Give an adjacency-list representation for a complete binary tree on 7 vertices. Give an equivalent adjacency-matrix representation. Assume that vertices are numbered from 1 to 7 as in a binary heap.

**Solution**:

A complete binary tree looks like:




An adjacency-list representation of this tree is shown below:

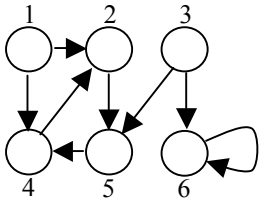



An equivalent adjacency-matrix representation of this tree is shown below:

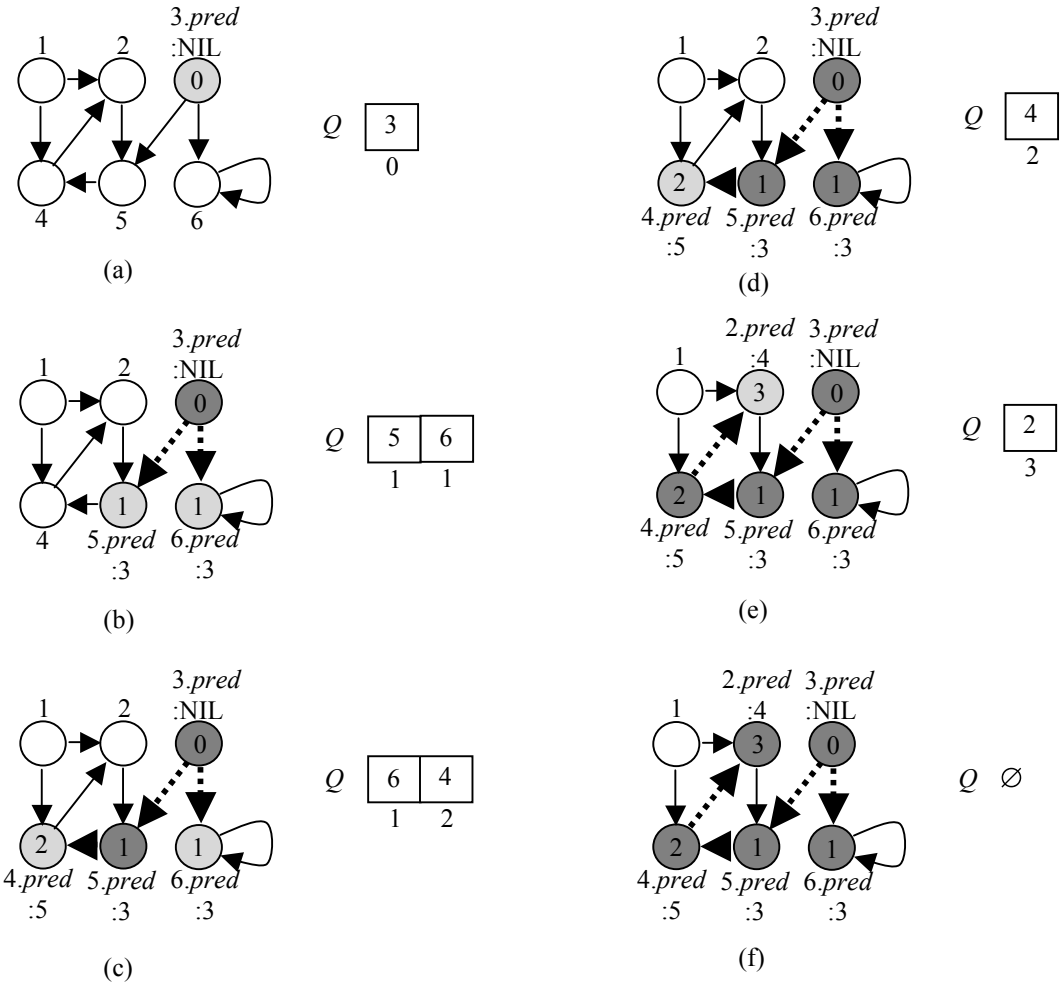
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 2 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| 3 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| 4 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 6 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 7 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |

## Exercise 22.2-1

Show the *dist* and *pred* values that result from running breadth-first search on the directed graph below, using vertex 3 as the source.




**Solution:**

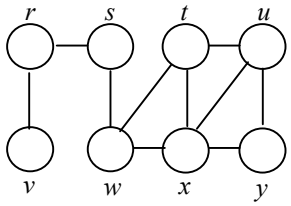



The procedure of the breadth-first search is shown above. From the result, we can see that:

| Node | *Pred* | *dist* |
|---|---|---|
| 3 | NIL | 0 |
| 5 | 3 | 1 |
| 6 | 3 | 1 |
| 4 | 5 | 2 |
| 2 | 4 | 3 |
| 1 | NIL | ∞ |

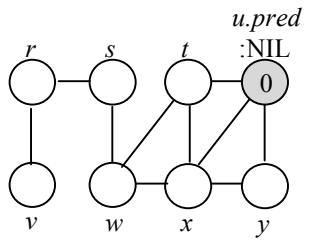
## Exercise 22.2-2

Show the *dist* and *pred* values that result from running breadth-first search on the undirected graph below, using vertex *u* as the source.



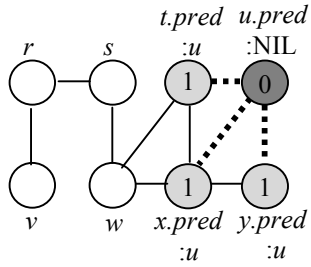

**Solution:**

(a)



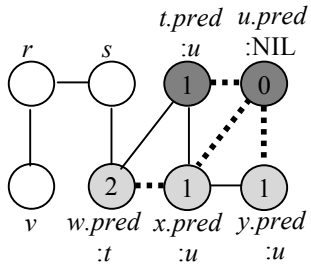

$Q$ | $u$ | 0

(b)



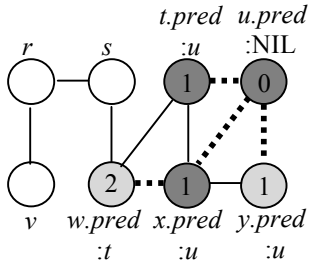

$Q$ | $t$ | $x$ | $y$ | 1 | 1 | 1

(c)



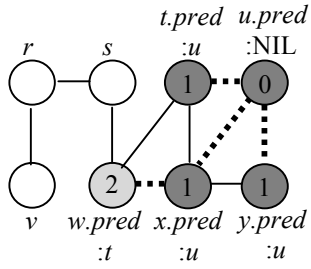

$Q$ | $x$ | $y$ | $w$ | 1 | 1 | 2

(d)



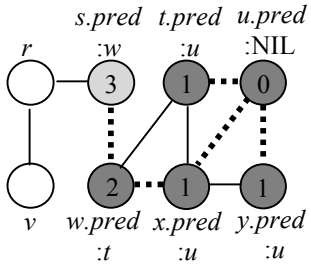

$Q$ | $y$ | $w$ | 1 | 2

(e)



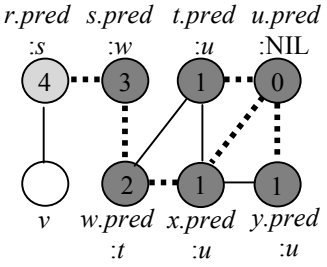

$Q$ | $w$ | 2

(f)



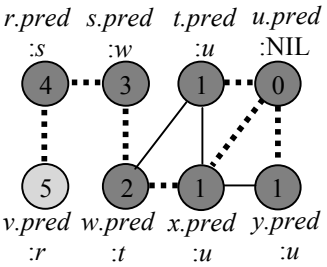

$Q$ | $s$ | 3

(g)



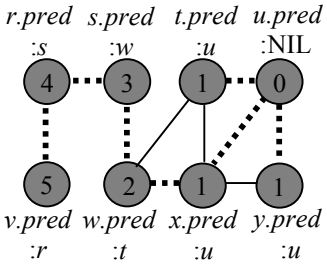

$Q$ | $r$ | 4

(h)




$Q$ | $v$ | 5

(i)




$Q$ | $\varnothing$

The procedure of the breadth-first search is shown above. From the result, we can see that:

| Node | *Pred* | *dist* |
|---|---|---|
| *u* | NIL | 0 |
| *t* | *u* | 1 |
| *x* | *u* | 1 |
| *y* | *u* | 1 |
| *w* | *t* | 2 |
| *s* | *w* | 3 |
| *r* | *s* | 4 |
| *v* | *r* | 5 |

## Exercise 22.3-1

Make a 3-by-3 chart with row and column labels WHITE, GRAY, and BLACK. In each cell (*i*, *j*), indicate whether, at any point during a depth-first search of directed graph, there can be an edge from a vertex of color *i* to a vertex of color *j*. For each possible edge, indicate what edge types it can be. Make a second such chart for depth-first search of an undirected graph.

**Solution**:

Directed graph:

| | WHITE | GRAY | BLACK |
|---|---|---|---|
| WHITE | tree, back, forward, and cross | back and cross | cross |
| GRAY | tree and forward | tree, forward, back | tree, forward, and cross |
| BLACK | × | back and cross | tree, forward, back and cross |

Undirected graph:

| | WHITE | GRAY | BLACK |
|---|---|---|---|
| WHITE | tree and back | tree and back | × |
| GRAY | tree and back | tree and back | tree and back |
| BLACK | × | tree and back | tree and back |

## Exercise 22.3-2

Show how depth-first search works on the graph of Figure 22.6. Assume that the **for** loop of lines 5–7 of the DFS procedure considers the vertices in alphabetical order, and assume that each adjacency list is ordered alphabetically. Show the discovery and finishing times for each vertex, and show the classification of each edge.

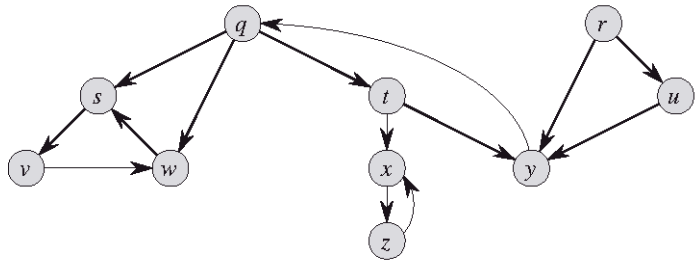
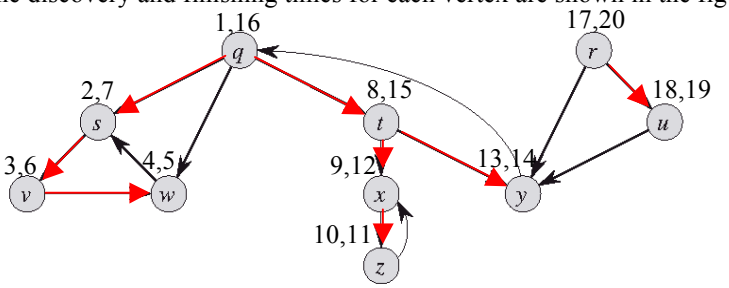



**Figure 22.6** A directed graph for use in Exercises 22.3-2 and 22.5-2.

**Solution**:

The discovery and finishing times for each vertex are shown in the figure below:




Tree edges: $(q, s)$, $(s, v)$, $(v, w)$, $(q, t)$, $(t, x)$, $(x, z)$, $(t, y)$, $(r, u)$
Back edges: $(w, s)$, $(z, x)$, $(y, q)$
Forward edges: $(q, w)$
Cross edges: $(r, y)$, $(u, y)$

## Exercise 22.3-3

Show the parenthesis structure of the depth-first search of Figure 22.4.

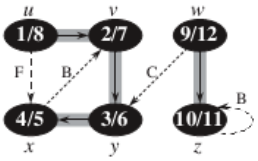
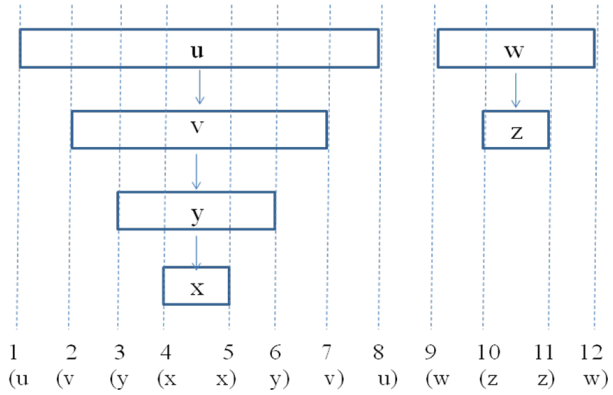



Figure 22.4 (p)

**Solution**:




## Exercise 22.3-7

Rewrite the procedure DFS, using a stack to eliminate recursion.

**Solution**:

Assume that the stack has following operations:
PUSH($S$, $v$) – pushes $v$ into the stack;
POP($S$) – returns the top of the stack and removes it;
TOP($S$) – returns the top of the stack without removing it.

Denote an empty stack by EMPTY.

Then the pseudocode of this algorithm becomes as follows:

```
DFS_STACK (G, s)
    for each vertex u ∈ V(G) – {s}
    {
       u.color  = WHITE
       u.pred = NIL
    }
    time = 0
    S = EMPTY

    time = time + 1
    s.t_d = time
    s.color = GRAY
    PUSH(S, s)
    while S != EMPTY
    {
       t = TOP(S)
       if ∃v ∈ V(G).Adj[t] s.t. v.color == WHITE // v's adjacency list hasn't been fully examined
       {
          v.pred = t
          time = time + 1
          v.t_d = time
          v.color = GRAY
          PUSH(S, v)
       }
       else                                          // v's adjacency list has been fully examined
       {
          t = POP(S)
          time = time + 1
          v.t_f = time
          v.color = BLACK
       }
    }
```

## Exercise 22.4-1

Show the ordering of vertices produced by TOPOLGICAL-SORT when it is run on the dag of Figure 22.8, under the assumption of Exercise 22.3-2.

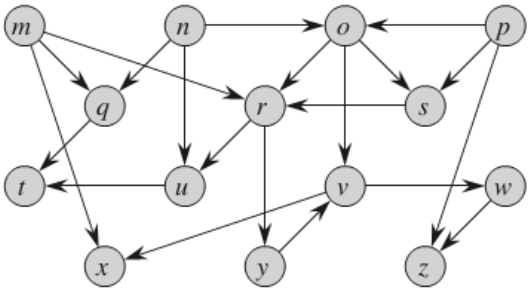


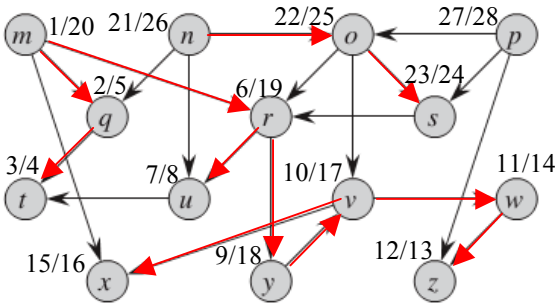

**Figure 22.8** A dag for topological sorting.

**Solution:**

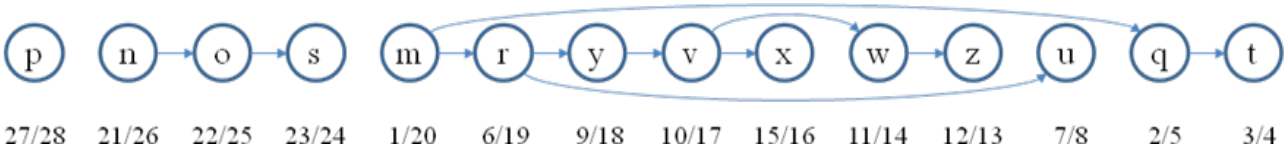
According to the assumption of Exercise 22.3-2, the **for** loop of lines 5–7 of the DFS procedure

considers the vertices in alphabetical order, and that each adjacency list is ordered alphabetically.

DFS on Figure 22.8:




Ordering of vertices:




## Exercise 22.4-2

Give a linear-time algorithm that takes as input a directed acyclic graph $G = (V, E)$ and two vertices $s$ and $t$ , and returns the number of simple paths from $s$ to $t$ in $G$. For example, the directed acyclic graph of Figure 22.8 contains exactly four simple paths from vertex $p$ to vertex $v$: $pov$, $poryv$, $posryv$, and $psryv$.
(Your algorithm needs only to count the simple paths, not list them.)

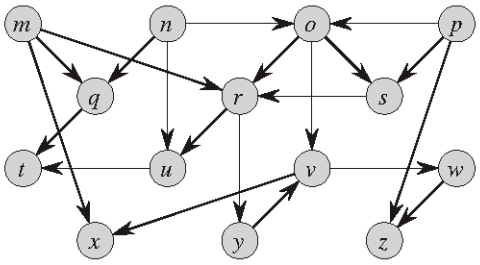



**Figure 22.8** A dag for topological sorting.

**Solution**:

Add a field to the vertex representation to hold an integer count. Initially, set vertex $t$'s count to 1 and other vertices' count to 0. Start running DFS with $s$ as the start vertex. When $t$ is discovered, it should be immediately marked as finished (BLACK), without further processing starting from it. Subsequently, each time DFS finishes a vertex $v$, set $v$'s count to the sum of the counts of all vertices adjacent to $v$. When DFS finishes vertex $s$, stop and return the count computed for $s$.

# Homework Solutions – Unit 4: Chapter 23

CMPSC 465

Disclaimer: This is a draft of solutions that has been prepared by the TAs and the instructor makes no guarantees that solutions presented here contain the level of detail that would be expected on an exam. Any errors or explanations you find unclear should be reported to either of the TAs for corrections first.

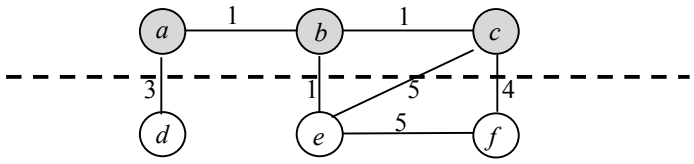
## Exercise 23.1-2

Professor Sabatier conjectures the following converse of Theorem 23.1. Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function $w$ defined on $E$. Let $A$ be a subset of $E$ that is included in some minimum spanning tree for $G$, let $(S, V - S)$ be any cut of $G$ that respects $A$, and let $(u, v)$ be a safe edge for $A$ crossing $(S, V - S)$. Then, $(u, v)$ is a light edge for the cut. Show that the professor's conjecture is incorrect by giving a counterexample.

**Solution**:

That is false. A safe edge (with respect to the set of edges $A$) is defined to be any edge such that when we add it to $A$, we still have a subset of some minimum spanning tree. This definition does not reference light edges in any way. In the situation from a counterexample, there may be multiple safe edges from $S$ to $V - S$, not all of them have to be light.

A counterexample is shown below. Let $S = \{a, b, c\}$, $A = \{(a, b), (b, c\}$, we can see that $(a, d)$, $(b, e)$, $(c, f)$ are all safe edges for $A$ crossing $(S, V - S)$, but only $(b, e)$ is a light edge for the cut.




## Exercise 23.1-3

Show that if an edge $(u, v)$ is contained in some minimum spanning tree, then it is a light edge crossing some cut of the graph.

**Solution**:

Suppose $(u, v) \in T$, a minimal spanning tree. Let $A = \{T - (u, v)\}$. $A$ contains two trees: $A = T_u \cup T_v$. $T_u$ is the tree in which vertex $u$ appears, and $T_v$ is the tree in which vertex $v$ appears. Moreover, $V_u \cup V_v = V$, where $V_u$ contains the vertices of $T_u$ and $V_v$ contains the vertices of $T_v$. That is, $(V_u, V_v)$ is a cut, which is crossed by $(u, v)$ and which respects $A$. Any edge crossing the cut rejoins the two trees. If there is a crossing edge $(x, y)$ with $w(x, y) < w(u, v)$, then $T' = T_u \cup T_v \cup (x, y)$ is a spanning tree with weight:

$$
\begin{aligned}
w(T') &= w(T_u \cup T_v) + w(x, y) && \text{as } T' = T_u \cup T_v \cup (x, y) \\
&= w(T_u \cup T_v) + w(u, v) + [w(x, y) - w(u, v)] && \text{by adding 0, that is } w(u, v) - w(u, v) \\
&= w(T) - [w(u, v) - w(x, y)] && \text{as } T = T_u \cup T_v \cup (u, v) \\
&< w(T)
\end{aligned}
$$

This contradicts with that $(u, v)$ is contained in some minimum spanning tree. So, $w(u, v) < w(x, y)$ for all $(x, y)$ crossing the cut $(V_u, V_v)$. That is, $(u, v)$ is a light edge crossing the cut.

## Exercise 23.2-1

Kruskal's algorithm can return different spanning trees for the same input graph $G$, depending on how it breaks ties when the edges are sorted into order. Show that for each minimum spanning tree $T$ of $G$, there is a way to sort the edges of $G$ in Kruskal's algorithm so that the algorithm returns $T$.

**Solution**:

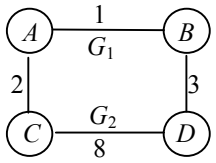
We would start by sorting the edges in of $G$ in non-descending order. In addition, we would want that among the edges of same weight, the edges which are contained in $T$ are placed in first positions.

## Exercise 23.2-8

Professor Borden proposes a new divide-and-conquer algorithm for computing minimum spanning trees, which goes as follows. Given a graph $G = (V, E)$, partition the set $V$ of vertices into two sets $V_1$ and $V_2$ such that $|V_1|$ and $|V_2|$ differ by at most 1. Let $E_1$ be the set of edges that are incident only on vertices in $V_1$, and let $E_2$ be the set of edges that are incident only on vertices in $V_2$. Recursively solve a minimum-spanning-tree problem on each of the two subgraphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. Finally, select the minimum-weight edge in $E$ that crosses the cut $(V_1, V_2)$, and use this edge to unite the resulting two minimum spanning trees into a single spanning tree.
Either argue that the algorithm correctly computes a minimum spanning tree of $G$, or provide an example for which the algorithm fails.

**Solution**:

We argue that the algorithm fails. Consider the graph $G$ below. We partition $G$ into $V_1$ and $V_2$ as follows: $V_1 = \{A, B\}$, $V_2 = \{C, D\}$. $E_1 = \{(A, B)\}$. $E_2 = \{(C, D)\}$. The set of edges that cross the cut is $E_c = \{(A, C), (B, D)\}$.




Now, we must recursively find the minimum spanning trees of $G_1$ and $G_2$. We can see that in this case, MST($G_1$) = $G_1$ and MST($G_2$) = $G_2$. The minimum spanning trees of $G_1$ and $G_2$ are shown below on the left.

The minimum weighted edge of the two edges across the cut is edge $(A, C)$. So $(A, C)$ is used to connect $G_1$ and $G_2$. This is the minimum spanning tree returned by Professor Borden's algorithm. It is shown below and to the right.


We can see that the minimum-spanning tree returned by Professor Borden's algorithm is not the minimum spanning tree of $G$, therefore, this algorithm fails.

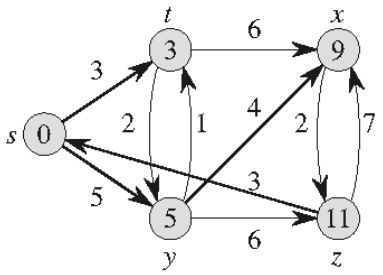
# Homework Solutions – Unit 4: Chapter 24

CMPSC 465

Disclaimer: This is a draft of solutions that has been prepared by the TAs and the instructor makes no guarantees that solutions presented here contain the level of detail that would be expected on an exam. Any errors or explanations you find unclear should be reported to either of the TAs for corrections first.

## Exercise 24.3-1

Run Dijkstra's algorithm on the directed graph below, first using vertex $s$ as the source and then using vertex $z$ as the source. In the style of Figure 24.6, show the $d$ and $\pi$ values and the vertices in set $S$ after each iteration of the while loop.




**Solution**:

Omitted

## Exercise 24.3-3

Suppose we change line 4 of Dijkstra's algorithm to the following:

line 4: **while** $|Q| > 1$

This change causes the while loop to execute $|V| - 1$ times instead of $|V|$ times. Is this proposed algorithm correct?

**Solution**:

Yes, the algorithm still works. Let $u$ be the leftover vertex that does not get extracted from the priority queue $Q$. If $u$ is not reachable from $s$, then $d[u] = \delta(s, u) = \infty$. If $u$ is reachable from $s$, there is a shortest path $p = s \sim x \rightarrow u$. When the node $x$ was extracted, $d[x] = \delta(s, x)$ and then the edge $(x, u)$ was relaxed; thus, $d[u] = \delta(s, u)$.

**Design and Analysis of Algorithms, Fall 2014**
**Exercise I: Solutions**

---

**I-1** Let $T(n) = M$ for $n \leq M$ for some constant $M \geq 1$ independent of $n$, and $2T(n) = 2T(n/2) + 3T(n/3) + n$ otherwise. Show that $T(n) = \Theta(n \log n)$.

As a clarification, we assume that the recurrence is defined for all positive reals. To show $T(n) = \Theta(n \log n)$, we show separately that $T(n) = O(n \log n)$ and $T(n) = \Omega(n \log n)$. All logarithms used are 2-based.

For the "$O$" direction, we show by induction that $T(n) \leq n \log n + cn = \Theta(n \log n)$ for $n \geq 1/3$ for some $c > 0$.

1. Case $1/3 \leq n \leq M$: It's easy to see that $n \log n$ is bounded from below by some constant $c_0$. Now, if we choose any $c \geq 3(M - c_0)$, we have that $T(n) = M \leq c_0 + (1/3)c \leq n \log n + cn$.
2. Induction step $n > M$: assume that the claim holds for $n/2$ and $n/3$. Note that, since $M \geq 1$, the smallest $n$ for which the claim is assumed to hold is $1/3$, which is proven in the base case. Under these assumptions, we have that

$$\begin{array}{rcll} 2T(n) & = & 2T(n/2) + 3T(n/3) + n & \\ & \leq & 2(\frac{n}{2}\log\frac{n}{2} + \frac{n}{2}c) + 3(\frac{n}{3}\log\frac{n}{3} + \frac{n}{3}c) + n & \text{induction assumption} \\ & = & n(\log\frac{n}{2} + \log\frac{n}{3} + 1) + 2nc & \\ & = & n(2\log n - \log 6 + 1) + 2nc & \text{properties of logarithm} \\ & \leq & 2(n \log n + nc) & \log_2 6 > 1 \end{array}$$

   holds regardless of our choice of $c$.

For the "$\Omega$" direction, we show by induction that $T(n) \geq cn \log n = \Theta(n \log n)$ for some $c > 0$.

1. Case $n \leq M$: by choosing $c \leq 1/\log M$, we have that $T(n) = M \geq cM \log M \geq cn \log n$.
2. Induction step $n > M$: assume that the claim holds for $n/2$ and $n/3$. Then,

$$\begin{array}{rcll} 2T(n) & = & 2T(n/2) + 3T(n/3) + n & \\ & \geq & 2(c\frac{n}{2}\log\frac{n}{2}) + 3(c\frac{n}{3}\log\frac{n}{3}) + n & \text{induction assumption} \\ & = & 2cn \log n + n(1 - c\log 6) & \text{properties of logarithm} \\ & \geq & 2cn \log n & \end{array}$$

   where the last inequality holds if $1 - c\log 6 \geq 0 \iff c \leq 1/\log 6$.

Since we want both of steps 1 and 2 to hold, we can choose $c = \min(1/\log M, 1/\log 6)$.

---

**I-2** (**CLRS 2.3-7**) Describe a $\Theta(n \log n)$-time algorithm that, given a set $S$ of $n$ integers and another integer $x$, determines whether or not there exist two elements in $S$ whose sum is exactly $x$.

Sort the set $S$ using merge sort. Then for each $y \in S$ separately use binary search to check if integer $x - y$ exists in $S$. Sorting takes time $\Theta(n \log n)$. Binary search takes time $O(\log n)$ and is executed $n$ times. The total time is thus $\Theta(n \log n)$.

If $S$ is sorted, the problem can also be solved in linear time by scanning the list $S$ at the same time forward and backward directions:

**Input**: list $S$ sorted in ascending order, $x$
**Output**: true if there exist two elements in $S$ whose sum is exactly $x$, false otherwise

```
1 i ← 1, j ← n
2 while i ≤ j do
3 |   if S[i] + S[j] = x then
4 |   |   return true
5 |   if S[i] + S[j] < x then
6 |   |   i ← i + 1
7 |   else
8 |   |   j ← j − 1
9 return false
```

Note: The above solutions assume that the two elements can actually be the same element (for examples, if $S = \{1,2,5\}$ and $x = 4$, the algorihms return true since $2+2=4$). If this is not allowed, then small changes to algorithms are needed. (In the first solution skip $y$ if $2y = x$. In the second solution replace $\leq$ by $<$.)

---

**I-3** (**CLRS 2-4 Inversions**) Let $A[1..n]$ be an array of $n$ distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair $(i, j)$ is called an *inversion* of $A$.

**a.** List the five inversions of the array $(2,3,8,6,1)$.

(1,5), (2,5), (3,4) (3,5), (4,5)

**b.** What array with elements from the set $\{1,2,\ldots,n\}$ has the most inversions? How many does it have?

If the array is sorted in descending order, then every pair $(i, j)$ with $i < j$ is an inversion. The number of such pairs is $n(n-1)/2$.

**c.** What is the relationship between the running time of insertion sort and the number of inversions in the input array? Justify your answer.

Assume a version of insertion sort that does not use a temporary variable but instead uses swaps to move the current element into the correct place in the already sorted part of the array. Now each execution of the inner loop body (that is, each swap) eliminates one inversion. Sorting eliminates all inversion and therefore the inner loop is executed exactly $I$ times, where $I$ is the number of inversions in the array. Since the outer loop is executed $n$ times the total running time is $\Theta(I+n)$.

**d.** Give an algorithm that determines the number of inversions in any permutation of $n$ elements in $\Theta(n \log n)$ worst-case time. (*Hint:* Modify merge sort.)

We modify merge sort to count the number of inversion while sorting the array. The number of inversion in array $A = BC$ is

$$I(A) = I(B) + I(C) + I(B,C),$$

where $I(X)$ is the number of inversion in array $X$ and $I(X,Y)$ is the number of pairs $(x,y) \in X \times Y$ such that $x > y$. Let $head(X)$ be the first element in array $X$. We can compute term $I(B,C)$ in the following way while merging arrays $B$ and $C$:

- If $head(B) \leq head(C)$, we append $head(B)$ to the merged array and remove it from array $B$ as usual.
- If $head(B) > head(C)$, we append $head(C)$ to the merged array, remove it from array $C$, and increment $I(B,C)$ by the number of elements remaining in array $B$.

Terms $I(B)$ and $I(C)$ are computed recursively. Counting the number of inversions does not increase the asymptotic time complexity of the merge sort.

---

**I-4** (**CLRS 4.2-6**) How quickly can you multiply a $kn \times n$ matrix by an $n \times kn$ matrix, using Strassen's algorithms as a subroutine? Answer the same question with the order of input matrices reversed.

Let the input be $A = [A_1^T \ \cdots \ A_k^T]^T$ and $B = [B_1 \ \cdots \ B_k]$, where $A_i$ and $B_i$ are $n \times n$ submatrices. The product $AB$ is a $kn \times kn$ matrix

$$AB = \begin{bmatrix} A_1B_1 & \cdots & A_1B_k \\ \vdots & \ddots & \vdots \\ A_kB_1 & \cdots & A_kB_k \end{bmatrix},$$

where each product $A_iB_j$ can be computed in $\Theta(n^{\log_2 7})$ time using Strassen's algorithm. There are $k^2$ such products, so the total time requirement is $\Theta(k^2 n^{\log_2 7})$.

The product $BA$ is a $n \times n$ matrix $BA = \sum_{i=1}^{k} A_iB_i$. There are $k$ products requiring $\Theta(n^{\log_2 7})$ time and $k-1$ summations requiring $\Theta(n^2)$ time. The total time is thus $\Theta(kn^{\log_2 7})$.

**I-5** (**CLRS 4.2-7**) Show how to multiply the complex numbers $a+bi$ and $c+di$ using only three multiplications of real numbers. The algorithm should take $a$, $b$, $c$, and $d$ as input and produce the real component $ac-bd$ and the imaginary component $ad+bc$ separately.

Compute the products $ac$, $bd$ and $(a+b)(c+d)$. Now the real component is $ac-bd$ and imaginary component is $(a+b)(c+d)-ac-bd$.

Alternatively, compute e.g. $a(c-d)$, $d(a-b)$ and $b(c+d)$. Now the real component is obtained as the sum of the first two, and the imaginary component as the sum of the last two.

Both solutions use 3 multiplications and 5 additions/substractions.

**Design and Analysis of Algorithms, Fall 2014**
**Exercise II: Solutions**

---

**II-1** Where in the matrix multiplication-based DP algorithm for the all-pairs shortest paths problem do we need the associativity of matrix multiplication?

The algoritm computes the product $W^{n-1}$ with the exception that instead of the usual definition, the product of matrices $A$ and $B$ is defined by

$$A \cdot B = C, \quad \text{where} \quad C_{ij} = \min_k \{A_{ik} + B_{kj}\}.$$

The slow version of the algorithm uses a recurrence $W^{i+1} = W^i \cdot W$, which gives the correct result. The fast version uses repeated squaring: $W^{2i} = W^i \cdot W^i$. We do not know a priori that the second recurrence computes $W^{2i}$ correctly. However, from the associativity of multiplication it follows that $W^i \cdot W^i = W^{2i-1} \cdot W$, that is, the fast version works correctly.

---

**II-2** Give an $O(n^2)$-time algorithm to find the maximum length of monotonically increasing subsequences of a given sequence of $n$ numbers. For instance, if given $(3,1,2,5,2,6,8,6,7,3,5)$ as input, the algorithm should output 6 (the length of the subsequence $(1,2,5,6,6,7)$). (Side note: Even an $O(n \log n)$-time algorithm exists.)

We write "MIS" short for monotonically increasing subsequence. Let $A[i]$ denote the $i$th number in the sequence. For all $1 \leq i \leq n$ we define $L[i]$ as the length of the longest MIS that ends in $A[i]$. Then $L[1] = 1$ and for all $i > 1$ we have that

$$L[i] = \max\{L[k] : 1 \leq k \leq i-1,\ A[k] \leq A[i]\} + 1.$$

In other words, $L[i]$ is found by finding the longest MIS among the preceding numbers that can be continued by $A[i]$. Assuming the values $L[k]$ for $1 \leq k \leq i-1$ are already computed, $L[i]$ is easily computed in linear time. This yields a simple dynamic programming algorithm that computes all $L[i]$ in increasing order of $i$ in $O(n^2)$ time. The solution is then obtained as $\max\{L[i] : 1 \leq i \leq n\}$. To find the actual longest MIS (instead of just the length), the algorithm should also keep track of the maximizing indices $k$, which can then be used to reconstruct the MIS.

The problem can also be solved in $O(n \log n)$ time by formulating the dynamic programming in a different manner. The algorithm initializes an array $E$ and performs $n$ iterations, maintaining the following invariant: After the $i$th iteration, each $E[\ell]$ for $1 \leq \ell \leq i$ contains the smallest number that ends a MIS of length exactly $\ell$ among the first $i$ elements of $A$, or $\infty$ if no such subsequence exists.

Assuming the invariant holds, it's easy to see that $E$ is always increasing: If there's a MIS of length $\ell$ ending at $A[i]$, there's also a MIS of length $\ell+1$ ending at $A[j]$ for $A[i] < A[j]$. Therefore $A[i]$ couldn't be the smallest number that ends a MIS of length $\ell$.

To maintain the invariant, for iteration $i$ the algorithm uses binary search to find the largest $E[\ell] \leq A[i]$. Since $E[\ell]$ ends a MIS of length $\ell$, it is extended by $A[i]$ to produce a MIS of length $\ell+1$, and $A[i]$ is clearly the smallest number ending such a MIS so far. Hence, the algorithm updates $E[\ell+1] = A[i]$. If no $E[\ell] \leq A[i]$ exists, the algorithm updates $E[1] = A[i]$. It turns out no other changes are required.

Therefore, for each of the $n$ iterations the algorithm performs a $O(\log n)$ amount of work, giving the claimed time complexity. After all iterations, the answer is the largest index $k$ for which $L[k] < \infty$. Again, slight modifications are required to find the actual MIS.

**II-3** (**CLRS 15-3 Bitonic euclidean traveling-salesman problem**) The euclidean traveling-salesman problem is the problem of determining the shortest closed tour that connects a given set of $n$ points in the plane. Figure 15.11(a) shows the solution to a 7-point problem. The general problem is NP-complete, and its solution is therefore believed to require more than polynomial time (see Chapter 34).
J. L. Bentley has suggested that we simplify the problem by restricting our attention to bitonic tours, that is, tours that start at the leftmost point, go strictly left to right to the rightmost point, and then go strictly right to left back to the starting point. Figure 15.11(b) shows the shortest bitonic tour of the same 7 points. In this case, a polynomial-time algorithm is possible.
Describe an $O(n^2)$-time algorithm for determining an optimal bitonic tour. You may assume that no two points have the same $x$-coordinate. (*Hint:* Scan left to right, maintaining optimal possibilities for the two parts of the tour.)

The first step is to sort the points according to $x$-coordinate. Let the sorted points be $1,\ldots,n$ and let $d(i,j)$ be the distance between points $i$ and $j$.

A bitonic tour starts at the leftmost point and ends at the rightmost point. It consists of two paths, the upper and lower (imaging a line connecting the starting and end points), such that each point is visited by at least one of the paths. We describe a dynamic programming algorithm which uses partially constructed bitonic tours.

For $i,j \in \{1,\ldots,n\}$ and $i$, $j$ on separate paths, let $B[i,j]$ be the minimum total cost of two paths. Now the length of the optimal bitonic tour is given by $B[n,n]$. Since $B[i,j] = B[j,i]$, we are only interested in pairs $i,j$ with $1 \le i \le j \le n$. The base case is $B[1,1] = 0$. For the rest of the cases we compute $B[i,j]$ as follows (the idea is deciding the predecessor of point $j$):

- **Case 1:** If $i < j-1$, then the path ending in $j$ must also visit $j-1$, because the other path cannot visit $j-1$ and then backtrack to $i$. Thus we have

$$B[i,j] = B[i,j-1] + d(j-1,j), \text{ if } i < j-1.$$

- **Case 2:** If $i = j-1$ or $i = j$, then the optimal solution must have a path which ends in $j$ and comes from some node $k$ with $1 \le k < j$. The optimal solution is therefore given by selecting the optimal predecessor by setting

$$B[i,j] = \min_{1 \le k < j} \{B[i,k] + d(k,j)\}.$$

  where $B[i,k]$ can be replaced by $B[k,i]$ (already computed), since $B[i,j] = B[j,i]$.

There are $O(n^2)$ entries that fall under the first case, and each of these takes a constant time to handle. The second case occurs only $O(n)$ times, with each taking $O(n)$ time. Therefore the total running time is $O(n^2)$.

To get the actual optimal bitonic tour, we modify the algorithm to keep track of the optimal predecessor chosen in the case 2 in a separate table. The optimal tour can then be constructed from this information.

---

**II-4** (**CLRS 15-4 Printing neatly**) Consider the problem of neatly printing a paragraph on a printer. The input text is a sequence of $n$ words of lengths $l_1, l_2, \ldots, l_n$, measured in characters. We want to print this paragraph neatly on a number of lines that hold a maximum of $M$ characters each. Our criterion of "neatness" is as follows. If a given line contains words $i$ through $j$, where $i < j$, and we leave exactly one space between words, the number of extra space characters at the end of the line is $M - j + i - \sum_{k=i}^{j} l_k$, which must be non-negative so that the words fit on the line. We wish to minimize the sum, over all lines except the last, of the cubes of the numbers of extra space characters at the ends of lines. Give a dynamic-programming algorithm to print a paragraph of $n$ words neatly on a printer. Analyze the running time and space requirements of your algorithm.

Let

$$s(i,j) = M - j + i + 1 - \sum_{k=i+1}^{j} l_k$$

be the number of trailing spaces required when the words $i+1,\ldots,j$ are put on a same line. Observe that is possible to put these words on a line only if $s(i,j) \geq 0$. We define the cost of putting the words from $i+1$ to $j$ on a same line as

$$c(i,j) = \begin{cases} \infty, & \text{if } s(i,j) < 0 \\ 0, & \text{if } j = n \text{ and } s(i,j) \geq 0 \\ s(i,j)^3, & \text{otherwise.} \end{cases}$$

The dynamic programming is now straightforward to formulate. Let $C[j]$ be the optimal cost of printing the words from 1 to $j$, such that word $j$ ends a line. Then we have

$$\begin{aligned} C[0] &= 0 \\ C[j] &= \min_{0 \leq i < j} \{C[i] + c(i,j)\} \quad \text{for } j > 0. \end{aligned}$$

Computing $C$ takes $\Theta(n^2)$ time, as $c(i-1,j)$ can be computed from $c(i,j)$ in constant time. The space requirement is $\Theta(n)$. A straightforward modification gives us $O(Mn)$ time, as there cannot be more than $(M+1)/2$ words on a single line.

Again, to actually get the optimal line lengths and print the paragraph, we need to keep track of minimizing values $i$, which indicate the last word on the previous line, given that the last word on the current line is $j$.

---

**II-5** (**CLRS 25.1-9**) Modify Faster-All-Pairs-Shortest-Paths so that it can determine whether the graph contains a negative-weight cycle.

We observe that if the graph contains a negative-weight cycle, then there is a negative-weight path of length $k \leq n$ from some vertex $i$ to $i$. This means that in the matrix-multiplication-based all-pairs shortest path algorithm we will get a negative value on the diagonal of the matrix $W^m$ for all $m \geq k$. Therefore we can detect the existence of negative cycles simply by checking whether there are negative values on the diagonal of the matrix $W^{2^{\lceil \log_2 n \rceil}}$. Note that we have to modify the algorithm to compute matrices up to at least $W^n$ instead of $W^{n-1}$, as the shortest negative weight cycle might have length $n$.

**Design and Analysis of Algorithms, Fall 2014**
**Exercise III: Solutions**

---

**III-1** Show that the subset-sum problem is solvable in polynomial time if the target value $t$ is expressed in unary.

Let $S_i, \ldots, S_n$ be the sequence of positive integers. We want to find out if there is a (multi-)subset of $S$ that sums up to $t > 0$. For all $0 \leq i \leq n$ and $0 \leq p \leq t$, we define $L[i,p]$ to be true if there is a subset of $S_1, \ldots, S_i$ that sums up $p$, and false otherwise. Then $L[n,t]$ is the solution to the problem. Observe that $L[i,0]$ is true for all, $i$ and $L[0,p]$ is false for all $p > 0$. For $i \geq 1$ and $p \geq 1$, there is a subset of $S_1, \ldots, S_i$ summing up to $p$ if and only if there is a subset of $S_1, \ldots, S_{i-1}$ summing to either $p$ or $p - S_i$. Thus, we get

$$L[i,p] = \begin{cases} \text{true} & \text{if } p = 0 \\ \text{false} & \text{if } p > 0 \text{ and } i = 0 \\ L[i-1,p] \vee L[i-1,p-S_i] & \text{otherwise.} \end{cases}$$

The term $L[n,t]$ is easily evaluated in $O(nt)$ time using dynamic programming. Let $b$ be the size of the input in bits. If $t$ is expressed in binary, as is common, then $b = O(\log_2 t)$ and the running time is $O(2^b n)$. However, if we express $t$ in unary, then $b = O(t)$ and the running time is $O(nb)$, which is polynomial in the input $b$.

---

**III-2** (**CLRS 34.1-5**) Show that if an algorithm makes at most a constant number of calls to polynomial-time subroutines and performs an additional amount of work that also takes polynomial time, then it runs in polynomial time. Also show that a polynomial number of calls to polynomial-time subroutines may result in an exponential-time algorithm.

Let $k$ be the number of subroutines. The algorithm starts out with an input data of size $n$. Each subroutine takes (some of) the available data as an input, performs some steps, then returns some amount of data as an output. Every time a subroutine returns, the output accumulates the amount of data the algorithm has access to. Any or all of this data may then be given as an input to the next subroutine. Since each subroutine runs in polynomial time, the output must also have a size polynomial in the size of the input. Let $d$ be an upper bound on the degree of the polynomials. Then there is a function $p(n) = n^d + c$, where $c$ is a constant, such that $p(n)$ is an upper bound for the size of the output of any subroutine when given an input of size $n$.

Let $n_0 = n$ and $n_i = n_{i-1} + p(n_{i-1})$ for all $1 \leq i \leq k$. We show by induction that $n_i$ is an upper bound for the amount of data available to the algorithm after the $i$th subroutine call. The base case is trivial. Assume the claim holds for $i-1$ and let $n'$ be the exact amount of data available before the $i$th call. Then we have $n_i \leq n' + p(n')$, since the $i$th call accumulates the amount of the data by at most $p(n')$. Since $p$ is increasing, by assumption we have $n' \leq n_{i-1}$ and $p(n') \leq p(n_{i-1})$, from which the claim follows.

We use induction again to show that each $n_i$ is polynomial in $n$. The base case is again trivial. Assume $n_{i-1}$ is polynomial in $n$. Since the composition of two polynomials is also polynomial, we have that $p(n_{i-1})$ is polynomial in $n$. Since also the sum of two polynomials is polynomial, we have that $n_{i-1} + p(n_{i-1}) = n_i$ is polynomial in $n$. Therefore $n_k$, which is an upper bound for the amount of data after the final subroutine, is also polynomial in $n$, and the time must also be polynomial.

For the second part, observe that if we have a subroutine whose output is always twice the size of its input, and we call this subroutine $n$ times, starting with input of size 1 and always feeding the previous output back into the subroutine, the final output will have size $2^n$. This means that the algorithm will take exponential time.

---

**III-3** (**CLRS 34.5-8**) In the half 3-CNF satisfiability problem, we are given a 3-CNF formula $\phi$ with $n$ variables and $m$ clauses, where $m$ is even. We wish to determine whether there exists a truth assignment to the variables of $\phi$ such that exactly half the clauses evaluate to 0 and exactly half the clauses evaluate to 1. Prove that the half 3-CNF satisfiability problem is NP-complete. (You may assume that the 3-CNF formula has *at most* 3 literals per clause, not necessarily exactly 3.)

First observe that given a 3-CNF formula and an assignment, it is easy to check in polynomial time if the assignment satisfies exactly half of the clauses. Therefore the half 3-CNF satisfiability is in NP.

We show NP-completeness by reduction from 3-CNF satisfiability. Let $\varphi$ be a 3-CNF-SAT formula with $n$ variables and $m$ clauses. We construct a 3-CNF-SAT formula $\psi$ such that exactly half of the clauses in $\psi$ can be satisfied if and only if $\varphi$ can be satisfied. Suppose that $y_i$ and $z_i$ for $i = 1, \ldots, m+1$ as well as $p$ are variables that do not appear in $\varphi$. We add to $\psi$ by all the clauses of $\varphi$, $m(m+1)+1$ distinct clauses of form $\{q, \neg q, p\}$, where $q$ can be any variable (we call these *type 1 clauses*) and clause $\{y_i, z_j, p\}$ for all $i, j \in \{1, \ldots, m+1\}$ (we call these *type 2 clauses*). Constructing $\psi$ clearly takes polynomial time.

We observe that all type 1 clauses are always satisfied. Since there are total of $2(m+1)^2$ clauses, we have to satisfy precisely $m$ other clauses to satisfy half of the clauses of $\psi$. If we tried to satisfy any type 2 clause $\{y_i, z_j, p\}$, we would also satisfy all type 2 clauses with variable $y_i$ or all type 2 clauses with variable $z_j$. This means that we would satisfy at least $m+1$ additional clauses, that is, total of at least $(m+1)^2+1$ clauses. Thus the only way to satisfy exactly $(m+1)^2$ clauses in $\psi$ is to satisfy all the clauses of $\varphi$. This implies that exactly half of the clauses in $\psi$ can be satisfied if and only if $\varphi$ can be satisfied.

Thus, given a polynomial-time algorithm for the half 3-CNF-SAT problem, we could solve 3-CNF-SAT in polynomial time. Since we know 3-CNF-SAT to be NP-complete, it follows that the half 3-CNF-SAT is NP-complete as well.

---

**III-4** (**CLRS 34.4-6**) Suppose someone gives you a polynomial-time algorithm to decide formula satisfiability. Describe how to use this algorithm to find satisfying assignments in polynomial time.

Let $\varphi$ be the input SAT formula that is satisfiable and contains $n$ variables. Let $\varphi|_{x_i=0}$ and $\varphi|_{x_i=1}$ be the simplified SAT formulas obtained by replacing variable $x_i$ by values 0 and 1 respectively and eliminating constants 0 and 1 by partially evaluating the formula. These can be computed in polynomial time. The results are SAT formulas containing $n-1$ variables. For example, if $\varphi = ((x_1 \to x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$ then

$$\varphi|_{x_2=0} = ((x_1 \to 0) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg 0 = (\neg x_1 \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \text{ and}$$
$$\varphi|_{x_2=1} = ((x_1 \to 1) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg 1 = 0.$$

Clearly $\varphi$ is satisfiable by assignment containing $x_i = c$ if and only if $\varphi|_{x_i=c}$ is satisfiable. The algorithm now takes any variable $x_i$ and asks the oracle whether $\varphi|_{x_i=0}$ is satisfiable. If the answer is yes, then the algorithm sets $x_i = 0$ and recursively repeats the procedure with $\varphi|_{x_i=0}$. Otherwise $\varphi|_{x_i=1}$ must be satisfiable, so the algorithm sets $x_i = 1$ and recursively repeats the procedure with $\varphi|_{x_i=1}$. Once all variables have been assigned a value, this assignment satisfies the original SAT formula. The algorithm takes $n$ polynomial time steps and thus works in polynomial time.

Instead of reducing the formula, one may alternatively augment it with conjunctions, producing formulas of form $\varphi \wedge x_i$ and $\varphi \wedge \neg x_i$. Again, one consults the oracle and recurses on a satisfiable formula until for each variable either the variable or its negation has been added, yielding a satisfying assignment.

---

**III-5** (**CLRS 34.5-6**) Show that the hamiltonian-path problem is NP-complete. (You may assume that you know that HAM-CYCLE is NP-complete.)

Again, observe that given a sequence of vertices it is easy to check in polynomial time if the sequence is a hamiltonian path, and thus the problem is in NP.

We reduce from the hamiltonian cycle problem. Let $G = (V, E)$ be a graph. The reduction transforms graph $G$ into $G'$ as follows. We pick an arbitrary vertex $v \in V$ and add a new vertex $v'$ that is connected to all the neighbors of $v$. We also add new vertices $u$ and $u'$ so that $u$ is adjacent to $v$ and $u'$ is adjacent to $v'$. This reduction clearly takes a polynomial time.

To complete the proof, we have to show that $G$ has a hamiltonian cycle if and only if $G'$ has a hamiltonian path. Now if there is a hamiltonian cycle $(v, v_2, \ldots, v_n, v)$ in $G$, then $(u, v, v_2, \ldots, v_n, v', u')$ is a hamiltonian path in $G'$. On the other hand, if there is a hamiltonian path in $G'$, its endpoints have to be $u$ and $u'$, because these have only on neighbor and thus cannot be in a middle of the path. Thus, the path has form $(u, v, v_2, \ldots, v_n, v', u')$ and we have that $(v, v_2, \ldots, v_n, v)$ is a hamiltonian cycle in $G$.

**Design and Analysis of Algorithms, Fall 2014**
**Exercise IV: Solutions**

---

**IV-1** (**CLRS 17.1-1**) If the set of stack operations included a MULTIPUSH operation, which pushes $k$ items onto the stack, would the $O(1)$ bound on the amortized cost of stack operations continue to hold?

No. The time complexity of such a series of operations depends on the number of pushes (pops vise versa) could be made. Since one MULTIPUSH needs $\Theta(k)$ time, performing $n$ MULTIPUSH operations, each with $k$ elements, would take $\Theta(kn)$ time, leading to amortized cost of $\Theta(k)$.

---

**IV-2** (**CLRS 17.1-3**) Suppose we perform a sequence of $n$ operations on a data structure in which the $i$th operation costs $i$ if $i$ is an exact power of 2, and 1 otherwise. Use aggregate analysis to determine the amortized cost per operation.

In a sequence of $n$ operations there are $\lfloor \log_2 n \rfloor + 1$ exact powers of 2, namely $1, 2, 4, \ldots, 2^{\lfloor \log_2 n \rfloor}$. The total cost of these is a geometric sum $\sum_{i=0}^{\lfloor \log_2 n \rfloor} 2^i = 2^{\lfloor \log_2 n \rfloor + 1} - 1 \leq 2^{\log_2 n + 1} = 2n$. The rest of the operations are cheap, each having a cost of 1, and there are less than $n$ such operations. The total cost of all operations is thus $T(n) \leq 2n + n = 3n = O(n)$, which means $O(1)$ amortized cost per operation.

---

**IV-3** (**CLRS 17.2-2** and **CLRS 17.3-2**) Redo the previous exercise using (a) an accounting method of analysis and (b) a potential method of analysis.

(a) The actual cost of the $i$th operation is

$$c_i = \begin{cases} i & \text{if } i \text{ is an exact power of 2} \\ 1 & \text{otherwise.} \end{cases}$$

We assign the amortized costs as follows:

$$\hat{c}_i = \begin{cases} 2 & \text{if } i \text{ is an exact power of 2} \\ 3 & \text{otherwise.} \end{cases}$$

Now an operation, which is an exact power of 2 uses all previously accumulated credit plus one unit of its own amortized cost to pay its true cost. It then assigns the remaining unit as credit. On the other hand, if $i$ is not an exact power of 2, then the operation uses one unit to pay its actual cost and assigns the remaining two units as credit. This covers the actual cost of the operation. We still have to show that there will always be enough credit to pay the cost of any power-of-2 operation. Clearly this is the case for the first operation ($i = 1$), which happens to be an exact power of two. Let now $j > 0$. After $2^{j-1}$:th operation there is one unit of credit. Between operations $2^{j-1}$ and $2^j$ there are $2^{j-1} - 1$ operations none of which is an exact power of 2. Each assigns two units as credit resulting to total of $1 + 2 \cdot (2^{j-1} - 1) = 2^j - 1$ accumulated credit before $2^j$th operation. This, together with one unit by its own, is just enough to cover its true cost. Therefore the amount of credit stays nonnegative all the time, and the total amorized cost is an upper bound for the total actual cost.

(b) We define the potential function as follows:

$$\Phi(D_0) = 0 \quad \text{and} \quad \Phi(D_i) = 2i - 2^{\lfloor \log_2 i \rfloor + 1} + 1 \text{ for } i > 0.$$

(Note that $\Phi(D_i)$ actually equals to the amount of credit after $i$th operation in previous exercise.) This potential is always nonnegative, so we have $\Phi(D_i) \geq 0 = \Phi(D_0)$ for all $i$. Now

- For $i$, which is an exact power of 2, the potential difference is

$$\Phi(D_i) - \Phi(D_{i-1}) = (2i - 2i + 1) - (2(i-1) - i + 1) = 2 - i$$

Note that this also holds for $i = 1$. Thus, the amortized cost of $i$th operation is

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = i + 2 - i = 2.$$

- For $i$, which is not an exact power of 2, the potential difference is

$$\Phi(D_i) - \Phi(D_{i-1}) = (2i - i + 1) - (2(i-1) - i + 1) = 2.$$

Thus, the amortized cost of $i$th operation is

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 2 = 3.$$

As seen above, the amortized cost of each operation is $O(1)$.

---

**IV-4** (**CLRS 17.3-4**) What is the total cost of executing $n$ of the stack operations PUSH, POP, and MULTIPOP, assuming that the stack begins with $s_0$ objects and finishes with $s_n$ objects?

Let $\Phi$ be the potential function that returns the number of elements in the stack. We know that for this potential function, we have amortized cost 2 for PUSH operation and amortized cost 0 for POP and MULTIPOP operations.

The total amortized cost is

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0).$$

Using the potential function and the known amortized costs, we can rewrite the equation as

$$\begin{aligned}\sum_{i=1}^{n} c_i &= \sum_{i=1}^{n} \hat{c}_i + \Phi(D_0) - \Phi(D_n) \\ &= \sum_{i=1}^{n} \hat{c}_i + s_0 - s_n \\ &\leq 2n + s_0 - s_n,\end{aligned}$$

which gives us the total cost of $O(n + (s_0 - s_n))$. If $s_n \geq s_0$, then this equals to $O(n)$, that is, if the stack grows, then the work done is limited by the number of operations.

(Note that it does not matter here that the potential may go below the starting potential. The condition $\Phi(D_n) \geq \Phi(D_0)$ for all $n$ is only required to have $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$, but we do not need for that to hold in this application.)

---

**IV-5** (**CLRS 17.4-3**) Suppose that instead of contracting a table by halving its size when its load factor drops below $1/4$, we contract it by multiplying its size by $2/3$ when its load factor drops below $1/3$. Using the potential function $\Phi(T) = |2 \cdot N(T) - S(T)|$, show that the amortized cost of a TABLE-DELETE that uses this strategy is bounded above by a constant. Here $N(T)$ and $S(T)$ denote the number of items stored in table $T$ and the size of $T$, respectively.

Let $c_i$ denote the actual cost of the $i$th operation, $\hat{c}_i$ its amortized cost and $n_i$, $s_i$ and $\Phi_i$ the number of items stored in the table, the size of the table and the potential after the $i$th operation, respectively. The potential $\Phi_i$ cannot get negative values and we have $\Phi_0 = 0$. Therefore $\Phi_i \geq \Phi_0$ for all $i$ and the total amortized cost provides an upper bound on the actual cost.

Now if the $i$th operation is TABLE-DELETE, then $n_i = n_{i-1} - 1$. Consider first the case when the load factor does not drop below $1/3$, i.e. $\frac{n_i}{s_{i-1}} \geq \frac{1}{3}$. Then the table isn't contracted and $s_i = s_{i-1}$. We pay $c_i = 1$ for deleting one item. Thus

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + |2n_i - s_i| - |2n_{i-1} - s_{i-1}| \\ &= 1 + |2n_{i-1} - s_{i-1} - 2| - |2n_{i-1} - s_{i-1}| && n_i = n_{i-1} - 1,\ s_i = s_{i-1} \\ &\leq 1 + |(2n_{i-1} - s_{i-1} - 2) - (2n_{i-1} - s_{i-1})| && \text{(reverse) triangle inequality} \\ &= 1 + 2 = 3\end{aligned}$$

Now consider the case when the load factor drops below $1/3$, i.e.

$$\frac{n_i}{s_{i-1}} < \frac{1}{3} \leq \frac{n_{i-1}}{s_{i-1}} \Rightarrow 2n_i < \frac{2}{3}s_{i-1} \leq 2n_i + 2. \tag{1}$$

Now we contract the table and have

$$s_i = \lfloor 2/3 \cdot s_{i-1} \rfloor \Rightarrow \frac{2}{3}s_{i-1} - 1 \leq s_i \leq \frac{2}{3}s_{i-1}. \tag{2}$$

By combining (1) and (2) we get $2n_i - 1 \leq s_i \leq 2n_i + 2$ and thus $|2n_i - s_i| \leq 2$. Furthermore, from (1) we get $s_{i-1} > 3n_i$ and thus $|2n_{i-1} - s_{i-1}| = -2n_{i-1} + s_{i-1} \geq n_{i-1}$. We pay $c_i = n_i + 1$ for deleting one item and moving the remaining $n_i$ items into the contracted table. Then,

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= (n_i + 1) + |2n_i - s_i| - |2n_{i-1} - s_{i-1}| \\
&\leq (n_i + 1) + 2 - n_i = 3
\end{aligned}$$

In both cases the amortized cost is at most 3 and thus bounded above by a constant.

# Math 416 Homework 2 Solutions

Updated 9 October, 2005

(1) (CRLS 3-4) **Suppose that $f$ and $g$ are asymptotically positive. Prove or disprove each of the following statements.**

(a) **If $f(n) = O(g(n))$, then $g(n) = O(f(n))$.** This is false. If $f(n) = 1$ and $g(n) = n$ for all natural numbers $n$, then $f(n) \leq g(n)$ for all natural numbers $n$, so $f(n) = O(g(n))$. However, suppose $g(n) = O(f(n))$. Then there are a natural number $n_0$ and a constant $c > 0$ such that $n = g(n) \leq cf(n) = c$ for all $n \geq n_0$, which is impossible.

(b) **$f(n) + g(n) = O(\min\{f(n), g(n)\})$.** This is false. Let $f$ and $g$ be as in (1a). Suppose that $f(n) + g(n) = O(\min\{f(n), g(n)\})$. Then there are a natural number $n_0$ and a constant $c > 0$ such that $n + 1 = f(n) + g(n) \leq c \min\{f(n), g(n)\} = c$ for all $n \geq n_0$, which is impossible.

(c) **If $f(n) = O(g(n))$ *and* if $f(n), \log g(n) \geq 1$ for sufficiently large $n$, then $\log(f(n)) = O(\log(g(n)))$.** Notice that this is *not* true if we remove the condition that $\log g(n) \geq 1$ for sufficiently large $n$. For example, suppose the logarithm is taken base 2 and we put $f(n) = 2$, $g(n) = 1$. Then obviously $f(n) = O(g(n))$, but $\log f(n) = 1 \neq O(0) = O(\log g(n))$. Thus any argument that did not use the condition that $\log g(n) \geq 1$ was incorrect. (The reason for the condition $f(n) \geq 1$ is that we want to be sure that $\log f(n) \geq 0$.)
With the additional conditions imposed, the statement is true. Since $f(n) = O(g(n))$, there are a natural number $n_0$ and a constant $c > 0$ such that $f(n) \leq cg(n)$ for $n \geq n_0$. Since log is increasing, $\log f(n) \leq \log(cg(n)) = \log(c) + \log(g(n))$ for $n \geq n_0$. Put $c' = \log(c) + 1$. Then

$$\log f(n) \leq \log(c) \log(g(n)) + \log(g(n)) = c' \log(g(n))$$

for $n \geq n_0$ (since $\log g(n) \geq 1$), so $\log f(n) = O(\log g(n))$.

(d) **If $f(n) = O(g(n))$, then $2^{f(n)} = O(2^{g(n)})$.** This is false. If $f(n) = 2n$ and $g(n) = n$ for all natural numbers $n$, then $f(n) \leq 2g(n)$ for all natural numbers $n$, so $f(n) = O(g(n))$. However, suppose $2^{f(n)} = O(2^{g(n)})$. Then there are a natural number $n_0$ and a constant $c > 0$ such that $4^n = 2^{2n} = 2^{f(n)} \leq c2^{g(n)} = c2^n$, i.e., $2^n = (4/2)^n \leq c$, for all $n \geq n_0$, which is impossible.

(e) **$f(n) = O((f(n))^2)$.** This is false. Let $f(n) = \frac{1}{n}$ for all natural numbers $n$. Suppose that $f(n) = O((f(n))^2)$. Then there are a natural number $n_0$ and a constant $c > 0$ such that $\frac{1}{n} = f(n) \leq c(f(n))^2 = c\frac{1}{n^2}$, i.e., $n = \frac{n^2}{n} \leq c$, for all $n \geq n_0$, which is impossible.

(f) **If $f(n) = O(g(n))$, then $g(n) = \Omega(f(n))$.** This is true. There are a natural number $n_0$ and a constant $c > 0$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$. Put $c' = \frac{1}{c} > 0$. Then $g(n) \geq c'f(n)$ for all $n \geq n_0$, so $g(n) = \Omega(f(n))$.

(g) **$f(n) = \Theta(f(n/2))$.** This is false. Let $f$ be as in (1d). Suppose $f(n) = \Theta(f(n/2))$. Then there are a natural number $n_0$ and a constant $c_2 > 0$ such that $2^n = f(n) \leq c_2 f(n/2) = c_2 2^{n/2} = c_2\sqrt{2}^n$, i.e., $\sqrt{2}^n = (2/\sqrt{2})^n \leq c_2$, for all $n \geq n_0$, which is impossible. (Of course there would also be a constant $c_1 > 0$ such that $f(n) \geq c_1 f(n/2)$, but we don't need that to get a contradiction.)

(h) $f(n)+o(f(n)) = \Theta(f(n))$. This is true. In fact, the stronger statement $f(n)+O(f(n)) = \Theta(f(n))$ is true. (This statement is stronger because there are more functions allowed on the left-hand side; that is, we are asserting that more functions belong to $\Theta(f(n))$.) Suppose that $g(n) = O(f(n))$. Then there are a natural number $n_0$ and a constant $c > 0$ such that $g(n) \leq cf(n)$ for all $n \geq n_0$. By making $n_0$ larger, if necessary, we may also assume that $g(n) \geq 0$ for all $n \geq n_0$. Put $c' = c + 1$. Then $f(n) \leq f(n) + g(n) \leq f(n) + cf(n) = c'f(n)$ for all $n \geq n_0$, so $f(n) + g(n) = \Theta(f(n))$.

(2) (CLRS 4.1-5) **Show that, if** $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$, **then** $T(n) = O(n \log n)$. In this problem, we *cannot*, in the proof, ignore the 17 in the recurrence just because it is a lower-order term. This strategy is OK when making a *guess*, but it doesn't prove anything.

We need to choose a base for the logarithm in the statement (although any base will do); for convenience, we will take the logarithm base 2. It turns out that it is very difficult to prove using the substitution method that $T(n) \leq cn \log_2 n$ for $n \geq n_0$, no matter how big we choose $c$ and $n_0$ to be. Remembering a trick from the text, we think of subtracting lower-order terms. Ordinarily the most natural guess would be that we would want to subtract a multiple of $n$, but we will actually subtract an even lower-order term. Thus our guess will be that there are a natural number $n_0$ and constants $c, b > 0$ such that $T(n) \leq c(n - b) \log_2 n$. The recurrence relation doesn't even make sense for $n > 34$ (otherwise, $\lfloor n/2 \rfloor + 17 \geq n$), so we will try $n_0 = 35$. In order to ensure that the right-hand side is nonzero, we will then require that $b \leq 34$. As long as $c \geq \frac{T(35)}{\log_2 35}$, we have that $T(n) \leq c(n - b) \log_2 n$ for $n = 35$. This is our base case.

Now suppose that $n_1 \geq 36$ and we have proven the result for $35 \leq n < n_1$. Notice that $35 \leq \lfloor n_1/2 \rfloor + 17 < n_1$, so

$$T(n_1) = 2T(\lfloor n_1/2 \rfloor + 17) + n_1 \leq 2c(\lfloor n_1/2 \rfloor + 17 - b) \log_2(\lfloor n_1/2 \rfloor + 17) + n_1. \tag{1}$$

What we really want is

$$T(n_1) \leq c(n_1 - b) \log_2 n_1. \tag{2}$$

The easiest way for this to be true is if the constant in front of the logarithm in the right-hand side of (1) is no more than the constant in front of the logarithm in the right-hand side of (2), i.e., if $2c(\lfloor n_1/2 \rfloor + 17 - b) \leq c(n_1 - b)$. By cancelling the common $c$ and replacing $\lfloor n_1/2 \rfloor$ on the left-hand side by $n_1/2$ (which makes it bigger, at worst), we see that it suffices to have $2(n_1/2 + 17 - b) \leq n_1 - b$, i.e., $b \geq 34$. Since we also required $b \leq 34$ above, this gives the unique choice $b = 34$. Now notice that, since $n_1 \geq 36$, we have $n_1 \leq 18(n_1 - 34) \leq 18(n_1 - b)$ and $17 \leq \frac{17}{36} n_1$, so $\lfloor n_1/2 \rfloor + 17 \leq (1/2)n_1 + (17/36)n_1 = (35/36)n_1$. Thus, by (1),

$$T(n_1) \leq c(n_1 - b) \log_2((35/36)n_1) + 18(n_1 - 34) = c(n_1 - b)(\log_2 n_1 + \log_2(35/36) + 18/c).$$

Since what we want is $T(n_1) \leq c(n_1 - b) \log_2 n_1$, we are really asking that the remaining terms be negative, i.e., that $18/c \leq -\log_2(35/36)$. This

looks bad until we remember that $-\log_2(35/36) = \log_2(36/35) > 0$. (Here the specific numbers aren't important, only that $36/35 > 1$.) Thus, so long as $c \geq 18/\log_2(36/35)$ (a condition which does not involve $n_1$, hence is legal), the remaining terms are indeed negative and we have $T(n_1) \leq c(n_1 - b)\log_2 n_1$, as desired.

(3) (CLRS 4.2-4) **Suppose that $T(n) = T(n-a) + T(a) + cn$, where $a \geq 1$ and $c > 0$ are constants. By using a recurrence tree, find an explicit (and simple) function $f$ so that $T(n) = \Theta(f(n))$. Justify your answer by using the substitution method.** As usual, when drawing the recurrence tree, we will suppose that $n$ is an exact multiple of $a$. We will not actually reproduce the recurrence tree in this solution set, but note that every non-leaf node in the tree, corresponding to a subproblem of size $m$, sprouts two children, corresponding to subproblems of size $m-a$ and $a$. One of these problems (the one of size $a$) is a leaf; the other is a leaf only if $m - a = a$, i.e., $m = 2a$. There are $(n/a) - 1$ levels in the tree, labelled $i = 0, \ldots, (n/a) - 2$, where the non-leaf node at level $i$ corresponds to a problem of size $n - ia$. (If we try to go to level $i = (n/a) - 1$, then both subproblems are leaves.) Since the overhead for a problem of size $n - ia$ is $c(n - ia)$, the cost of the non-leaf nodes is

$$\sum_{i=0}^{(n/a)-2} c(n-ia) = cn((n/a)-1) - ca \sum_{i=0}^{(n/a)-2} a$$
$$= cn((n/a)-1) - ca\frac{((n/a)-1)((n/a)-2)}{2} = \Theta(n^2).$$

(Notice that there are $((n/a) - 2) - 0 + 1 = (n/a) - 1$ terms in the sum, not $(n/a) - 2$.) There are $(n/a)$ leaves in the tree, one at each of the levels labelled $i = 1, \ldots, (n/a) - 2$ and two at the level labelled $i = (n/a) - 1$. Thus the total cost of the leaves is $\Theta(n/a) = \Theta(n) = o(n^2)$, so the cost of the entire tree is $T(n) = \Theta(n^2) + o(n^2) = \Theta(n^2)$ (by (1h)).

However, this is just a guess. (We ignored the case where $n$ is not a multiple of $a$. Although it is possible in this simple case to handle that possibility explicitly, usually it will not be possible.) Let us try to prove that $T(n) = \Theta(n^2)$, that is, that there are a natural number $n_0$ and constants $c_1, c_2 > 0$ such that $c_1 n^2 \leq T(n) \leq c_2 n^2$ for $n \geq n_0$. (Be careful not to confuse the constants $c_1, c_2$ here with the constant $c$ occurring in the problem statement!) Since the right-hand side is never 0, we will try $n_0 = 1$. Notice that the recurrence relation makes sense only for $n > a$, so we will have to handle the terms $n \leq a$ separately. If $c_1 \leq \frac{T(1)}{1^2}, \ldots, \frac{T(a)}{a^2}$ and $c_2 \geq \frac{T(1)}{1^2}, \ldots, \frac{T(a)}{a^2}$, then $c_1 n^2 \leq T(n) \leq c_2 n^2$ for $n \leq a$. These $a$ statements will serve as our "base case". (This precision about which statements will serve as the base case is more than is necessary in the solution of a homework or exam problem, unless specifically otherwise stated. We include it here only for completeness.)

Now suppose that we have proven the statement for all $n < n_1$, and we want to prove it for $n = n_1$. By our choice of base case, we may suppose that $n_1 > a$. Then $T(n_1) = T(n_1 - a) + T(a) + cn_1 \leq c_2(n_1 - a)^2 + T(a) + cn_1 = c_2 n_1^2 + (c - 2ac_2)n_1 + (a^2 c_2 + T(a))$. It is tempting at this point to say

that the terms $(c - 2ac_2)n_1 + (a^2c_2 + T(a))$ are lower-order terms, hence negligible. *This is not OK.* Remember that we saw in class that this kind of reasoning, apparently perfectly reasonable, can be used to "prove" that (say) $T(n) = 2T(n/2) + n$ has solution $T(n) = \Theta(n)$, which we know is untrue.

Thus we must work a little harder. Now remember that $n_1 > a$, so $a^2c_2 < ac_2n_1$. Therefore, $T(n_1) < c_2n_1^2 + (c - ac_2)n_1 + T(a)$. What we really want is $T(n_1) \leq c_2n_1^2$. This is true if $(c - ac_2)n_1 + T(a) \leq 0$ or, rearranging, if $c_2 \geq \frac{c}{a} + \frac{T(a)}{n_1 a}$. However, this condition is not OK, because the right-hand side involves the variable $n_1$. To get rid of the dependence, just note that $n_1 > a$, so it's enough to require $c_2 \geq \frac{c}{a} + \frac{T(a)}{a^2}$ (a stronger requirement, and one that doesn't involve $n_1$).

We must also show that $T(n_1) \geq c_1n_1^2$. This proof looks similar, but is actually a little simpler. We have that $T(n_1) = T(n_1 - a) + T(a) + cn_1 \geq c_1(n_1 - a)^2 + T(a) + cn_1 = c_1n_1^2 + (c - 2ac_1)n_1 + (a^2c_1 + T(a))$. Since $a, c_1, T(a) \geq 0$, we may drop all these terms and conclude that $T(n_1) \geq c_1n_1^2 + (c - 2ac_1)n_1$. What we really want is $T(n_1) \geq c_1n_1^2$. This is true if $(c - 2ac_1)n_1 \geq 0$, i.e., if $c - 2ac_1 \geq 0$. Rearranging, we see that it is enough to require that $c_1 \leq \frac{c}{2a}$. Since this condition doesn't depend on $n_1$, we are finished.

(4) (CLRS 4-5) **The Fibonacci numbers $F_i$ satisfy the recurrence $F_i = F_{i-2} + F_{i-1}$, $i \geq 2$, and $F_0 = 0$, $F_1 = 1$. Define the generating function $\mathcal{F}$ by $\mathcal{F}(z) = \sum_{i=0}^{\infty} F_i z^i$.**

(a) **Show that $\mathcal{F}(z) = z + z\mathcal{F}(z) + z^2\mathcal{F}(z)$.** In this setting, writing out the two series and showing that the first few terms "match up" is not sufficient. What is needed is an algebraic proof. Actually, this is not much harder.
We have

$$z + z\mathcal{F}(z) + z^2\mathcal{F}(z) = z + \sum_{i=0}^{\infty} F_i z^{i+1} + \sum_{i=0}^{\infty} F_i z^{i+2} = z + \sum_{i=1}^{\infty} F_{i-1} z^i + \sum_{i=2}^{\infty} F_{i-2} z^i,$$

by replacing the variable $i$ in the first sum by $i - 1$, and the variable $i$ in the second sum by $i - 2$. Let's look at the $z^1$-terms separately. We get

$$z + z\mathcal{F}(z) + z^2\mathcal{F}(z) = (1 + F_{1-1})z^1 + \sum_{i=2}^{\infty} (F_{i-1} + F_{i-2}) z^i.$$

Remembering that $F_0 = 0$, $F_1 = 1$ and $F_i = F_{i-1} + F_{i-2}$ for $i \geq 2$, this becomes

$$z + z\mathcal{F}(z) + z^2\mathcal{F}(z) = F_0 z^0 + F_1 z^1 + \sum_{i=2}^{\infty} F_i z^i;$$

but this last expression is just $\mathcal{F}(z)$, as desired.

(b) **Show that $\mathcal{F}(z) = \frac{1}{\sqrt{5}}\left(\frac{1}{1-\phi z} - \frac{1}{1-\hat{\phi} z}\right)$, where $\phi = \frac{1+\sqrt{5}}{2}$ and $\hat{\phi} = \frac{1-\sqrt{5}}{2}$.** Since $\mathcal{F}(z) = z + z\mathcal{F}(z) + z^2\mathcal{F}(z)$, we have $(1 - z - z^2)\mathcal{F}(z) = z$,

i.e., $\mathcal{F}(z) = \frac{z}{1-z-z^2}$. On the other hand,

$$\frac{1}{\sqrt{5}}\Big(\frac{1}{1-\phi z} - \frac{1}{1-\hat{\phi}z}\Big) = \frac{((1-\hat{\phi}(z)) - (1-\phi(z)))/\sqrt{5}}{(1-\phi z)(1-\hat{\phi}z)} = \frac{(\phi-\hat{\phi})z/\sqrt{5}}{1-(\phi+\hat{\phi})z+\phi\hat{\phi}z^2}.$$

Since $\phi - \hat{\phi} = \sqrt{5}$, $\phi + \hat{\phi} = 1$ and $\phi\hat{\phi} = -1$, these two expressions are equal, as desired. (In general, one would have to factor $1 - z - z^2 = -(z - \phi^{-1})(z - \hat{\phi}^{-1})$ – that is, $1 - z - z^2 = -(z + \hat{\phi})(z + \phi)$ – and perform a partial fractions decomposition to find constants $A$ and $B$ such that $\frac{z}{1-z-z^2} = \frac{A}{z-\phi^{-1}} + \frac{B}{z-\hat{\phi}^{-1}}$.)

(c) **Show that** $\mathcal{F}(z) = \sum_{i=0}^{\infty} \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i)z^i$**.** We have that $\frac{1}{1-\phi z} = \sum_{i=0}^{\infty}(\phi z)^i$ and $\frac{1}{1-\hat{\phi}z} = \sum_{i=0}^{\infty}(\hat{\phi}z)^i$ (as can be verified of multiplying both sides of the first equation by $1 - \phi z$, and similarly for the second equation). Therefore,

$$\mathcal{F}(z) = \frac{1}{\sqrt{5}}\Big(\frac{1}{1-\phi z} - \frac{1}{1-\hat{\phi}z}\Big) = \sum_{i=0}^{\infty}\frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i)z^i,$$

as desired.

(d) **Prove that, for** $i > 0$**,** $\phi^i/\sqrt{5}$**, rounded to the nearest integer, is** $F_i$**.** By (4c), $F_i = \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i)$ for $i > 0$ (in fact for $i \geq 0$). Since $5 < 9$, $\sqrt{5} < \sqrt{9} = 3$, so $\hat{\phi} = \frac{1-\sqrt{5}}{2} > -1$. Thus $|\hat{\phi}^i| \leq |\hat{\phi}| = \frac{\sqrt{5}-1}{2} < \frac{\sqrt{5}}{2}$, $i \geq 0$, so $F_i = \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i)$ is less than $\frac{1}{2}$ away from $\frac{1}{\sqrt{5}}\phi^i$. Since $F_i$ is also an integer, this means that it is the nearest integer to $\frac{1}{\sqrt{5}}\phi^i$. (In fact, we have shown that there is no ambiguity; that is, that there are not two nearest integers.)

(e) **Prove that** $F_{i+2} \geq \phi^i$ **for** $i \geq 0$**.** By (4d), $F_{i+2} = \phi^i\big(\frac{\phi^2}{\sqrt{5}} - \big(\frac{\hat{\phi}}{\phi}\big)^i\frac{\hat{\phi}^2}{\sqrt{5}}\big)$. Since $|\hat{\phi}| = \frac{\sqrt{5}-1}{2} \leq \frac{\sqrt{5}+1}{2} = \phi$, we have that $|(\hat{\phi}/\phi)^i| \leq 1$. Thus, using (4c) again, $F_{i+2} \geq \phi^i\big(\frac{\phi^2-\hat{\phi}^2}{\sqrt{5}}\big) = \phi^i F_2 = \phi^i$, as desired.

(5) (CLRS 4.3-2) **The running time** $T(n)$ **of an algorithm** $A$ **on an input of size** $n$ **satisfies** $T(n) = 7T(n/2) + n^2$**, whereas the running time** $T'(n)$ **of an algorithm** $A'$ **on an input of size** $n$ **satisfies** $T'(n) = aT(n/4) + n^2$ **for some constant** $a$**. What is the largest integer** $a$ **such that** $A'$ **is asymptotically faster than** $A$**?** Since $4 < 7$, we have $2 = \log_2 4 < \log_2 7$, so $n^2 = O(n^{\log_2 7 - \varepsilon})$ for some $\varepsilon > 0$ (specifically, $\varepsilon = \log_2 7 - 2$). Thus, by the Master Theorem, $T(n) = \Theta(n^{\log_2 7})$.

For $T'(n)$, there are three possibilities: $2 < \log_4 a$, $2 = \log_4 a$ or $2 > \log_4 a$. Since we are interested in $a$ as large as possible, if there are any integral values of $a$ such that $2 < \log_4 a$ and $A'$ runs asymptotically faster than $A$ (i.e., $T'(n) = o(T(n))$), then we need not consider the other two cases. In fact, if $2 < \log_4 a$, i.e., $16 = 4^2 < 4^{\log_4 a} = a$, then $n^2 = O(n^{\log_4 a - \varepsilon'})$ for some $\varepsilon' > 0$, so $T'(n) = \Theta(n^{\log_4 a})$. Thus $T'(n) = o(T(n)) = o(n^{\log_2 7})$ if and only if $\log_4 a < \log_2 7$, i.e., if and only if

$$a = 4^{\log_4 a} < 4^{\log_2 7} = (2^2)^{\log_2 7} = (2^{\log_2 7})^2 = 49.$$

Thus there are integers $a > 16$ such that $A'$ runs asymptotically faster than $A$, so we need not consider the possibility that $2 = \log_4 a$ or $2 > \log_4 a$.

The computation above shows that the largest *integral* value of $a$ such that $A'$ runs asymptotically faster than $A$ is $a = 48$.

(6) (CRLS 4-6) **Suppose one has a collection of $n$ computer chips, some good and some bad, and a device which accepts a pair of chips and uses each to test the other. (A more vivid description is given in the text.) If two good chips test one another, each is reported good; if a good chip tests a bad chip, the bad chip is reported bad; and a bad chip returns a random answer when testing any other chip, good or bad.**

(a) **Show that, if more than $n/2$ chips are bad, this device is not sufficient to identify which chips are bad.** Note that any test results, resulting from any combination of chips, could have been mimicked by a collection of all-bad chips. Thus we cannot know for sure that we do *not* have all bad chips.

We show also that we cannot necessarily know for sure that we have all bad chips. (There are some test results that will definitively indicate bad chips – for example, if the tests $c_1c_2, c_1c_2, \ldots c_1c_n, c_1c_n$, performed successively, yield results $GG, BB, \ldots, GG, BB$ – so the best we can do is to show that the test results *might* be such that we cannot draw a conclusion.)

(b) **Consider the problem of finding one good chip. Show that, if more than $n/2$ of the chips are good, then $\lfloor n/2 \rfloor$ pairwise tests using the device are sufficient to reduce the problem to one of nearly half the size.** Label the chips $c_i$, $i = 1, \ldots, n$. We suppose we have an algorithm TEST$(i, j)$ which returns a two-entry array $T$ such that $T[1]$ is the result ($G$ or $B$) of using $c_j$ to test $c_i$, and $T[2]$ is the result of using $c_i$ to test $c_j$. If there are no more than two chips, then the fact that more than half of them are good means that they are all good, so we can pick (say) the first one. Otherwise, we consider choosing pairs of chips, testing them against one another, and discarding them unless each reports the other is good. This leaves a smaller pile of chips, which we test recursively. In other words, we have the algorithm

```
FIND-GOOD-CHIP(A)
 1 ◁ The input array A must be non-empty, and its entries must
     index more good chips than bad.
 2 if length[A] ≤ 2
 3     return A[1]
 4 endif
 5 m ← length[A]
 6 for j = 1 to ⌊length[A]/2⌋
 7     T ← TEST(A[2j − 1], A[2j])
 8     if T ≠ ⟨G, G⟩
 9         m ← m − 2
10         A[2j − 1] ← B
11         A[2j] ← B
12     endif
13 endfor
```

```
14 m' ← 2⌊m/2⌋
15 if ⌊m/2⌋ is odd
16       m ← m − 1
17       A[n] ← B
18 endif
19 for j = 1 to ⌊length[A]/2⌋
20       if A[2j − 1] ≠ B
21             m ← m − 1
22             A[2j] ← B
23       endif
24 endfor
25 create array A'[1 . . . m]
26 A' ← READNB(A, m)
27 return FIND-GOOD-CHIP(A')
```

We start this procedure with input $\langle 1, \ldots, n \rangle$, meaning that it tests all chips. The **if** test on line 15 is unexpected but important (consider running this algorithm on three chips, of which the first two are good and the last is bad, without it). READNB$(A, m)$ is an algorithm which returns a new array containing the $m$ entries of $A$ which are not marked $B$. (If desired, it may be improved to read only the entries of $A$ with odd indices, since the entries with even indices are all marked $B$.) These are the indices of the chips which we have not yet discarded. It is easy to see that this can be done in time $\Theta(n)$, where $n = \text{length}[A]$, but we omit the details.

We claim that "More than half of the chips not marked $B$ are good" is a loop invariant for the **for** loop on line 6. Indeed, we are given that this is true upon initialisation. If the statement was true the last time $j$ incremented, then either TEST returned $\langle G, G \rangle$ and no changes were made, or TEST returned something else. In this case, not both of the chips tested can be good. Thus the number of unmarked chips decreased by 2, while the number of unmarked good chips decreased by at most 1. This is the maintenance step, and, at termination, we simply observe that it is still true that more than half of the chips not marked $B$ are good.

Suppose that, at line 15, exactly $g$ of the chips indexed by the unmarked entries of $A[1 \ldots 2\lfloor n/2 \rfloor]$ are good, and exactly $b$ of the chips indexed by the unmarked entries of $A[1 \ldots 2\lfloor n/2 \rfloor]$ are bad. (Notice that we are ignoring the last entry $A[n]$ if $n$ is odd.) Then $g + b = m'$ (defined on line 14) and

(i) $g > b$ or

(ii) $g = b$, $n$ is odd and the chip indexed by $A[n]$ is good.

For any $j$ between 1 and $\lfloor n/2 \rfloor$, if $A[2j-1]$ is unmarked, then the chips indexed by $A[2j-1]$ and $A[2j]$ are both good or both bad. Thus, when the **for** loop on line 19 has finished, exactly $g/2$ of the chips indexed by the still-unmarked entries of $A[1 \ldots 2\lfloor n/2 \rfloor]$ are good, and exactly $b/2$ of the chips indexed by the still-unmarked entries of $A[1 \ldots 2\lfloor n/2 \rfloor]$ are bad. Moreover, since the **for** loop marks one of the remaining

unmarked entries bad at each iteration, after it has completed there are $m - \lfloor m/2 \rfloor = \lceil m/2 \rceil \leq \lceil n/2 \rceil$ unmarked entries remaining. Then

(i) $g/2 > b/2$ or

(ii) $g/2 = b/2$, $n$ is odd and the chip indexed by $A[n]$ is good.

In the latter case, a total of $(g/2)+1$ chips indexed by unmarked entries of $A$ are good, whereas a total of $b/2$ chips indexed by unmarked entries of $A$ are bad. Since $(g/2) + 1 = (b/2) + 1 > b/2$, we still have more good chips than bad. In the former case, either $g/2 = (b/2) + 1$ or $g/2 > (b/2) + 1$.

Suppose $g/2 = (b/2) + 1$. Then $\lfloor m/2 \rfloor = m'/2 = (g/2) + (b/2) = 2(b/2) + 1$ is odd, so $A[n]$ is marked $B$. Thus a total of $g/2$ chips indexed by unmarked entries of $A$ are good, whereas a total of $b/2$ chips indexed by unmarked entries of $A$ are bad. Since $g/2 > b/2$, once again we have more good chips than bad. Suppose, on the other hand, that $g/2 > (b/2) + 1$. Then at least $g/2$ chips indexed by unmarked entries of $A$ are good, whereas at most $(b/2) + 1$ chips indexed by unmarked entries of $A$ are bad. Thus in this case too we have more good chips than bad.

A particular (if unexpected) consequence is that we have *not* marked all the entries of $A$. Indeed, if we had marked all the entries of $A$, then exactly 0 of the chips indexed by unmarked entries of $A$ are good and exactly 0 of the chips indexed by unmarked entries of $A$ are bad, a contradiction of the fact that there should be more of the former than the latter.

Thus the recursive call to FIND-GOOD-CHIP on line 27 accepts as input a non-empty array more of whose entries index good chips than index bad ones, which is precisely the legal input.

(c) **By setting up and solving a recurrence describing the number of tests required, show that, if more than $n/2$ of the chips are good, then the good chips can be identified with $\Theta(n)$ pairwise tests.** Consider the algorithm FIND-GOOD-CHIP of (6b). Denote its worst-case running time on an input of size $n$ by $T(n)$. Then it is easy to see that

$$T(n) = \max\{T(k) : k = 1, \ldots, \lceil n/2 \rceil\} + f(n),$$

where $f(n) = \Theta(n)$. We cannot say that $T(n) = T(\lceil n/2 \rceil) + f(n)$, since all we know is that the recursive call in FIND-GOOD-CHIP takes an input of size *no more than* $\lceil n/2 \rceil$. The master theorem is thus not applicable here. (Even if it were applicable, we would have to verify that $f(n)$ satisfied the condition $af(n/b) \leq \gamma f(n)$ for some $\gamma < 1$. Since this is difficult to do without knowing exactly what form the function $f$ takes, it would be easier to do our usual trick of over- and under-estimating $T$ with asymptotically identical estimates.) In particular, there are a natural number $n_0$ and a constant $c > 0$ such that $f(n) \leq cn$ for $n \leq n_0$. By making $c$ larger, if necessary, we may ensure that $f(n) \leq cn$ for all $n$ and $c \geq \frac{T(1)}{3}$. We claim that $T(n) \leq 3cn$ for all $n$.

Since $c \geq 1$, the statement is true for $n = 1$. Now suppose that $n_1 > 1$ and we have proven the statement for all $n < n_1$. Then $\lceil n_1/2 \rceil < n_1$

and $\lceil n_1/2 \rceil \le (2/3)n_1$, so

$$\begin{aligned} T(n_1) &= \max\{T(k) : k = 1, \ldots, \lceil n_1/2 \rceil\} + f(n_1) \\ &\le \max\{3ck : k = 1, \ldots, \lceil n_1/2 \rceil\} + cn_1 \\ &= 3c\lceil n_1/2 \rceil + cn_1 \le 2cn_1 + cn_1 = 3cn_1. \end{aligned}$$

Thus $T(n) = O(n)$. Since also $T(n) = \Omega(f(n)) = \Omega(n)$, we have $T(n) = \Theta(n)$. By the master theorem (with $a = 1$ and $b = 2$), since $n = \Omega(n^{\log_2 1})$ and $1(n/2) \le (1/2)n$, we have that $T(n) = \Theta(n)$. Once we know a single good chip, we can use it to definitively identify every other chip as good or bad. The algorithm

FIND-GOOD-CHIPS($n$)

```
 1  g ← 0
 2  i ← FIND-GOOD-CHIP(⟨1, ..., n⟩)
 3  for j = 1 to n
 4      T ← TEST(i, j)
 5      if T[2] = G ◁ T[2] is the result of using the known-good
                          chip c_i to test chip c_j
 6          g ← g + 1
 7          A[i] ← G
 8      endif
 9  endfor
10  return READG(A, g)
```

identifies the (indices of the) good chips. Here, READG$(A, g)$ is an algorithm which returns a new array containing the $g$ entries of $A$ which are marked $G$. As before, we omit the details. Since the worst-case running time of this algorithm is $T(n) + \Theta(n) = \Theta(n)$, we are finished.

# Math 416 Homework 3 Solutions

Last updated 9 October 2005

(1) (CRLS 5.3-2) **Consider the following algorithm:**

PERMUTE-WITHOUT-IDENTITY($A$)

**1** $n \leftarrow \text{length}[A]$
**2** **for** $i = 1$ **to** $n - 1$
**3** **swap** $A[i]$ **and** $A[\text{RANDOM}(i + 1, n)]$
**4** **endfor**

**Does this code produce, at random, any permutation other than the identity permutation? That is, does it produce a uniform random non-identity permutation?** It is easy to see that, as long as $\text{length}[A] > 1$, the first entry of the output array is not the same as the first entry of $A$, so the output is a non-identity permutation of the input. (We assume tacitly, as usual, that the entries of $A$ are distinct.) Note that any choice of random numbers in this algorithm produces the same output as if those same random numbers were chosen in the algorithm PERMUTE-IN-PLACE. In particular, each possible output occurs for exactly one of the possible choices of random numbers. That is, each possible output occurs with the same probability.

Thus the only question is whether every non-identity output occurs. For $n = 2$, the only non-identity permutation of $A$ is $\langle A[2], A[1]\rangle$, and this permutation does occur. However, for $n > 1$, $\langle A[1], \ldots, A[n-2], A[n], A[n-1]\rangle$ is a non-identity permutation which cannot occur as the output of this algorithm (since the algorithm always swaps the first entry of $A$ with another entry, and can never thereafter swap the first entry back into place). Thus, for $n = 1$ or $n > 2$, this algorithm does not produce a uniform random non-identity permutation.

(2) (CRLS 5.3-3) **Consider the following algorithm:**

PERMUTE-WITH-ALL($A$)

**1** $n \leftarrow \text{length}[A]$
**2** **for** $i = 1$ **to** $n$
**3** **swap** $A[i]$ **and** $A[\text{RANDOM}(1, n)]$
**4** **endfor**

**Does this code produce a uniform random permutation? Why or why not?** Notice that there are $n^n$ choices of random numbers that could be made (for each value of $i$ from 1 to $n$, we have $n$ choices), whereas there are $n!$ possible outputs. For $n = 1$ or 2, one can verify that $n^n = n \cdot n!$, and each of the $n!$ possible outputs occurs $n$ times. Thus, in this case, the code produces a uniform random permutation of the output.

Now suppose that $n > 2$. If this code produces a uniform random permutation, then each of the $n!$ possible outputs should appear the same number of times, say $c$ times, so $n^n = cn!$. Now $n - 1$ divides the right-hand side, so it should also divide the left-hand side. However, we actually have that $n^n = (n^{n-1} - 1)(n - 1) + 1$, so the remainder of $n^n$ upon division by $n - 1$ is 1, not 0. (The reader is invited to see why this same method of proof doesn't work for $n = 1$ or 2.)

(3) (CRLS C.1-11) **Show that, for any $n, j, k \geq 0$ with $j + k \leq n$,**

$$\binom{n}{j+k} \leq \binom{n}{j}\binom{n-j}{k}.$$

**Give both an algebraic proof and an argument interpreting $\binom{\cdot}{\cdot}$ as counting the number of ways of choosing a collection of objects. Give an example of $n, j, k$ for which equality does not hold.** We have that $\binom{n}{j+k} = \frac{n!}{(j+k)!(n-j-k)!}$, whereas $\binom{n}{j}\binom{n-j}{k} = \frac{n!}{j!(n-j)!}\frac{(n-j)!}{k!(n-j-k)!} = \frac{n!}{j!k!(n-j-k)!}$. Since all the numbers involved are positive, we have that $\binom{n}{j+k} \leq \binom{n}{j}\binom{n-j}{k}$ if and only if $j!k! \leq (j+k)!$, or, equivalently, $k! \leq \frac{(j+k)!}{j!}$. Since $1 \leq j+1$, $2 \leq j+2$, $\ldots$ $k \leq j+k$, we have that

$$k! = (1)(2)\ldots(k) \leq (j+1)(j+2)\ldots(j+k) = \frac{(j+k)!}{j!}.$$

(4) (CRLS C.2-9) **Suppose that you are a contestant in a game show in which a prize is hidden behind one of three curtains. You will win the prize if you select the correct curtain. After you have picked one curtain but before the curtain is lifted, the host of the show lifts one of the other curtains, revealing that the prize is not behind it, and asks if you would like to switch your choice to the third curtain. How would your chances of winning change if you switched?** This is a famous problem in probability, called the Monty Hall problem, and has been the subject of much controversy (see `http://en.wikipedia.org/wiki/Monty_hall_problem` for a history, as well as a detailed discussion of the sometimes-unstated assumptions which go into this problem).

The correct answer is that the probability of winning by remaining with the original curtain is 1/3, whereas the probability of winning by switching is (therefore) 2/3. A simple proof comes from comparing possibilities: If the three curtains in question are labelled $A$, $B$ and $C$, then the twelve possible outcomes are as follows:

(a) The prize is behind curtain $A$.
  (i) We picked $A$ and the host opened $B$. We win by remaining with the original curtain.
  (ii) We picked $A$ and the host opened $C$. We win by remaining with the original curtain.
  (iii) We picked $B$ and the host opened $C$. We win by switching.
  (iv) We picked $C$ and the host opened $B$. We win by switching.

(b) The prize is behind curtain $B$.
  (i) We picked $B$ and the host opened $A$. We win by remaining with the original curtain.
  (ii) We picked $B$ and the host opened $C$. We win by remaining with the original curtain.
  (iii) We picked $A$ and the host opened $C$. We win by switching.
  (iv) We picked $C$ and the host opened $A$. We win by switching.

(c) The prize is behind curtain $C$.
  (i) We picked $C$ and the host opened $A$. We win by remaining with the original curtain.

(ii) We picked $C$ and the host opened $B$. We win by remaining with the original curtain.
(iii) We picked $A$ and the host opened $B$. We win by switching.
(iv) We picked $B$ and the host opened $A$. We win by switching.

Upon counting, one sees that, in six of the twelve possible outcomes, we win by remaining with the original curtain. Thus it *seems* that the probability of winning with this strategy is $\frac{6}{12} = \frac{1}{2}$. However, this is not correct! If we have picked the curtain hiding the prize, the host has to make a decision about which of the remaining curtains he will open; one unstated assumption is that he does so uniformly randomly. Thus the first two events in each of the three sub-lists above have half the probability of the remaining two, so our probability calculation changes a little. We see that the probability of winning by remaining with the original curtain is in fact

$$\frac{(1/2+1/2)+(1/2+1/2)+(1/2+1/2)}{(1/2+1/2+1+1)+(1/2+1/2+1+1)+(1/2+1/2+1+1)} = \frac{3}{9} = \frac{1}{3}.$$

Thus, as stated, the probability of winning by switching must be 2/3, so ours odds double if we switch.

Although the above calculation shows that we can compute the probability with only the most basic counting, some people prefer to use Bayes's theorem on conditional probabilities to find the probability in question. Thus, if we label the three curtains $A$, $B$ and $C$, and if we let $X, Y, W$ be random variables returning, respectively, the curtain chosen by the player, the curtain opened by the host and the winning curtain, then we want to compute $\Pr(X = W|Y = A)$, $\Pr(X = W|Y = B)$ and $\Pr(X = W|Y = C)$. We will compute only the first, since the calculations are the same. By Bayes's theorem,

$$\Pr(X = W|Y = A) = \frac{\Pr(X = W)\Pr(Y = A|X = W)}{\Pr(Y = A)}.$$

Another unstated assumption of the problem is that the prize has been placed uniformly randomly behind one of the curtains, so that $\Pr(X = W) = \frac{1}{3}$. On the other hand, the fact that the chosen curtain ($X$) is the winning curtain ($W$) does not change whether or not the curtain the host opened is $A$. (If we knew more – what, specifically, the winning curtain is – then this statement would fail.) That is, the events $X = W$ and $Y = A$ are independent, so $\Pr(Y = A|X = W) = \Pr(Y = A)$. Thus $\Pr(X = W|Y = A) = \Pr(X = W) = \frac{1}{3}$.

Finally, there are many explanations of this problem which purport to give some intuition that might lead one to believe the somewhat unexpected calculation above. To this author's mind, the most instructive of these is to realise that we are really being given the choice between two alternatives, namely opening a single curtain (the one we originally chose) or opening two curtains (the one chosen by the host, and the remaining curtain). It is clear that the latter decision has twice the chances of winning. **However, it must be emphasised that intuitive explanations like this should never substitute for actual calculations of probability.** There is much in probability that is counterintuitive, and as well many problems in probability which have two 'intuitive' but *contradictory* explanations. Only

formal mathematical calculations can guarantee a *correct* path through the intuition.

(5) (CRLS C.3-2) **An array $A$ has distinct, randomly ordered entries, which each permutation of the entries being equally likely. That is, the entries are uniformly randomly ordered. What are the expectations of the indices of the maximum and minimum entries in the array?** Let $m$ and $M$ be the random variables giving the index of the minimum and maximum entries of $A$, respectively. By assumption, $\Pr(m = i) = \frac{1}{n} = \Pr(M = i)$ for any $1 \leq i \leq n$, where $n = \text{length}[A]$ (and $\Pr(m = i) = 0 = \Pr(M = i)$ otherwise). Therefore, the expectations of these random variables are the same, namely

$$\sum_{i=1}^{n} i\frac{1}{n} = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n}\,\frac{n(n+1)}{2} = \frac{n+1}{2}.$$

That is, one expects the maximum *or* minimum (or second-largest, or seventh-smallest, or ...) entry of a uniformly randomly permuted array to appear in the middle.

(6) (CRLS C.3-9) **Show that, if $X$ is a random variable which assumes only the values 0 and 1, then $\text{Var}[X] = E[X]E[1 - X]$.** By definition, $\text{Var}[X] = E[X^2] - E[X]^2$. Since $X$ assumes only the values 0 and 1, $X^2 = X$, so

$$\text{Var}[X] = E[X] - E[X]^2 = E[X](1 - E[X]) = E[X]E[1 - X]$$

(since $E[1] = 1$).

# Math 416 Homework 5 Solutions

Due 26 October

(1) (CLRS 6-1) **The procedure** BUILD-MAX-HEAP **can be implemented by repeatedly using** MAX-HEAP-INSERT **to insert the elements into the heap, as follows:**

BUILD-MAX-HEAP$'(A)$
**1** heap-size$[A] \leftarrow 1$
**2** **for** $i = 2$ **to** length$[A]$
**3** $\quad A \leftarrow$ MAX-HEAP-INSERT$(A, A[i])$
**4** **endfor**
**5** **return** $A$

(a) **Do the procedures** BUILD-MAX-HEAP **and** BUILD-MAX-HEAP$'$ **always create the same heap when run on the same input?** No. Remember that the algorithm BUILD-MAX-HEAP looks like

BUILD-MAX-HEAP$(A)$
1 heap-size$[A] \leftarrow$ length$[A]$
2 **for** $i = \lfloor$length$[A]/2\rfloor$ to 1
3 $\quad A \leftarrow$ MAX-HEAPIFY$(A, i)$
4 **endfor**
5 **return** $A$

Consider the array $A = \langle 1, 2, 3\rangle$. Then length$[A] = 3$, so $\lfloor$length$[A]/2\rfloor = 1$, and the single execution of MAX-HEAPIFY on line 3 of BUILD-MAX-HEAP exchanges $A[1]$ with the largest of its children, $A[3]$, to obtain the max-heap $\langle 3, 2, 1\rangle$. This is the output of BUILD-MAX-HEAP.
On the other hand, the successive max-heaps produced by BUILD-MAX-HEAP$'$ are $\langle 1\rangle$, $\langle 2, 1\rangle$ and $\langle 3, 1, 2\rangle$ (where we have represented the heaps as arrays for convenience). This last is the output of BUILD-MAX-HEAP$'$.

(b) **Show that the worst-case running time of** BUILD-MAX-HEAP$'$ **is** $O(n \log n)$**.** We know from the text that the worst-case running time of MAX-HEAP-INSERT$(A, key)$ is $O(\log$ heap-size$[A])$. It is easy to see inductively that heap-size$[A] = i-1$ on line 3 of BUILD-MAX-HEAP$'$, so the total worst-case running time of BUILD-MAX-HEAP$'$ is $\sum_{i=2}^{n} O(\log(i-1))$. Since the constant hidden in each "$O$" is the same, this bound can be rewritten as $O\left(\sum_{i=2}^{n} \log(i-1)\right) = O((n-1)!) = O(n \log n)$, by Stirling's approximation.

(2) (CLRS 6-2)

(a) **How should one represent a** $d$**-ary heap as an array?** Given a heap, we define inductively the array $A$ by letting $A[1]$ be the root of the heap and, if we have put the value at node $n$ into $A[i]$, letting $A[d(i-1)+j]$ be the value at the $j$th child of node $n$ for $1 \leq j \leq d$. Put PARENT$(i) = \lceil i/d\rangle$ if $i > 1$ and PARENT$(i)$ = NULL otherwise. Instead of the LEFT and RIGHT functions, we have a single function, CHILD, accepting two arguments, the index $i$ of the parent and an index $1 \leq j \leq d$ indicating the desired child. Then CHILD$(i, j) = d(i-1)+j$.

(b) **What is the height of a** $d$**-ary heap of** $n$ **nodes in terms of** $n$ **and** $d$**?** Let us say that the $d$-ary heap consisting only of a root has height 0. (One could also say it has height 1. Either is OK, as long

as we're consistent!) Then the children of the first node (if it has $d$ of them) are the 2nd, ..., $(d+1)$st nodes, so any heap with between 2 and $d+1$ nodes has height 1. The children of the $(d+1)$st node (if it has $d$ of them) are the $(d^2+1)$st, ..., $(d^2+d+1)$st nodes, so any heap with between $d+2$ and $d^2+d+1$ nodes has height 2. Inductively, one sees that any heap with between $(d^{h-1}+\cdots+d+1)+1 = \frac{d^h-1}{d-1}+1$ and $(d^h+d^{h-1}+\cdots+d+1) = \frac{d^{h+1}-1}{d-1}$ nodes has height $h$. Thus, the height of a tree with $n$ elements is $h = \lfloor \log_d(n(d-1)) \rfloor$. (If we had said that the tree with only a root had height 1, then we would have $h = \lceil \log_d(n(d-1)+1) \rceil$.)

(c) **Give, and compute the running time of, an efficient implementation of** EXTRACT-MAX **for a $d$-ary max-heap.** Our implementation of EXTRACT-MAX looks just like the one for binary heaps:

```
EXTRACT-MAX(A)
 1 if heap-size[A] < 1
 2     return heap underflow error
 3 endif
 4 swap A[1] and A[heap-size[A]]
 5 heap-size[A] ← heap-size[A] − 1
 6 A ← MAX-HEAPIFY(A, 1)
 7 return A
```

but the MAX-HEAPIFY algorithm is (slightly) different. For $d$-ary heaps, it looks like

```
MAX-HEAPIFY(A, i)
 1 create C[1 ... d]
 2 for j = 1 to d
 3     C[j] ← CHILD(i, j)
 4 endfor
 5 largest ← i
 6 for j = 1 to d
 7     if C[j] ≤ heap-size[A]
 8         if A[C[j]] > A[largest]
 9             largest ← C[j]
10         endif
11     else
12         break
13     endif
14 endfor
15 if largest ≠ i
16     swap A[i] and A[largest]
17     A ← MAX-HEAPIFY(A, largest)
18 endif
19 return A
```

If the **if** test on line 15 is successful, then the tree rooted at *largest* has height one less than the height of the tree rooted at $i$. Thus, if we let $T(h)$ be a solution to the recurrence $T(h) = T(h-1) + O(d) = T(h-1) + O(1)$ (with appropriate boundary conditions), then $T(h)$ is an overestimate for the running time of MAX-HEAPIFY$(A, i)$ on a

heap $A$ such that the subheap rooted at $i$ has height $h$. One checks by the substitution method that $T(h) = O(h)$. Thus the worst-case running time of MAX-HEAPIFY$(A, 1)$ is $O(h)$, where $h$ is the height of the max-heap $A$. By part (2b), $h = O(\log n)$. Since EXTRACT-MAX does only a constant amount of work in addition to MAX-HEAPIFY, we have that its running time is also $O(\log n)$.

# Math 416 Homework 6 Solutions

Due 2 November, 2005

(1) (CLRS 7.1-2) **What value of $q$ does** PARTITION$(A, p, r)$ **return when all entries in $A[p \ldots r]$ are equal? Modify** PARTITION **so that $q = \lfloor (p+r)/2 \rfloor$ in this case.** We describe the modified algorithm PARTITION$'$ first.

PARTITION$'(A, p, r)$

```
 1  x ← A[r]
 2  k ← p − 1
 3  ℓ ← p − 1
 4  for j = p to r − 1
 5      if A[j] < x
 6          k ← k + 1
 7          swap A[j] and A[k]
 8          ℓ ← ℓ + 1
 9      else
10          if A[j] = x
11              ℓ ← ℓ + 1
12              swap A[j] and A[ℓ]
13          endif
14      endif
15  endfor
16  swap A[ℓ + 1] and A[r]
17  return ⟨A, ⌊(ℓ + k)/2⌋ + 1⟩
```

In the original algorithm PARTITION, the **if** test on the analogue of line 5 tests whether $A[j] \leq x$; lines 8–13 are not present; line 16 swaps $A[k+1]$ and $A[r]$; and line 17 returns $\langle A, k+1 \rangle$. Since the **if** test on the analogue of line 5 of the original algorithm is thus successful when all entries are equal, in that case the value of $k$ when the **for** loop terminates is $k = r-1$, so the value of $q$ returned is $q = k+1 = r$. In our modification, the **if** test on line 5 is always unsuccessful, but the **if** test on line 10 is always successful, if all entries are equal. Thus in that case the value of $k$ when the **for** loop terminates is $k = p-1$, whereas the value of $\ell$ when the **for** loop terminates is $\ell = r-1$, so the value of $q$ returned is $q = \lfloor (\ell+k)/2 \rfloor = \lfloor (p+r-2)/2 \rfloor + 1 = \lfloor (p+r)/2 \rfloor$.

The reader should check that a loop invariant for the **for** loop in our modified algorithm is "All entries of $A[p \ldots k]$ are less than $A[r]$; all entries of $A[(k+1) \ldots \ell]$ are equal to $A[r]$; and all entries of $A[(\ell+1) \ldots (j-1)]$ are greater than $A[r]$." On termination, this shows that our modified algorithm partitions correctly.

(2) (CLRS 7.2-2) **What is the running time of** QUICKSORT **on an array all of whose entries are equal?** Remember that the algorithm QUICKSORT looks like

QUICKSORT$(A, p, r)$

```
1  if p ≤ r
2      return A
3  endif
4  ⟨A, q⟩ ← PARTITION(A, p, r)
5  A ← QUICKSORT(A, p, q − 1)
```

6 **return** QUICKSORT$(A, q+1, r)$

Recall that the running time of PARTITION is $\Theta(n)$. By (1), PARTITION$(A, 1, n)$ returns $q = n$. Moreover, all entries of $A[1 \ldots (n-1)]$ are equal. (This second sentence is important to allow us to assert the recurrence relation below.) Thus, if $T(n)$ denotes the running time of QUICKSORT on an $n$-entry array all of whose entries are equal, then $T(n) = T(n-1) + \Theta(n)$. One then shows using the substitution method that $T(n) = \Theta(n^2)$.

(3) (CLRS 7.2-3) **Show that the running time of** QUICKSORT **on an array with distinct entries, sorted in decreasing order, is** $\Theta(n^2)$**.** Suppose that $A$ satisfies the conditions in the statement. Then the analogue in PARTITION of the **if** test on line 5 of PARTITION$'$ always fails, so, on completion of the **for** loop, $k = 0$. Thus we swap $A[1]$ and $A[n]$, and return $\langle A, 1\rangle$. Note that $A$ is no longer sorted in decreasing order, so we can't set up a recurrence yet! (The reader is encouraged to consider the case $A = \langle 5, 4, 3, 2, 1\rangle$ as an example throughout this problem.)

To fix this problem, let's look at what happens next. We run QUICKSORT$(A, 2, n)$, which in turn calls PARTITION$(A, 2, n)$. Since the new $A[n]$ is the old $A[1]$, which is the largest element of the array, the analogue in PARTITION of the **if** test on line 5 of PARTITION$'$ is now always successful, so we swap each entry $A[2], \ldots, A[n-1]$ with itself, and, on completion of the **for** loop, $k = n-1$. Thus we swap $A[n]$ with itself, and return $\langle A, n\rangle$. Note that $A$ has not changed at all in this call to PARTITION. In particular, the new entries of $A[2 \ldots (n-1)]$ are the same as the old entries, so they are sorted in decreasing order. Since our next call is to QUICKSORT$(A, 2, n-1)$, it is now appropriate to set up our recurrence.

Since the two calls to PARTITION still involved only $\Theta(n)$ running time, we have that, if $T(n)$ denotes the running time of QUICKSORT on an array sorted in decreasing order, then $T(n) = T(n-2) + \Theta(n)$. Once again, one verifies by the substitution method that $T(n) = \Theta(n^2)$.

(4) (CLRS 7.4-2) **Show that** QUICKSORT**'s best-case running time is** $\Omega(n \log n)$**.** If $T$ is always nonnegative and satisfies the recurrence $T(n) = \min\{T(q-1) + T(n-q) : q = 1, \ldots, n\} + \Theta(n)$, then, since the running time of PARTITION is $\Theta(n)$, we have that $T(n)$ is an underestimate for the running time of QUICKSORT on an input of length $n$. Let $d$ be a constant such that the $\Theta(n)$ term is eventually bounded below by $dn$. By making $d$ smaller if necessary, we may assume that this bound actually holds for all $n$.

Suppose that we have a constant $c$ and a number $N > 1$ such that $T(n) \geq cn \ln n$, $1 \leq n < N$. (Note we don't have our usual problem with the logarithm being 0 at 1, since we are trying to establish a *lower* bound! However, we do have to be careful to avoid plugging in $n = 0$.) By taking $c$ smaller if necessary, we may ensure that $c < d$. (The reason for this will become clearer later.) Then we have that

$$\begin{aligned} &T(N) \\ &\geq \min\{T(0)+c(N-1)\ln(N-1), c(q-1)\ln(q-1)+c(N-q)\ln(N-q) : q = 2, \ldots, N-1\} \\ &\qquad + dN. \end{aligned}$$

Now consider the function $x \mapsto c(x-1)\ln(x-1)+c(N-x)\ln(N-x)$ defined for $1 < x < N$. Its derivative is $x \mapsto c + c\ln(x-1) - c - c\ln(N-x) = c\ln(x-1) - c\ln(N-x)$, which is 0 precisely when $x - 1 = N - x$, i.e., $x = (N+1)/2$. The second derivative test reveals that this is a local minimum. Since $x = (N+1)/2$ is a critical point, it is actually a global minimum. Thus

$$\min\{c(q-1)\ln(q-1) + c(N-q)\ln(N-q) : q = 2, \ldots, N-1\} \geq 2c\frac{N-1}{2}\ln\Big(\frac{N-1}{2}\Big) = c(N-1)\ln\Big(\frac{N-1}{2}\Big).$$

Since also $T(0)+c(N-1)\ln(N-1) \geq c(N-1)\ln\big(\frac{N-1}{2}\big)$ (because $T(0) \geq 0$), we have that in fact

$$\begin{aligned} T(N) &\geq c(N-1)\ln\Big(\frac{N-1}{2}\Big) + dN \\ &= cN\ln(N-1) + \big(dN - cN\ln(N/(N-1)) + c\ln(2)(N-1)\big). \end{aligned}$$

Since $N/(N-1) \leq 2$, we have

$$T(N) \geq cN\ln N + (dN - c).$$

Since $c < d$ and $N > 1$, we have that $dN - c > 0$, so $T(N) \geq cN\ln N$. Thus (by the substitution method) $T(n) = \Omega(n\log n)$.

(5) (CLRS 7-1) **Consider the following variant partitioning algorithm:**

HOARE-PARTITION$(A, p, r)$

```
 1 if p > r
 2       return ⟨A, error⟩
 3 endif
 4 x ← A[p]
 5 i ← p
 6 j ← r
 7 do
 8       while A[j] > x
 9             j ← j − 1
10       endwhile
11       while A[i] < x
12             i ← i + 1
13       endwhile
14       if i < j
15             swap A[i] and A[j]
16             j ← j − 1
17             i ← i + 1
18       else
19             return ⟨A, j⟩
20       endif
21 enddo
```

**Prove the following.**

(b) **If $p \leq r$, then the indices $i$ and $j$ are such that we never access an element of $A$ outside the subarray $A[p \ldots r]$.** Since any change to $i$ in the program increments it, and any change to $j$ in the program decrements it, and since we start with $i = p$ and $j = r$, obviously at

all times $i \geq p$ and $j \leq r$, so we need only prove that at all times $j \geq p$ and $i \leq r$. The test in the **while** loop on line 8 fails on the first pass through the **do** loop if $j = p$ (since $A[p] = x$), so $j$ is not decremented past $p$ on the first loop. On the other hand, since, on the first pass through the **do** loop, $i = p$, we have that $A[i] = x$, so the test in the **while** loop on line 11 fails immediately on the first pass through the **do** loop.

After the **while** loops on lines 8 and 11 have terminated, we have that $A[i] \leq x$ and $A[j] \geq x$. Now there are two possibilities. If the **if** test on line 14 is unsuccessful, then execution ceases with the **return** instruction, so we are done. Note for use below that, in this case, $j = p$.

If the test is successful, we swap $A[i]$ and $A[j]$, then increment $i$ and decrement $j$. Thus there are indices $s = i - 1 \geq p$ and $t = j + 1 \leq r$ such that $A[s] \leq x$ and $A[t] \geq x$. Then $t > j \geq s$ and $s < i \leq t$. In particular, for the remainder of the program we have that $j < t$ and $i < s$. Note for use below that, in particular, $j < r$ for the remainder of the program. Now the only way that $A[s]$ or $A[t]$ will ever be moved in the remainder of the program is if, at some point, we reach the **if** test on line 14 when $i = t$ or $j = s$. However, in this case $i = t > j$ or $j = s > i$, respectively, so the **if** test fails. Thus neither $A[s]$ nor $A[t]$ will ever be moved. Then the **while** test on line 8 fails whenever $j = s$, and the **while** test on line 11 fails whenever $i = t$. Thus $i$ is never incremented past $t \leq r$, and $j$ is never decremented past $s \geq p$.

(c) **If $p < r$, then, when** HOARE-PARTITION **terminates, $p \leq j < r$.** In part (5b), we showed that, on termination, $j = p$ or $j < r$. If $p < r$, in either case $j < r$.

(d) **If $p \leq r$, then, when** HOARE-PARTITION **terminates, every element of $A[p \ldots j]$ is less than or equal to every element of $A[(j+1) \ldots r]$.** We claim that the following statement is true at every execution of line 14: "Every entry of $A[p \ldots (i-1)]$ is less than or equal to $x$, every entry of $A[(j+1) \ldots r]$ is greater than or equal to $x$, $A[i] \geq x$ and $A[j] \leq x$." (Notice that we are *not* claiming that this statement is true – because it needn't be! – when we begin the **do** loop, only on line 14.) We prove that this statement is a loop invariant in the usual way:

- **Initialisation**. The first time we reach line 14, the test $A[i'] < x$ was successful for every $p \leq i' < i$, and the test $A[j'] > x$ was successful for every $j < j' \leq r$, but the tests $A[i] < x$ and $A[j] > x$ were unsuccessful.
- **Maintenance**. Consider reaching line 14 after the first time. Denote by $i_{\text{old}}$ and $j_{\text{old}}$ the values of $i$ and $j$ the last time we reached line 14. That we have reached line 14 again means that $i_{\text{old}} < j_{\text{old}}$ and we performed the swap on line 15. Then all entries of $A[1 \ldots (i_{\text{old}} - 1)]$ are less than or equal to $x$ and all entries of $A[(j_{\text{old}} + 1) \ldots r]$ are greater than or equal to $x$; but in fact, since we swapped $A[i_{\text{old}}]$ and $A[j_{\text{old}}]$, even all entries of $A[1 \ldots i_{\text{old}}]$ are less than or equal to $x$ and all entries

of $A[j_{\text{old}} \ldots r]$ are less than or equal to $x$. On the most recent executions of the **while** loops on lines 8 and 11, we found that the test $A[i'] < x$ was successful for every $i_{\text{old}} + 1 \leq i' < i$, and the test $A[j'] > x$ was successful for every $j < j' \leq j_{\text{old}} - 1$, but the tests $A[i] < x$ and $A[j] > x$ failed. Thus we have maintained the invariant.

- **Termination**. For the program to terminate, we must have $i \geq j$. Then the loop invariant says that every entry of $A[p \ldots (i-1)]$ is less than or equal to $x$ and every entry of $A[(j+1) \ldots r]$ is greater than or equal to $x$, while $A[i] \geq x$ and $A[j] \leq x$. If $i = j$, this means that $A[j] = x$, so every entry of $A[p \ldots j]$ is either an entry of $A[p \ldots (i-1)]$ or $A[j]$, hence is less than or equal to $x$, which is in turn less than or equal to every entry of $A[(j+1) \ldots r]$. If $i > j$, then every entry of $A[p \ldots j]$ is an entry of $A[p \ldots (i-1)]$, hence is less than or equal to $x$, which is in turn less than or equal to every entry of $A[(j+1) \ldots r]$.

# Math 416 Homework 7

Due 9 November, 2005

(1) (CLRS 8.1-1) **What is the smallest possible depth of a leaf in a decision tree for a comparison sort?** Suppose that we reached the leaf $\langle i_1, i_2, \ldots, i_n \rangle$, but never passed through a node of the form $i_1 : i_2$. Then switching the $i_1$st and $i_2$nd entries of $A$ would still bring us to the leaf $\langle i_1, i_2, \ldots, i_n \rangle$, but then this would *not* correctly indicate how to sort the array $A$, contrary to the fact that we are working with a decision tree for a comparison sort. Thus we must have passed through the node $i_1 : i_2$ and, similarly, $i_2 : i_3, \ldots, i_{n-1} : i_n$. Thus the smallest possible depth of a leaf in a decision tree is at least $n-1$. Since it is easy to see that there are decision trees (for INSERTION-SORT, for example) in which some leaves have leaves of depth $n-1$, the smallest possible depth is in fact exactly $n-1$.

(2) (CLRS 8.1-3) **Show that there is no comparison sort whose running time is linear for at least half of the $n!$ permutations of a given $n$-element array, all of whose entries are distinct. What if we replace 'half' by '$1/n$'? $1/2^n$?** It suffices to show that there is no comparison sort whose running time is linear for at least $n!/2^n$ of the $n!$ permutations of such an array. Suppose that there is some comparison sort which sorts $n!/2^n$ of the permutations in no more than time $cn$. Then there are $n!/2^n$ leaves at depth no greater than $cn$. On the other hand, in any binary tree, there are fewer than $2^{cn}$ nodes (*a fortiori* leaves) at depth no greater than $cn$. Thus we have $n!/2^n \leq 2^{cn}$, or $n! \leq 2^{(c+1)n} = (2^{c+1})^n$. However, $n! = \omega(k^n)$ for any $k$, in particular, for $k = 2^{c+1}$, so this is impossible.

(3) (CLRS 8.2-4) **Describe an algorithm that, given $n$ integers in the range 0 to $k$, preprocesses its input in time $\Theta(n+k)$ and then answers in $O(1)$ time any query of the form "how many of the $n$ integers are between $a$ and $b$?".** Suppose the $n$ integers are stored in an array $A$. Remember our subroutine for counting sort:

PRE-COUNTING-SORT$(A, k)$

```
1 create C[(-1)...k]
2 C[-1] ← 0
3 for j = 1 to length[A]
4       C[A[j]] ← C[A[j]] + 1 ▷ At the end of this for loop, C[i] will
  count the number of times i occurs in A.
5 endfor
6 for i = 1 to k
7       C[i] ← C[i − 1] + C[i] ▷ At the end of this for loop, C[i] will
  count the number of times an integer ≤ i occurs in A.
8 endfor
9 return C
```

Obviously the running time of this preprocessing algorithm is $\Theta(n+k)$. Notice that we have included the count $C[-1]$, for reasons which will become clearer below. Once we have the array $C$, we can answer queries as in the statement in problem time:

INTERVAL-MEMBERSHIP$(C)$

```
1 if a > b
2       swap a and b
```

```
 3 endif
 4 if b > k
 5       b ← k
 6 endif
 7 if a < 0
 8       a ← 0
 9 endif
10 return C[b] − C[a − 1].
```

If $a < 0$ or $b > k$, it will not hurt to set $a = 0$ or $b = k$, since no entries of $A$ are outside the range $[0, k]$. Then the number of entries between $a$ and $b$ (inclusive) is the number of entries $\leq b$, minus the number of entries $< a$, which is just $C[b] - C[a-1]$.

(4) (CLRS 8-1) **Consider an array $C = \langle c_1, \ldots, c_n \rangle$ with distinct entries. Let $A$ be a (deterministic) comparison sort with decision tree $T_A$.**

(a) **Suppose that each leaf of $T_A$ is labelled with the probability that it is the leaf reached when we execute $A$ on a uniform random permutation of $C$. Show that exactly $n!$ leaves are labelled $1/n!$, and the rest are labelled 0.** Since the entries of $C$ are distinct, every permutation of $C$ leads to a different leaf, for a total of $n!$ reachable leaves. If we choose a uniform random permutation of $C$, then we have probability $1/n!$ of reaching any fixed one of these leaves, and probability 0 of reaching the others.

(b) **For any decision tree $T$, let $D(T)$ denote the sum of the depths of all the leaves of $T$. Suppose that $T$ is a decision tree with exactly $k > 1$ leaves, and let $T_{\mathbf{left}}$ and $T_{\mathbf{right}}$ be its left and right subtrees. Prove that $D(T) = D(T_{\mathbf{left}}) + D(T_{\mathbf{right}}) + k$.** Since $T$ has more than 1 leaf, its leaves are just the leaves of $T_{\text{left}}$, together with those of $T_{\text{right}}$. However, consider a leaf in $T_{\text{left}}$ of depth $d$. Then, considered as a leaf in $T$, it has depth $d+1$ (since we must also count the root of $T$). Similar reasoning holds for $T_{\text{right}}$. Thus $D(T) = (D(T_{\text{left}}) + i) + (D(T_{\text{right}}) + j)$, where $i$ is the number of leaves in $T_{\text{left}}$ and $b$ is the number of leaves in $T_{\text{right}}$. Since $i + j = k$, the total number of leaves in the tree, we are done.

(c) **For $k \geq 1$, let $d(k)$ be the minimum value of $D(T)$, taken over all decision trees with exactly $k$ leaves. Show that, if $k > 1$, then $d(k) = \min_{1 \leq i \leq k-1}(d(i) + d(k-i) + k)$.** Suppose that $T$ is a tree with $k > 1$ leaves. By part (4b), $D(T) = D(T_{\text{left}}) + D(T_{\text{right}}) + k$. If $T_{\text{left}}$ has $i$ leaves, then $D(T_{\text{left}}) \geq d(i)$; and $T_{\text{right}}$ has $k - i$ leaves, so $D(T_{\text{right}}) \geq d(k-i)$. Thus $D(T) \geq d(i) + d(k-i) + k$, so, in particular, it is at least as big as the desired minimum.

On the other hand, let $1 \leq i \leq k-1$ be the value that minimises $d(i) + d(k-i) + k$. Let $T_{\text{left, min}}$ be a tree with $i$ leaves such that $D(T_{\text{left, min}}) = d(i)$, and $T_{\text{right, min}}$ be a tree with $k$ leaves such that $D(T_{\text{right, min}}) = d(k-i)$. These trees exist by the definition of the function $d$. Let $T_{\text{min}}$ be the tree whose left subtree is $T_{\text{left, min}}$ and whose right subtree is $T_{\text{right, min}}$. Then $D(T) = D(T_{\text{left}}) + D(T_{\text{right}}) + k = d(i) + d(k-i) + k$, which, by assumption, is equal to the desired minimum.

(d) **Prove that, for a given value of $k > 1$, the minimum of the function $x \mapsto x \lg x + (k-x)\lg(k-x)$ is achieved at $x = k/2$. Conclude that $d(k) = \Omega(k \lg k)$.** The derivative of the indicated function is $x \mapsto (\lg e) + \lg x - (\lg e) - \lg(k-x) = \lg x - \lg(k-x)$, which is 0 precisely when $x = k - x$, i.e., $x = k/2$. The second derivative test shows that this is a local minimiser. Since it is the only critical point, it is actually a global minimiser.

Now suppose that $c$ is a constant and $K > 1$ an integer such that $d(k) \geq ck \lg k$ for $1 \leq k < K$. By taking $c$ smaller if necessary, we may ensure that $c < 1$. Then

$$\begin{aligned} d(K) &= \min\{d(i) + d(K-i) : i = 1, \ldots, K-1\} + k \\ &\geq \min\{ci \lg i + c(K-i)\lg(K-i) : i = 1, \ldots, K-1\} + K. \end{aligned}$$

By our calculations above,

$$d(K) \geq 2c\frac{K}{2}\lg\left(\frac{K}{2}\right) + K = cK \lg K + (K - cK) > cK \lg K.$$

(We have used the fact that $c < 1$.) Then, by the substitution method, $d(k) = \Omega(k \lg k)$.

(e) **Prove that $D(T_A) = \Omega(n! \lg(n!))$, and conclude that the expected time to sort a uniform random permutation of $C$ is $\Omega(n \lg n)$.** Notice that we can prune all the unreachable nodes from $T_A$, and still have a decision tree for a comparison sort. This will only decrease $D(T_A)$ and will not change the expected time to sort a uniform random permutation (since the unreachable nodes don't affect the running time), so we will be done if we can complete this problem for the pruned tree.

By part (4a), $T_A$ has $n!$ leaves. Therefore, by part (4d), $D(T_A) = \Omega(n! \lg(n!))$. The expected time to sort a uniform random permutation of $C$ is $\sum_{\text{all leaves}} \Pr(\text{reaching that leaf}) \cdot (\text{depth of the leaf})$. By part (4a), the probability in question is $1/n!$ (since we have got rid of the leaves labelled with probability 0). Therefore, we can pull it out front to get $\frac{1}{n!}\sum_{\text{all leaves}}(\text{depth of the leaf})$. Now, however, the sum is just $D(T_A)$, so the expected running time is $\frac{1}{n!}D(T_A) = \frac{1}{n!}\Omega(n! \lg(n!)) = \Omega(\lg n!) = \Omega(n \lg n)$ (by Stirling's approximation, or – for a lower bound such as this one – by the elementary calculation shown in class).

(f) **Show that, for any randomised comparison sort $B$, there exists a deterministic comparison sort $A$ such that, on average, the number of comparisons performed by $A$ on a uniform random permutation of $C$ is no greater than the number of comparisons performed by $B$.** We will show how to get rid of one randomiser node without increasing the average number of comparisons performed by our comparison sort. Performing this operation repeatedly will yield a deterministic comparison sort. Suppose our randomiser node chooses uniformly randomly among $r$ subtrees $T_1, \ldots, T_r$. Then the expected running time of our comparison sort on a uniform random permutation of $C$ is $\sum_{\text{all leaves}} \Pr(\text{reaching that leaf}) \cdot (\text{depth of the leaf})$. Let's break this sum up into those leaves which are in some tree $T_i$, and those which are not. Since the second sum

will not change when we get rid of the randomiser node, we ignore it for the moment. (However, don't forget it – we'll refer to it several paragraphs below.) Let $d$ be the depth of the randomiser node, and $p$ the probability of reaching the randomiser node.
Now let's look at a fixed leaf in some tree $T_i$. Denote by $E_i$ the event of reaching the randomiser node and choosing subtree $i$. Then

$$\begin{aligned}\Pr(\text{reaching that leaf}) &= \Pr(\text{reaching the randomiser node})\\ &\quad\times \Pr(\text{choosing subtree } T_i|\text{reaching the randomiser node})\\ &\qquad\qquad\times \Pr(\text{reaching that leaf}|E_i).\end{aligned}$$

In other words, since the subtree is uniformly randomly chosen among $r$ choices once we have reached the randomiser node,

$$\Pr(\text{reaching that leaf}) = p\cdot\frac{1}{r}\cdot\Pr(\text{reaching that leaf}|E_i).$$

Notice that

$$\Pr(\text{reaching that leaf}|E_i) = \Pr(\text{reaching that leaf, starting in tree } T_i).$$

Finally, notice that

$$\text{depth of the leaf} = (\text{depth of the leaf in tree } T_i) + d.$$

Thus the first sum (over leaves occurring in some tree $T_i$) becomes

$$\frac{p}{r}\sum_{i=1}^{r}\sum_{\text{leaves in } T_i}\Pr(\text{reaching that leaf, starting in tree } T_i)\big((\text{depth of the leaf in tree } T_i)+d\big).$$

Since the sum of all the probabilities in the inner sum is 1, we can pull out $d$ to get

$$p\Big[\Big(\frac{1}{r}\sum_{i=1}^{r}\sum_{\text{leaves in } T_i}\Pr(\text{reaching that leaf, starting in tree } T_i)(\text{depth of the leaf in tree } T_i)\Big)+d\Big].$$

Now let $j$ be the value of $i$ which makes the inner sum (over leaves in $T_i$) smallest. (There might be several choices; make any one.) Then the sum above is at least

$$\begin{aligned}&p\Big[\Big(\sum_{\text{leaves in } T_j}\Pr(\text{reaching that leaf, starting in tree } T_j)(\text{depth of the leaf in tree } T_j)\Big)+d\Big]\\ &= p\sum_{\text{leaves in } T_j}\Pr(\text{reaching that leaf, starting in tree } T_j)(\text{depth of the leaf in tree } T).\end{aligned}$$

Now here's the trick – if we *replace* the randomiser node, with all its subtrees, by the subtree $T_j$, then one can check that now the expected running time of our algorithm on a uniform random permutation of $C$ is equal to the big sum on the last line of the previous paragraph, plus the "second sum" (the one over leaves not in some $T_i$, which we have so far been ignoring). However, by the way we chopse $j$ above, this is guaranteed to be no greater than the average running time of the original algorithm (with randomiser in place). This is just what was desired.

# Math 416 Homework 8

Due 23 November, 2005

(1) (CLRS 8-2) **Suppose that we have an array of $n$ data records to sort, and that the key of each record has the value 0 or 1. An algorithm for sorting (the keys of) such a set of records might possess some of the following three desirable characteristics:**

1. **the algorithm runs in $O(n)$ time;**
2. **the algorithm is stable;**
3. **the algorithm sorts in place.**

(a) **Give an algorithm that has characteristics 1 and 2.** COUNTING-SORT is stable and runs in $O(n)$ time.

(b) **Give an algorithm that has characteristics 1 and 3.** This was problem 1 on exam 2.

Several people also noted that QUICKSORT's PARTITION subroutine, which runs in $O(n)$ time and uses $O(1)$ memory, will sort an array whose entries contain only two values.

(c) **Give an algorithm that has characteristics 2 and 3.** INSERTION-SORT is stable and sorts in place.

(2) (CLRS 8-6)

(a) **Show that, given $2n$ distinct numbers, there are $\binom{2n}{n}$ possible ways to divide them into two sorted arrays, each with $n$ numbers.** Note that the described action is the *same* as simply choosing $n$ numbers from the $2n$ original numbers. Indeed, obviously the action described involves choosing $n$ numbers – say, the $n$ numbers belonging to the first array. On the other hand, if we specify that choosing $n$ numbers is choosing the entries of the first array, then we have already chosen the entries of the second array (just the remaining $n$ numbers), and there is no question of ordering (since the arrays are sorted, and contain distinct entries). By definition, there are $\binom{2n}{n}$ ways to choose $n$ numbers from a pool of $2n$ distinct numbers.

(b) **Using a decision tree, show that any algorithm that correctly merges two sorted lists uses at least $2n-o(n)$ comparisons.** The original version of this problem didn't state, but should have, that this many comparisons are used *in the worst case.* It is possible to use far fewer comparisons in the best case. For example, if a comparison shows that the $n$th entry of the first array is smaller than the first element of the second array, then no further comparisons are necessary to merge the two arrays.

We consider a decision tree of the usual sort, with non-leaf nodes labelled by comparisons, the left child indicating a successful comparison and the right child indicating an unsuccessful one; but now we label the leaves by the set of indices in the sorted array of the entries of the first array. (For example, if our arrays were $\langle 3,5,9\rangle$ and $\langle 2,8,10\rangle$, then the sorted array would be $\langle 2,3,5,8,9,10\rangle$, in which the entries of the first array are the second, third and fifth; so the label on the leaf we would reach is $\{2,3,5\}$. Note that we are only concerned with the *set*, so the label could as well be $\{5,2,3\}$.) Since there are $\binom{2n}{n}$ possible labels (one for each $n$-element subset of $\{1,\ldots,2n\}$), and every possible label

must occur on at least one reachable leaf, the 'pruned' decision tree (in which non-reachable nodes are removed) has at least $\binom{2n}{n}$ reachable leaves. Since a (binary) tree of height $h$ has at most $2^h$ nodes, we have that the height of the decision tree is at least $\log_2 \binom{2n}{n}$. Since the height of the decision tree is the same as the worst-case number of comparisons, we would like to show that $\log_2 \binom{2n}{n} = 2n - o(n)$.

It is tempting to write $\log_2 \binom{2n}{n} = \log_2(2n)! - 2\log_2 n!$ and try to use the asymptotic formula $\log_2 m! = \Theta(m \log m)$, but this doesn't work, for the following reason. That $\log_2 m! = \Theta(m \log m)$ means that there are constants $c_1 < c_2$ such that $c_1 m \log_2 m \leq \log(m!) \leq c_2 m \log_2 m$ for all $m$ sufficiently large. By replacing $c_1$ by a smaller constant and $c_2$ by a larger one if necessary, we may assume that the inequality actually holds for all $m > 1$. Normally we'd run into a problem at $m = 1$, since $\log(m!) = 0$, but, since $m \log_2 m = 0$ also at $m = 1$, our inequality even holds for $m = 1$. Thus we have $\log_2(2n)! - 2\log_2 n! \geq c_1(2n)\log_2(2n) - 2c_2 n \log_2(n) = 2c_1 n - 2(c_2 - c_1)n \log n$. Even if we had $c_1 = 1$, the error term would be $\omega(n)$, not $o(n)$.

It is possible to show the desired result using the full force of Stirling's approximation (instead of just its consequence mentioned above), but Daniel Konson provided the following simpler argument. By the binomial theorem,

$$\sum_{i=0}^{2n} \binom{2n}{i} = \sum_{i=0}^{2n} \binom{2n}{i} 1^i 1^{2n-i} = (1+1)^{2n}.$$

A simple calculation shows that $\binom{2n}{i} - \binom{2n}{i-1} = \frac{n!}{i!(n-i+1)!}(2n+1-2i)$. In particular, $\binom{2n}{i} - \binom{2n}{i-1} > 0$ if $i \leq n$, and $\binom{2n}{i} - \binom{2n}{i-1} < 0$ if $i \geq n$. This means that $\binom{2n}{n} \geq \binom{2n}{i}$, $i = 0, \ldots, 2n$. Thus

$$(2n+1)\binom{2n}{n} \geq \sum_{i=0}^{2n} \binom{2n}{i} = 2^{2n},$$

so $\log_2 \binom{2n}{n} \geq 2n - \log_2(2n+1)$. Since $\log_2(2n+1) = o(n)$, we are finished.

(c) **Show that, if two elements are consecutive in the sorted order and come from opposite arrays, then they must be compared.** Suppose that the original arrays are $\langle a_1, \ldots, a_n \rangle$ and $\langle b_1, \ldots, b_n \rangle$ and that $a_i$ and $b_j$ are consecutive in the sorted order. By switching the arrays if necessary, we may suppose that $a_i < b_j$. Suppose that no comparison is made between $a_i$ and $b_j$. Since comparing any element, other than $a_i$ or $b_j$, to $b_j$ is the same as comparing it to $a_i$ (since $a_i$ and $b_j$ are consecutive), and similarly for comparing any element, other than $a_i$ or $b_j$, to $a_i$, we follow the same path through the decision tree, and hence reach the same node, if we run our merging algorithm on the two (sorted) arrays $\langle a_1, \ldots, a_{i-1}, b_j, a_{i+1}, \ldots, a_n \rangle$ and $\langle b_1, \ldots, b_{j-1}, a_i, b_{j+1}, \ldots, b_n \rangle$. This will have the effect of switching $a_i$ and $b_j$ in the output, giving an incorrectly sorted array. Since this is a contradiction, there must after all have been some comparison made between $a_i$ and $b_j$.

(d) **Use your answer to** (2c) **to show that, in the worst case, $2n-1$ comparisons must be made when merging two sorted arrays, each containing $n$ entries.** Consider the two sorted arrays $\langle 0, 2, \ldots, 2n-2\rangle$ and $\langle 1, 3, \ldots, 2n-1\rangle$. Then any two consecutive entries in the sorted array come from opposite arrays. Since there are $2n-1$ such pairs (the $i$th and $(i+1)$st, for $i = 0, \ldots, 2n-1$), at least $2n-1$ comparisons must be made.

(3) (CLRS 8.4-2) **What is the worst-case running time for** BUCKET-SORT**? What simple change to the algorithm preserves its linear expected running time and makes its worst-case running time $O(n \log n)$?** If all the entries fall into one bucket, then BUCKET-SORT is (aside from $\Theta(n)$ overhead) just INSERTION-SORT. Since we know the worst-case running time of INSERTION-SORT is $\Theta(n^2)$, this means that the worst-case running time of BUCKET-SORT is $\Omega(n^2)$. We only say $\Omega(n^2)$, rather than $\Theta(n^2)$, because we don't know until we reason a little farther that the case where all entries fall into a single bucket is actually the worst case.

The following remark will come in handy: If $f$ is a continuous function on $[0, +\infty)$ such that $f(0) = 0$ and $f''(x) > 0$ for all $x > 0$, then $f(x) + f(y) \le f(x+y)$ for all $x, y \in [0, \infty)$ [1]. Then, by induction, we have that $\sum_{i=0}^{k} f(x_i) \le f\big(\sum_{i=0}^{k} x_i\big)$ for any $x_1, \ldots, x_k \ge 0$.

We know from class (see also p. 175 of the text) that the running time of BUCKET-SORT on an input of length $n$ is $\Theta(n) + \sum_{i=0}^{n-1} O(m_i^2)$, where $m_i$ is the number of entries in the input in $[(i-1)/n, i/n)$ and the constant hidden in each "$O$" is the same. Since the constants are the same, we have that

$$\sum_{i=0}^{n-1} O(m_i^2) = O\Big(\sum_{i=0}^{n-1} m^2{}_i\Big).$$

Since $f : x \mapsto x^2$ satisfies the conditions of the previous paragraph, we have that

$$\sum_{i=0}^{n-1} m_i^2 \le \Big(\sum_{i=0}^{n-1} m_i\Big)^2 = n^2.$$

Thus the running time is $\Theta(n) + O(n^2) = O(n^2)$. Putting this together with the previous bound, we have that the worst-case running time of BUCKET-SORT is $\Theta(n^2)$.

Suppose that we replace the calls to INSERTION-SORT (on each bucket) with a call to MERGE-SORT, or any other fast sorting algorithm. Then the same reasoning as in class shows that the running time becomes

$$\Theta(n) + \sum_{i=0}^{n-1} O(m_i \log m_i) = \Theta(n) + O\Big(\sum_{i=0}^{n-1} m_i \log m_i\Big),$$

with notation as above. Since $O(m_i \log m_i) = O(m_i^2)$, we still have that the running time is $\Theta(n) + \sum_{i=0}^{n-1} O(m_i^2)$, so we still have that the expected

[1] Indeed, fix $y \in [0, \infty)$ and consider the function $x \mapsto f(x+y) - (f(x) + f(y))$. Since $f(y)$ is constant, The derivative at $x > 0$ is $f'(x+y) - f'(x)$. Since $f''$ is everywhere positive, we have that $f'$ is increasing, so $f'(x+y) - f'(x) \ge 0$. Since this is true for all $x > 0$, $x \mapsto f(x+y) - f(x) - f(y)$ is a nondecreasing function. Thus, for any $x \ge 0$, $f(x+y) - f(x) - f(y) \ge f(0+y) - f(0) - f(y) = 0$.

running time is $O(n)$. On the other hand, note that the function $f$ on $[0, \infty)$ defined by

$$f(x) = \begin{cases} x \ln x, & x > 0 \\ 0, & x = 0 \end{cases}$$

satisfies the conditions of the paragraph above, so that

$$\sum_{i=0}^{n-1} m_i \log m_i \leq \Big(\sum_{i=0}^{n-1} m_i\Big) \log\Big(\sum_{i=0}^{n-1} m_i\Big) = n \log n.$$

Thus the (worst-case) running time is $\Theta(n) + O(n \log n) = O(n \log n)$. We can show as above that it is also $\Omega(n \log n)$, hence that the worst-case running time for this modified version of BUCKET-SORT is $\Theta(n \log n)$.

(4) (CLRS 8.4-4) **We are given $n$ nonzero points $p_i = (x_i, y_i)$, $1 \leq i \leq n$, in the unit disk. Suppose that the points are uniformly distributed, i.e., that the probability of finding any fixed point in a subset of the disk is proportional to the area of that subset.**

**Suppose also that the points are independently distributed, that is, that the probability of finding some subset of the points in a subset of the disk is the product of the individual probabilities. (That sentence doesn't appear in the text, but it should!)**

**Design an algorithm with linear expected running time to sort the $n$ points by their distances from the origin.** The text also included the proviso that $x_i^2 + y_i^2 > 0$ (i.e., that $\langle x_i, y_i \rangle \neq \langle 0, 0 \rangle$), but this is unnecessary. Consider the following algorithm. For convenience, we suppose that the $n$ points are stored in an array $P$; that each point $p_i$ is an array $\langle x_i, y_i \rangle$ of length 2. We will treat the points $p_i$ as 'records', and will not do any sorting on them. We will assign 'key' values in the algorithm, on which we will do the actual sorting.

Remember from class that we have supposed we have algorithms INSERT$(val, B, i)$, which inserts value $val$ at the end of the (partially filled) $i$th row of a two-dimensional array $B$, and CONCATENATE$(B, n)$, which reads off the $n$ filled entries of a two-dimensional array $B$ into a single output array of length $n$.

CIRCLE-BUCKET-SORT$(P)$

```
 1  n ← length[P]
 2  create B[0...n][1...n]  ▷ A two-dimensional, (n + 1) × n, array.
 3  for j = 1 to n
 4      k_j ← p_j[1]^2 + p_j[2]^2
 5      attach k_j to P[j] as key
 6      B ← INSERT(P[j], B, ⌈n k_j⌉)
 7  endfor
 8  for i = 1 to n
 9      B[i] ← INSERTION-SORT(B[i])  ▷ Sort the ith row of the two-
        dimensional array B, using the k_j as keys.
10  endfor
11  return CONCATENATE(B, n)
```

Notice we do not need to sort $B[0]$, because it consists of points all at the same distance (0) from the origin.

Our first instinct is to assign the $j$th point to the $\lceil nd_j \rceil$th bucket, where $d_j = \sqrt{k_j}$. While this approach suffers from the æsthetic defect that the buckets are not expected to be of equal sizes (since, for example, the annulus of inner radius $0/n$ and outer radius $1/n$ – a small disc with its centre removed – is much smaller than the annulus of inner radius $(n-1)/n$ and outer radius $n/n$ – a large annulus near the edge of the disc), the original statement here that the expected running time is $\Theta(n^2)$ was incorrect. In fact this algorithm has expected running time $\Theta(n)$ as well.

The reason our approach works is that the annuli consisting of points whose *squared* distance from the origin is in $((i-1)/n, i/n]$ have the same area for all $i = 1, \ldots, n$ (whereas this would not be true if we removed the word 'squared'). As in the ordinary BUCKET-SORT, we see that the expected running time of CIRCLE-BUCKET-SORT is $\Theta(n) + \sum_{i=1}^{n} O(E[m_i^2])$, where $m_i$ is the size of the $i$th bucket and the hidden constant in each "$O$" is the same. Let $X_{ij}$ be the indicator random variable which is 1 if $p_j$ lands in the $i$th bucket, and 0 otherwise. Then $m_i = \sum_{j=1}^{n} X_{ij}$. Note that the $X_{ij}$, $j = 1, \ldots, n$, are independent random variables [2]. Also,

$$E[X_{ij}] = \Pr(X_{ij} = 1) = \Pr(p_j \text{ lands in bucket } i)$$

$$= \Pr(\boxed{\begin{array}{l} p_j \text{ is outside the disk of squared} \\ \text{radius } (i-1)/n \text{ but inside the disk} \\ \text{of squared radius } i/n \end{array}})$$

By the definition of a uniform random distribution, this last line is

$$\frac{1}{\pi}(\text{area of outer disk} - \text{area of inner disk})$$

(the factor of $1/\pi$ in front is because the total area is $\pi$, not 1, and we need to normalise probabilities). Since the area of a disk is $\pi$ times its squared radius, the probability is just

$$(\text{squared outer radius} - \text{squared inner radius}) = \frac{1}{n}.$$

Thus these random variables $X_{ij}$ are the same as the ones that occurred in class (the ones defined on p. 175 of the text), and the calculation proceeds as before to show that the expected running time is $\Theta(n) + n(2 - 1/n) = \Theta(n)$.

(5) (CLRS 27.1-2) **Suppose that $n$ is a power of 2. Show how to construct an $n$-input, $n$-output comparison network of depth $\log_2 n$ in which the top output wire always carries the minimum input value and the bottom output wire always carries the maximum input value.** We will construct, by induction on $\log_2 n$, a comparison network MINMAX[$n$] which has the desired properties, and is such that the topmost output wire has depth $\log_2 n$. (That is, we are specifying not just the maximum depth of an output wire, but the wire on which it occurs.) MINMAX[1] will be the unique comparison network on $n = 1$ wires. If MINMAX[$n/2$] has been constructed, define MINMAX[$n$] to be the comparison network which first passes the top and bottom $n/2$ wires through two copies of MINMAX[$n/2$] running in parallel, then passes the 1st and

[2] For indicator random variables, this just means that the events they 'indicate' are independent. The consequence for us is that $E[X_{ij}X_{ik}] = E[X_{ij}]E[X_{ik}]$.

$(n/2+1)$st, and the $(n/2)$th and $n$th, wires through two comparators. Since the two copies of MinMax$[n/2]$ run in parallel, as do the two new comparators, the depth of MinMax$[n]$ is no more than $\log_2(n/2)+1 = \log_2 n$. On the other hand, since the top output wire of the top copy of MinMax$[n/2]$ has depth $\log_2(n/2)$, and since it passes through one more comparator in MinMax$[n]$, the depth of the top output wire of MinMax$[n]$ is exactly $\log_2(n/2)+1 = \log_2(n)$, so the depth of MinMax$[n]$ is at least $\log_2 n$. Thus the depth of the network is precisely $\log_2 n$.

Now we show that the comparison network puts the minimum and maximum entries where we want them. Indeed, the top copy of MinMax$[n/2]$ has placed the minimum among the first $(n/2)$ entries on the 1st wire, and the bottom copy of MinMax$[n/2]$ has placed the minimum among the last $(n/2)$ entries on the $(n/2+1)$st wire. Thus the minimum among all the entries is on the 1st or $(n/2+1)$st wire, so the comparator connecting these two wires moves the minimum to the 1st output wire. Similar reasoning shows that the maximum is on the $n$th output wire.

(6) (CLRS 27.1-4) **Prove that any sorting network on $n$ inputs has depth at least $\log_2 n$.** Several people proved that a comparison network of depth $d$ has at most $d(n/2)$ comparators (since there can be at most $(n/2)$ simultaneous non-competing comparisons); that a comparison network with $c$ comparators can be converted into a (serial) comparison sort for $n$-element inputs which makes $c$ comparisons in the worst case; that the worst-case number of comparisons of any such sort is $\Omega(n \log n)$; and that therefore $d(n/2) = \Omega(n \log n)$, so $d = \Omega(\log n)$. This is perfectly correct, but it doesn't show the exact result that $d \geq \log_2 n$. (It leaves open the possibility, for example, that $\frac{3}{4}\log_2 n \geq d \geq \frac{1}{4}\log_2 n$.)

To get the exact result, we use Stirling's approximation, $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n(1 + \Theta(1/n))$, in the weaker form $n! \geq (n/e)^n$. Thus

$$(*) \qquad \log_2 n! \geq n\log_2 n - n\log_2 e = \left(\frac{n}{2}\log_2 n\right)\cdot 2\left(1 - \frac{\log_2 e}{\log_2 n}\right).$$

If $n \geq 2$, then, since $e \leq 4$, we have $n \geq \sqrt{e}$; so $\frac{\log_2 e}{\log_2 n} \leq \frac{1}{2}$ and (*) shows that $\log_2 n! \geq \frac{n}{2}\log_2 n$. If $n = 1$, one verifies easily that $\log_2 n! = 0 = \frac{n}{2}\log_2 n$. Thus, reasoning as in the previous paragraph, we have that $d(n/2) \geq \log_2 n! \geq (n/2)\log_2 n$, hence that $d \geq \log_2 n$, for all $n$.

(7) (CLRS 27.2-2) **Prove that a comparison network with $n$ inputs correctly sorts the input array $\langle n, n-1, \ldots, 2, 1\rangle$ if and only if it correctly sorts the $(n-1)$ zero-one sequences $\langle 1,0,\ldots,0\rangle$, $\langle 1,1,0,\ldots,0\rangle$, $\langle 1,\ldots,1,0\rangle$ (i.e., the sequences consisting of $i$ 1's followed by $(n-i)$ 0's, for $i = 1,2,\ldots,n-1$).** The problem as originally written here incorrectly included an $n$th zero-one sequence, $\langle 1,\ldots,1\rangle$, but, since every comparison network sorts this sequence correctly (the one possible output is the correct one!), including or excluding it makes no difference.

For any $1 \leq i < n$, consider the monotonically increasing function $f_i$ given by

$$f_i(x) = \begin{cases} 0, & x \leq n-i \\ 1, & x > n-i. \end{cases}$$

By Lemma 27.1 (which says that, if we transform the input wires by a monotonically increasing function – such as $f_i$ – then we transform the output wires in the same way), if $\langle n, \ldots, n-i+1, n-i, \ldots, 1\rangle$ is sorted correctly, then so is $\langle f_i(n), \ldots, f_i(n-i+1), f_i(n-i), \ldots, f_i(1)\rangle = \langle 1, \ldots, 1, 0, \ldots, 0\rangle$ (a sequence with $i$ 1's, followed by $(n-i)$ 0's.) This is the 'only if' half of the 'if and only if' statement.

Now we consider the 'if' half. By contraposition, it suffices to prove: If $\langle n, n-1, \ldots, 1, 0\rangle$ is incorrectly sorted, then so is one of the zero-one sequences mentioned above. Indeed, if $\langle n, n-1, \ldots, 1\rangle$ is incorrectly sorted, then there are $1 \le i < j \le n$ such that the output wire carrying $j$ is above the output wire carrying $i$. Now consider transforming the input by $f_{n-i}$. The input becomes one of the zero-one sequences in question, whereas the output now has $f_{n-i}(j) = 1$ (since $j > i = n-(n-i)$) on a wire above $f_{n-i}(i) = 0$ (since $i \le i = n-(n-i)$). However, this is incorrectly sorted, as desired.

(8) (CLRS 27.5-3) **Suppose that we have an array of length $2n$ which we wish to partition into its $n$ smallest and $n$ largest entries. Prove that we can do this in constant additional time after first sorting the first $n$, and the last $n$, entries.** We claim that a comparison network which has the desired effect on pairs of sorted zero-one sequences, has the desired effect for *any* pairs of sorted sequences. As usual, it suffices to prove the contrapositive. Thus, suppose that our comparison network does not correctly merge $\langle a_1, \ldots, a_{n/2}; b_1, \ldots, b_{n/2}\rangle$ (where $\langle a_1, \ldots, a_{n/2}\rangle$ and $\langle b_1, \ldots, b_{n/2}\rangle$ are sorted). Then there are entries $r < s$ from the inptu such that the output wire carrying $s$ is above the output wire carrying $r$. (This solution set originally, incorrectly, stated that $r$ and $s$ must come from different halves of the input. In fact, consider the four-input, four-output comparison network with a single comparator connecting the second and fourth wires. On input $\langle 1, 4; 2, 3\rangle$, this network produces output $\langle 1, 3, 2, 4\rangle$. Here, it is *not* the case that the entries $r$ and $s$ above may be taken to have come from different halves of the input.) In this case, consider the monotonically increasing function $f$ given by

$$f(x) = \begin{cases} 0, & x \le (r+s)/2 \\ 1, & x > (r+s)/2. \end{cases}$$

The input $\langle f(a_1), \ldots, f(a_n); f(b_1), \ldots, f(b_n)\rangle$ consists of a pair of sorted zero-one sequences (since monotonically increasing functions preserve the property of being sorted), but the output wire carrying $f(s) = 1$ is above the output wire carrying $f(r) = 0$, so that the sequences have not been merged correctly.

Thus it suffices to find a comparison network which works for zero-one sequences. However, we know from class that, on an input consisting of two sorted zero-one sequences, HALF-CLEANER$'[n]$ (which is illustrated for $n = 8$ as figure 27.10 in the text) produces an output in which the first $(n/2)$ elements are smaller than the last $(n/2)$. Thus this is just the comparison network we need.

# Math 416 Homework 9

Last updated 21 November, 2005

Due 30 November, 2005

(1) (CLRS 27.3-6) **Prove that a comparison network that can sort any bitonic zero-one sequence can sort any bitonic sequence.** As usual, we proceed by contraposition. Suppose that we have a comparison network which, on input some bitonic sequence $a$, produces an incorrectly sorted output. Then there are entries $a_i < a_j$ of the input sequence such that $a_j$ appears on an output wire above $a_i$. Consider the monotone increasing 0-1 function $f$ given by

$$f(x) = \begin{cases} 0, & x \leq (a_i + a_j)/2 \\ 1, & x > (a_i + a_j)/2. \end{cases}$$

Since monotone increasing functions carry increasing sequences to increasing sequences and decreasing sequences to decreasing sequences, they also carry bitonic sequences to bitonic sequences. (One can see this by breaking the bitonic sequence into two pieces, one of which is increasing and one of which is decreasing, and transforming them individually.) In particular, on input the 0-1 sequence $f(a)$ (the sequence whose $i$th entry is $f(a_i)$), the comparison network produces output with $f(a_j) = 1$ on an output wire above $f(a_i) = 0$.

Incidentally, many people claimed that a monotone increasing sequence transforms an incorrectly sorted sequence into another incorrectly sorted sequence. This is not quite true. For example, consider the comparison network with two wires and no comparators. This sorts the input $\langle 1, 0\rangle$ incorrectly, but, if $f$ is the monotone increasing function which is 0 for every input, then the comparison network sorts the input $\langle f(1), f(0)\rangle = \langle 0, 0\rangle$ correctly.

(2) (CLRS 27.4-1) **Prove that a comparison network that can merge any two sorted (in increasing order) zero-one sequences into a single sorted sequence can merge any two sorted (in increasing order) sequences into a single sorted sequence.** As in Problem 1, we argue by contraposition. Suppose that $\langle a; b\rangle$ is an input consisting of two sorted (in increasing order) sequences $a$ and $b$, and that we have a comparison network which does not sort this input correctly. Then there are entries $c_i < c_j$ of these two sequences such that $c_j$ appears on an output wire above $c_i$. (Although $c_i$ and $c_j$ cannot both come from $a$, they might both come from $b$, or one might come from $a$ and one from $b$. However, this doesn't really matter.) Consider the monotone increasing 0-1 function

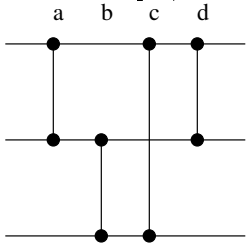
$$f(x) = \begin{cases} 0, & x \leq (c_i + c_j)/2 \\ 1, & x > (c_i + c_j)/2. \end{cases}$$

Then $f(a)$ and $f(b)$ (defined as in Problem 1) are still monotone increasing sequences (i.e., sorted sequences), but, on input the 0-1 sequence $\langle f(a); f(b)\rangle$, our comparison network outputs $f(c_j) = 1$ on an output wire above $f(c_i) = 0$.

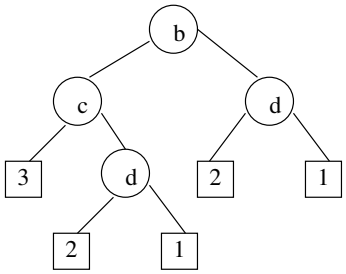
(3) (CLRS 27.4-3) **Show that any network that can merge a single entry with a sorted sequence of length $(n-1)$ into a single sorted**

**sequence has depth at least** $\log_2 n$**.** It is tempting to reason as follows: "We are trying to determine the proper position of the 'extra' element. Initially, there are $n$ possibilities for this position. Each comparator can cut the number of possibilities at most in half, so we need at least $\log_2 n$ comparators." There are two problems with this: First, that it confuses the depth of the network with the number of comparators. This problem doesn't seem so serious, since it seems in some informal sense that, because the comparators "at a given depth" are "non-competing", only one of them can involve the 'extra' element, hence provide useful information. Second, that we can get lucky with a comparison. For example, if an initial comparator connecting the penultimate and ultimate wires shows that the input on the ultimate wire (the 'extra' element) is bigger than the input on the penultimate wire (the last entry of the sorted array), then we have merged the 'extra' element in depth 1. What we need to show is that we don't have this kind of luck on *all* inputs.

We do this using a binary tree which behaves like a decision tree. Consider tracing the movement of the 'extra' element through our comparison network. Each time it passes through a comparator, it moves to either the top or the bottom of that comparator. The non-leaf nodes of our tree will be labelled by the comparators through which the extra element might pass. The left child of a node will correspond to the 'extra' element moving to the bottom of the comparator, and the right child to the 'extra' element moving to the top. The leaf nodes are labelled by the wire on which the 'extra' element belongs. For example, the comparison network




has the tree




.

Notice that the height of a reachable leaf is no more than the depth of the network. Since there are at least $n$ reachable leaves (the 'extra' element has to be able to be merged into any position), the height is at least $\log_2 n$. Thus the depth is also at least $\log_2 n$.

# Math 416 Homework 10 Solutions

Due 7 December, 2005

(1) **Consider the problem of evaluating a polynomial $A(x)$ at a point $x_0$. There are a polynomial $Q(x)$ and a number $r$ such that $A(x) = (x - x_0)Q(x) + r$, and clearly $A(x_0) = r$. Given a degree-bound $n$ for $A(x)$ and the coefficient vector $\langle a_n, \ldots, a_1, a_0 \rangle$, write an algorithm to compute $r$ and the coefficient vector of $Q(x)$ in time $\Theta(n)$.** Note that, since $A(x)$ has degree-bound $n$, dividing by a polynomial $x - x_0$ of degree-bound 1 gives a quotient of degree-bound $n - 1$. Suppose that the quotient is $Q(x) = \sum_{i=0}^{n-1} q_i x^i$. Then multiplying by $x - x_0$ and collecting like terms gives

$$(x - x_0)Q(x) = \sum_{i=0}^{n-1} q_i x^{i+1} + \sum_{i=0}^{n-1} (-q_i x_0) x^i$$
$$= q_{n-1} x^n + \sum_{i=1}^{n-1} (q_{i-1} - q_i x_0) x^i + (-q_0 x_0).$$

If this is to equal $A(x) - r$, which has coefficient vector $\langle a_n, \ldots, a_1, a_0 + r \rangle$, then the coefficients must be equal, so $q_{n-1} = a_n$, $q_{i-1} - q_i x_0 = a_i$ and $-q_0 x_0 = a_0 - r$. We will assemble the quotient in the usual way, starting with the most significant term and working towards the least significant.

PolyQuotient$(a, n, x_0)$
1 create vector $q$
2 $q_{n-1} \leftarrow a_n$
3 **for** $i = n - 1$ downto 1
4 $\quad q_{i-1} \leftarrow a_i + q_i x_0$
5 **endfor**
6 $r \leftarrow a_0 + q_0 x_0$
7 **return** $\langle q, r \rangle$

The reader should check that $r$, together with the polynomial $Q(x)$ whose coefficient vector is $q$, satisfy the equations above, hence are a correct solution. Since the **for** loop executes $n - 1$ times, and each instruction takes $\Theta(1)$ time, PolyQuotient has runing time $\Theta(n)$.

(2) **Suppose that we have $(n + 1)$ points $\langle x_i, y_i \rangle$, $i = 0, \ldots, n$, where $x_i \neq x_j$ if $i \neq j$. Show that the coefficient vector $\langle a_n, \ldots, a_1, a_0 \rangle$ of**

$$A(x) = \sum_{i=0}^{n} y_i \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j}$$

**can be computed in time $\Theta(n^2)$.** The bottleneck is the computation of the products $\prod_{j \neq i} (x - x_j)$, each of which (at least according to the natural approach) already takes time $\Theta(n^2)$. Notice, however, that, if we compute the full product (with no omitted terms) once, then we can recover the product with any one term omitted by a single division, which we can do quickly by Problem 1. (The reader should verify for itself that we cannot compute the denominators using this trick.)

Note that, if we have so far computed the product $P(x) = p_{j-1} x^{j-1} + \ldots + p_1 x + p_0$ up to just before the $j$th term, then, as in Problem 1, the

product including that term is

$$P(x)(x - x_j) = p_{j-1}x^j + \sum_{i=1}^{j-1}(p_{i-1} - p_i x_j)x^i + (-p_0 x_j).$$

Thus we can compute the big product as follows. The argument here is the array $\mathcal{P}[0 \ldots n]$ whose $i$th entry is $\langle x_i, y_i \rangle$; so $\mathcal{P}[i][1] = x_i$ and $\mathcal{P}[i][2] = y_i$.

BIGPROD($\mathcal{P}$)
```
 1 create vector p
 2 p_1 ← 1
 3 for j = 0 to n − 1
 4       p_j ← p_{j−1}
 5       for i = j − 1 downto 1
 6             p_i ← p_{i−1} − p_i P[j][1]
 7       endfor
 8       p_0 ← −p_0 P[j][1]
 9       return p
10 endfor
```
The reader should check that "$p$ is the coefficient vector of $\prod_{j'=0}^{j-1}(x - x_j)$" is a loop invariant for the outer **for** loop. On **termination**, this says that $p$ is the coefficient vector of the big product that we want. Since the inner **for** loop executes $O(n)$ times for each execution of the outer **for** loop; the outer **for** loop executes $n$ times; and every instruction takes time $\Theta(1)$, we have that BIGPRODUCT has running time $O(n^2)$. (In fact, it's easy to show that it actually has running time $\Theta(n^2)$, but we don't need to do so here.)

Even though the algorithm below packs a notational punch, it's just carrying out the straightforward idea of how to build up $A(x)$ in $\Theta(n^2)$ additional time.

LAGRANGE($\mathcal{P}$)
```
 1 n ← length[P] − 1
 2 p ← BIGPRODUCT(P)
 3 create vector a
 4 for i = 1 to n
 5       p' ← POLYQUOTIENT(p, P[i][1])[1]
 6       for j = 0 to n − 1
 7             c ← 1
 8             if j ≠ i
 9                   c ← c(x_i − x_j)
10             endif
11       endfor
12       for j = 0 to n − 1
13             a_j ← a_j + P[i][2]p'_j/c
14       endfor
15 endfor
16 return a
```
(POLYQUOTIENT is the algorithm described in Problem 1. Note that line 5 is just assigning to $p'$ the coefficient vector of the quotient of the big product computed above by $x - \mathcal{P}[i][1]$, i.e., $x - x_i$. The extra '[1]' on the end is because we are throwing away the term $r$ which POLYQUOTIENT

also produces.) The reader should check that "$a$ is the coefficient vector of $\sum_{i'=0}^{i-1} y_{i'} \prod_{j' \neq i'} \frac{x - x_{j'}}{x_{i'} - x_{j'}}$" is a loop invariant for the outer **for** loop. On **termination**, this says that $a$ holds the coefficient vector of the desired polynomial. Both of the inner **for** loops execute $n$ times for every execution of the outer **for** loop; the outer **for** loop executes $n$ times; and, except for the single call to BIGPRODUCT (which runs in time $O(n^2)$), every instruction takes time $\Theta(1)$. Thus the algorithm has running time $\Theta(n^2)$.

# Solution Manual for: Introduction to ALGORITHMS (Second Edition) by T. Cormen, C. Leiserson, and R. Rivest

John L. Weatherwax*

December 17, 2013

**Acknowledgments**

Special thanks to Avi Cohenal for finding typos in these notes. All comments were (and are) much appreciated.

*wax@alum.mit.edu

# Chapter 1 (The Role of Algorithms in Computing)

## 1.1 (Algorithms)

### Exercise 1.1-1 (sorting, optimally multiply matrices, and convex hulls)

Sorting is done in all sorts of computational problems. It is especially helpful with regard to keeping data in a understood ordering so that other algorithms can then work easily and efficiently on the underlying sorted items. One such example of such an algorithm is searching for a specific key in a sequence of elements. When the elements are sorted searching can be done more efficiently.

Selecting the optimal order to multiply matrices can occur in programs/algorithms that update their "state" through linear transformations. When this is the case, and the transformations can be cashed, in other words they don't need to be performed immediately, then computing an optimal ordering in which to calculate the individual matrix products could radically reduce the total computational time.

Finding the convex hull occurs in many graphics programs where the convex hull finding algorithm needs to determine the largest "box" required to contain all the given data points.

### Exercise 1.1-2 (measures of efficiency)

Other common measures of efficiency used to compare algorithms could be anything that might be constrained in a real world setting. Examples of this are memory constraints both disk and random access memory (RAM), the number of memory accesses, determinism as opposed to a randomize algorithm, number of files created, number of sockets opened, number of Internet connections established etc.

### Exercise 1.1-3 (an example data structure)

A common data structure often used is a linked list. Such a data structure can easily insert items into any location within the data structure once the desire insertion point is known. A linked list structure cannot locate new elements or locations quickly since it must effectively look at each element one at a time until the desired one is found.

**Exercise 1.1-4 (shortest-path v.s. traveling-salesman problems)**

In the shortest-path problem the path through the network is often only one way. It is not required to end up at the starting location, but just to end at the last destination desired. In the traveling-salesman problem the algorithm must start and *end* at the *same* location while visiting all other destinations en-route. This requirement that we start and end at the same location while visiting all intermediate locations makes the problem more difficult.

**Exercise 1.1-5 (when only the optimal will do)**

There are relatively few situations in real life where only the optimal will do. This is because normally in formulating a physical problem into a framework that an algorithm can solve involves approximations and simplifications and using an approximate algorithm (assuming that it is not way off of optimal) does not introduce errors greater than have already been introduced in the approximations up to that point.

# Chapter 28 (Matrix Operations)

## 28.1 (Properties of Matrices)

### Exercise 28.1-1 (elementary properties of transposes)

These two facts are a simple consequences of the definition of a transpose: the $(i,j)$th element in $M^T$ is the $(j,i)$th element in $M$. For $A+B$ we have that the $(i,j)$th element in $(A+B)^T$ is the $(j,i)$th element in $A+B$, which is the sum of the $(j,i)$th elements in $A$ and $B$ individually, which is also the sum of the $(i,j)$th elements in $A^T$ and $B^T$ individually. So we have that the $(i,j)$th element in $(A+B)^T$ is the same as the sum of the $(i,j)$th element from $A^T$ and $B^T$. Since $A$ and $B$ are symmetric we have that

$$(A+B)^T = A^T + B^T = A + B$$

and the matrix $A+B$ is symmetric. The proof for the difference of $A$ and $B$ is done in the same way.

### Exercise 28.1-2 (transpose of a product)

To prove that $(AB)^T = B^T A^T$ we begin by looking at the components of the product $AB$. For simplicity assume $A$ and $B$ are $n$ by $n$. Then the $(i,j)$th entry of $AB$ is given by

$$(AB)_{i,j} = \sum_{k=1}^{n} A_{i,k} B_{k,j} \,.$$

Then the transpose of $AB$ has components given by

$$((AB)^T)_{i,j} = (AB)_{j,i} = \sum_{k=1}^{n} A_{j,k} B_{k,i} \,.$$

Note that the components of $A$ in the above can be expressed in term of $A$'s transpose since $A_{j,k} = (A^T)_{k,j}$. The same can be said for $B$ and when this substitution is done we have

$$((AB)^T)_{i,j} = (AB)_{j,i} = \sum_{k=1}^{n} (A^T)_{j,k} (B^T)_{i,k} = \sum_{k=1}^{n} (B^T)_{i,k} (A^T)_{j,k} \,.$$

Where in the first summation we have replace $A_{j,k}$ with $(A^T)_{k,j}$ (similarly for $B$) and in the second summation we just placed the term $B^T$ before the term involving $A^T$. The above can be recognized as the $(i,j)$th element of the product $B^T A^T$ as expected.

Using the fact that $(AB)^T = B^T A^T$ (proven above), and that $(A^T)^T = A$, for the product $A^T A$ we have that

$$(A^T A)^T = (A)^T (A^T)^T = A^T A \,.$$

thus since $A^T A$ when transposed equals itself we have that it is a symmetric matrix as requested.

**Exercise 28.1-3 (uniqueness of a matrix inverse)**

If we assume (to reach a contradiction) that *both* $B$ and $C$ are inverses of $A$ then we must have that

$$\begin{aligned} AB &= I \quad \text{and} \quad BA = I \\ AC &= I \quad \text{and} \quad CA = I\,, \end{aligned}$$

Multiplying the equation $AB = I$ on the left by $C$ gives $CAB = C$. Using the fact that $CA = I$, the above simplifies to $B = C$, showing that the inverse of a matrix must be unique.

**Exercise 28.1-4 (triangular matrices)**

We will prove that the product of two lower triangular matrices is lower triangular by induction. We begin with $n = 2$ for which we have

$$\begin{bmatrix} l_{11} & 0 \\ l_{21} & l_{22} \end{bmatrix} \begin{bmatrix} m_{11} & 0 \\ m_{21} & m_{22} \end{bmatrix} = \begin{bmatrix} l_{11}m_{11} & 0 \\ l_{21}m_{11} + l_{22}m_{21} & l_{22}m_{22} \end{bmatrix}$$

which is lower triangular. Assume the product of two lower triangular matrices of size $\hat{n} \leq n$ is also lower triangular and consider two lower triangular matrices of size $n+1$. Performing a "bordered" block partitioning of the two lower triangular matrices we have that

$$L_{n+1} = \begin{bmatrix} L_n & 0 \\ l^T & l_{n+1,n+1} \end{bmatrix} \quad \text{and} \quad M_{n+1} = \begin{bmatrix} M_n & 0 \\ m^T & m_{n+1,n+1} \end{bmatrix}$$

where the single subscripts denote the order of the matrices. With bordered block decomposition of our two individual matrices we have a product given by

$$L_{n+1}M_{n+1} = \begin{bmatrix} L_nM_n & 0 \\ l^TM_n + l_{n+1,n+1}m^T & l_{n+1,n+1}m_{n_1,n+1} \end{bmatrix}.$$

Since by the induction hypotheses the product $L_nM_n$ is lower triangular we see that our product $L_{n+1}M_{n+1}$ is lower triangular.

The fact that the determinant of a triangular matrix is equal to the product of the diagonal elements, can easily be proved by induction. Lets assume without loss of generality that our system is *lower* triangular (upper triangular systems are transposes of lower triangular systems) and let $n = 1$ then $|G| = g_{11}$ trivially. Now assume that for a triangular system of size $n \times n$ that the determinant is given by the product of its $n$ diagonal elements and consider a matrix $\tilde{G}$ of size $(n+1) \times (n+1)$ partitioned into a leading matrix $G_{11}$ of size $n \times n$.

$$G = \begin{bmatrix} G_{11} & 0 \\ h^T & g_{n+1,n+1} \end{bmatrix}.$$

Now by expanding the determinant of $G$ along its last column we see that

$$|G| = g_{n+1,n+1}|G_{11}| = g_{n+1,n+1}\prod_{i=1}^{n} g_{ii} = \prod_{i=1}^{n+1} g_{ii}\,,$$

proving by induction that the determinant of a triangular matrix is equal to the product of its diagonal elements.

We will prove that the inverse of a lower triangular matrix $L$ (if it exists) is lower triangular by induction. We assume that we are given an $L$ that is non-singular and lower triangular. We want to prove that $L^{-1}$ is lower triangular. We will do this by using induction on $n$ the dimension of $L$. For $n = 1$ $L$ is a scalar and $L^{-1}$ is also a scalar. Trivially both are lower triangular. Now assume that if $L$ is non-singular and lower triangular of size $n \times n$, then $L^{-1}$ has the same property. Let $L$ be a matrix of size $(n+1) \times (n+1)$ and partition $L$ as follows

$$L = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix}.$$

Where $L_{11}$ and $L_{22}$ are both lower triangular matrices of sizes less than $n \times n$, so that we can apply the induction hypothesis. Let $M = L^{-1}$ and partition $M$ con formally i.e.

$$M = \begin{bmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{bmatrix}.$$

We want to show that $M_{12}$ must be zero. Now since $ML = I$ by multiplying the matrices above out we obtain

$$\begin{aligned} LM &= \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{bmatrix} \\ &= \begin{bmatrix} L_{11}M_{11} & L_{11}M_{12} \\ L_{21}M_{11} + L_{22}M_{21} & L_{21}M_{12} + L_{22}M_{22} \end{bmatrix} = \begin{bmatrix} I & 0 \\ 0 & I \end{bmatrix} \end{aligned}$$

Equating block components gives

$$\begin{aligned} L_{11}M_{11} &= I \\ L_{11}M_{12} &= 0 \\ L_{21}M_{11} + L_{22}M_{21} &= 0 \\ L_{21}M_{12} + L_{22}M_{22} &= I\,. \end{aligned}$$

By the induction hypothesis both $L_{11}$ and $L_{22}$ are invertible. Thus the equation $L_{11}M_{11} = I$ gives $M_{11} = L_{11}^{-1}$, and the equation $L_{11}M_{12} = 0$ gives $M_{12} = 0$. With these two conditions the equation $L_{21}M_{12} + L_{22}M_{22} = I$ becomes $L_{22}M_{22} = I$. Since $L_{22}$ is invertible we compute that $M_{22} = L_{22}^{-1}$. As both $L_{11}$ and $L_{22}$ are lower triangular of size less than $n \times n$ by the induction hypothesis their inverse are lower triangular and we see that $M$ itself is then lower triangular since

$$M = \begin{bmatrix} L_{11}^{-1} & 0 \\ M_{21} & L_{22}^{-1} \end{bmatrix}.$$

Thus by the principle of induction we have shown that the inverse of a lower triangular matrix is lower triangular.

**Exercise 28.1-5 (permutation matrices)**

From the definition of a permutation matrix (a matrix with only a single one in each row/column and all other elements zero) the product $PA$ can be computed by recognizing

that each row of $P$ when multiplied into $A$ will select a single row of $A$ and thus produces a permutation of the rows of $A$. In the same way the product $AP$ will produce a permutation of the columns of $A$. If we have two permutation matrices $P_1$ and $P_2$, then $P_1$ acting on $P_2$ will result in a permutation of the rows of $P_2$. This result is a further permutation matrix, since it is the combined result of two permutations, that from $P_2$ and then that from $P_1$. If $P$ is a permutation, then it represents a permutation of the rows of the identity matrix. The matrix $\hat{P}$ representing the permutation which reorders the rows of $P$ back into the identity matrix would be the inverse of $P$ and from this argument we see that $P$ is invertible and has an inverse that is another permutation matrix. The fact that $P^{-1} = P^T$ can be recognized by considering the product of $P$ with $P^T$. When row $i$ of $P$ is multiplied by any column of $P^T$ not equal to $i$ the result will be zero since the location of the one in each vector won't agree in the location of their index. However, when row $i$ of $P$ is multiplied by column $i$ the result will be one. Thus we see by looking at the components of $PP^T$ that $PP^T = I$ and $P^T$ is the inverse of $P$.

**Exercise 28.1-6 (Gauss transformations)**

Lets assume (without loss of generality) that $j > i$, then we will let $M$ be the elementary matrix (of type 1) that produces $A'$ from $A$ i.e. it adds row $i$ to $j$ and the sum replaces row $j$. Then since $AB = I$ multiplying this system by $M$ on the left we obtain (since $MA = A'$) that $A'B = M$. From this we see that by multiplying this expression above by $M^{-1}$ on the left we obtain $A'BM^{-1} = I$, showing that the inverse of $A'$ is $BM^{-1}$. Now the matrix $M$ is like the identity by with an additional one at position $(j, i)$ rather than a zero there. Specifically we have

$$M = \begin{bmatrix} 1 & & & & & & \\ & 1 & & & & & \\ & & \ddots & & & & \\ & & & 1 & & & \\ & & & & 1 & & \\ & & 1 & & & \ddots & \\ & & & & & & 1 \end{bmatrix}.$$

Where the one in the above matrix is at location $(j, i)$. In this case the inverse of $M$ is then easy to compute; it is the identity matrix with a *minus* one at position $(j, i)$, specifically we have

$$M^{-1} = \begin{bmatrix} 1 & & & & & & \\ & 1 & & & & & \\ & & \ddots & & & & \\ & & & 1 & & & \\ & & & & 1 & & \\ & & -1 & & & \ddots & \\ & & & & & & 1 \end{bmatrix}.$$

Now multiplying $B$ by this matrix on the left will operate on the columns of $B$, thus $B' = BM^{-1}$ is the same matrix as $B$ but with the $j$th and the negative of the $i$th column added

together and placed in column $j$. That is we are subtracting the $i$th column from the $j$th column and placing it in column $j$.

**Exercise 28.1-7 (the realness of $A^{-1}$)**

Lets begin by assuming that every entry of $A$ is real and then show that every entry of $A^{-1}$ is real. The easiest way to see that all entries of $A^{-1}$ must be real is to recall the adjoint theorem from linear algebra. The adjoint theorem gives the inverse of a matrix in terms of the adjoint of that matrix. Specifically, the adjoint theorem states that

$$A^{-1} = \frac{1}{\det(A)} C^T ,$$

where $C$ is the matrix of *cofactors* of $A$. This cofactor matrix $C$ is the matrix such that its $(i,j)$ element is given by the cofactor of the element $a_{ij}$ or

$$C_{i,j} = (-1)^{i+j} \det(A_{[ij]}) .$$

Where $A_{[ij]}$ is the $ij$th minor of the matrix $A$, i.e. the submatrix that is produced from $A$ by deleting its $i$th row and its $j$th column. Because the determinants of the minors of $A$ only involve additions and multiplications, if $A$ has only real elements then all cofactors and determinants of $A$ must be real. By the adjoint theorem above, $A^{-1}$ can have only real elements. The other direction, where we argue that if $A^{-1}$ has only real elements then $A$ must have only real elements can be seen by applying the above arguments to the matrix $A^{-1}$.

**Exercise 28.1-8 (symmetry of the inverse)**

Let $A$ be symmetric and invertible. Then by the definition of the inverse, $A^{-1}$ satisfies $AA^{-1} = I$. Now taking the transpose of both sides and remembering that the transpose of a product is the product of the transposes but in the opposite order we have $(A^{-1})^T A^T = I^T$ which simplifies to $(A^{-1})^T A = I$, since both $A$ and $I$ are symmetric. By multiplying both sides on the left by $A^{-1}$ we have that

$$(A^{-1})^T = A^{-1}$$

showing that $A^{-1}$ is symmetric.

That the product $BAB^T$ is symmetric is just an exercise in taking transposes. We have

$$(BAB^T)^T = (B^T)^T A^T B^T = BAB^T ,$$

and this product of matrices is symmetric.

**Exercise 28.1-9 (full column rank)**

Lets assume that $A$ has full column rank and that $Ax = 0$. Note that $Ax = 0$ is equivalent to the statement that a linear combination of the columns of $A$ sums to zero. Specifically if $v_i$ represents the $i$th column of $A$ then $Ax = 0$ is equivalent to

$$\sum_{i=1}^{n} x_i v_i = 0 \,.$$

Since $A$ is full column rank its columns are linearly independent (this is the definition of full column rank). Because of this fact, the only way these columns can sum to zero is if $x = 0$ and we have proven one direction. Now lets assume that $Ax = 0$ implies that $x = 0$. This statement is equivalent to the fact that the columns of $A$ are linearly independent which again implies that $A$ is of full column rank.

**Exercise 28.1-10 (rank inequalities)**

To show this we will first show that

$$\text{rank}(AB) \leq \text{rank}(A) \,.$$

and then use this result to show that

$$\text{rank}(AB) \leq \text{rank}(B) \,.$$

When these two results are combined the rank of the product $AB$ must be *less* than the smaller of the two rank$(A)$ and rank$(B)$ giving the requested inequality of

$$\text{rank}(AB) \leq \min(\text{rank}(A), \text{rank}(B)) \,.$$

To show that rank$(AB) \leq$ rank$(A)$, we first note that every vector in the column space of $AB$ is in the column space of $A$. This is true since $AB$ is the action of the matrix $A$ on every column of the matrix $B$, and the action of a matrix $(A)$ on a vector is a linear superposition of the columns of that matrix $(A)$. Thus the number of linearly independent columns of the *product* matrix $AB$ cannot be greater than that of the number of linear independent columns in the multiplying matrix $(A)$ and we recognize the desired result of

$$\text{rank}(AB) \leq \text{rank}(A) \,.$$

To show the second inequality rank$(AB) \leq$ rank$(B)$, we apply the expression proved above on the matrix $(AB)^T$. Since this matrix equals $B^T A^T$ we have that

$$\text{rank}((AB)^T) = \text{rank}(B^T A^T) \leq \text{rank}(B^T) = \text{rank}(B) \,.$$

But since rank$((AB)^T) =$ rank$(AB)$ replacing the first term in the above gives

$$\text{rank}(AB) \leq \text{rank}(B) \,,$$

showing the second of the desired inequalities. If $A$ or $B$ is invertible, it must be *square* and its rank is equal to its number of columns (equivalently the number of rows). With out loss of generality assume that $A$ is invertible. By the arguments above when we multiply $B$, the dimension of the column space of $AB$ cannot change from the dimension of the column space of $B$. Thus we have that

$$\text{rank}(AB) = \text{rank}(B)\,.$$

The same argument applied to $(AB)^T$ (as outlined in the inequality proof above) can be used to show that if $B$ is invertible that

$$\text{rank}(AB) = \text{rank}(A)\,.$$

**Exercise 28.1-11 (the determinant of the Vandermonde matrix)**

Following the hint given, we will multiply column $i$ by $-x_0$ and add it to column $i+1$ for $i = n-1, n-2, \cdots, 1$. This action will not change the value of the determinant but will make it simpler to evaluate as we will soon see. We begin by first multiplying column $n-1$ by $-x_0$ and adding to column $n$. We find that

$$\begin{aligned}\det(V(x_0, x_1, x_2, \cdots, x_{n-3}, x_{n-2}, x_{n-1})) &= \begin{vmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-2} & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-2} & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-2} & x_2^{n-1} \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-2} & x_{n-1}^{n-1} \end{vmatrix} \\ &= \begin{vmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-2} & 0 \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-2} & (x_1 - x_0)x_1^{n-2} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-2} & (x_2 - x_0)x_2^{n-2} \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-2} & (x_{n-1} - x_0)x_{n-1}^{n-2} \end{vmatrix}.\end{aligned}$$

Where we see that this action has introduced a zero in the $(1, n)$ position. Continuing, we now multiply column $n-2$ by $-x_0$ and add it to column $n-1$. We find that the above determinant now becomes

$$\begin{vmatrix} 1 & x_0 & x_0^2 & \cdots & 0 & 0 \\ 1 & x_1 & x_1^2 & \cdots & (x_1 - x_0)x_1^{n-3} & (x_1 - x_0)x_1^{n-2} \\ 1 & x_2 & x_2^2 & \cdots & (x_2 - x_0)x_2^{n-3} & (x_2 - x_0)x_2^{n-2} \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & (x_{n-1} - x_0)x_{n-1}^{n-3} & (x_{n-1} - x_0)x_{n-1}^{n-2} \end{vmatrix}.$$

Where we see that this action has introduced another zero this time at the $(1, n-1)$ position. Continuing in this manner we will obtain the following determinant where the first row has

only one non-zero element

$$\begin{vmatrix} 1 & 0 & 0 & 0 & \cdots & 0 & 0 \\ 1 & x_1 - x_0 & (x_1 - x_0)x_1 & (x_1 - x_0)x_1^2 & \cdots & (x_1 - x_0)x_1^{n-3} & (x_1 - x_0)x_1^{n-2} \\ 1 & x_2 - x_0 & (x_2 - x_0)x_2 & (x_2 - x_0)x_2^2 & \cdots & (x_2 - x_0)x_2^{n-3} & (x_2 - x_0)x_2^{n-2} \\ \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots \\ 1 & x_{n-1} - x_0 & (x_{n-1} - x_0)x_{n-1} & (x_{n-1} - x_0)x_{n-1}^2 & \cdots & (x_{n-1} - x_0)x_{n-1}^{n-3} & (x_{n-1} - x_0)x_{n-1}^{n-2} \end{vmatrix}.$$

We can expand this determinant about the first row easily since there is only a single nonzero element in this row. After this reduction we see that we can factor $x_1 - x_0$ from the first row of our reduced matrix, factor $x_2 - x_0$ from the second row, $x_3 - x_0$ from the third row, and so on til we get to the $n-1$th row of our reduced matrix where we will factor $x_{n-1} - x_0$. This gives us

$$\det(V(x_0, x_1, x_2, \cdots, x_{n-2}, x_{n-1})) = \prod_{i=1}^{n-1}(x_i - x_0) \times \begin{vmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{n-3} & x_1^{n-2} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-3} & x_2^{n-2} \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-3} & x_{n-1}^{n-2} \end{vmatrix}.$$

Note that the remaining determinant is of size $(n-1) \times (n-1)$ and is of exactly the same form as the previous determinant. Thus using the same manipulations performed earlier we see that

$$\begin{aligned} \det(V) &= \prod_{i=1}^{n-1}(x_i - x_0) \times \det(V(x_1, x_2, \cdots, x_{n-1})) \\ &= \prod_{i=1}^{n-1}(x_i - x_0) \times \prod_{j=2}^{n-1}(x_j - x_1) \times \det(V(x_2, x_3, \cdots, x_{n-1})) \\ &= \prod_{i=1}^{n-1}(x_i - x_0) \times \prod_{j=2}^{n-1}(x_j - x_1) \times \prod_{k=3}^{n-1}(x_k - x_2)\det(V(x_3, \cdots, x_{n-1})) \\ &= \prod_{i=1}^{n-1}(x_i - x_0) \times \prod_{j=2}^{n-1}(x_j - x_1) \times \prod_{k=3}^{n-1}(x_k - x_2) \cdots \prod_{l=n-2}^{n-1}(x_l - x_{n-3}) \times (x_{n-1} - x_{n-2}). \end{aligned}$$

As a bit of shorthand notation the above can be written as

$$\det(V(x_0, x_1, \cdots, x_{n-2}, x_{n-1})) = \prod_{0 \le j < k \le n-1}(x_k - x_j),$$

as claimed in the book.

## 28.2 (Strassen's algorithm for matrix multiplication)

### Exercise 28.2-6 (multiplying complex numbers)

We are given $a$, $b$, $c$, and $d$ and we desire to compute the complex product

$$(a + ib)(c + id) = (ac - bd) + i(ad + bc),$$

using only three real multiplications. To perform this computation define the some intermediate variables $t_1$, $t_2$, and $t_3$ in terms of our inputs as

$$\begin{aligned} t_1 &= ad \\ t_2 &= bc \\ t_3 &= (a+b)(c-d)\,, \end{aligned}$$

which involve only *three* real multiplications. Note that the product obtained by computing $t_3$ is algebraically equivalent to

$$t_3 = (a+b)(c-d) = ac - ad + bc - bd = (ac - bd) + (-ad + bc)\,,$$

where we have grouped specific terms to help direct the manipulations we will perform below. Note also that the sums $t_1 + t_2$ and $t_3 + t_1 - t_2$ are given by

$$\begin{aligned} t_1 + t_2 &= ad + bc \\ t_3 + t_1 - t_2 &= ac - bd\,. \end{aligned}$$

so that our full complex product can be written in terms of these sums as

$$(a+ib)(c+id) = (t_3 + t_1 - t_2) + i(t_1 + t_2)\,.$$

## 28.3 (Solving systems of linear equations)

### Exercise 28.3-1 (an example of forward substitution)

Our problem is to solve

$$\begin{bmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ -6 & 5 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 14 \\ -7 \end{bmatrix},$$

which upon performing forward substitution gives

$$\begin{aligned} x_1 &= 3 \\ x_2 &= 14 - 4x_1 = 14 - 12 = 2 \\ x_3 &= -7 + 6x_1 - 5x_2 = -7 + 18 - 10 = 1\,. \end{aligned}$$

## 28.5 (Symmetric PD matrices and LS approximations)

### Exercise 28.5-1 (diagonal elements of positive-definite matrices)

Since $A$ is positive definite we must have for all non-zero $x$'s the condition $x^T A x > 0$. Let $x = e_i$, a vector of all zeros but with a one in the $i$-th spot. Then $x^T A x = e_i^T A e_i = a_{ii} > 0$, proving that the diagonal elements of a positive definite matrix must be positive.

# Chapter 31 (Number-Theoretic Algorithms)

## 31.1 (Elementary number-theoretic notations)

### Exercise 31.1-1

Following the hint, given a sequence of increasing primes $p_1, p_2, \cdots, p_k$ we can form another prime $p$ by computing "the product plus one" or $p = p_1 p_2 \cdots p_k + 1$. Note that this new number $p$ is prime since none of $p_1, p_2, \cdots, p_k$ divide it. If one did then it would also have to divide the number $p - p_1 p_2 \cdots p_k = 1$ which is impossible. Thus the newly formed number is prime and as we can repeat this procedure an infinite number of times there must be an infinite number of primes.

### Exercise 31.1-2

If $a|b$ then we have that there exists an integer $k_1$ such that $b = k_1 a$. Since $b|c$ we have that there exists an integer $k_2$ such that $c = k_2 b$. Using the first of these two relationships into the second we have

$$c = k_2 b = k_2(k_1 a) = k_1 k_2 a\,,$$

showing that $a|c$.

### Exercise 31.1-3

Assume that $\gcd(k, p) = d \neq 1$. Then $d|k$ and $d|p$. Since $p$ is prime if $d|p$ then $d = 1$ or $d = p$. As we assumed that $d \neq 1$ we must have $d = p$. The fact that $d|k$ means that $|d| \leq |k|$ or $|p| \leq |k|$. This last fact is a contraction to the problem assumption that $k < p$. Thus the assumption that $d \neq 1$ must be false.

### Exercise 31.1-4

Since $\gcd(a, n) = 1$ then by Theorem 31.2 there must exist integers $x$ and $y$ such that $1 = ax + ny$. If we multiply this equation by $b$ we get

$$b = abx + nbx\,.$$

Note that the right-hand-side of the above is divisible by $n$ since it is a linear combination of two things (the terms $abx$ and $nbx$) both of which are divisible by $n$. We know that $abx$ is divisible by $n$ since $ab$ is and $nbx$ is divisible by $n$ since $n$ is a factor. Since the right-hand-side equals $b$ this shows that $b$ is divisible by $n$.

**Exercise 31.1-5**

Using Equation 4 we can write

$$\binom{p}{k} = \frac{p}{k}\binom{p-1}{k-1},$$

which since the right-hand-side is a multiple of $p$ shows that $p$ divides $\binom{p}{k}$ or $p|\binom{p}{k}$. Next note that using the binomial theorem we can write $(a+b)^p$ as

$$(a+b)^p = a^p + b^p + \sum_{k=1}^{p-1} \binom{p}{k} a^k b^{p-k}.$$

Since $p$ divides $\binom{p}{k}$ it divides the summation on the right-hand-side of the above equation. Thus the remainder of $(a+b)^p$ and $a^p + b^p$ when dividing by $p$ is the same or

$$(a+b)^p = a^p + b^p \pmod{p}.$$

**Exercise 31.1-6**

Now the definition of the "mod" operators is that

$$\begin{aligned} x \bmod b &= x - \left\lfloor \frac{x}{b} \right\rfloor b \\ x \bmod a &= x - \left\lfloor \frac{x}{a} \right\rfloor a. \end{aligned} \tag{1}$$

Since $a|b$ there exists an integer $k$ such that $b = ka$. Thus Equation 1 becomes

$$x \bmod b = x - \left\lfloor \frac{x}{b} \right\rfloor ka. \tag{2}$$

Thus taking the "mod" $a$ operator to both sides of Equation 2 we get

$$(x \bmod b) \bmod a = x \bmod a,$$

as we were to show. Next if we assume that $x \equiv y \pmod{b}$ then taking the "mod" $a$ operator to both sides of this expression and using the just derived result we get

$$(x \bmod b) \bmod a = (y \bmod b) \bmod a \quad \text{or} \quad x \bmod a = y \bmod a,$$

as we were to show.

**Exercise 31.1-8**

Recall that Theorem 31.2 states that that $\gcd(a,b)$ is the smallest positive element of the set $\{ax + by : x, y \in \mathbb{Z}\}$. Since this set definition of the greatest common divisor is symmetric in $a$ and $b$ we see that

$$\gcd(a,b) = \gcd(b,a).$$

and because $x$ and $y$ are arbitrary integers that

$$\gcd(a,b) = \gcd(-a,b) = \gcd(|a|,|b|)\,.$$

We also have that $\gcd(a,0)$ is the smallest element in the set $\{ax : x \in \mathbb{Z}\}$ which would be when $x = \pm 1$ such that the product of $ax$ is positive or $|a|$. Next we have that $\gcd(a,ka)$ is the smallest element of the set $\{ax + kay : x,y \in \mathbb{Z}\} = \{a(x+ky) : x,y \in \mathbb{Z}\}$. This smallest element would be when $x + ky = \pm 1$ in order that $a(x+ky) = |a|$.

**Exercise 31.1-9**

Let $d = \gcd(a, \gcd(b,c))$ i.e. the left-hand-side of the equation we wish to show. Then $d|a$ and $d|\gcd(b,c)$. This second statement $d|\gcd(b,c)$ means that $d|b$ and $d|c$. Since $d$ divides $a$ and $b$ by the above this means that $d|\gcd(a,b)$. Thus $d$ is a number that $\gcd(a,b)$ and $c$. Lets show that $d$ is the largest number that divides both $\gcd(a,b)$ and $c$. Then if so we have that $d = \gcd(\gcd(a,b),c)$ and we have shown the desired equivalence. To show this assume that there is another integer $d'$ such that $d' > d$ that divides $\gcd(a,b)$ and $c$. This means that $d'|a$ and $d'|b$, and $d'|c$ so that $d'|\gcd(b,c)$. Thus since $d'$ divides $a$ and $\gcd(b,c)$ it must be smaller than or equal to the greatest common divisor of $a$ and $\gcd(b,c)$ or

$$d' \leq d = \gcd(a, \gcd(b,c))\,,$$

giving a contradiction to the initial assumption that $d' > d$. Thus means $d$ must be the largest integer already and so $d = \gcd(\gcd(a,b),c)$.

## 31.2 (Greatest common divisor)

## Notes on Euclid's algorithm

Recall that $a \bmod b = a - \left\lfloor \frac{a}{b} \right\rfloor b$ i.e. $a \bmod b$ is the remainder when $a$ is divided by $b$. This is the common understanding when $a > b$. We now ask what does this mean if $a < b$? In that case we would have $\left\lfloor \frac{a}{b} \right\rfloor = 0$ since the fraction $\frac{a}{b} < 1$ and the above formula would give

$$a \bmod b = a \quad \text{when} \quad a < b\,. \tag{3}$$

To numerical examples of this are

$$5 \bmod 10 = 5$$
$$2 \bmod 5 = 2\,.$$

The point of these comments is to give some understanding to Euclid's algorithm for finding the greatest common divisor when the inputs are $\text{EUCLID}(a,b)$ where $a < b$ then in this case the first recursive call performed is

$$\text{EUCLID}(a,b) = \text{EUCLID}(b, a \bmod b) = \text{EUCLID}(b,a)\,.$$

From the above we see that now all recursive calls to EUCLID will have the first argument larger than the second argument. In `python`, the greatest common divisor is available in the `fractions` module i.e.

```
import fractions
print fractions.gcd(30,21) # the example from the book gives 3
```

At the time of writing the source code of this function shows that its in fact using Euclid's algorithm for its implementation.

**Exercise 31.2-1**

Let $d = \gcd(a, b)$ then $d$ must have a prime factorization in terms of the same primes as $a$ and $b$ namely

$$d = p_1^{g_1} p_2^{g_2} \cdots p_r^{g_r} .$$

Since $d$ is a divisor of $a$ and $b$ we must have $d|a$ and $d|b$ which means that

$$p_i^{g_i}|a \quad \text{and} \quad p_i^{g_i}|b\,,$$

for $i = 1, 2, \cdots, r$. This in tern means that

$$p_i^{g_i}|p_i^{e_i} \quad \text{and} \quad p_i^{g_i}|p_i^{f_i} .$$

Thus we have that

$$g_i \leq e_i \quad \text{and} \quad g_i \leq f_i\,.$$

For $d$ to be as large as possible (since it is the *greatest* common divisor the powers $g_i$ must be as large as possible such that the above two relations hold or

$$g_i = \min(e_i, f_i)\,,$$

as we were to show.

**Exercise 31.2-2**

See the output in Table 1 where we follow the output format given in this section of the notes where as we move down the table we are moving in increasing stack depth. From that output we see that $\gcd(899, 493) = 29$ and that $29 = 899(-6) + 493(11)$.

**Exercise 31.2-3**

One way to see that these two expressions are equal is to use one step of EUCLID's algorithm to evaluate both of them and see that after the first step both of these algorithms are

| stack level | $a$ | $b$ | $\lfloor a/b \rfloor$ | $d$ | $x$ | $y$ |
|---|---|---|---|---|---|---|
| 0 | 899 | 493 | 1 | 29 | -6 | 11 |
| 1 | 493 | 406 | 1 | 29 | 5 | -6 |
| 2 | 406 | 87 | 4 | 29 | -1 | 5 |
| 3 | 87 | 58 | 1 | 29 | 1 | -1 |
| 4 | 58 | 29 | 2 | 29 | 0 | 1 |
| 5 | 29 | 0 | - | 29 | 1 | 0 |

Table 1: The recursive calls resulting from executing EXTENDED-EUCLID(899,493)

computing the same thing. For example, EUCLID to compute $\gcd(a, n)$ would first call (assuming $n \neq 0$)

$$\text{EUCLID}(n, a \bmod n)\,.$$

The first step of EUCLID's algorithm on the second expression $\gcd(a + kn, n)$ would call

$$\text{EUCLID}(n, (a + kn) \bmod n)\,.$$

Since $(a + kn) \bmod n = a \bmod n$ we have that the two expressions will evaluate to the same number and are therefore the same. If $n = 0$ the we can see that both sides are equal.

**Exercise 31.2-4**

One way to do this would be the following in `python`

```
def iterative_euclid(a,b):
    while b != 0 :
        a, b = b, a % b # swap in place
    return a
```

Here we have assigned $b$ to $a$ and $a \bmod b$ to b in one step using `pythons` tuple assignment.

**Exercise 31.2-5**

For the Fibonacci sequence defined as $F_0 = 0$, $F_1 = 1$, and

$$F_k = F_{k-1} + F_{k-2} \quad \text{for} \quad k \geq 2\,,$$

Then it can be shown that $F_{k+1} \sim \frac{\phi^{k+1}}{\sqrt{5}}$ for large $k$ where $\phi$ is the "golden ratio" defined as

$$\phi = \frac{1 + \sqrt{5}}{2} = 1.61803\,.$$

From Lame's theorem in the book we have less than $k$ recursive calls in EUCLID$(a,b)$ when $F_{k+1} > b$ or using the approximation above for $F_{k+1}$ when

$$\frac{\phi^{k+1}}{\sqrt{5}} > b\,.$$

Solving the above for $k$ gives

$$k > \log_\phi(b) + \log_\phi(\sqrt{5}) - 1\,.$$

Note that we can evaluate the constants in the above using

$$\log_\phi(\sqrt{5}) = \frac{\ln(\sqrt{5})}{\ln(\phi)} = 1.67\,,$$

Thus we conclude that $k > \log_\phi(b) + 0.67$. Thus we have shown that the invocation EUCLID$(a,b)$ makes at most $1 + \log_\phi(b)$ recursive calls.

**Exercise 31.2-7**

One can show that the function returns the same answer independent of the order of the arguments by induction. It is easiest to see how to write

$$\gcd(a_0, a_1, a_2, \cdots, a_n) = a_0x_0 + a_1x_1 + a_2x_2 + \cdots + a_nx_n\,,$$

by looking at an example. Consider

$$\gcd(a_0, a_1, a_2, a_3) = \gcd(a_0, \gcd(a_1, a_2, a_3)) = \gcd(a_0, \gcd(a_1, \gcd(a_2, a_3)))\,.$$

We can now use Theorem 31.2 to write $\gcd(a_2, a_3)$ as $a_2x_2 + a_3x_3$ for two integers $x_2$ and $x_3$. This would give

$$\gcd(a_0, a_1, a_2, a_3) = \gcd(a_0, \gcd(a_1, a_2x_2 + a_3x_3))\,.$$

We can again use use Theorem 31.2 to write $\gcd(a_1, a_2x_2 + a_3x_3)$ as

$$x_1a_1 + y_1(a_2x_2 + a_3x_3) = x_1a_1 + y_1x_2a_2 + y_1x_3a_3\,.$$

for two integers $x_1$ and $y_1$. Thus we have shown that

$$\gcd(a_0, a_1, a_2, a_3) = \gcd(a_0, x_1a_1 + y_1x_2a_2 + y_1x_3a_3)\,.$$

One more application of Theorem 31.2 gives

$$\begin{aligned}\gcd(a_0, a_1, a_2, a_3) &= x_0a_0 + y_2(x_1a_1 + y_1x_2a_2 + y_1x_3a_3) \\ &= x_0a_0 + y_2x_1a_1 + y_1y_2x_2a_2 + y_1y_2x_3a_3\,.\end{aligned}$$

As each coefficient in front of $a_i$ is the product of integers it itself is an integer. This gives the desired representation when $n = 3$. In general, we use the definition of the greatest common divisor for $n > 1$ to "nest" gcd function calls until we get a function call with only two arguments $\gcd(a_{n-1}, a_n)$. We can then use Theorem 31.2 to write this as $a_{n-1}x_{n-1} + a_nx_n$ for two integers $x_{n-1}$ and $x_n$. We can then "unnest" the gcd function calls each time using Theorem 31.2 to get the desired result.

**Exercise 31.2-8 (the least common multiple)**

We can first compute the greatest common divisor of the given $n$ integers

$$d = \gcd(a_1, a_2, \ldots, a_n) .$$

This can be done using the recursive definition of the gcd for more than two arguments given in a previous problem. Then since $d$ is a divisor of each of $a_i$ it is a factor of each of them (by definition). Once we have this number then the least common multiple is given by

$$\operatorname{lcm}(a_1, a_2, \ldots, a_n) = d \left(\frac{a_1}{d}\right) \left(\frac{a_2}{d}\right) \cdots \left(\frac{a_n}{d}\right) .$$

In the right-hand-side of the above expression by grouping $d$ with a given term in parenthesis we have that $d\left(\frac{a_i}{d}\right) = a_i$ and thus the above number is a multiple of $a_i$ for each $i$.

# Appendix C (Counting and Probability)

## Appendix C.1 (Counting)

### a lower bound on the binomial coefficients (book notes page 1097)

The proof of the lower bound on the binomial coefficients given in the book, requires that

$$\frac{n-1}{k-1} \geq \frac{n}{k},$$

for $1 \leq k \leq n$. We can show this expression is true, by starting with it and seeing if we can convert it using reversible transformations to another expression known to be true. If this can be done, since each transformation is reversible, the original expression must also be true. For this expression we can demonstrate it is true by using some simple manipulations. We have

$$\begin{aligned} \frac{n-1}{k-1} &\geq \frac{n}{k} \\ k(n-1) &\geq n(k-1) \\ kn-k &\geq nk-n \\ -k &\geq -n \\ n &\geq k \end{aligned}$$

Which is known to be true. Using the same manipulations we can show show that

$$\frac{n-2}{k-2} \geq \frac{n}{k}.$$

This inductive process can continue, subtracting one each time from the numerator and denominator. As a check we can verify the correctness of the final inequality which is given by

$$\frac{n-k+1}{1} \geq \frac{n}{k}.$$

Using the same approach as above

$$\begin{aligned} \frac{n-k+1}{1} &\geq \frac{n}{k} \\ kn-k^2+k &\geq n \\ n(k-1)-k(k-1) &\geq 0 \\ (k-1)(n-k) &\geq 0 \end{aligned}$$

and the last equation is true. Given this sequence of results we can derive the lower bounds on the binomial coefficients as

$$\begin{aligned}\binom{n}{k} &= \frac{n(n-1)\ldots(n-k+1)}{k(k-1)\ldots 1} \\ &= \left(\frac{n}{k}\right)\left(\frac{n-1}{k-1}\right)\cdots\left(\frac{n-k+1}{1}\right) \\ &\geq \left(\frac{n}{k}\right)\left(\frac{n}{k}\right)\cdots\left(\frac{n}{k}\right) \\ &= \left(\frac{n}{k}\right)^k\end{aligned}$$

**a upper bound on the binomial coefficients (book notes page 1097)**

We have from the definition of the binomial coefficients that

$$\begin{aligned}\binom{n}{k} &= \frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 2\,1} \\ &\leq \frac{n^k}{k!} \\ &\leq \frac{e^k n^k}{k^k} \\ &\leq \left(\frac{en}{k}\right)^k .\end{aligned}$$

Where in the above we have used the $k! \geq \left(\frac{k}{e}\right)^k$ (obtained from Stirling's approximation), in the form

$$\frac{1}{k!} \leq \left(\frac{e}{k}\right)^k$$

to simplify the expression $\frac{n^k}{k!}$ in the above.

**the binomial coefficients and the entropy function (book notes page 1097)**

If we assign $k = \lambda n$, then we have the following

$$\begin{aligned}
\left( \begin{array}{c} n \\ \lambda n \end{array} \right) &\leq \frac{n^n}{(\lambda n)^{\lambda n}(n - \lambda n)^{n-\lambda n}} \\
&= \frac{n^n}{(\lambda n)^{\lambda n}((1-\lambda)n)^{(1-\lambda)n}} \\
&= \left( \frac{n}{(\lambda n)^{\lambda}((1-\lambda)n)^{1-\lambda}} \right)^n \\
&= \left( \frac{1}{\lambda^{\lambda}(1-\lambda)^{1-\lambda}} \right)^n \\
&= \left( \left( \frac{1}{\lambda} \right)^{\lambda} \left( \frac{1}{1-\lambda} \right)^{1-\lambda} \right)^n \\
&\equiv 2^{nH(\lambda)} ,
\end{aligned}$$

where the introduction of the expression $H(\lambda)$ above requires that

$$nH(\lambda) = n \lg( \left( \frac{1}{\lambda} \right)^{\lambda} \left( \frac{1}{1-\lambda} \right)^{1-\lambda} )$$

or

$$H(\lambda) = -\lambda \lg(\lambda) - (1-\lambda) \lg(1-\lambda) .$$

This is the (binary) entropy function. We note before continuing that since $\lambda = n/k$ the above bound can also be written as

$$\left( \begin{array}{c} n \\ k \end{array} \right) \leq 2^{nH(\frac{n}{k})} .$$

**Exercise C.1-1 (counting substrings)**

Assuming $k < n$, then the first substring occupies the locations $1, 2, \ldots, k$, the second substring occupies the locations $2, 3, \ldots, k+1$, and so on. The last possible substring would occupy the locations $n-k+1, n-k, \ldots, n$. Thus we have in total

$$n - k + 1$$

substrings of size $k$ in a string of size $n$.

To calculate how many substrings of size $k$ a string of size $n$ has, we from the above formula that we have $n$ size one substrings ($k = 1$), $n-1$ size two substrings ($k = 2$), $n-2$ size three substrings, etc. Continuing this pattern the total number of substrings is the sum of all these numbers given by

$$\sum_{k=1}^{n} (n - k + 1) = \sum_{l=1}^{n} l = \frac{n(n+1)}{2} .$$

**Exercise C.1-2 (counting Boolean functions)**

Each Boolean function is defined by how it maps its inputs to outputs. For an $n$-input, 1-output Boolean function we have a total of $2^n$ possible inputs, to which we can assign a TRUE or a FALSE output. Thus for each possible unique input specification we have 2 possible output specifications. So we have for the number of possible inputs (counting the number of possible outputs for each input)

$$2 * 2 \cdots 2 * 2 \,.$$

With one factor of 2 for each of the possible input specification. Since there are $2^n$ possible input specifications, this gives a total of

$$2^{2^n} \,,$$

possible Boolean functions with $n$-input variables and 1-output variable.

For a $n$-input $m$-output we have $2^m$ possible choices for each possible output, so using the same logic as before we have

$$(2^m)^{2^n} = 2^{m2^n} \,,$$

possible $n$-input $m$-output Boolean functions.

**Exercise C.1-3 (counting professors)**

Let the number of such ways our professors can sit be denoted by $N_t$, with $t$ for "table". Then for each one of these seatings we can generate all permutations of $n$ objects in the following way. First consider a specific table ordering of professors say *abcd*. Then all the other equivalent table orderings are given by circularly shifting the given ordering i.e. producing *bcda*, *cdab*, *dabc*. This circular shifts produce $n$ different. Since by performing this operation on each of the $N_t$ table orderings we get all possible permutations of which there are $n!$, so in equation form we have that $nN_t$ must equal $n!$, which when we solve for $N_t$ gives

$$N_t = \frac{n!}{n} = (n-1)! \,.$$

**Exercise C.1-4 (an even sum from distinct subsets of size three)**

Our set is given by $S = 1, 2, \ldots, 100$ and we want three distinct numbers that sum to an even number. The total number of distinct size 3 subsets (independent of the order of the elements in the subset) drawn from $S$ is given by $\begin{pmatrix} 100 \\ 3 \end{pmatrix}$. Half of these will sum to an even number and the other half will sum to an odd number. Thus the total number is given by

$$\frac{1}{2}\begin{pmatrix} 100 \\ 3 \end{pmatrix} = 80850 \,.$$

Another way to obtain this same number is to recognize that to have an even sum I must pick either three even numbers or one even number and two odd numbers. Since these two outcomes are mutually exclusive using the rule of sum the total number of ways to pick a even summable subset is the sum of the number of ways to select each of the previous sets. This number is

$$\left( \begin{array}{c} 50 \\ 3 \end{array} \right) + \left( \begin{array}{c} 50 \\ 1 \end{array} \right) \left( \begin{array}{c} 50 \\ 2 \end{array} \right),$$

where the first combination is the number of ways to select three even numbers and the second combination product is the number of ways to select a single odd number and then two even numbers. One can check that the sum above reduces to 80850 also.

**Exercise C.1-5 (factoring a fraction from the binomial coefficients)**

Using the definition of the binomial coefficient, we have the following

$$\left( \begin{array}{c} n \\ k \end{array} \right) = \frac{n!}{k!(n-k)!} = \frac{n(n-1)!}{k(k-1)!(n-1-(k-1))!} = \frac{n}{k} \left( \begin{array}{c} n-1 \\ k-1 \end{array} \right). \tag{4}$$

**Exercise C.1-6 (factoring another fraction from the binomial coefficients)**

Using the definition of the binomial coefficient, we have the following

$$\left( \begin{array}{c} n \\ k \end{array} \right) = \frac{n!}{(n-k)!k!} = \frac{n(n-1)!}{(n-k)(n-k-1)!k!} = \frac{n}{n-k} \left( \begin{array}{c} n-1 \\ k \end{array} \right).$$

**Exercise C.1-7 (choosing $k$ subsets from $n$ by drawing subsets of size $k-1$)**

Considering the group of $n$ objects with one object specified as distinguished or "special". Then the number of ways to select $k$ objects from $n$ can be decomposed into two distinct occurrences. The times when this "special" object *is* selected in the subset of size $k$ and the times when its *not*. When it is *not* selected in the subset of size $k$ we are specifying our $k$ subset elements from the $n-1$ remaining elements giving $\left( \begin{array}{c} n-1 \\ k \end{array} \right)$ total subsets in this case. When it *is* selected into the subset of size $k$ we have to select $k-1$ other elements from the $n-1$ remaining elements, giving $\left( \begin{array}{c} n-1 \\ k-1 \end{array} \right)$ additional subsets in this case. Summing the counts from these two occurrences we have that factorization can be written as the following

$$\left( \begin{array}{c} n \\ k \end{array} \right) = \left( \begin{array}{c} n-1 \\ k \end{array} \right) + \left( \begin{array}{c} n-1 \\ k-1 \end{array} \right).$$

**Exercise C.1-8 (Pascal's triangle)**

We have (evaluating our binomial coefficients as we place them in the table)

$$\begin{array}{ccccccccccccc}
 & & & & & 1 & & & & & & & \\
 & & & & 1 & & 1 & & & & & & \\
 & & & 1 & & 2 & & 1 & & & & & \\
 & & 1 & & 3 & & 3 & & 1 & & & & \\
 & 1 & & 4 & & 6 & & 4 & & 1 & & & \\
1 & & 5 & & 10 & & 10 & & 5 & & 1 & & \\
1 & & 6 & & 15 & & 20 & & 15 & & 6 & & 1
\end{array}$$

Note that the top row only has one element $\begin{pmatrix} 0 \\ 0 \end{pmatrix} \equiv 1$. The second row has two elements $\begin{pmatrix} 1 \\ 0 \end{pmatrix} = 1$, and $\begin{pmatrix} 1 \\ 1 \end{pmatrix} = 1$. Subsequent rows are obtained by beginning and ending with a one and summing the values of the two binomial coefficients in the row above.

**Exercise C.1-9 (sum of the integers up to $n$)**

Expressing each side of this expression, we have that the left hand side is given by

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

while the right hand side is given by

$$\begin{pmatrix} n+1 \\ 2 \end{pmatrix} = \frac{n(n+1)}{2}$$

since the two sides are equal we have the proven identity.

**Exercise C.1-10 (maximum of $\begin{pmatrix} n \\ k \end{pmatrix}$ as a function of $k$)**

Because the binomial coefficients satisfy

$$\begin{pmatrix} n \\ k \end{pmatrix} = \begin{pmatrix} n \\ n-k \end{pmatrix} \quad \text{for} \quad 0 \leq k \leq n$$

as a function of $k$ the values of the binomial coefficients repeat once $k \geq \frac{n}{2}$. A specific example of this can be seen in the Pascal triangle construction in Exercise C.1-8. If we can show that $\begin{pmatrix} n \\ k \end{pmatrix}$ is an increasing function of $k$, the $k$ value that will maximize the binomial coefficient will be the one corresponding to $k \approx \frac{n}{2}$. If $n$ is even this maximum will occur the value floor($\frac{n}{2}$) while if $n$ is odd this maximum will be found at floor($\frac{n}{2}$) and ceiling($\frac{n}{2}$).

All one has to do is show that $\left( \begin{array}{c} n \\ k \end{array} \right)$ is an increasing function of $k$, which means proving the following chain of reasoning is reversible

$$\begin{aligned}
\left( \begin{array}{c} n \\ k+1 \end{array} \right) &> \left( \begin{array}{c} n \\ k \end{array} \right) \\
\frac{n!}{(k+1)!(n-k-1)!} &> \frac{n!}{(n-k)!k!} \\
\frac{(n-k)!}{(n-k-1)!} &> \frac{(k+1)!}{k!} \\
n-k &> k+1 \\
n &> 2k+1 \\
k &< \frac{n-1}{2}
\end{aligned}$$

which says that $\left( \begin{array}{c} n \\ k \end{array} \right)$ is increasing as a function of $k$ as long as $k < \frac{n-1}{2}$, which equals the floor and ceiling expressions above if $n$ is odd or even respectively.

**Exercise C.1-11 (a product upper bounds on binomial coefficients)**

We want to show that for any $n \geq 0$, $j \geq 0$, $k \geq 0$ and $j + k \leq n$ that

$$\left( \begin{array}{c} n \\ j+k \end{array} \right) \leq \left( \begin{array}{c} n \\ j \end{array} \right) \left( \begin{array}{c} n-j \\ k \end{array} \right) .$$

To show this using an algebraic proof, we will evaluate the left hand side of this expression, convert it into the right hand side multiplied by a correction factor and then show that the correction factor is less than one. To begin we have

$$\begin{aligned}
\left( \begin{array}{c} n \\ j+k \end{array} \right) &= \frac{n!}{(j+k)!(n-j-k)!} \\
&= \frac{n!}{j!(n-j)!} \frac{j!(n-j)!}{(j+k)!(n-j-k)!} \\
&= \left( \begin{array}{c} n \\ j \end{array} \right) \frac{(n-j)!}{k!(n-j-k)!} \frac{j!k!}{(j+k)!} \\
&= \left( \begin{array}{c} n \\ j \end{array} \right) \left( \begin{array}{c} n-j \\ k \end{array} \right) \frac{(j(j-1)(j-2)\cdots 2\,1)(k(k-1)\cdots 2\,1)}{(j+k)(j+k-1)(j+k-2)\cdots 21}
\end{aligned}$$

Note that the top and bottom of the factor multiplying the terms $\left( \begin{array}{c} n \\ j \end{array} \right) \left( \begin{array}{c} n-j \\ k \end{array} \right)$ both have $j + k$ elements in their product. With out loss of generality we will assume that $j \leq k$. The product above can be written as

$$\begin{aligned}
&\frac{(j(j-1)(j-2)\cdots 2\,1)(k(k-1)\cdots 2\,1)}{(j+k)(j+k-1)(j+k-2)\cdots 21} = \frac{j!k!}{(k+j)(k+j-1)\cdots(k+1)k!} \\
= \quad &\frac{j(j-1)(j-2)\cdots 21}{(j+k)(j+k-1)(j+k-2)\cdots(k+2)(k+1)} < 1 ,
\end{aligned}$$

since for every factor in the numerator, we can find a term in the denominator that is larger than it, for example the ratio of the first product in the numerator and denominator above satisfy

$$\frac{j}{j+k} < 1\,.$$

As the combined correction factor (with all these ratios) is less than one, we have shown the inequality we started with.

As a combinatorial proof, the number of ways to select $j+k$ items from a set of $n$ will be smaller than if we first select $j$ items from $n$, and for each of those sets select $k$ items from $n-j$. That the former is a larger number can be seen by the following example where equality does not hold. Consider the case $n=4$, $k=3$, and $j=1$, then

$$\left(\begin{array}{c} 4 \\ 4 \end{array}\right) = 1 < \left(\begin{array}{c} 4 \\ 1 \end{array}\right)\left(\begin{array}{c} 3 \\ 3 \end{array}\right) = 4\,.$$

There the number of ways to select 4 items from 4 is only one which is smaller than if we are allowed to select 1 of the 4 items first (of which there are four ways) and then combine this with the number of ways to select three remaining elements (of which there is only one).

**Exercise C.1-12 (a algebraic upper bound on the binomial coefficients)**

We desire to prove by induction that

$$\left(\begin{array}{c} n \\ k \end{array}\right) \leq \frac{n^n}{k^k(n-k)^{n-k}}\,.$$

Assuming that $0^0 = 1$, we can show that $k=0$ satisfies this easily since it reduces to $1 \leq 1$. However, to begin with a more realistic case, to anchor our induction proof to, we start with $k=1$. This value gives $\left(\begin{array}{c} n \\ 1 \end{array}\right) = n$ for the left hand side of this expression, while the right hand-side evaluates to

$$\frac{n^n}{(n-1)^{n-1}}\,.$$

This can be simplified as follows

$$\begin{aligned} \frac{n^n}{(n-1)^{n-1}} &= n\left(\frac{n^{n-1}}{(n-1)^{n-1}}\right) \\ &= n\left(\frac{n}{n-1}\right)^{n-1} \\ &= n\left(\frac{1}{1-\frac{1}{n}}\right)^{n-1} \end{aligned}$$

Now since we assume that $n > 1$, the expression $1 - \frac{1}{n}$ is bounded by

$$0 < 1 - \frac{1}{n} < 1$$

and more specifically

$$1 < \frac{1}{1-\frac{1}{n}}\,.$$

Taking positive powers of this expression (i.e. to the $n-1$ power) can only increase its value and we have that

$$n < n\left(\frac{1}{1-\frac{1}{n}}\right)^{n-1}$$

which provides the anchoring point from which to begin mathematical induction. Having shown that this inequality is true for $k=1$ we next proceed to assume it is true for $\hat{k} < k$ and show that it is true for $\hat{k} < k+1$. To do this consider the following

$$\begin{aligned}
\left(\begin{array}{c} n \\ k+1 \end{array}\right) &= \frac{n!}{(n-k-1)!(k+1)!} \\
&= \frac{n!}{(n-k)!k!}\left(\frac{n-k}{k+1}\right) \\
&= \left(\begin{array}{c} n \\ k \end{array}\right)\left(\frac{n-k}{k+1}\right) \\
&\leq \frac{n^n}{k^k(n-k)^{n-k}}\left(\frac{n-k}{k+1}\right) = \frac{n^n}{k^k(k+1)(n-k)^{n-k-1}}\,.
\end{aligned}$$

Here on the last step we have used the induction hypothesis. Our induction proof will be finished if we can show that the last expression above is less than the following

$$\frac{n^n}{(k+1)^{k+1}(n-k-1)^{n-k-1}}\,.$$

Towards this end we will show this by *assuming* that it is true and then through reversible transformations reduce this expression to one that is known to be true. Since all transformation are reversible the original inequality must be true. So to begin we assume that

$$\begin{aligned}
\frac{n^n}{k^k(k+1)(n-k)^{n-k-1}} &\leq \frac{n^n}{(k+1)^{k+1}(n-k-1)^{n-k-1}} \\
\frac{(k+1)^k}{k^k} &\leq \frac{(n-k)^{n-k-1}}{(n-k-1)^{n-k-1}} \\
\left(1+\frac{1}{k}\right)^k &\leq \left(1+\frac{1}{n-k-1}\right)^{n-k-1}
\end{aligned}$$

From the above if we define the function $f(x)$ as

$$f(x) = \left(1+\frac{1}{x}\right)^x$$

The above expression becomes the following functional relationship, which if we can prove is true, will complete the inductive proof

$$f(k) \leq f(n-k-1)\,.$$

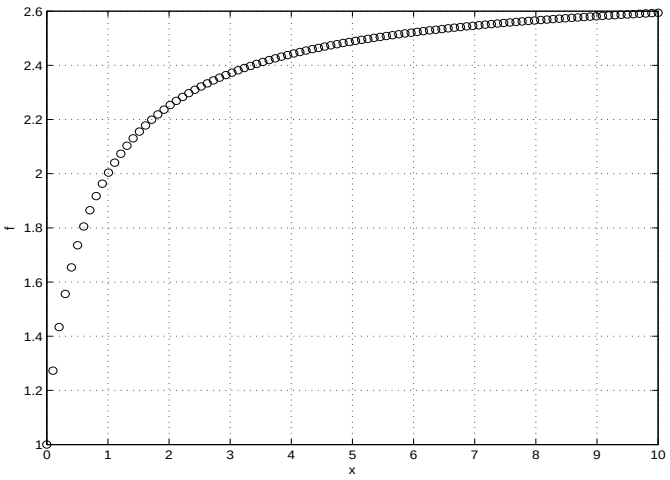



Figure 1: Graphical plot showing the monotonicity of $f(x)$ required for the inductive proof in Exercise C.1-12.

Further assuming that $f$ has a monotone inverse (to be discussed below) then we can apply the inverse to this expression to obtain the region of validity for this inequality as

$$k \leq n - k - 1$$

or $k \leq \frac{n-1}{2}$. For all the statements above to be valid (and our induction proof to be complete) all one must do now is to show that $f(x)$ has an inverse or equivalently that $f(x)$ is monotonic over its domain. This could be done with derivatives but rather than that Figure 1 shows a plot of $f(x)$ showing its monotonic behavior.

Note also from the symmetry relationship possessed by the binomial coefficients, i.e,

$$\left( \begin{array}{c} n \\ k \end{array} \right) = \left( \begin{array}{c} n \\ n-k \end{array} \right)$$

we can extend our result above to all $0 \leq k \leq n$.

**Exercise C.1-13 (Stirling's approximate for some binomial coefficients)**

We desire to show that

$$\left( \begin{array}{c} 2n \\ n \end{array} \right) = \frac{2^{2n}}{\sqrt{\pi n}} \left(1 + O(1/n)\right)$$

using Stirling's approximation. From the definition of the binomial coefficients we know that

$$\left( \begin{array}{c} 2n \\ n \end{array} \right) = \frac{(2n)!}{n!\, n!} .$$

Now Stirling's approximation gives an estimate of $n!$. Its expression is

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \theta(\frac{1}{n})\right) .$$

Therefore substituting $2n$ for $n$ we have an approximation to $(2n)!$ given by

$$(2n)! = \sqrt{4\pi n}\left(\frac{2n}{e}\right)^{2n}\left(1+\theta(\frac{1}{n})\right).$$

We also need an expression for $(n!)^2$, which is given by

$$\begin{aligned}(n!)^2 &= 2\pi n\left(\frac{n}{e}\right)^{2n}\left(1+\theta(\frac{1}{n})\right)^2 \\ &= 2\pi n\left(\frac{n}{e}\right)^{2n}\left(1+\theta(\frac{1}{n})\right).\end{aligned}$$

Thus we have that

$$\begin{aligned}\left(\begin{array}{c}2n\\ n\end{array}\right) &= \frac{\sqrt{4\pi n}\left(\frac{2n}{e}\right)^{2n}\left(1+\theta(\frac{1}{n})\right)}{2\pi n\left(\frac{n}{e}\right)^{2n}\left(1+\theta(\frac{1}{n})\right)} \\ &= \frac{1}{\sqrt{\pi n}}\left(\frac{2n}{n}\right)^{2n}\frac{1+\theta(\frac{1}{n})}{1+\theta(\frac{1}{n})} \\ &= \frac{2^{2n}}{\sqrt{\pi n}}\left(1+\theta(\frac{1}{n})\right)\left(1+\theta(\frac{1}{n})\right) \\ &= \frac{2^{2n}}{\sqrt{\pi n}}\left(1+\theta(\frac{1}{n})\right).\end{aligned}$$

As was to be shown. In deriving this result we have made repeated use of the algebra of order symbols.

**Exercise C.1-14 (the maximum of the entropy function $H(\lambda)$)**

The entropy function is given by

$$H(\lambda) = -\lambda\lg(\lambda) - (1-\lambda)\lg(1-\lambda).$$

Remembering the definition of $\lg(\cdot)$ in terms of the natural logarithm $\ln(\cdot)$ as

$$\lg(x) = \frac{\ln(x)}{\ln(2)}$$

we have that the derivative of the $\lg(\cdot)$ function is given by

$$\frac{\lg(x)}{dx} = \frac{1}{x\ln(2)}.$$

Using this expression we can take the derivative of the binary entropy function giving

$$\begin{aligned}H'(\lambda) &= -\lg(\lambda) - \frac{\lambda}{\lambda\ln(2)} + \lg(1-\lambda) + \frac{(1-\lambda)}{(1-\lambda)\ln(2)} \\ &= -\lg(\lambda) + \lg(1-\lambda).\end{aligned}$$

When we set this equal to zero looking for an extrema we have that

$$-\lg(\lambda) + \lg(1-\lambda) = 0$$

which has as its solution $\lambda = \frac{1}{2}$. We can check that this point is truly a maximum and not a minimum by computing the second derivative of $H(\lambda)$. We have that

$$H''(\lambda) = -\frac{1}{\lambda \ln(2)} - \frac{1}{(1-\lambda)\ln(2)}\,,$$

which is negative when $\lambda = \frac{1}{2}$ implying a maximum. We also have that

$$H(\frac{1}{2}) = \frac{1}{2} + \frac{1}{2} = 1\,.$$

**Exercise C.1-15 (moment summing of binomial coefficients)**

Consider the binomial expansion learned in high school

$$(x+y)^n = \sum_{k=0}^{n} \left( \begin{array}{c} n \\ k \end{array} \right) x^k y^{n-k}\,.$$

Taking the derivative of this expression with respect to $x$ gives the following

$$n(x+y)^{n-1} = \sum_{k=0}^{n} \left( \begin{array}{c} n \\ k \end{array} \right) k x^{k-1} y^{n-k}\,,$$

evaluating at $x = y = 1$, then gives

$$n2^{n-1} = \sum_{k=0}^{n} \left( \begin{array}{c} n \\ k \end{array} \right) k\,,$$

as was desired to be shown.

## Appendix C.2 (Probability)

**Exercise C.2-1 (Boole's inequality)**

We begin by decomposing the countable union of sets $A_i$ i.e.

$$A_1 \cup A_2 \cup A_3 \ldots\,,$$

into a countable union of disjoint sets $C_j$. Define these disjoint sets as

$$\begin{aligned}
C_1 &= A_1 \\
C_2 &= A_2 \backslash A_1 \\
C_3 &= A_3 \backslash (A_1 \cup A_2) \\
C_4 &= A_4 \backslash (A_1 \cup A_2 \cup A_3) \\
&\vdots \\
C_j &= A_j \backslash (A_1 \cup A_2 \cup A_3 \cup \cdots \cup A_{j-1})
\end{aligned}$$

Then by construction

$$A_1 \cup A_2 \cup A_3 \cdots = C_1 \cup C_2 \cup C_3 \cdots ,$$

and the $C_j$'s are disjoint, so that we have

$$\Pr(A_1 \cup A_2 \cup A_3 \cup \cdots) = \Pr(C_1 \cup C_2 \cup C_3 \cup \cdots) = \sum_j \Pr(C_j) .$$

Since $\Pr(C_j) \leq \Pr(A_j)$, for each $j$, this sum is bounded above by

$$\sum_j \Pr(A_j) ,$$

and Boole's inequality is proven.

**Exercise C.2-2 (more head for professor Rosencrantz)**

For this problem we can explicitly enumerate our sample space and then count the number of occurrences where professor Rosencrantz obtains more heads than professor Guildenstern. Denoting the a triplet of outcomes from the one coin flip by Rosencrantz and the two coin flips by Guildenstern as $(R, G_1, G_2)$, we see that the total sample space for this experiment is given by

$$\begin{array}{cccc} (H,H,H) & (H,H,T) & (H,T,H) & (H,T,T) \\ (T,H,H) & (T,H,T) & (T,T,H) & (T,T,T) \end{array}$$

From which the only outcome where professor Rosencrantz obtains more heads than professor Guildenstern is the sample $(H, T, T)$. Since this occurs once from eights possible outcomes, Rosencrantz has a $1/8$ probability of winning.

**Exercise C.2-3 (drawing three ordered cards)**

There are $10 \cdot 9 \cdot 8 = 720$ possible ways to draw three cards from ten when the order of the drawn cards matters. To help motivate how to calculate then number of *sorted* three card draws possible, we'll focus how the number of ordered draws depends on the numerical value of the *first* card drawn. For instance, if a ten or a nine is drawn as a first card there are no possible ways to draw two other cards and have the total set ordered. If an eight is drawn as the first card, then it is possible to draw a nine for the second and a ten for the third producing the ordered set $(8, 9, 10)$. Thus if an eight is the first card drawn there is one possible sorted hand. If a seven is the first card drawn then we can have second and third draws consisting of $(8, 9)$, $(8, 10)$, or a $(9, 10)$ and have an ordered set of three cards. Once the seven is drawn we now are looking for the number of ways to draw two sorted cards from $\{8, 9, 10\}$. So if an seven is drawn we have three possible sorted hands. Continuing, if a six is drawn as the first card we have possible second and third draws of: $(7, 8)$, $(7, 9)$, $(7, 10)$, $(8, 9)$, $(8, 10)$, or $(9, 10)$ to produce an ordered set. Again we are looking for a way to draw two sorted cards from a list (this time $\{7, 8, 9, 10\}$).

After we draw the first card an important subcalculation is to compute the number of ways to draw *two* sorted cards from a list. Assuming we have $n$ ordered items in our list, then we have $n$ ways to draw the first element and $n-1$ ways to draw the second element giving $n(n-1)$ ways to draw two elements. In this pair of elements *one* ordering result in a correctly sorted list (the other will not). Thus half of the above are incorrectly ordered what remains are

$$N_2(n) = \frac{n(n-1)}{2}$$

The number of total ways to draw three sorted cards can now be found. If we first draw an eight we have two cards from which to draw the two remaining ordered cards in $N_2(2)$ ways. If we instead first drew a seven we have three cards from which to draw the two remaining ordered cards in $N_2(3)$ ways. If instead we first drew a six then we have four cards from which to draw two ordered cards in $N_2(4)$ ways. Continuing, if we first draw a one, we have nine cards from which to draw two ordered cards in $N_2(9)$ ways. Since each of these options is mutually exclusive the total number of ways to draw three ordered cards is given by

$$N_2(3) + N_2(4) + N_2(5) + \cdots + N_2(9) = \sum_{n=2}^{9} N_2(n) = \sum_{n=2}^{9} \frac{n(n-1)}{2} = 120\,.$$

Finally with this number we can calculate the probability of drawing three ordered cards as

$$\frac{120}{720} = \frac{1}{6}\,.$$

**Exercise C.2-4 (simulating a biased coin)**

Following the hint, since $a < b$, the quotient $a/b$ is less than one. If we express this number in binary we will have an (possibly infinite) sequence of zeros and ones. The method to determine whether to return a biased "heads" (with probability $a/b$) or a biased "tails" (with probability $(b-a)/b$) is best explained with an example. To get a somewhat interesting binary number, lets assume an example where $a = 76$ and $b = 100$, then our ratio $a/b$ is 0.76. In binary that has a representation given by (see the Mathematica file `exercise_C_2_4.nb`)

$$0.76 = 0.1100001010001111011_2$$

We then begin flipping our unbiased coin. To a "head" result we will associate a one and to a tail result we shall associate a zero. As long as the outcome of the flipped coin matches the sequence of ones and zeros in the binary expansion of $a/b$, we continue flipping. At the point where the flips of our unbiased coins diverges from the binary sequence of $a/b$ we stop. If the divergence produces a sequence that is *less than* the ratio $a/b$ we return this result by returning a biased "head" result. If the divergence produces a sequence that is *greater than* the ratio $a/b$ we return this result by returning a biased "tail" result. Thus we have used our fair coin to determine where a sequence of flips "falls" relative to the ratio of $a/b$.

The expected number of coin flips to determine our biased result will be the expected number of flips required until the flips from the unbiased coin don't match the sequence of zeros and ones in the binary expansion of the ratio $a/b$. If we assume that a success is when our

unbiased coin flip *does not* match the digit in the binary expansion of $a/b$. The for each flip a success can occur with probability $p = 1/2$ and a failure can occur with probability $q = 1 - p = 1/2$, we have that the number of flips $n$ needed before a "success" is a geometric random variable with parameter $p = 1/2$. This is that

$$P\{N = n\} = q^{n-1}p\,.$$

So that the expected number of flips required for a success is then the expectation of the geometric random variable and is given by

$$E[N] = \frac{1}{p} = 2\,,$$

which is certainly $O(1)$.

**Exercise C.2-5 (closure of conditional probabilities)**

Using the definition of conditional probability given in this section we have that

$$P\{A|B\} = \frac{P\{A \cap B\}}{P\{B\}}\,,$$

so that the requested sum is then given by

$$\begin{aligned} P\{A|B\} + P\{\bar{A}|B\} &= \frac{P\{A \cap B\}}{P\{B\}} + \frac{P\{\bar{A} \cap B\}}{P\{B\}} \\ &= \frac{P\{A \cap B\} + P\{\bar{A} \cap B\}}{P\{B\}} \end{aligned}$$

Now since the events $A \cap B$ and $\bar{A} \cap B$ are mutually exclusive the above equals

$$\frac{P\{(A \cap B) \cup (\bar{A} \cap B)\}}{P\{B\}} = \frac{P\{(A \cup \bar{A}) \cap B\}}{P\{B\}} = \frac{P\{B\}}{P\{B\}} = 1\,.$$

**Exercise C.2-6 (conditioning on a chain of events)**

This result follows for the two set case $P\{A \cap B\} = P\{A|B\}P\{B\}$ by grouping the sequence of $A_i$'s in the appropriate manner. For example by grouping the intersection as

$$A_1 \cap A_2 \cap \cdots \cap A_{n-1} \cap A_n = (A_1 \cap A_2 \cap \cdots \cap A_{n-1}) \cap A_n$$

we can apply the two set result to obtain

$$P\{A_1 \cap A_2 \cap \cdots \cap A_{n-1} \cap A_n\} = P\{A_n | A_1 \cap A_2 \cap \cdots \cap A_{n-1}\}\, P\{A_1 \cap A_2 \cap \cdots \cap A_{n-1}\}\,.$$

Continuing now to peal $A_{n-1}$ from the set $A_1 \cap A_2 \cap \cdots \cap A_{n-1}$ we have the second probability above equal to

$$P\{A_1 \cap A_2 \cap \cdots \cap A_{n-2} \cap A_{n-1}\} = P\{A_{n-1} | A_1 \cap A_2 \cap \cdots \cap A_{n-2}\} P\{A_1 \cap A_2 \cap \cdots \cap A_{n-2}\}\,.$$

Continuing to peal off terms from the back we eventually obtain the requested expression i.e.

$$\begin{aligned} P\{A_1 \cap A_2 \cap \cdots \cap A_{n-1} \cap A_n\} &= P\{A_n | A_1 \cap A_2 \cap \cdots \cap A_{n-1}\} \\ &\times P\{A_{n-1} | A_1 \cap A_2 \cap \cdots \cap A_{n-2}\} \\ &\times P\{A_{n-2} | A_1 \cap A_2 \cap \cdots \cap A_{n-3}\} \\ &\vdots \\ &\times P\{A_3 | A_1 \cap A_2\} \\ &\times P\{A_2 | A_1\} \\ &\times P\{A_1\}\,. \end{aligned}$$

**Exercise C.2-7 (pairwise independence does not imply independence)**

**Note:** As requested by the problem I was *not* able to find a set of events that are pairwise independent but such that no subset $k > 2$ of them are mutually independent. Instead I found a set of events that are all pairwise independent but that are not mutually independent. If you know of a solution to the former problem please contact me.

Consider the situation where we have $n$ distinct people in a room. Let $A_{i,j}$ be the event that person $i$ and $j$ have the same birthday. We will show that any two of these events are pairwise independent but the totality of events $A_{i,j}$ are not mutually independent. That is we desire to show that the two events $A_{i,j}$ and $A_{r,s}$ are independent but the totality of all $\begin{pmatrix} n \\ 2 \end{pmatrix}$ events are not independent. Now we have that

$$P(A_{i,j}) = P(A_{r,s}) = \frac{1}{365}\,,$$

since for the specification of either one persons birthday the probability that the other person will have that birthday is $1/365$. Now we have that

$$P(A_{i,j} \cap A_{r,s}) = P(A_{i,j} | A_{r,s}) P(A_{r,s}) = \left(\frac{1}{365}\right)\left(\frac{1}{365}\right) = \frac{1}{365^2}\,.$$

This is because $P(A_{i,j}|A_{r,s}) = P(A_{i,j})$ i.e. the fact that persons $r$ and $s$ have the same birthday has no effect on whether the persons $i$ and $j$ have the same birthday. This is true even if one of the people in the pairs $(i,j)$ and $(r,s)$ is the same. When we consider the intersection of *all* the sets $A_{i,j}$, the situation changes. This is because the event $\cap_{(i,j)} A_{i,j}$ (where the intersection is over all pairs $(i,j)$) is the event that *every* pair of people have the same birthday, i.e. that everyone considered has the same birthday. This will happen with probability

$$\left(\frac{1}{365}\right)^{n-1}\,,$$

while if the events $A_{i,j}$ were independent the required probability would be

$$\prod_{(i,j)} P(A_{i,j}) = \left(\frac{1}{365}\right)^{\binom{n}{2}} = \left(\frac{1}{365}\right)^{\frac{n(n-1)}{2}} .$$

Since $\binom{n}{2} \neq n-1$, these two results are not equal and the totality of events $A_{i,j}$ are not independent.

**Exercise C.2-8 (conditional but not independence)**

Consider the situation where we have to coins. The first coin $C_1$ is biased and has probability $p$ of landing heads when flipped where $p > 1/2$. The second coin $C_2$ is fair and has probability of landing heads $1/2$. For our experiment one of the two coins will be selected (uniformly and at random) and presented to two people who will each flip this coin. We will assume that the coin that is selected is not known. Let $H_1$ be the event that the first flip lands heads, and let $H_2$ be the event that the second flip lands heads. We can show that the events $H_1$ and $H_2$ are not independent by computing $P(H_1)$, $P(H_2)$, and $P(H_1, H_2)$ by conditioning on the coin selected. We have for $P(H_1)$

$$\begin{aligned} P(H_1) &= P(H_1|C_1)P(C_1) + P(H_1|C_2)P(C_2) \\ &= p\frac{1}{2} + \frac{1}{2}\frac{1}{2} \\ &= \frac{1}{2}(p + \frac{1}{2}) , \end{aligned}$$

Here $C_1$ is the event that we select the first (biased) coin while $C_2$ is the event we select the second (unbiased) coin. In the same way we find that

$$P(H_2) = \frac{1}{2}(p + \frac{1}{2}) ,$$

while $P(H_1, H_2)$ is given by

$$\begin{aligned} P(H_1, H_2) &= P(H_1, H_2|C_1)P(C_1) + P(H_1, H_2|C_2)P(C_2) \\ &= p^2\frac{1}{2} + \frac{1}{4}\frac{1}{2} \\ &= \frac{1}{2}(p^2 + \frac{1}{4}) . \end{aligned}$$

The events $H_1$ and $H_2$ will *only* be independent if $P(H_1)P(H_2) = P(H_1, H_2)$ or

$$\frac{1}{4}\left(p + \frac{1}{2}\right)^2 = \frac{1}{2}(p^2 + \frac{1}{4}) .$$

or simplifying this result equality only holds if $p = 1/2$ which we are assuming is not true. Now let event $E$ denote the event that we are *told* that the coin selected and flipped by both

parties is the fair one, i.e. that event $C_2$ happens. Then we have

$$\begin{aligned} P(H_1|E) &= \frac{1}{2} \quad \text{and} \quad P(H_2|E) = \frac{1}{2} \\ P(H_1, H_2|E) &= \frac{1}{4}. \end{aligned}$$

Since in this case we do have $P(H_1, H_2|E) = P(H_1|E)P(H_2|E)$ we have the required conditional independence. Intuitively, this result makes sense because once we *know* that the coin flipped is fair we expect the result of each flip to be independent.

**Exercise C.2-9 (the Monte-Hall problem)**

Once we have selected a curtain we are two possible situations we might find ourself in. The first situation $A$, is where we have selected the curtain with the prize behind it or situation $B$ where we have *not* selected the curtain with the prize behind it. The initial probability of event $A$ is $1/3$ and that of event $B$ is $2/3$. We will calculate our probability of winning under the two choices of actions. The first is that we choose *not* to switch curtains after the emcee opens a curtain that does not cover the prize. The probability that we win, assuming the "no change" strategy, can be computed by conditioning on the events $A$ and $B$ above as

$$P(W) = P(W|A)P(A) + P(W|B)P(B).$$

Under event $A$ and the fact that we don't switch curtains we have $P(W|A) = 1$ and $P(W|B) = 0$, so we see that $P(W) = 1/3$ as would be expected. If we now assume that we follow the second strategy where by we *switch* curtains after the emcee revels a curtain behind which the prize does not sit. In that case $P(W|A) = 0$ and $P(W|B) = 1$, since in this strategy we switch curtains. The the total probability we will is given by

$$P(W) = 0 + 1 \cdot \frac{2}{3} = \frac{2}{3},$$

since this is greater than the probability we would win under the strategy were we *don't switch* this is the action we should take.

**Exercise C.2-10 (a prison delima)**

I will argue that $X$ has obtained some information from the guard. Before asking his question the probability of event $X$ ($X$ is set free) is $P(X) = 1/3$. If prisoner $X$ is told that $Y$ (or $Z$ in fact) is to be executed, then to determine what this implies about the event $X$ we need to compute $P(X|\neg Y)$. Where $X$, $Y$, and $Z$ are the events that prisoner $X$, $Y$, or $Z$ is to be set free respectively. Now from Bayes' rule

$$P(X|\neg Y) = \frac{P(\neg Y|X)P(X)}{P(\neg Y)}.$$

We have that $P(\neg Y)$ is given by

$$P(\neg Y) = P(\neg Y|X)P(X) + P(\neg Y|Y)P(Y) + P(\neg Y|Z)P(Z) = \frac{1}{3} + 0 + \frac{1}{3} = \frac{2}{3}\,.$$

So the above probability then becomes

$$P(X|\neg Y) = \frac{1(1/3)}{2/3} = \frac{1}{2} > \frac{1}{3}\,.$$

Thus the probability that prisoner $X$ will be set free has increased and prisoner $X$ has learned from his question.

**proof of the cumulative summation formula for $E[X]$ (book notes page 1109)**

We have from the definition of the expectation that

$$E[X] = \sum_{i=0}^{\infty} iP\{X = i\}\,.$$

expressing this in terms of the the complement of the cumulative distribution function $P\{X \geq i\}$ gives $E[X]$ equal to

$$\sum_{i=0}^{\infty} i(P\{X \geq i\} - P\{X \geq i+1\})\,.$$

Using the theory of difference equations we write the difference in probabilities above in terms of the $\Delta$ operator defined on a discrete function $f(i)$ as

$$\Delta g(i) \equiv g(i+1) - g(i)\,,$$

giving

$$E[X] = -\sum_{i=0}^{\infty} i\Delta_i P\{X \geq i\}\,.$$

No using the discrete version of integration by parts (demonstrated here for two discrete functions $f$ and $g$) we have

$$\sum_{i=0}^{\infty} f(i)\Delta g(i) = f(i)g(i)|_{i=1}^{\infty} - \sum_{i=1}^{\infty} \Delta f(i)g(i)\,,$$

gives for $E[X]$ the following

$$\begin{aligned} E[X] &= -iP\{X \geq i\}|_{i=1}^{\infty} + \sum_{i=1}^{\infty} P\{X \geq i\} \\ &= \sum_{i=1}^{\infty} P\{X \geq i\}\,. \end{aligned}$$

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | (2,1) | (3,2) | (4,3) | (5,4) | (6,5) | (7,6) |
| 2 | (2,2) | (4,2) | (5,3) | (6,4) | (7,5) | (8,6) |
| 3 | (4,3) | (5,3) | (6,3) | (7,4) | (8,5) | (9,6) |
| 4 | (5,4) | (6,4) | (7,4) | (8,4) | (9,5) | (10,6) |
| 5 | (6,5) | (7,5) | (8,5) | (9,5) | (10,5) | (11,6) |
| 6 | (7,6) | (8,6) | (9,6) | (10,6) | (11,6) | (12,6) |

Table 2: The possible values for the sum (the first number) and the maximum (the second number) observed when two die are rolled.

which is the desired sum. To see this another way we can explicitly write out the summation given above and cancel terms as in

$$\begin{aligned} E[X] &= \sum_{i=0}^{\infty} i(P\{X \geq i\} - P\{X \geq i+1\}) \\ &= 0 + P\{X \geq 1\} - P\{X \geq 2\} \\ &+ 2P\{X \geq 2\} - 2P\{X \geq 3\} \\ &+ 3P\{X \geq 3\} - 3P\{X \geq 4\} + \ldots \\ &= P\{X \geq 1\} + P\{X \geq 2\} + P\{X \geq 3\} + \ldots , \end{aligned}$$

verifying what was claimed above.

## Appendix C.3 (Discrete random variables)

### Exercise C.3-1 (expectation of the sum and maximum of two die)

We will begin by computing all possible sums and maximum that can be obtained when we roll two die. These numbers are computed in table 2, where the row corresponds to the first die and the column corresponds to the second die. Since each roll pairing has a probability of $1/36$ of happening we see that the expectation of the sum $S$ is given by

$$\begin{aligned} E[S] &= 2\left(\frac{1}{36}\right) + 3\left(\frac{2}{36}\right) + 4\left(\frac{3}{36}\right) + 5\left(\frac{4}{36}\right) + 6\left(\frac{5}{36}\right) + 7\left(\frac{6}{36}\right) \\ &+ 8\left(\frac{5}{36}\right) + 9\left(\frac{4}{36}\right) + 10\left(\frac{3}{36}\right) + 11\left(\frac{2}{36}\right) + 12\left(\frac{1}{36}\right) \\ &= 6.8056\,. \end{aligned}$$

While the expectation of the maximum $M$ is given by a similar expression

$$\begin{aligned} E[M] &= 1\left(\frac{1}{36}\right) + 2\left(\frac{3}{36}\right) + 3\left(\frac{5}{36}\right) + 4\left(\frac{7}{36}\right) + 5\left(\frac{9}{36}\right) + 6\left(\frac{11}{36}\right) \\ &= 4.4722\,. \end{aligned}$$

**Exercise C.3-2 (expectation of the index of the max and min of a random array)**

Since the array elements are assumed random the maximum can be at any of the $n$ indices with probability $1/n$. Thus if we define $X$ to be the random variable representing the location of the maximum of our array we see that the expectation of $X$ is given by

$$\begin{aligned} E[X] &= 1\left(\frac{1}{n}\right) + 2\left(\frac{1}{n}\right) + \cdots + n\left(\frac{1}{n}\right) \\ &= \frac{1}{n}\sum_{k=1}^{n} k = \frac{1}{n}\left(\frac{n(n+1)}{2}\right) \\ &= \frac{n+1}{2}\,. \end{aligned}$$

The same is true for the expected location of the minimum.

**Exercise C.3-3 (expected cost of playing a carnival game)**

Define $X$ to be the random variable denoting the payback from one play of the carnival game. Then the payback depends on the possible outcomes after the player has guessed a number. Let $E_0$ be the event that the player's number does not appear on any die, $E_1$ the event that the player's number appears on only one of the three die, $E_2$ the event that the player's number appears on only two of the die, and finally $E_3$ the event that the player's number appears on all three of the die. The the expected payback is given by

$$E[X] = -P(E_0) + P(E_1) + 2P(E_2) + 3P(E_3)\,.$$

We now compute the probability of each of the events above. We have

$$\begin{aligned} P(E_0) &= \left(\frac{5}{6}\right)^3 = 0.5787 \\ P(E_1) &= 3\left(\frac{1}{6}\right)\left(\frac{5}{6}\right)^2 = 0.3472 \\ P(E_2) &= 3\frac{1}{6}\cdot\frac{1}{6}\cdot\frac{5}{6} = 0.0694 \\ P(E_3) &= \frac{1}{6^3} = 0.0046\,. \end{aligned}$$

Where the first equation expresses the fact that each individual die will not match the player's selected die with probability of $5/6$, so the three die will not match the given die with probability $(5/6)^3$. The other probabilities are similar. Using these to compute the expected payoff using the above formula gives

$$E[X] = -\frac{17}{216} = -0.0787\,.$$

To verify that these numbers are correct in the Matlab file `exercise_C_3_3.m`, a Monte-Carlo simulation is developed verifying these values. This function can be run with the command `exercise_C_3_3(100000)`.

**Exercise C.3-4 (bounds on the expectation of a maximum)**

We have for $E[\max(X,Y)]$ computed using the joint density of $X$ and $Y$ that

$$\begin{aligned} E[\max(X,Y)] &= \sum_x \sum_y \max(x,y) P\{X=x, Y=y\} \\ &\leq \sum_x \sum_y (x+y) P\{X=x, Y=y\}\,, \end{aligned}$$

since $X$ and $Y$ are non-negative. Then using the linearity of the above we have

$$\begin{aligned} E[\max(X,Y)] &\leq \sum_x \sum_y x P\{X=x, Y=y\} + \sum_x \sum_y y P\{X=x, Y=y\} \\ &= \sum_x x P\{X=x\} + \sum_y y P\{Y=y\} \\ &= E[X] + E[Y]\,, \end{aligned}$$

which is what we were to show.

**Exercise C.3-5 (functions of independent variables are independent)**

By the independence of $X$ and $Y$ we know that

$$P\{X=x, Y=y\} = P\{X=x\}P\{Y=y\}\,.$$

But by definition of the random variable $X$ has a realization equal to $x$ then the random variable $f(X)$ will have a realization equal to $f(x)$. The same statement hold for the random variable $g(Y)$ which will have a realization of $g(y)$. Thus the above expression is equal to (almost notationally)

$$P\{f(X)=f(x), g(Y)=g(y)\} = P\{f(X)=f(x)\}P\{g(Y)=g(y)\}$$

Defining the random variable $F$ by $F=f(X)$ (an instance of this random variable $f$) and the random variable $G=g(Y)$ (similarly and instance of this random variable $g$) the above shows that

$$P\{F=f, G=g\} = P\{F=f\}P\{G=g\}$$

or

$$P\{f(X)=f, g(Y)=g\} = P\{f(X)=f\}P\{g(Y)=g\}\,,$$

showing that $f(X)$ and $g(Y)$ are independent as requested.

**Exercise C.3-6 (Markov's inequality)**

To prove that

$$P\{X \geq t\} \leq \frac{E[X]}{t}\,,$$

is equivalent to proving that

$$E[X] \geq tP\{X \geq t\}\,.$$

To prove this first consider the expression for $E[X]$ broken at $t$ i.e.

$$\begin{aligned} E[X] &= \sum_x xP\{X = x\} \\ &= \sum_{x<t} xP\{X = x\} + \sum_{x \geq t} xP\{X = x\}\,. \end{aligned}$$

But since $X$ is non-negative we can drop the expression $\sum_{x<t} xP\{X = x\}$ from the above and obtain a lower bound. Specifically we have

$$\begin{aligned} E[X] &\geq \sum_{x \geq t} xP\{X = x\} \\ &\geq t\sum_{x \geq t} P\{X = x\} \\ &= tP\{X \geq t\}\,, \end{aligned}$$

or the desired result.

**Exercise C.3-7 (if $X(s) \geq X'(s)$ then $P\{X \geq t\} \geq P\{X' \geq t\}$)**

Lets begin by defining two sets $A_t$ and $B_t$ as follows

$$\begin{aligned} A_t &= \{s \in S : X(s) \geq t\} \\ B_t &= \{s \in S : X'(s) \geq t\}\ . \end{aligned}$$

Then lets consider an element $\tilde{s} \in B_t$. Since $\tilde{s}$ is in $B_t$ we know that $X'(\tilde{s}) > t$. From the assumption on $X$ and $X'$ we know that $X(\tilde{s}) \geq X'(\tilde{s}) \geq t$, and $\tilde{s}$ must also be in $A_t$. Thus the set $B_t$ is a subset of the set $A_t$. We therefore have that

$$P\{X \geq t\} = \sum_{s \in A_t} P\{s\} \geq \sum_{s \in B_t} P\{s\} = P\{X' \geq t\}\,,$$

and the desired result is obtained.

**Exercise C.3-8 ($E[X^2] > E[X]^2$)**

From the calculation of the variance of the random variable $X$ we have that

$$\mathrm{Var}(X) = E[(X - E[X])^2] > 0\,,$$

since the square in the expectation is always a positive number. Expanding this square we find that

$$E[X^2] - E[X]^2 > 0\,.$$

Which shows on solving for $E[X^2]$ that

$$E[X^2] > E[X]^2 \,,$$

or that the expectation of the square of the random variable is larger than the square of the expectation. This make sense because if $X$ where to take on both positive and negative values in computing $E[X]^2$ the expectation would be decreased by $X$ taking on instances of both signs. The expression $E[X^2]$ would have no such difficulty since $X^2$ is aways positive regardless of the sign of $X$.

**Exercise C.3-9 (the variance for binary random variables)**

Since $X$ takes on only value in $\{0, 1\}$, lets assume that

$$\begin{aligned} P\{X = 0\} &= 1 - p \\ P\{X = 1\} &= p \,. \end{aligned}$$

Then the expectation of $X$ is given by $E[X] = 0(1-p) + 1p = p$. Further the expectation of $X^2$ is given by $E[X^2] = 0(1-p) + 1p = p$. Thus the variance of $X$ is given by

$$\begin{aligned} \mathrm{Var}(X) &= E[X^2] - E[X]^2 \\ &= p - p^2 = p(1-p) \\ &= E[X](1 - E[X]) \\ &= E[X]\, E[1 - X] \,. \end{aligned}$$

Where the transformation $1 - E[X] = E[1 - X]$ is possible because of the linearity of the expectation.

**Exercise C.3-10 (the variance for $aX$)**

From the suggested equation we have that

$$\begin{aligned} \mathrm{Var}(aX) &= E[(aX)^2] - E[aX]^2 \\ &= E[a^2X^2] - a^2E[X]^2 \\ &= a^2(E[X^2] - E[X]^2) \\ &= a^2\mathrm{Var}(X) \,, \end{aligned}$$

by using the linearity property of the expectation.

## Appendix C.4 (The geometric and binomial distributions)

**an example with the geometric distribution (book notes page 1112)**

All the possible outcomes for the sum on two die are given in table 2. There one sees that the sum of a 7 occurs along the right facing diagonal (elements $(6,1), (5,5), (4,3), (3,4), (2,5), (1,6)$),

while the sum of 11 occurs for the elements $(6,5),(5,6)$. The the probability we roll a 7 or an 11 is given by

$$p = \frac{6}{36} + \frac{2}{36} = \frac{8}{36} = \frac{2}{9}\,.$$

if we take this as the probability of success then the remaining results from the book follow.

**Exercise C.4-1 (the closure property for the geometric distribution)**

The geometric distribution gives the probability our first success occurs at trail numbered $k$ when each trial has $p$ as a probability of success. It has a distribution function given by

$$P\{X = k\} = p(1-p)^{k-1} = pq^{k-1}\,.$$

Where we have defined $q = 1 - p$. If we sum this probability distribution for all possible location of the first success we obtain

$$\begin{aligned}
\sum_{k=1}^{\infty} p(1-p)^{k-1} &= \frac{p}{q}\sum_{k=1}^{\infty} q^k \\
&= \frac{p}{q}\left(\sum_{k=0}^{\infty} q^k - 1\right) \\
&= \frac{p}{q}\left(\frac{1}{1-q} - 1\right) \\
&= \frac{p}{q}\left(\frac{1}{p} - 1\right) \\
&= \frac{p}{q}\left(\frac{1-p}{p}\right) = 1\,,
\end{aligned}$$

as we were to show.

**Exercise C.4-2 (average number of times to obtain three heads and tails)**

Let one trial consist of flipping six coins. Then we consider a success when we have obtained three head and three tails. Thus since there are $2^6$ possible outcomes of six flips when the flips are ordered there are then $\binom{6}{3}$ the probability of "success" ($p$) for this experiment is given by

$$p = \frac{\binom{6}{3}}{2^6} = \frac{20}{64} = 0.3125\,.$$

Since the number of trials needed to obtain a success is a geometric distribution we have that the average number of times we must perform this experiment is given by the average for a geometric distribution with probability of success $p$ or

$$\frac{1}{p} = \frac{1}{0.3125} = 3.2\,.$$

**Exercise C.4-3 (an equality with the binomial distribution)**

Using the definition of $b(k;n,p)$ we have that

$$b(k;n,p) = \left( \begin{array}{c} n \\ k \end{array} \right) p^k (1-p)^{n-k} .$$

Since $\left( \begin{array}{c} n \\ k \end{array} \right) = \left( \begin{array}{c} n \\ n-k \end{array} \right)$ the above can be written as

$$b(k;n,p) = \left( \begin{array}{c} n \\ n-k \end{array} \right) (1-p)^{n-k} p^k .$$

Defining $q = 1-p$ we see the right hand side of the above is equal to

$$\left( \begin{array}{c} n \\ n-k \end{array} \right) q^{n-k} (1-q)^k = b(n-k;n,q) ,$$

and the desired result is shown.

**Exercise C.4-4 (the maximum of the binomial coefficient)**

From the discussion in the text the binomial coefficient has a maximum at the integer $k$ that lies in the range $np - q < k < (n+1)p$. To evaluate the approximate maximum of the binomial coefficient we will evaluate it at $k = np$ which is certainly between the two limits $np-q$ and $(n+1)p$. Our binomial coefficient evaluated at its approximate maximum $k = np$ is given by

$$\begin{aligned} b(np;n,p) &= \left( \begin{array}{c} n \\ np \end{array} \right) p^{np} (1-p)^{n-np} \\ &= \frac{n!}{(n-np)!(np)!} p^{np} (1-p)^{nq} \\ &= \frac{n!}{(nq)!(np)!} p^{np} q^{nq} . \end{aligned}$$

where we have introduced $q = 1-p$ into the above to simplify the notation. Using Stirling's approximation for $n!$ given by

$$n! = \sqrt{2\pi n} \left( \frac{n}{e} \right)^n \left( 1 + \theta(\frac{1}{n}) \right) ,$$

we can simplify the factorial expression above (and dropping the order symbols $\theta$) we have

$$\begin{aligned} \frac{n!}{(nq)!(np)!} &\approx \sqrt{2\pi n} \left( \frac{n}{e} \right)^n \times \frac{1}{\sqrt{2\pi nq}} \left( \frac{e}{nq} \right)^{nq} \times \frac{1}{\sqrt{2\pi np}} \left( \frac{e}{np} \right)^{np} \\ &= \frac{1}{\sqrt{2\pi}} \left( \frac{n}{(nq)(np)} \right)^{1/2} \frac{n^n}{(nq)^{nq}(np)^{np}} \\ &= \frac{1}{\sqrt{2\pi}} \left( \frac{1}{nqp} \right)^{1/2} \left( \frac{1}{q^q p^p} \right)^n . \end{aligned}$$

upon multiplying the above by $p^{np}q^{nq}$ we find that

$$b(np;n,p) \approx \left(\frac{1}{2\pi npq}\right)^{1/2},$$

as we were to show.

**Exercise C.4-5 (the limiting probability of no successes)**

The probability of no successes in a sequence of $n$ Bernoulli trials with (probability of success $p = 1/n$) is given by $b(0;n,1/n)$. This is equal to

$$\begin{aligned} b(0;n,1/n) &= \binom{n}{0}\left(\frac{1}{n}\right)^0\left(1-\frac{1}{n}\right)^n \\ &= \left(1-\frac{1}{n}\right)^n \end{aligned}$$

Remembering a famous limit from calculus

$$\lim_{x\to+\infty}\left(1+\frac{x}{n}\right)^n = e^x,$$

we see that for large $n$ the probability $b(0;n,1/n)$ is approximately $e^{-1}$. The probability of at least one success is given by $b(1;n,1/n)$. This is equal to

$$\begin{aligned} b(1;n,1/n) &= \binom{n}{1}\left(\frac{1}{n}\right)^1\left(1-\frac{1}{n}\right)^{n-1} \\ &= n\left(\frac{1}{n}\right)\left(1-\frac{1}{n}\right)^{n-1} \\ &= \left(1-\frac{1}{n}\right)^{n-1} \\ &= \frac{1}{(1-\frac{1}{n})}\left(1-\frac{1}{n}\right)^n \\ &\to \left(1-\frac{1}{n}\right)^n \quad \text{as} \quad n\to+\infty. \end{aligned}$$

Using the limiting argument above we have this probability equal to $e^{-1}$ as $n$ goes to infinity.

**Exercise C.4-6 (the probability of obtaining the same number of heads)**

For a total of $2n$ flips we have $2^{2n} = 4^n$ possible sequences of heads and tails in $2n$ flips. Let the first $n$ of these correspond to the flips of professor Rosencrantz and the remaining $n$ corresponds to those of professor Guildenstern. Following the hint in the book, we will call a success for professor Rosencrantz when his flip lands *heads* and a success for professor

Guildenstern when his flip lands *tails*. Then both professors will obtain the *same* number of heads if say professor Rosencrantz has $k$ successes while professor Guildenstern has $n-k$ successes. Thus the number of sequences (from our $4^n$) that have the same number of heads for both professors is an equivalent problem to selecting $k+(n-k)=n$ total locations from among $2n$ possible and declaring these to be the locations of the successes for both professors. This means that if a success is placed before the $n+1$th flip it is considered to be a success for professor Rosencrantz and denotes a head (the remaining flips for Rosencrantz are then all tails). If a success falls after the $n$th flip it is considered to be a success for professor Guildenstern and is considered to be a tail (the remaining flips for Guildenstern are considered to be all heads). The number of sequences with $n$ total successes is given by $\left( \begin{array}{c} 2n \\ n \end{array} \right)$ and so the probability of obtaining the same number of heads is given by

$$\frac{\left( \begin{array}{c} 2n \\ n \end{array} \right)}{4^n} ,$$

as claimed. Using the result from Exercise C.1-13 which derives the approximate

$$\left( \begin{array}{c} 2n \\ n \end{array} \right) = \frac{4^n}{\sqrt{\pi n}} \left( 1 + O(\frac{1}{n}) \right) .$$

we can simplify the probability of obtaining the same number of heads and find that is is approximately equal to

$$\frac{1}{\sqrt{\pi n}} \left( 1 + O(\frac{1}{n}) \right) .$$

Another way to approach this problem is to explicitly sum the probability that professor Rosencrantz has $k$ successes while professor Guildenstern has $n-k$ successes. The probability that both these events happen (since they are independent) is given by the product of the appropriate binomial distribution i.e.

$$b(k; n, 1/2) b(n-k; n, 1/2) .$$

So the total probability of that the two professors get the same number of heads is given by the sum of that probability for all possible $k$'s ($k=0$ to $k=n$) or

$$\sum_{k=0}^{n} b(k; n, 1/2) b(n-k; n, 1/2) .$$

This expression simplifies in the obvious ways

$$\begin{aligned} \sum_{k=0}^{n} b(k; n, 1/2) b(n-k; n, 1/2) &= \sum_{k=0}^{n} b(k; n, 1/2)^2 \\ &= \sum_{k=0}^{n} \left( \begin{array}{c} n \\ k \end{array} \right)^2 \left( \frac{1}{2} \right)^{2n} \\ &= \frac{1}{4^n} \sum_{k=0}^{n} \left( \begin{array}{c} n \\ k \end{array} \right)^2 . \end{aligned}$$

In the above we have used the symmetry of the binomial distribution with respect to $k$ and $n-k$, i.e. $b(k;n,1/2) = b(n-k;n,1/2)$, which a special case of the result shown in Exercise C.4-3 above. Equating these two results we have a nice combinatorial identity

$$\sum_{k=0}^{n} \left( \begin{array}{c} n \\ k \end{array} \right)^2 = \left( \begin{array}{c} 2n \\ n \end{array} \right) ,$$

as claimed.

**Exercise C.4-7 (the entropy bound on the binomial distribution)**

From the binomial bound derived in Appendix C.1 (also see the notes above that go with that section) we have that

$$\left( \begin{array}{c} n \\ k \end{array} \right) \leq 2^{nH(\frac{n}{k})} .$$

Using this expression we can immediately bound $b(k;n,1/2)$ as

$$\begin{aligned} b(k;n,1/2) &= \left( \begin{array}{c} n \\ k \end{array} \right) \left(\frac{1}{2}\right)^k \left(\frac{1}{2}\right)^{n-k} \\ &= \left( \begin{array}{c} n \\ k \end{array} \right) \left(\frac{1}{2}\right)^n \\ &\leq 2^{nH(\frac{n}{k})} \left(\frac{1}{2}\right)^n \\ &= 2^{nH(\frac{n}{k})-n} , \end{aligned}$$

which is the desired upper bound on $b(k;n,1/2)$.

**Exercise C.4-8 (sums of Bernoulli random variables with different values of $p$)**

Since $X$ is a random variable representing the total number of success in $n$ Bernoulli trials (with each trial having $p_i$ as its probability of success) we can explicitly express $X$ as the sum of $n$ indicator random variables $I_i$ as

$$X = \sum_{i=1}^{n} I_i .$$

Here where $I_i$ is *one* if the $i$th Bernoulli trial is a success and zero otherwise. Then since the definition of $P\{X < k\}$ means the sum of the probability that $X$ takes the values $0, 1, \ldots, k-1$ or

$$P\{X < k\} = \sum_{i=0}^{k-1} P\{X = i\} ,$$

the probability we are attempting to bound is given in terms of "equality" probabilities i.e. expressions like $P\{X = i\}$. Now for the random variable $X$ to be *exactly* $i$ means that only

$i$ of the Bernoulli trials were a success (no more and no less) and the remaining trials were a failure. Thus if we select a subset of size $i$ from the $n$ total Bernoulli random variables, the above probability is expressed as the sum over all such subsets of size $i$ i.e.

$$P\{X = i\} = \sum p_{l_1} p_{l_2} \cdots p_{l_i} (1 - p_{l_{i+1}})(1 - p_{l_{i+2}}) \cdots (1 - p_{l_n}) \,. \tag{5}$$

Here the sum is over all possible subsets of size $i$ from $n$ and the indices $l_i$ select which of the $i$ Bernoulli indicator variables from $n$ were successful. Now since $p_i \leq p$ it follows that $1 - p_i \geq 1 - p$ and thus using these for each factor in the product we have that the term in the sum above is bounded as

$$p_{l_1} p_{l_2} \cdots p_{l_i} (1 - p_{l_{i+1}})(1 - p_{l_{i+2}}) \cdots (1 - p_{l_n}) \leq p^i (1 - p)^{n-i} \,, \tag{6}$$

Since there are $\left( \begin{array}{c} n \\ i \end{array} \right)$ total terms in the sum above we see that

$$P\{X = i\} \leq \left( \begin{array}{c} n \\ i \end{array} \right) p^i (1 - p)^{n-i} = b(i; n, p) \,.$$

Thus using this we have just shown that

$$P\{X < k\} = \sum_{i=0}^{k-1} P\{X = i\} \leq \sum_{i=0}^{k-1} b(i; n, p) \,,$$

which is the desired result.

**Exercise C.4-9 (a lower bound on our success)**

We can work this exercise in much the same way as the previous exercise C.4-8. Here we can still write $P\{X = i\}$ in the form as given by Equation 5. Rather than use a single value of $p$ such that $p \geq p_i$ for all $i$ we now have several $p'_i$ where $p'_i \geq p_i$ and thus Equation 6 in this case becomes

$$p_{l_1} p_{l_2} \cdots p_{l_i} (1 - p_{l_{i+1}})(1 - p_{l_{i+2}}) \cdots (1 - p_{l_n}) \leq p'_{l_1} p'_{l_2} \cdots p'_{l_i} (1 - p'_{l_{i+1}})(1 - p'_{l_{i+2}}) \cdots (1 - p'_{l_n}) \,.$$

Thus

$$\begin{aligned} P\{X = i\} &= \sum p_{l_1} p_{l_2} \cdots p_{l_i} (1 - p_{l_{i+1}})(1 - p_{l_{i+2}}) \cdots (1 - p_{l_n}) \\ &\leq \sum p'_{l_1} p'_{l_2} \cdots p'_{l_i} (1 - p'_{l_{i+1}})(1 - p'_{l_{i+2}}) \cdots (1 - p'_{l_n}) = P\{X' = i\} \,. \end{aligned}$$

The cumulative probabilities now follow. We have

$$P\{X' \geq k\} = \sum_{i=k}^{n} P\{X' = i\} \geq \sum_{i=k}^{n} P\{X = i\} = P\{X \geq k\} \,,$$

which is the desired result.

## Appendix C.5 (The tails of the binomial distribution)

### Exercise C.5-1 (flip $n$ tails or obtain fewer than $n$ heads)

We know that the probability of obtaining $n$ failures when flipping a fair coin is given by

$$\left(\frac{1}{2}\right)^n$$

This is to be compared to the probability of flipping fewer than $n$ heads when we flip our fair coin $4n$ times or

$$\sum_{i=0}^{n-1} b(i; 4n, 1/2)$$

From Corollary C.5 we have that (with $X$ the number of success obtained in our $4n$ flips) that

$$P\{X < n\} = \sum_{i=0}^{n-1} b(i; 4n, 1/2) < b(n; 4n, 1/2)\,.$$

Now for $b(n; 4n, 1/2)$ we have

$$\begin{aligned} b(n; 4n, 1/2) &= \binom{4n}{n} \left(\frac{1}{2}\right)^n \left(\frac{1}{2}\right)^{4n-n} \\ &= \frac{4n!}{3n!n!} \left(\frac{1}{2}\right)^{4n} \\ &= \frac{(4n)(4n-1)(4n-2)(4n-3)\cdots(3n+2)(3n+1)}{n!} \left(\frac{1}{2}\right)^{4n}\,. \end{aligned}$$

Since the numerator in the above has $n$ factors we can break this factor up into $n$ products (each of which is less than 4) like the following

$$\left(\frac{4n}{n}\right)\left(\frac{4n-1}{n-1}\right)\left(\frac{4n-2}{n-2}\right)\left(\frac{4n-3}{n-3}\right)\cdots\left(\frac{3n+2}{2}\right)\left(\frac{3n+1}{1}\right)\,.$$

Therefore since each term in the above is bounded by 4 and there are $n$ of them we can further bound $b(n; 4n, 1/2)$ as follows

$$b(n; 4n, 1/2) < 4^n \left(\frac{1}{2}\right)^{4n} = \left(\frac{1}{4}\right)^n\,.$$

Thus since

$$\sum_{i=0}^{n-1} b(i; 4n, 1/2) < \left(\frac{1}{4}\right)^n < \left(\frac{1}{2}\right)^n\,,$$

we see that obtaining fewer than $n$ heads in $4n$ flips of a fair coin is less likely then obtaining no heads when flipping a fair coin $n$ times.

**Exercise C.5-2 (bounds on the right tail of the binomial distribution)**

We begin by proving Corollary C.6 which claims that given a sequence of $n$ Bernoulli trials each with probability $p$ of success that

$$P\{X > k\} = \sum_{i=k+1}^{n} b(i;n,p) < \frac{(n-k)p}{k-np} b(k;n,p)\,.$$

To prove this lets first consider the ratio $b(i+1;n,p)/b(i;n,p)$ which is given by

$$\begin{aligned}\frac{b(i+1;n,p)}{b(i;n,p)} &= \frac{\binom{n}{i+1} p^{i+1}(1-p)^{n-i-1}}{\binom{n}{i} p^{i}(1-p)^{n-i}} \\ &= \left(\frac{n-i}{i+1}\right)\frac{p}{1-p}\,.\end{aligned}$$

Thus (since we assume that $i$ is taken between $k$ and $n$ i.e. that $k < i < n$) we have

$$\frac{b(i+1;n,p)}{b(i;n,p)} \leq \left(\frac{n-k}{k+1}\right)\left(\frac{p}{1-p}\right)\,.$$

Defining $x$ to be the above expression we see that $x < 1$ through the following arguments

$$\begin{aligned} x &< \left(\frac{n-np}{np+1}\right)\left(\frac{p}{1-p}\right) \\ &= \frac{n(1-p)p}{(np+1)(1-p)} = \frac{np}{np+1} \\ &< \frac{np}{np+1} < \frac{np}{np} = 1\,.\end{aligned}$$

Now using the ratio above and the definition of $x$ we have that $b(i+1;n,p)$ is bounded by $b(i;n,p)$ as

$$b(i+1;n,p) < x b(i;n,p)\,.$$

In the same way, $b(i+2;n,p)$ is bounded by $b(i;n,p)$ as

$$b(i+2;n,p) < x b(i+1;n,p) < x^2 b(i;n,p)\,.$$

Continuing this iteration scheme for $i+3, i+4, \cdots$ we see that the probability $X > k$ (the right tail probability) is bounded above by

$$\sum_{i=k+1}^{n} b(i;n,p) < x b(k;n,p) + x^2 b(k;n,p) + \cdots + x^n b(k;n,p)\,.$$

This can be further manipulated (by summing to infinity) as follows

$$\begin{aligned}\sum_{i=k+1}^{n} b(i;n,p) &< b(k;n,p)\sum_{i=1}^{n} x^i \\ &< b(k;n,p)\sum_{i=1}^{\infty} x^i \\ &= b(k;n,p)\left(\frac{x}{1-x}\right)\,.\end{aligned}$$

Remembering the definition of $x$ we can compute that

$$\frac{x}{1-x} = \frac{(n-k)p}{1-p+k-np}\,,$$

which is itself less than $\frac{(n-k)p}{k-np}$ so we can finally conclude that

$$\sum_{i=k+1}^{\infty} b(i;n,p) \leq \frac{(n-k)p}{k-np} b(k;n,p)\,,$$

as we were asked to show.

We next prove Corollary C.7 which is the statement that given a sequence of $n$ Bernoulli trials each with probability $p$ of being a success and a specific number of success $k$ in the right tail of our distribution ($\frac{np+n}{2} < k < n$) that the probability of obtaining more than $k$ success is less than one half the probability of obtaining more than $k-1$ success. This is really a statement that as we require having more and more successes from our $n$ Bernoulli trials the probabilities decrease geometrically (with rate 1/2). We will begin by showing that the coefficient of $b(k;n,p)$ in Corollary C.6 is less than one. We have since $\frac{np+n}{2} < k < n$ that

$$\left(\frac{n-k}{k-np}\right) p < \left(\frac{n-\frac{np+n}{2}}{\frac{np+n}{2}-np}\right) p = p < 1\,.$$

Thus from Corollary C.6 we have that

$$P\{X > k\} = \sum_{i=k+1}^{n} b(i;n,p) < b(k;n,p)\,,$$

or equivalently that

$$\frac{b(k;n,p)}{\sum_{i=k+1}^{n} b(i;n,p)} > 1\,.$$

Now consider the ratio of $P\{X > k\}$ to $P\{X > k-1\}$

$$\begin{aligned}
\frac{P\{X > k\}}{P\{X > k-1\}} &= \frac{\sum_{i=k+1}^{n} b(i;n,p)}{\sum_{i=k}^{n} b(i;n,p)} \\
&= \frac{\sum_{i=k+1}^{n} b(i;n,p)}{b(k;n,p) + \sum_{i=k+1}^{n} b(i;n,p)} \\
&= \frac{1}{1 + \frac{b(k;n,p)}{\sum_{i=k+1}^{n} b(i;n,p)}} \\
&< \frac{1}{1+1} = \frac{1}{2}\,,
\end{aligned}$$

and the desired result is proven.

**Exercise C.5-3 (upper bounds on a geometric sum)**

Let $a > 0$ be given, and define $p = \frac{a}{a+1} < 1$, then

$$q = 1 - p = 1 - \frac{a}{a+1} = \frac{1}{a+1}\,.$$

Now consider the $pq$ product found in the binomial coefficient $b(i;n,p) = \left( \begin{array}{c} n \\ i \end{array} \right) p^i q^{n-i}$, i.e. $p^i q^{n-i}$. With the definition given above for $p$ and $q$ we have

$$p^i q^{n-i} = \left(\frac{a}{a+1}\right)^i \left(\frac{1}{a+1}\right)^{n-i} = \frac{a^i}{(a+1)^n} .$$

From this we see that $a^i = (a+1)^n p^i q^{n-i}$ and thus our desired sum is given by

$$\sum_{i=0}^{k-1} \left( \begin{array}{c} n \\ i \end{array} \right) a^i = (a+1)^n \sum_{i=0}^{k-1} \left( \begin{array}{c} n \\ i \end{array} \right) p^i q^{n-i} .$$

Which can be seen as the left tail of the binomial distribution for which we have developed bounds for in the text. Specifically if $X$ is a binomial random variable (with parameters $(n,p)$) then using the bounds that

$$P\{X < k\} = \sum_{i=0}^{k-1} b(i;n,p) < \frac{kq}{np-k} b(k;n,p) ,$$

the above becomes

$$\begin{aligned} \sum_{i=0}^{k-1} b(i;n,\frac{a}{a+1}) &\leq \frac{k\left(\frac{1}{a+1}\right)}{n\left(\frac{a}{a+1}\right)-k} b(k;n,\frac{a}{a+1}) \\ &= \frac{k}{na-k(a+1)} b(k;n,\frac{a}{a+1}) . \end{aligned}$$

Which gives in summary (including the factor $(a+1)^n$) then that

$$\sum_{i=0}^{k-1} \left( \begin{array}{c} n \\ i \end{array} \right) a^i \leq (a+1)^n \frac{k}{na-k(a+1)} b(k;n,\frac{a}{a+1}) ,$$

as we were requested to show.

**Exercise C.5-4 (an upper bound on a geometric like sum)**

Consider the sum we are asked to study

$$\sum_{i=0}^{k-1} p^i q^{n-i} .$$

Since $1 \leq \left( \begin{array}{c} n \\ i \end{array} \right)$ for $i = 0, 1, 2, \cdots, n$ the above sum is less than or equal to

$$\sum_{i=0}^{k-1} \left( \begin{array}{c} n \\ i \end{array} \right) p^i q^{n-i} = \sum_{i=0}^{k-1} b(i;n,p) ,$$

which by Theorem C.4 is bounded above by

$$\left(\frac{kq}{np-k}\right) b(k;n,p)\,.$$

From Lemma C.1 we can further bound the individual binomial coefficient $b(k;n,p)$ as

$$b(k;n,p) \leq \left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k}\,.$$

Thus incorporating each of these bounds we have that

$$\sum_{i=0}^{k-1} p^i q^{n-i} < \left(\frac{kq}{np-k}\right)\left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k}\,,$$

as we were requested to show.

**Exercise C.5-5 (bounds on the number of failures)**

We begin by defining the random variable $Y$ to be the number of *failures* in $n$ Bernoulli trials where the probability of success for each trial is $p_i$. Then in terms of $X$ (the number of successes in these $n$ Bernoulli trials) we have that $Y = n - X$, since each success is *not* a failure and vice versa. This expression simply states that if you know the number of success (failures) in a sequence of $n$ Bernoulli trials it is easy to calculate the number of failures (successes). Taking the expectation of this expression and remembering that we defined $\mu$ as $\mu = E[X]$, we see that $E[Y] = n - E[X] = n - \mu$. Now applying Theorem C.8 to the random variable $Y$ we have that

$$P\{Y - E[Y] \geq r\} \leq \left(\frac{E[Y]e}{r}\right)^r\,.$$

Writing the above expression in terms of $X$ and $\mu$ we have

$$P\{(n-X) - (n-\mu) \geq r\} \leq \left(\frac{(n-\mu)e}{r}\right)^r\,.$$

or

$$P\{\mu - X \geq r\} \leq \left(\frac{(n-\mu)e}{r}\right)^r\,,$$

the desired result.

To show the second identity, we will apply Corollary C.9 to the random variable $Y$, but in this case the probability of success for each Bernoulli trial is a constant $p$. Thus the probability of failure in each Bernoulli trial is also a constant $q = 1 - p$. We thus obtain (since $E[Y] = nq$)

$$P\{Y - nq \geq r\} \leq \left(\frac{nqe}{r}\right)^r\,.$$

Again using the fact that $Y = n - X$ to write the above expression in terms of $X$ we find

$$P\{n - X - nq \geq r\} \leq \left(\frac{nqe}{r}\right)^r ,$$

or

$$P\{n(1-q) - X \geq r\} \leq \left(\frac{nqe}{r}\right)^r ,$$

or

$$P\{np - X \geq r\} \leq \left(\frac{nqe}{r}\right)^r ,$$

the desired result.

**Exercise C.5-6 (an exponential bound on the right tail)**

**Note:** I was not able to prove the inequality

$$p_i e^{\alpha q_i} + q_i e^{-\alpha p_i} \leq e^{\alpha^2/2} ,$$

as suggested in the text. If anyone has such a proof please contact me.

Using the notation in the text we have that (by using Markov's inequality)

$$P\{X - \mu \geq r\} \leq E[e^{\alpha(X-\mu)}]e^{-\alpha r} .$$

Since $X$ is the number of success in $n$ independent Bernoulli trials $X = \sum_{i=1}^n X_i$ the expectation of the exponential can be written as the product of the individual indicator random variables $X_i$ as

$$E[e^{\alpha(X-\mu)}] = \prod_{i=1}^n E[e^{\alpha(X_i - p_i)}] .$$

The explicit expectation of each term in the product is easily computed using the definition of expectation and gives

$$E[e^{\alpha(X_i - p_i)}] = p_i e^{\alpha q_i} + q_i e^{-\alpha p_i} .$$

If we now assume that $p_i e^{\alpha q_i} + q_i e^{-\alpha p_i} \leq e^{\alpha^2/2}$, then the product expectation can be written as

$$E[e^{\alpha(X-\mu)}] \leq \prod_{i=1}^n e^{\alpha^2/2} = e^{n\alpha^2/2} .$$

so that we can bound $P\{X - \mu \geq r\}$ as

$$P\{X - \mu \geq r\} \leq e^{n\alpha^2/2} e^{-\alpha r} = e^{-(\alpha r - n\alpha^2/2)} .$$

To compute the tightest possible upper bound on this probability we can minimize the right hand side of this expression with respect to $\alpha$. This is equivalent to maximizing the expression $\alpha r - n\alpha^2/2$ with respect to $\alpha$. Taking the $\alpha$ derivative of this expression and setting it equal to zero gives

$$r - n\alpha = 0 ,$$

which has $\alpha = r/n$ as its solution. The second derivative of $\alpha r - n\alpha^2/2$ with respect to $\alpha$ begin negative imply that this value of $\alpha$ corresponds to a maximum. The value of this quadratic at this maximal $\alpha$ is given by

$$\frac{r}{n}r - \frac{n}{2}\frac{r^2}{n^2} = \frac{r^2}{2n}\,.$$

This result intern implies that we have an upper bound on $P\{X - \mu \geq r\}$ given by

$$P\{X - \mu \geq r\} \leq e^{-\frac{r^2}{2n}}\,,$$

as we were required to show.

**Exercise C.5-7 (minimizing the upper bound in Markov's inequality)**

The equation C.45 is given by

$$P\{X - \mu \geq r\} \leq \exp(\mu e^\alpha - \alpha r)\,.$$

If we consider the right-hand side of this expression to be an function of $\alpha$, we can find the $\alpha$ that minimizes this function by taking the first derivative and setting it equal to zero. We find that the first derivative is given by

$$\frac{d}{d\alpha}\exp(\mu e^\alpha - \alpha r) = \exp(\mu e^\alpha - \alpha r)\,(\mu e^\alpha - r)\,.$$

Which when we set this equal to zero and solve for $\alpha$ gives

$$\alpha = \ln(\frac{r}{\mu})\,.$$

We can check that this point is indeed a minimum of our right-hand side by computing the sign of the second derivative at that point. The second derivative is given by

$$\begin{aligned}\frac{d^2}{d\alpha^2}\exp(\mu e^\alpha - \alpha r) &= \exp(\mu e^\alpha - \alpha r)(\mu e^\alpha - r)^2 + \exp(\mu e^\alpha - \alpha r)(\mu e^\alpha) \\ &= \exp(\mu e^\alpha - \alpha r)(\mu^2 e^{2\alpha} - 2\mu r e^\alpha + r^2 + \mu e^\alpha) \\ &= \exp(\mu e^\alpha - \alpha r)(\mu^2 e^{2\alpha} - (2r-1)\mu e^\alpha + r^2)\,.\end{aligned}$$

Evaluating this expression at $\alpha = \ln(r/\mu)$ (and ignoring the exponential factor which does not affect the sign of the second derivative) gives

$$\mu^2\left(\frac{r^2}{\mu^2}\right) - (2r-1)r + r^2 = r^2 - 2r^2 + r + r^2 = r > 0\,.$$

Proving that for $\alpha = \ln(r/\mu)$ the expression $\exp(\mu e^\alpha - \alpha r)$ is a minimum.

**Problem C-1 (Balls and bins)**

**Part (a):** We have $b$ choices for the location of the first ball, $b$ choices for the location of the second ball, $b$ choices for the location of the third ball and so on down to $b$ choices for the $n$-th ball. Since the balls are distinguishable each of these placements represent different configurations. Thus the total number of possible configurations in this case is $b^n$.

**Part (b):** Following the hint, since we have a total of $n$ distinguishable balls and $b$ bins we can imagine the requested problem as equivalent to that of counting the number of configurations of $n$ balls and $b-1$ sticks. The $b-1$ sticks represents the internal bin edges and the linear ordering of the balls between sticks represents the ball ordering within the bin. Now to count this number we note that we can order $n+b-1$ distinguishable objects in $(n+b-1)!$ ways, but since the $b-1$ sticks are indistinguishable we need to divide by the number of unique orderings of $b-1$ objects giving a total count of

$$\frac{(n+b-1)!}{(b-1)!}\,.$$

**Part (c):** If the balls in each bin are in fact identical in the same manner in which we needed to divide $(n+b-1)!$ by the number of unique orderings of distinguishable internal sticks $(b-1)!$ we need to divide the result above by the number of unique orderings of these $n$ balls or $n!$. This gives

$$\frac{(n+b-1)!}{(b-1)!n!} = \left( \begin{array}{c} n+b-1 \\ n \end{array} \right).$$

**Part (d):** We assume in this part that we have more bins than balls or $b > n$. If we assume for the moment that the balls are *not* identical then we have $b$ choices for the location of the first ball, $b-1$ choices for the location of the second ball, $b-2$ choices for the location of the third ball and so on down to $b-n+1$ choices for the $n$-th ball. This gives

$$b(b-1)(b-2)\cdots(b-n+2)(b-n+1) = \frac{b!}{(b-n)!}\,,$$

Ways to place $n$ unique balls. Since the balls are in fact identical this number over counts the total possible by $n!$. So we have

$$\frac{b!}{(b-n)!n!} = \left( \begin{array}{c} b \\ n \end{array} \right),$$

placements of identical balls where no more than one ball can go in any given bin.

**Part (e):** We assume in this part that we have more balls than bins or $n > b$. Consider all of our $n$ indistinguishable balls lined up in a row and note that our problem is equivalent to that of counting the number of ways we can select $b-1$ "spaces" (which will represent the $b-1$ internal bin boundaries) from all $n-1$ possible spaces in our linear ordering of the balls. The number of ways to select $b-1$ things from a set of $n-1$ is given by

$$\left( \begin{array}{c} n-1 \\ b-1 \end{array} \right),$$

or the claimed number.

**CS161: Design and Analysis of Algorithms** **Summer 2004**

# Problem Set #1 Solutions

## General Notes

- **Regrade Policy:** If you believe an error has been made in the grading of your problem set, you may resubmit it for a regrade. If the error consists of more than an error in addition of points, please include with your problem set a detailed explanation of which problems you think you deserve more points on and why. We reserve the right to regrade your entire problem set, so your final grade may either increase or decrease.
- If you use pseudocode in your assignment, please either actually use psuedocode or include a written explanation of your code. The TA does not want to parse through C or JAVA code with no comments.
- If you fax in your problem set, please make sure you write clearly and darkly. Some problem sets were very difficult to read. Also, make sure that you leave enough margins so that none of your work is cut off.
- The version of the Master Theorem that was given in lecture is slightly different from that in the book. We gave case 2 as

$$f(n) = \Theta(n^{\log_b a} \log^k n) \implies T(n) = \Theta(n^{\log_b a} \log^{k+1} n) \text{ for } k \geq 0$$

  On exams, using the Master Theorem is normally quicker than other methods. But, remember that cases 1 and 3 only apply when $f(n)$ is polynomially smaller or larger, which is different from asymptotically smaller or larger.

1. **[16 points] Ordering By Asymptotic Growth Rates**

   Throughout this problem, you do not need to give any formal proofs of why one function is $\Omega$, $\Theta$, etc... of another function, but please explain any nontrivial conclusions.

   (a) **[10 points]** Do problem 3-3(a) on page 58 of CLRS.

   Rank the following functions by order of growth; that is, find an arrangement $g_1, g_2, \ldots, g_{30}$ of the functions satisfying $g_1 = \Omega(g_2), g_2 = \Omega(g_3), \ldots, g_{29} = \Omega(g_{30})$. Partition your list into equivalence classes such that $f(n)$ and $g(n)$ are in the same class if and only if $f(n) = \Theta(g(n))$.

| | | | | | |
|---|---|---|---|---|---|
| $\lg(\lg^* n)$ | $2^{\lg^* n}$ | $(\sqrt{2})^{\lg n}$ | $n^2$ | $n!$ | $(\lg n)!$ |
| $(\frac{3}{2})^n$ | $n^3$ | $\lg^2 n$ | $\lg(n!)$ | $2^{2^n}$ | $n^{1/\lg n}$ |
| $\ln \ln n$ | $\lg^* n$ | $n \cdot 2^n$ | $n^{\lg \lg n}$ | $\ln n$ | $1$ |
| $2^{\lg n}$ | $(\lg n)^{\lg n}$ | $e^n$ | $4^{\lg n}$ | $(n+1)!$ | $\sqrt{\lg n}$ |
| $\lg^*(\lg n)$ | $2^{\sqrt{2 \lg n}}$ | $n$ | $2^n$ | $n \lg n$ | $2^{2^{n+1}}$ |

**Answer:** Most of the ranking is fairly straightforward. Several identities are helpful:

$$\begin{aligned} n^{\lg\lg n} &= (\lg n)^{\lg n} \\ n^2 &= 4^{\lg n} \\ n &= 2^{\lg n} \\ 2^{\sqrt{2\lg n}} &= n^{\sqrt{2/\lg n}} \\ 1 &= n^{1/\lg n} \\ \lg^*(\lg n) &= \lg^* n - 1 \text{ for } n > 1 \end{aligned}$$

In addition, asymptotic bounds for Stirling's formula are helpful in ranking the expressions with factorials:

$$\begin{aligned} n! &= \Theta(n^{n+1/2}e^{-n}) \\ \lg(n!) &= \Theta(n\lg n) \\ (\lg n)! &= \Theta((\lg n)^{\lg n+1/2}e^{-\lg n}) \end{aligned}$$

Each term gives a different equivalence class, where the $>$ symbol means $\omega$.

$$\begin{array}{ccccccccccc} 2^{2^{n+1}} & > & 2^{2^n} & > & (n+1)! & > & n! & > & e^n & > \\ n\cdot 2^n & > & 2^n & > & (\frac{3}{2})^n & > & \begin{array}{c} n^{\lg\lg n} \\ (\lg n)^{\lg n} \end{array} & > & (\lg n)! & > \\ n^3 & > & \begin{array}{c} n^2 \\ 4^{\lg n} \end{array} & > & \begin{array}{c} n\lg n \\ \lg(n!) \end{array} & > & \begin{array}{c} n \\ 2^{\lg n} \end{array} & > & (\sqrt{2})^{\lg n} & > \\ 2^{\sqrt{2\lg n}} & > & \lg^2 n & > & \ln n & > & \sqrt{\lg n} & > & \ln\ln n & > \\ 2^{\lg^* n} & > & \begin{array}{c} \lg^*(\lg n) \\ \lg^* n \end{array} & > & \lg(\lg^* n) & > & \begin{array}{c} 1 \\ n^{1/\lg n} \end{array} & & & \end{array}$$

(b) **[2 points]** Do problem 3-3(b) on page 58 of CLRS.
Give an example of a single nonnegative function $f(n)$ such that for all functions $g_i(n)$ in part (a), $f(n)$ is neither $O(g_i(n))$ nor $\Omega(g_i(n))$.
**Answer:** $f(n) = (1+\sin n)\cdot 2^{2^{n+2}}$.

(c) **[2 points]** Give an example of a single nonnegative function $f(n)$ such that for all functions $g_i(n)$ in part $(a)$, $f(n) = o(g_i(n))$.
**Answer:** $f(n) = 1/n$.

(d) **[2 points]** Give an example of a single nonnegative function $f(n)$ such that for all functions $g_i(n)$ in part $(a)$, $f(n) = \omega(g_i(n))$.
**Answer:** $f(n) = 2^{2^{2^n}}$.

2. **[16 points] Recurrences**

Give asymptotic upper and lower bounds for $T(n)$ in each of the following recurrences. Make your bounds as tight as possible, and justify your answers. You may assume $T(n)$ is constant for sufficiently small $n$.

(a) **[2 points]** $T(n) = T(9n/10) + n$.

**Answer:** $T(n) = \Theta(n)$. We have $a = 1$, $b = 10/9$, $f(n) = n$ so $n^{\log_b a} = n^0 = 1$. Since $f(n) = n = \Omega(n^{0+\epsilon})$, case 3 of the Master Theorem applies if we can show the regularity condition holds. For all $n$, $af(n/b) = \frac{9n}{10} \leq \frac{9}{10}n = cf(n)$ for $c = \frac{9}{10}$. Therefore, case 3 tells us that $T(n) = \Theta(n)$.

(b) **[2 points]** $T(n) = T(\sqrt{n}) + 1$.

**Answer:** $T(n) = \Theta(\lg \lg n)$. We solve this problem by change of variables. Let $m = \lg n$, and $S(m) = T(2^m)$. Then

$$\begin{aligned} T(n) &= T(n^{1/2}) + 1 \\ T(2^m) &= T(2^{m/2}) + 1 \\ S(m) &= S(m/2) + 1 \end{aligned}$$

We have $a = 1$, $b = 2$, $f(m) = 1$ so $m^{\log_b a} = m^0 = 1$. Since $f(m) = \Theta(1)$, case 2 of the Master Theorem applies and we have $S(m) = \Theta(\lg m)$. To change back to T(n), we have

$$T(n) = T(2^m) = S(m) = \Theta(\lg m) = \Theta(\lg \lg n).$$

Alternatively, we could solve this problem by algebraic substitution.

$$\begin{aligned} T(n) &= T(n^{1/2}) + 1 \\ &= (T(n^{1/4}) + 1) + 1 \\ &\cdots \\ &= T(n^{1/2^k}) + k \end{aligned}$$

When $n^{1/2^k} \leq 2$, we have that $k \geq \lg \lg n$. Then, $T(n) = \Theta(1) + \lg \lg n = \Theta(\lg \lg n)$.

(c) **[2 points]** $T(n) = 4T(n/2) + n^2 \lg n$.

**Answer:** $T(n) = \Theta(n^2 \lg^2 n)$. We have $a = 4$, $b = 2$, $f(n) = n^2 \lg n$ so $n^{\log_b a} = n^2$. Since $f(n) = \Theta(n^2 \lg n)$, case 2 of the Master Theorem applies and $T(n) = \Theta(n^2 \lg^2 n)$.

(d) **[2 points]** $T(n) = 5T(n/5) + n/\lg n$.

**Answer:** $T(n) = \Theta(n \lg \lg n)$. We have $a = 5$, $b = 5$, $f(n) = n/\lg n$ so $n^{\log_b a} = n$. None of the Master Theorem cases may be applied here, since $f(n)$ is neither polynomially bigger or smaller than $n$, and is not equal to $\Theta(n \lg^k n)$ for any $k \geq 0$. Therefore, we will solve this problem by algebraic substitution.

$$\begin{aligned} T(n) &= 5T(n/5) + n/\lg n \\ &= 5(5T(n/25) + (n/5)/(\lg(n/5))) + n/\lg n \end{aligned}$$

$$\begin{aligned} &= 25T(n/25) + n/\lg(n/5) + n/\lg(n) \\ &\cdots \\ &= 5^i T(n/5^i) + \sum_{j=1}^{i-1} n/\lg(n/5^j). \end{aligned}$$

When $i = \log_5 n$ the first term reduces to $5^{\log_5 n}T(1)$, so we have

$$\begin{aligned} T(n) &= n\Theta(1) + \sum_{j=1}^{\lg_5 n-1} (n/(\lg(n/5^{j-1}))) \\ &= \Theta(n) + n \sum_{j=1}^{\lg_5 n-1} (1/(\lg n - (j-1)\lg_2 5)) \\ &= \Theta(n) + n(1/\log_2 5) \sum_{j=1}^{\log_5 n-1} (1/(\log_5 n - (j-1))) \\ &= \Theta(n) + n \log_5 2 \sum_{i=2}^{\log_5 n} (1/i). \end{aligned}$$

This is the harmonic sum, so we have $T(n) = \Theta(n) + c_2 n \ln(\log_5 n) + \Theta(1) = \Theta(n \lg \lg n)$.

(e) **[2 points]** $T(n) = T(n/2) + T(n/4) + T(n/8) + n$.
**Answer:** $T(n) = \Theta(n)$. We solve this problem by guess-and-check. The total size on each level of the recurrance tree is less than $n$, so we guess that $f(n) = n$ will dominate. Assume for all $i < n$ that $c_1 n \le T(i) \le c_2 n$. Then,

$$\begin{aligned} c_1 n/2 + c_1 n/4 + c_1 n/8 + kn \le \quad & T(n) \quad \le c_2 n/2 + c_2 n/4 + c_2 n/8 + kn \\ c_1 n(1/2 + 1/4 + 1/8 + k/c_1) \le \quad & T(n) \quad \le c_2 n(1/2 + 1/4 + 1/8 + k/c_2) \\ c_1 n(7/8 + k/c_1) \le \quad & T(n) \quad \le c_2 n(7/8 + k/c_2) \end{aligned}$$

If $c_1 \ge 8k$ and $c_2 \le 8k$, then $c_1 n \le T(n) \le c_2 n$. So, $T(n) = \Theta(n)$.

In general, if you have multiple recursive calls, the sum of the arguments to those calls is less than $n$ (in this case $n/2 + n/4 + n/8 < n$), and $f(n)$ is reasonably large, a good guess is $T(n) = \Theta(f(n))$.

(f) **[2 points]** $T(n) = T(n-1) + 1/n$.
**Answer:** $T(n) = \Theta(\lg n)$. We solve this problem by algebraic substitution.

$$\begin{aligned} T(n) &= T(n-1) + 1/n \\ &= T(n-2) + 1/(n-1) + 1/n \\ &\cdots \\ &= \Theta(1) + \sum_{i=1}^{n} 1/i \\ &= \ln n + \Theta(1) \end{aligned}$$

(g) **[2 points]** $T(n) = T(n-1) + \lg n$.
**Answer:** $T(n) = \Theta(n \lg n)$. We solve this problem by algebraic substitution.

$$\begin{aligned} T(n) &= T(n-1) + \lg n \\ &= T(n-2) + \lg(n-1) + \lg n \\ &\cdots \\ &= \Theta(1) + \sum_{i=1}^{n} \lg i \\ &= \Theta(1) + \lg(\prod_{i=1}^{n} i) \\ &= \Theta(1) + \lg(n!) \\ &= \Theta(n \lg n) \end{aligned}$$

(h) **[2 points]** $T(n) = \sqrt{n}T(\sqrt{n}) + n$.
**Answer:** $T(n) = \Theta(n \lg \lg n)$. We solve this problem by algebraic substitution.

$$\begin{aligned} T(n) &= \sqrt{n}T(\sqrt{n}) + n \\ &= n^{1/2}(n^{1/4}T(n^{1/4}) + n^{1/2}) + n \\ &= n^{3/4}T(n^{1/4}) + 2n \\ &= n^{3/4}(n^{1/8}T(n^{1/8}) + n^{1/4}) + 2n \\ &= n^{7/8}T(n^{1/8}) + 3n \\ &\cdots \\ &= n^{1-1/2^k}T(n^{1/2^k}) + kn \end{aligned}$$

When, $n^{1/2^k}$ falls under 2, we have $k > \lg \lg n$. We then have $T(n) = n^{1-1/\lg n}T(2) + n \lg \lg n = \Theta(n \lg \lg n)$.

3. **[10 points] Integer Multiplication**

Let $u$ and $v$ be two $n$-bit numbers, where for simplicity $n$ is a power of 2. The traditional multiplication algorithm requires $\Theta(n^2)$ operations. A divide-and-conquer based algorithm splits the numbers into two equal parts, computing the product as

$$uv = (a2^{n/2} + b)(c2^{n/2} + d) = ac2^n + (ad + bc)2^{n/2} + bd$$

Here, $a$ and $c$ are the higher order bits, and $b$ and $d$ are the lower order bits of $u$ and $v$ respectively. Multiplications are done recursively, except multiplication by a power of 2, which is a simple bitshift and takes $\Theta(n)$ time for an $n$-bit number. Addition and subtraction also take $\Theta(n)$ time.

(a) **[4 points]** Write a recurrence for the running time of this algorithm as stated. Solve the recurrence and determine the running time.

**Answer:** $T(n) = 4T(n/2) + \Theta(n) = \Theta(n^2)$. We divide the problem into 4 subproblems ($ac$, $ad$, $bc$, $bd$) of size $n/2$ each. The dividing step takes constant time, and the recombining step involves adding and shifting $n$-bit numbers and therefore takes time $\Theta(n)$. This gives the recurrence relation $T(n) = 4T(n/2) + \Theta(n)$. For this case, we have $a = 4$, $b = 2$, $f(n) = n$ so $n^{\log_b a} = n^2$. $f(n) = O(n^{\log_b a - \epsilon})$ so case 1 of the Master Theorem applies. Therefore, $T(n) = \Theta(n^2)$.

(b) **[6 points]** You now notice that $ad+bc$ can be computed as $(a+b)(c+d)-ac-bd$. Why is this advantageous? Write and solve a recurrence for the running time of the modified algorithm.

**Answer:** $T(n) = 3T(n/2) + \Theta(n) = \Theta(n^{\log_2 3})$. If we write $ad + bc$ as $(a + b)(c + d) - ac - bd$, then we only need to compute three integer multiplications of size $n/2$, namely $ac$, $bd$, and $(a + c)(c + d)$. This is advantageous since we replace one multiplication with additions and subtractions, which are less expensive operations. The divide step will now take time $\Theta(n)$, since we need to calculate $a + c$ and $c + d$, and the recombining step will still take $\Theta(n)$. This leads to the recurrence relation $T(n) = 3T(n/2) + \Theta(n)$. We have $a = 3$, $b = 2$, and $f(n) = n$ so $n^{\log_b a} \approx n^{1.585}$. $f(n) = O(n^{1.585-\epsilon})$ so case 1 of the Master Theorem applies. Therefore, $T(n) = \Theta(n^{\lg 3})$.

4. **[20 points] Stable Sorting**

Recall that a sorting algorithm is **stable** if numbers with the same value appear in the output array in the same order as they do in the input array. For each sorting algorithm below, decide whether it is stable or not stable. If the algorithm is stable, give a formal proof of its stability (using loop invariants if necessary). If the algorithm is unstable, suggest a way to modify the algorithm to make it stable. This modification should not alter the fundamental idea of the algorithm or its running time! Explain (informally) why your modifications make the algorithm stable.

(a) **[6 points]** Insertion Sort: pseudocode given on page 17 of CLRS.

```
INSERTION-SORT(A)
1   for j ← 2 to length[A]
2       do key ← A[j]
3          ▷ Insert A[j] into the sorted sequence A[1 . . . j − 1]
4          i ← j − 1
5          while i > 0 and A[i] > key
6             do A[i + 1] ← A[i]
7                i ← i − 1
8          A[i + 1] ← key
```

**Answer:** Insertion sort is stable. We will prove this using the following loop invariant: At the start of each interation of the for loop of lines 1-8, if $A[a] = A[b], a < b \leq j - 1$ and distinct, then $A[a]$ appeared before $A[b]$ in the initial array.

**Initialization:** Before the first loop iteration, $j = 2$ so there are no distinct

elements $a < b \leq j - 1 = 1$. So, the condition holds trivially.
**Maintenance:** Given that the property holds before any iteration, we need to show that it holds at the end of the iteration. During each iteration, we insert $A[j]$ into the sorted sequence $A[1 \ldots j-1]$. For all $a < b \leq j - 1$, the property holds at the end of the loop since we simply shift elements in the subarray and don't change their respective order. For all $a < b = j$, the property holds at the end of the loop since we insert $j$ after all elements whose value is equal to $A[j]$. This is because the condition for shifting elements to the right is $A[i] > key$ (line 5), so all the elements whose value equals $key$ are not shifted and $A[j]$ is inserted after them. Thus, since we have shown the property is true for all $a < b \leq j - 1$ and $a < b = j$, we have shown it true for all $a < b \leq j$ and so the property is true at the end of the loop.
**Termination:** When the for loop ends, $j = n + 1$, and the loop invariant states that for all $A[a] = A[b], a < b \leq n$, implies $A[a]$ appeared before $A[b]$ in the initial array. This is the definition of stability and since it applies to the entire array $A$, insertion sort is stable.

(b) **[6 points]** Mergesort: pseudocode given on pages 29 and 32 of CLRS.

```
MERGE(A, p, q, r)
 1    n_1 ← q − p + 1
 2    n_2 ← r − q
 3    create arrays L[1 ... n_1 + 1] and R[1 ... n_2 + 1]
 4    for i ← 1 to n_1
 5       do L[i] ← A[p + i − 1]
 6    for j ← 1 to n_2
 7       do R[j] ← A[q + j]
 8    L[n_1 + 1] ← ∞
 9    R[n_2 + 1] ← ∞
10    i ← 1
11    j ← 1
12    for k ← p to r
13       do if L[i] ≤ R[j]
14          then A[k] ← L[i]
15               i ← i + 1
16          else A[k] ← R[j]
17               j ← j + 1
MERGE-SORT(A, p, r)
1   if p < r
2      then q ← ⌊(p + r)/2⌋
3         MERGE-SORT(A, p, q)
4         MERGE-SORT(A, q + 1, r)
5         MERGE(A, p, q, r)
```

**Answer:** Mergesort is stable. We prove this by induction on the fact that Mergesort is stable.

**Base Case:** When we call merge-sort with indices $p$ and $r$ such that $p \not< r$ (therefore $p == r$), we return the same array. So, calling mergesort on an array of size one returns the same array, which is stable.
**Induction:** We assume that calling merge-sort on an array of size less than $n$ returns a stably sorted array. We then show that if we call merge-sort on an array of size $n$, we also return a stably sorted array. Each call to mergesort contains two calls to mergesort on smaller arrays. In addition, it merges these two subarrays and returns. Since we assume that the calls to mergesort on smaller arrays return stably sorted arrays, we need to show that the merge step on two stably sorted arrays returns a stable array. If $A[i] = A[j], i < j$ in the initial array, we need $f(i) < f(j)$ in the new array, where $f$ is the function which gives the new positions in the sorted array. If $i$ and $j$ are in the same half of the recursive merge-sort call, then by assumption, they are in order when the call returns and they will be in order in the merged array (since we take elements in order from each sorted subarray). If $i$ and $j$ are in different subarrays, then we know that $i$ is in the left subarray and $j$ is in the right subarray since $i < j$. In the merge step, we take the elements from the left subarray while the left subarry element is less than or equal to the right subarray element (line 13). Therefore, we will take element $i$ before taking element $j$ and $f(i) < f(j)$, the claim we are trying to prove.
Therefore, Mergesort is stable.

(c) **[8 points]** Quicksort: pseudocode given on page 146 of CLRS.

```
QUICKSORT(A, p, r)
1    if p < r
2       then q ← PARTITION(A, p, r)
3            QUICKSORT(A, p, q − 1)
4            QUICKSORT(A, q + 1, r)

PARTITION(A, p, r)
1    x ← A[r]
2    i ← p − 1
3    for j ← p to r − 1
4       do if A[j] ≤ x
5          then i ← i + 1
6               exchange A[i] ↔ A[j]
7    exchange A[i + 1] ↔ A[r]
8    return i + 1
```

**Answer:** Quicksort is not stable. To make it stable, we can add a new field to each array element, which indicates the index of that element in the original array. Then, when we sort, we can sort on the condition (line 4) $A[j] < x$ OR ($A[j] == x$ AND $index(A[j]) < index(x)$). At the end of the sort, we are guaranteed to have all the elements in sorted order, and for any $i < j$, such that $A[i]$ equals $A[j]$, we will have $index(A[i]) < index(A[j])$ so the sort will be stable.

5. **[22 points] Computing Fibonacci Numbers**

Recall the Fibonacci numbers as defined in lecture and on page 56 of CLRS. Throughout this problem assume that the cost of integer addition, subtraction, multiplication, as well as reading from memory or writing to memory is $\Theta(1)$, independent of the actual size of the numbers. We will compare the efficiency of different algorithms for computing $F_n$, the $n$th Fibonacci number.

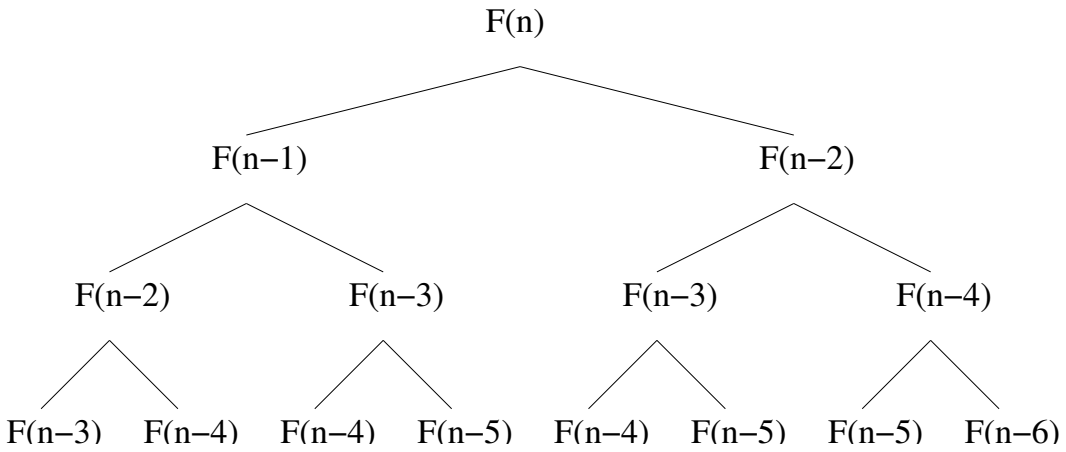
(a) **[5 points]** First, consider a naive recursive algorithm:

```
int Fib (int n) {
  // recursion
  if (n >= 2) return Fib(n-1) + Fib(n-2);

  // base cases
  if (n == 1) return 1;
  if (n == 0) return 0;

  // error if none of the above returns
}
```

Sketch a recursion tree for this algorithm. During the computation of Fib(n), how many times is Fib(n-1) called? Fib(n-2)? Fib(n-3)? Fib(2)? Use formula (3.23) on page 56 of CLRS to conclude that this algorithm takes time exponential in $n$.

$$F_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}}, \text{ where } \phi = \frac{1+\sqrt{5}}{2} \text{ and } \hat{\phi} = \frac{1-\sqrt{5}}{2} \tag{1}$$

**Answer:**




We call $Fib(n-1)$ one time during the recursion, $Fib(n-2)$ 2 times, $Fib(n-3)$ 3 times, $Fib(n-4)$ 5 times, ..., $Fib(2)$ $F_{n-1}$ times. Each node in the recurrence

tree takes constant time to evaluate, so we need to calculate the number of nodes in the tree. $T(n) = \Theta(\sum_{i=1}^{n} F_i)$. Using formula 3.23 in the book, we get

$$T(n) = \Theta(\sum_{i=1}^{n} \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}})$$

As $n$ grows large, the $\hat{\phi}^i$ term goes to zero, so we are left with

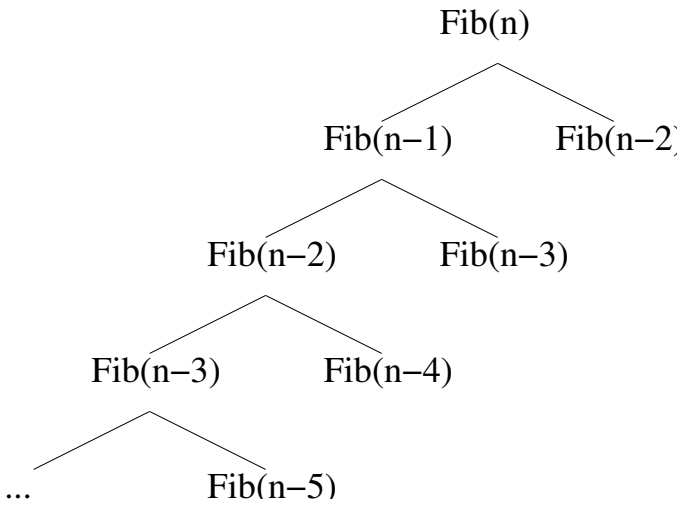
$$T(n) = \Theta(\sum_{i=1}^{n} \frac{\phi^i}{\sqrt{5}}) = \Omega(\phi^n)$$

So, the algorithm takes time exponential in $n$.

(b) **[5 points]** Notice that our recursion tree grows exponentially because the same computation is done many times over. We now modify the naive algorithm to only compute each Fib(i) once, and store the result in an array. If we ever need Fib(i) again, we simply reuse the value rather than recomputing it. This technique is called **memoization** and we will revisit it when we cover dynamic programming later in the course.

Prune your recursion tree from part (a) to reflect this modification. Considering the amount of computation done at each node is $\Theta(1)$, what is the total amount of computation involved in finding Fib(n)?

**Answer:**




As in part (a), the time to calculate the $n$th Fibonacci number $F_n$ is equal to the time to calculate $F_{n-1}$, plus the time to calculate $F_{n-2}$, plus the time to add $F_{n-1}$ to $F_{n-2}$. However, because of memoization, the time to calculate $F_{n-2}$ is constant once $F_{n-1}$ has already been calculated. Therefore, the recurrence for this modified algorithm is $T(n) = T(n-1) + \Theta(1)$. It is easy to use the algebraic substitution method to verify that the solution to this recurrence is $T(n) = \Theta(n)$.

(c) **[4 points]** Prove that

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$

for $n = 1, 2, 3, \ldots$ .

**Hint:** The last line of the question should immediately suggest what proof technique would be useful here.

**Answer:** We prove the statement by mathematical induction. For the base case of $n = 1$, we get

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^1 = \begin{pmatrix} F_2 & F_1 \\ F_1 & F_0 \end{pmatrix} \tag{2}$$

which certainly holds. Now, we assume the statement holds for $n-1$. Then,

$$\begin{aligned} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \\ &= \begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \\ &= \begin{pmatrix} F_n + F_{n-1} & F_n \\ F_{n-1} + F_{n-2} & F_{n-1} \end{pmatrix} \\ &= \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} \end{aligned}$$

Thus, we have shown that the statement holds for all $n$.

(d) **[8 points]** Use part (c) or formula (3.23) to design a simple divide-and-conquer algorithm that computes the $n$th Fibonacci number $F_n$ in time $\Theta(\lg n)$.

**Answer:** Let $A$ be the matrix $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ of part (c). To compute the $n$th Fibonacci number $F_n$, we can simply exponentiate the matrix $A$ to the $n$th power and take the element $A^n_{2,1}$. An efficient divide-and-conquer algorithm for this problem uses the notion of **repeated squaring**. In other words, to calculate $A^n$, observe that

$$A^n = \begin{cases} (A^{n/2})^2 & \text{if } n \text{ is even} \\ (A^{\lfloor n/2 \rfloor})^2 \cdot A & \text{if } n \text{ is odd} \end{cases}$$

This algorithm satisfies the recurrence $T(n) = T(n/2) + \Theta(1)$ because we consider multiplying two $2 \times 2$ matrices as a constant time operation and because we have reduced a problem of size $n$ to one of size $\lfloor n/2 \rfloor$. We can solve this recurrence using the master theorem case 2, and arrive at a solution of $T(n) = \Theta(\lg n)$.

6. **[26 points] Algorithm Design**

(a) **[10 points]** Design a $\Theta(n \lg n)$-time algorithm that, given an array $A$ of $n$ integers and another integer $x$, determines whether or not there exist two (not necessarily distinct) elements in $A$ whose sum is exactly $x$. This is problem 2.3-7 on page 37 of CLRS, slightly reworded for the sake of clarity.

**Answer:** We first sort the elements in the array using a sorting algorithm such as merge-sort which runs in time $\Theta(n \lg n)$. Then, we can find if two elements exist in $A$ whose sum is $x$ as follows. For each element $A[i]$ in $A$, set $y = A[i] - x$. Using binary search, find if the element $y$ exists in $A$. If so, return $A[i]$ and $y$. If we can't find $y$ for any $A[i]$, then return that no such pair of elements exists. Each binary search takes time $\Theta(\lg n)$, and there are $n$ of them. So, the total time for this procedure is $T(n) = \Theta(n \lg n) + \Theta(n \lg n)$ where the first term comes from sorting and the second term comes from performing binary search for each of the $n$ elements. Therefore, the total running time is $T(n) = \Theta(n \lg n)$.

An alternate procedure to find the two elements in the sorted array is given below:

```
SUM-TO-X(A)
 1    Merge-Sort(A)
 2    i ← 1
 3    j ← length(A)
 4    while i ≤ j
 5       if A[i] + A[j] equals x
 6          return A[i], A[j]
 7       if A[i] + A[j] < x
 8          then i ← i + 1
 9       if A[i] + A[j] > x
10          then j ← j − 1
```

We set counters at the two ends of the array. If their sum is $x$, we return those values. If the sum is less than $x$, we need a bigger sum so we increment the bottom counter. If the sum is greater than $x$, we decrement the top counter. The loop does $\Theta(n)$ iterations, since at each iteration we either increment $i$ or decrement $j$ so $j - i$ is always decreasing and we terminate when $j - i < 0$. However, this still runs in time $\Theta(n \lg n)$ since the running time is dominated by sorting.

(b) **[16 points]** The majority element of an array $A$ of length $n$ is the element that appears **strictly more than** $\lfloor n/2 \rfloor$ times in the array. Note that if such an element exists, it must be unique.

Design a $\Theta(n)$-time algorithm that, given an array $A$ of $n$ objects, finds the majority element or reports that no such element exists. Assume that two objects can be compared for equality, read, written and copied in $\Theta(1)$ time, but **no other operations are allowed** (for instance, there is no '<' operator). Explain why your algorithm is correct (no formal proof necessary).

**Hint:** It is possible to find a candidate solution using only one pass through the array, with the help of an auxiliary data structure such as a stack or a queue. Then with one more pass you can determine whether this candidate is indeed the majority element.

**Answer:** We will give an algorithm to solve this problem which uses an auxiliary

stack and a proof of why it is correct. The following is the pseudocode for the algorithm.

```
FIND-MAJORITY-ELEMENT(A)
 1   for i ← 1 to length[A]
 2       if stack is empty
 3           then push A[i] on the stack
 4       else if A[i] equals top element on the stack
 5           then push A[i] on the stack
 6           else pop the stack
 7   if stack is empty
 8       then return NoMajority
 9   candidate ← top element on the stack
10   counter ← 0
11   for i ← 1 to length[A]
12       if A[i] equals candidate
13           then counter ← counter + 1
14   if counter > ⌊length[A]/2⌋
15       then return candidate
16   return NoMajority
```

The procedure first generates a candidate object from the array. It finds this element using the stack by looping over the array. If the stack is empty or the current element is the same as the top object on the stack, the element is pushed onto the stack. If the top object of the stack is different from the current element, we pop that object off the stack.

**Claim:** If a majority element exists, it will be on the stack at the end of this part of the procedure.

**Proof:** (by contradiction) Call the majority element $x$ and say it appears $i > \lfloor n/2 \rfloor$ times. Each time $x$ is encountered, it is either pushed on the stack or an element (different from $x$) is popped off the stack. There are $n - i < i$ elements different from $x$. Assume $x$ is not on the stack at the end of the procedure. Then, each of the $i$ elements of $x$ must have either popped another element off the stack, or been popped off by another element. However, there are only $n - i < i$ other elements, so this is a contradiction. Therefore, $x$ must be on the stack at the end of the procedure.

Notice that this proof does not show that if an element is on the stack at the end of the procedure that it is the majority element. We can construct simple counterexamples ($A = [1, 2, 3]$) where this is not the case. Therefore, after obtaining our candidate majority element solution, we check whether it is indeed the majority element (lines 9-16). We do this by scanning through the array and counting the number of times the element *candidate* appears. We return *candidate* if it appears more than $\lfloor n/2 \rfloor$ times, else we report that no majority element exists.

This procedure scans through the array twice and does a constant amount of computation (pushing and popping elements on the stack take constant time) at each step. Therefore the running time is $T(n) = cn = \Theta(n)$.

**CS161: Design and Analysis of Algorithms** **Summer 2004**

# Problem Set #2 Solutions

## General Notes

- **Regrade Policy:** If you believe an error has been made in the grading of your problem set, you may resubmit it for a regrade. If the error consists of more than an error in addition of points, please include with your problem set a detailed explanation of which problems you think you deserve more points on and why. We reserve the right to regrade your entire problem set, so your final grade may either increase or decrease.

- Remember you can only split an expectation of a product into the product of the expectations if the two terms are **independent**. Many students did this on question 3c either without justification or justifying it as linearity of expectation, which is incorrect.

- For the skip list question 4b, many students found the expected number of nodes between $left[i+1].below$ and $right[i+1].below$ by saying the expected number of nodes at level $i+1$ is $np^{i+1}$ and the expected number of nodes at level $i$ is $np^i$. Then, if you divide the number of nodes at $i$ by the number at $i+1$, you get an average interval of $1/p$. While this gives the correct answer, it is not a valid argument as it stands. For example, let's say you always propagate the first nodes in level $i$ - you will get an interval of length 1 with probability $1/p$ and an interval of length $np^i(1-p)$ with probability $1-1/p$, which clearly gives you a different expectation. You need to use properties other than just the expected number of nodes to calculate the expected interval length.

1. **[18 points]** Probability and Conditional Probability

   Throughout this problem, please justify your answers mathematically as well as logically; however, you do not have to give any formal proofs.

   (a) **[5 points]** Do problem C.2-9 on page 1106 of CLRS. Assume that initially, the prize is placed behind one of the three curtains randomly.

   **Answer:** The answer is not 1/2, because now the two curtains are no longer equally likely to hide the prize. In fact, the original 3 cases are still equally likely, but now in two of the cases the prize is behind the curtain that is not revealed or chosen. Therefore, the probability of winning if you stay is 1/3, if you switch is 2/3.

   (b) **[5 points]** Do problem C.2-10 on page 1106 of CLRS.

   **Answer:** This is essentially the same problem, except now there is no option to switch. The prisoners are the curtains, freedom is the prize, and the guard's revelation is equivalent to the emcee's revelation in the previous question. Thus, the probability that $X$ will go free is 1/3.

(c) **[8 points]** Consider the following game using a standard 52-card deck (with 26 red cards and 26 black cards):

i. the deck $D$ is shuffled randomly.

ii. one at a time, the top card of $D$ is removed and revealed until you say "stop" or only one card remains.

iii. if the next card is red, you win the game. if the next card is black, you lose the game.

Consider the following algorithmic strategy for playing this game:

i. count the number of black cards (b) and red cards (r) revealed thus far.

ii. as soon as $b > r$, say "stop".

What is the probability of success for this strategy? Is there a better strategy? Can you prove a useful upper bound on the probability of success for any legal strategy?

**Hint:** Consider the following modification to the game: after you say "stop", rather than looking at the top card of $D$, we look at the bottom card.

**Answer:** For any given strategy $S$, let $Pr_S\{(b,r)\}$ be the probability that we stop after we have seen $b$ black cards and $r$ red cards. Note that some of these may be 0, for example, using the strategy in question we will only stop when $b = r + 1$ or $(b,r) = (25,26)$.

Since we assume all permutations of the deck to be equally likely, the remaining $52-b-r$ cards are in random order, so the probability of success (top remaining card is red) is just

$$Pr_S\{\text{success} \mid (b,r)\} = \frac{26-r}{52-b-r}.$$

Thus, we can write

$$Pr_S\{\text{success}\} = \sum_{(b,r)} Pr_S\{(b,r)\} \cdot \frac{26-r}{52-b-r}.$$

Now notice that $(26-r)/(52-b-r)$ is also the probability that the bottom card is red given that we stop at $(b,r)$. Thus, we can say

$$Pr_S\{\text{success}\} = \sum_{(b,r)} Pr_S\{(b,r)\} \cdot Pr\{\text{bottom} = \text{red} \mid (b,r)\}.$$

Now we manipulate the sum by multiplying the two probabilities:

$$Pr_S\{\text{success}\} = \sum_{(b,r)} Pr\{\text{bottom} = \text{red} \wedge (b,r)\} = Pr\{\text{bottom} = \text{red}\} = 1/2.$$

The argument applies for any legal strategy, thus, regardless of the strategy, we succeed with probability 1/2.

If you are uncomfortable with some of the probabilistic manipulation and reasoning in this solution, please review Appendix C in CLRS, particularly section $C.2$ and Bayes's theorem.

2. [**12 points**] Modifying Quicksort

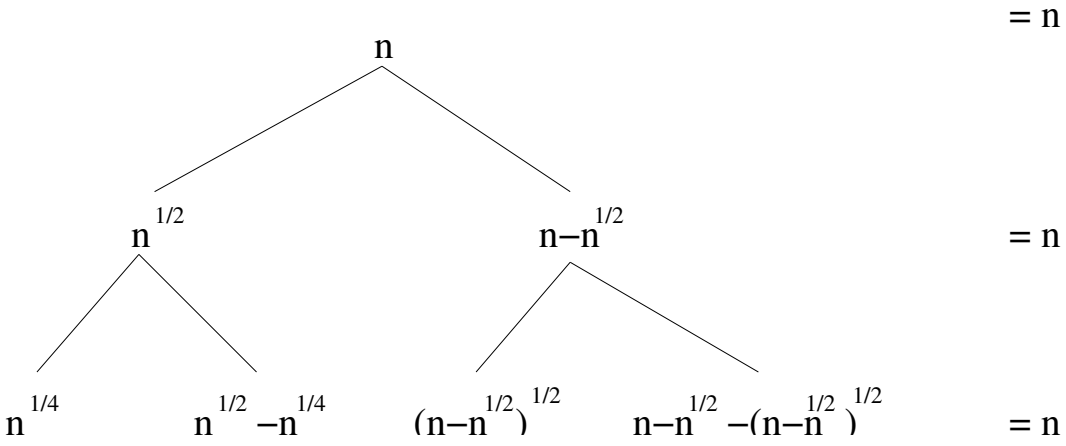
   Consider the following modification to the deterministic quicksort algorithm: instead of using $A[n]$ as pivot, take the last $2\sqrt{n}+1$ elements of $A$, sort them using insertion sort, then use the middle element of these as the pivot. Assume that this guarantees there are at least $\sqrt{n}$ elements in each subarray after partitioning - we will ignore the issue of repeated elements in this problem. As usual, ignore the issue of rounding as well.

   (a) [**3 points**] Write a recurrence for the worst-case running time $T(n)$ of this algorithm.

   **Answer:** Our partition guarantees that there are at least $\sqrt{n}$ elements in each subarray. In the worst case, all the remaining elements will go to one subarray. Then, we will need to solve one subproblem of the size $\sqrt{n}$ and one of the size $n-\sqrt{n}$. Also, we need to sort the $2\sqrt{n}+1$ elements to find the pivot. Using insertion sort, this is a worst case running time of $\Theta((2\sqrt{n}+1)^2)=\Theta(n)$. Then, we find the median of the sorted sample, a $\Theta(1)$ step. In addition, we need to partition the elements at each step, a $\Theta(n)$ operation. So in the worst-case, the recurrence relation will be $T(n)=T(\sqrt{n})+T(n-\sqrt{n})+\Theta(n)$.

   (b) [**2 points**] Draw the recursion tree for the expression you came up with in part (a). Label the work in the top 3 levels of the tree.

   **Answer:**




   (c) [**6 points**] How many levels does it take before the shortest branch reaches a leaf? the longest branch (an asymptotic upper bound is sufficient)?

   **Hint:** Consider writing a recurrence for the height of the longest branch. You may find the approximation $\sqrt{n-\sqrt{n}} \approx \sqrt{n}-1/2$ helpful in guessing a solution for this recurrence.

   **Answer:** The shortest branch is the one where everytime we call $T(\sqrt{n})$. At each level $i$, we will call $T(n^{1/2^i})$. We assume $T(n)$ is a constant for small values of $n$, so we find the value of $i$ for which $n^{1/2^i} \leq 2$. This is true for $i \geq \lg\lg n$. Thus,

the shortest branch reaches a leaf in $\lg \lg n$ steps.
The longest branch is the one where everytime we call $T(n-\sqrt{n})$. At the second level we will call $T(n-\sqrt{n}-\sqrt{n-\sqrt{n}}) \approx T(n-\sqrt{n}-\sqrt{n}-1/2)$. So, at this level, we essentially subtract another $\sqrt{n}$. Therefore, we can guess that at each level $i$, we call $T(n-i\sqrt{n})$ which will be a constant when $i = \sqrt{n}$. We can now check to see that this is in fact an upper bound for the number of levels. Define $h(n)$ to be the number of levels in the recurrence tree. We know that $h(n) = h(n-\sqrt{n})+1$, since we add one level each time we recurse. We guess that $h(i) \leq c\sqrt{i}$ for $i < n$ and check.

$$\begin{aligned} h(n) &= h(n-\sqrt{n})+1 \\ &\leq c\sqrt{n-\sqrt{n}}+1 \\ &\approx c(\sqrt{n}-1/2)+1 \\ &= c\sqrt{n}-c/2+1 \\ &\leq c\sqrt{n} \end{aligned}$$

which is true for $c \geq 2$. Therefore, we know that the longest branch is $O(\sqrt{n})$.

(d) **[1 point]** Using your answers from parts (b) and (c), give a big-O bound on $T(n)$.
**Answer:** $T(n) = O(n^{3/2})$. Looking at the recurrence tree from part (b), we see that we do $\Theta(n)$ work on each level. We showed in part (c) that the longest branch of the recurrence tree had a height of $O(\sqrt{n})$ levels. So, we know the recurrence is bounded by $T(n) = O(n^{3/2})$.

3. **[15 points]** Alternative Quicksort Analysis

In this problem, you will analyze the running time of the randomized quicksort algorithm using a different approach than in lecture. Instead of counting the expected number of comparisons, this approach focuses on the expected running time of each recursive call to randomized quicksort.

Throughout your analysis, please be as formal as possible.

(a) **[2 points]** Do problem 7-2(a) on page 160 of CLRS.
**Answer:** We know that we select pivots from the array uniformly at random. Since there are $n$ elements in the array, and since the sum of the probabilities of choosing any one element as the pivot must equal one, each element must be chosen with probability $1/n$. If we define the indicator random variable $X_i = I\{i^{th}$ smallest element chosen as the pivot$\}$, then $E[X_i]$ is simply $\sum_{j=1}^{n} Pr\{$select $j \wedge j$ is the $i^{th}$ smallest element$\}$. These are independent, so we can break apart the product and get $\sum_{j=1}^{n} Pr\{$select $j\} \cdot Pr\{j$ is the $i^{th}$ smallest element$\} = \sum_{j=1}^{n}(1/n)\cdot(1/n)$, since each element is equally likely to be the $i^{th}$ smallest element. This gives $E[X_i] = 1/n$.

(b) **[3 points]** Do problem 7-2(b) on page 161 of CLRS.

**Answer:** The expected running time of quicksort depends on the sizes of the subarrays after partitioning. We can sum over the sizes resulting from all possible pivots multiplied by the probability of that pivot. If the pivot is at $q$, then the resulting subarrays will have size $q-1$ and $n-q$. In addition, it takes $\Theta(n)$ time to do the partitioning. So, the total expected running time is:
$E[T(n)] = E[\sum_{q=1}^{n} X_q(T(q-1) + T(n-q) + \Theta(n))]$

(c) **[2 points]** Do problem 7-2(c) on page 161 of CLRS.

**Answer:** By linearity of expectation, we can move the summation outside of the expectation.

$$E[T(n)] = \sum_{q=1}^{n} E[X_q(T(q-1) + T(n-q) + \Theta(n))]$$

Because $X_q$ is independent from the time is takes to do the recursive calls, we can split the expectation of the product into the product of expectations.

$$E[T(n)] = \sum_{q=1}^{n} E[X_q] \cdot E[T(q-1) + T(n-q) + \Theta(n)]$$

Substituting in the value for $E[X_q]$ that we solved in part (a) and again changing the expectation of the sum into a sum of expectations, we have

$$\begin{aligned}
E[T(n)] &= \sum_{q=1}^{n} \frac{1}{n}(E[T(q-1)] + E[T(n-q)] + E[\Theta(n)]) \\
&= \frac{1}{n}\sum_{q=1}^{n}(E[T(q-1)] + E[T(n-q)] + \Theta(n)) \\
&= \frac{1}{n}(\sum_{q=1}^{n} E[T(q-1)] + \sum_{q=1}^{n} E[T(n-q)] + \sum_{q=1}^{n} \Theta(n)) \\
&= \frac{1}{n}((E[T(0)] + E[T(1)] + \cdots + E[T(n-1)]) + \\
&\quad (E[T(n-1)] + E[T(n-2)] + \cdots + E[T(0)]) + n\Theta(n)) \\
&= \frac{1}{n}(2\sum_{q=2}^{n-1} E[T(q)]) + \Theta(n)
\end{aligned}$$

since we consider $E[T(0)]$ and $E[T(1)]$ to be $\Theta(1)$.

(d) **[6 points]** Do problem 7-2(d) on page 161 of CLRS.

**Answer:**

$$\begin{aligned}
\sum_{k=2}^{n-1} k \lg k &= \sum_{k=2}^{\lceil n/2 \rceil - 1} k \lg k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \lg k \\
&\leq \sum_{k=2}^{\lceil n/2 \rceil - 1} k \lg(n/2) + \sum_{k=\lceil n/2 \rceil}^{n-1} k \lg n \\
&= \lg(n/2) \sum_{k=2}^{\lceil n/2 \rceil - 1} k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k \\
&= \lg(n/2) \left( \frac{(n/2-1)(n/2)}{2} - 1 \right) + \lg(n) \left( \frac{(n-1)(n)}{2} - \frac{(n/2-1)(n/2)}{2} \right) \\
&\leq (\lg n - \lg 2) \frac{(n/2)^2}{2} + (\lg n) \left( \frac{n^2}{2} - \frac{(n/2)^2}{2} \right) \\
&= (\lg n) \frac{n^2}{8} - \frac{n^2}{8} + (\lg n) \frac{n^2}{2} - (\lg n) \frac{n^2}{8} \\
&= \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2
\end{aligned}$$

(e) **[2 points]** Do problem 7-2(e) on page 161 of CLRS. Note that you are actually showing $E[T(n)] = O(n \lg n)$, but the $\Omega$ bound is trivial.

**Answer:** We will show this using the guess-and-check method. First, we guess that $E[T(q)] \leq cq \lg q$ for $2 \leq q < n$. Then, we show that the same is true for $E[T(n)]$.

$$\begin{aligned}
E[T(n)] &= \frac{2}{n} \sum_{q=2}^{n-1} E[T(q)] + kn \\
&\leq \frac{2}{n} \sum_{q=2}^{n-1} cq \lg q + kn \\
&\leq \frac{2}{n} c \left( \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2 \right) + kn \\
&= 2c \left( \frac{1}{2} n \lg n - \frac{1}{8} n \right) + kn \\
&= cn \lg n - \frac{1}{4} cn + kn \\
&\leq cn \lg n
\end{aligned}$$

for $c \geq 4k$.

4. **[19 points]** Skip List Analysis

In this problem, you will analyze the behavior of the skip list data structure introduced in lecture. Please see the Skip List Summary handout for more information, including detailed descriptions of the FIND, INSERT, and DELETE operations.

Throughout this problem, we will consider a skip list with $L$ levels and $n$ elements, in which elements are propagated to the next level with probability $p$, $0 < p < 1$.

(a) **[5 points]** What is the expected value of $L$ in terms of $n$ and $p$? For simplicity of analysis, consider levels that you expect to contain less than 1 cell to be nonexistent. In practice, how might you enforce this assumption?

**Answer:** $E[L] = \Theta(\log_{1/p} n)$. We can define the indicator variable $I_{i,k}$ to be equal to 1 if and only if the element $A[i]$ is present in the $k^{th}$ level of the skip list. Then,

$$E[L] = \min_k(E[\sum_{j=1}^{n} I_{j,k}] \leq 1)$$

the expected number of levels in the skip list is equal to the minimum level where the expected number of elements in the list is $\leq 1$. By linearity of expectation, we can move the summation outside of the expectation:

$$E[L] = \min_k(\sum_{j=1}^{n} E[I_{j,k}] \leq 1)$$

The expectation of an indicator variable is simply the probability of that event. What is the probability that element $A[i]$ is present in the $k^{th}$ level of the skip list? It is the probability that we propagated $A[i]$ up from level 0 to the level $k$. This is the product of $k$ independent propagation events, each of which has probability $p$. Thus, the equation for the expected number of levels is

$$\begin{aligned} E[L] &= \min_k(\sum_{j=1}^{n} p^k \leq 1) \\ &= \min_k(p^k \sum_{j=1}^{n} 1 \leq 1) \\ &= \min_k(np^k \leq 1) \end{aligned}$$

The value $np^k$ falls below 1 when $k \geq \log_{1/p} n$. So, we expect to have $E[L] = \Theta(\log_{1/p} n)$.

(b) **[8 points]** Consider the work done by the SEARCH operation at level $i$, where $i < L$. We must examine, in the worst case, all cells between $left[i+1].below$ and $right[i+1].below$ (since $left[L]$ and $right[L]$ are not actually defined, we can say that in level $L-1$ we look between *dummy* and NULL). Thus, we examine all cells immediately after some given cell in level $i$ that have not been propagated to level $i+1$. What is the expected number of such cells? What is the expected running time of the SEARCH operation in terms of $n$ and $p$?

**Answer:** We are trying to find the expected number of elements examined on level $i$, which is equal to the number of elements between $left[i+1].below$ and $right[i+1].below$. This is equal to the expected number of elements in level $i$ between two elements which are propagated up to level $i+1$. Since we know at least one element is propagated (the *dummy* element is always propagated), we simply compute the expected number of elements until another element is propagated. Since each element is propagated independently with probability $p$, the probability that the $k$th next element is the first to be propagated is $(1-p)^{k-1}p$. Therefore, the expected value of $k$ is $E[k] = \sum_{k=1}^{\infty} kp(1-p)^{k-1}$. This is the expectation of the geometric distribution, so the expectation is given by $E[k] = 1/p$ (p. 1112 CLRS). So, we expect to have to search through $1/p$ elements at each of the $i$ levels. Since we found in part (a) that there are $\Theta(\log_{1/p} n)$ levels, we expect SEARCH to take time $\Theta((1/p)\log_{1/p} n) = \Theta((1/p)\log_{1/p} n)$.

(c) **[3 points]** Use your answers to the previous parts to find the expected running time of the INSERT and DELETE operations.

**Answer:** INSERT and DELETE first call SEARCH, which runs in expected time $\Theta((1/p)lg_{1/p}n)$. Inserting and deleting an element from each level of the skip list takes $\Theta(1)$ time. The expected number of levels that each element is propagated is $E[levels] = \sum_{k=1}^{\infty} kp^{k-1}(1-p) = 1/(1-p)$, since this is once again the expectation of a geometric distribution. So, the total expected running time for INSERT and DELETE is $\Theta((1/p)\lg_{1/p} n) + (1/(1-p))\Theta(1) = \Theta((1/p)\lg_{1/p} n + 1/(1-p))$. Notice that the maximum number of levels is $log_{1/p}n$, so the first term dominates the second term.

(d) **[3 points]** Asymptotically, in terms of $n$ and $p$, what is the expected amount of memory used by the skip list data structure?

**Answer:** Each cell in the skip list takes a constant amount of memory since we need to store the four fields *next*, *above*, *below*, and *key*. The expected number of cells in the skip list is the sum over all elements $A[j]$ of the number of levels that $A[j]$ appears in.

$$E[\text{num cells}] = E[\sum_{j=1}^{n}(\text{num levels that } A[j] \text{ appears in})]$$

By linearity of expectation, we can pull the summation outside of the expectation.

$$E[\text{num cells}] = \sum_{j=1}^{n} E[\text{num levels that } A[j] \text{ appears in}]$$

We found the expected number of levels in part (c), names $E[levels] = 1/(1-p)$. So,

$$E[\text{num cells}] = \sum_{j=1}^{n} 1/(1-p) = n/(1-p).$$

So, the expected amount of memory is $\Theta(n/(1-p))$.

5. **[18 points]** Weighted Selection

   Consider some ways to generalize the notion of the median and the $i$th element.

   (a) **[1 point]** Do problem 9-2(a) on page 194 of CLRS.

   **Answer:** We have that the weighted median $x_k$ satisfies the formulas $\sum_{x_i<x_k} w_i < \frac{1}{2}$ and $\sum_{x_i>x_k} w_i \leq \frac{1}{2}$. For $w_i = 1/n$ for $i = 1, 2, \ldots, n$, these formulas reduce to $\sum_{x_i<x_k} w_i = \sum_{x_i<x_k} 1/n = y/n < 1/2$ where $y$ is the number of elements smaller than $x_k$ and $\sum_{x_i>x_k} 1/n = (n-y-1)/n \leq 1/2$ where $n-y-1$ is the number of elements greater than to $x_k$. This shows that $y < n/2$ and $y \geq n/2 - 1$. This states that exactly $n/2 - 1$ elements are smaller than $x_k$ for even $n$ and exactly $\lfloor n/2 \rfloor$ elements are smaller than $x_k$ for odd $n$. These is precisely the definition for the *lower* median.

   (b) **[2 points]** Do problem 9-2(b) on page 194 of CLRS.

   **Answer:** One can calculate the weighted median in $\Theta(n \lg n)$ worst case time using sorting. First, one sorts the array using merge sort which has worst-case running time $\Theta(n \lg n)$. Then we iterate through the array, computing $\sum_{i=1}^{k-1} w_i$, the total weight of the first $k-1$ elements in the array. We find the last element $k-1$ for which $\sum_{i=1}^{k-1} w_i < 1/2$. Then element $k$ satisfies the first equation. It also satisfies the second equation, since $\sum_{i=k+1}^{n} w_i = 1 - \sum_{i=1}^{k} w_i \leq 1/2$ since we know that $k-1$ was the *last* element for which the sum was less than $1/2$, so the sum up to $k$ must be $\geq 1/2$. Therefore, $k$ is the weighted median, and we have found it in $\Theta(n)$ time after sorting, so the total amount of time is $\Theta(n \lg n)$.

   (c) **[10 points]** Do problem 9-2(c) on page 194 of CLRS.

   **Answer:** The main idea is to use deterministic SELECT to compute the regular median in $\Theta(n)$ worst-case time. We can then use this median to partition our array into two (almost) equal halves, and look for the weighted median in either one or the other half. The recurrence will be $T(n) = T(n/2) + \Theta(n)$, which yields $T(n) = \Theta(n)$. In giving the pseudocode we will ignore floors and ceilings for clarity, thus, for instance, the median has index $n/2$.

   W-SELECT takes 4 parameters - the array $A$, $left$ and $right$ boundaries, and weight $w$. We look for an element $x_m$ that will satisfy $\sum_{x_i<x_m} w_i < w$ and $\sum_{x_i>x_m} w_i \leq 1 - w$.

   W-SELECT($A$, $left$, $right$, $w$)

   i. Call SELECT (the deterministic version) to find the median $m$ of $A$.
   ii. Partition $A$ around $m$.
   iii. Compute the total weight $W$ of all the elements in the lower partition.
   iv. If $W < w$ but $W + w_{n/2} \geq w$, return $m$.
   v. Else if $W < w$, recursively call W-SELECT($A$, $n/2$, $n$, $w - W - w_{n/2}$).
   vi. Else (if $W > w$), recursively call W-SELECT($A$, 1, $n/2$, $w$).

   This is a little more general than necessary for weighted median, but it helps in the next part of the problem. To compute the weighted median of an array $A$, we call W-SELECT($A$, 1, $n$, 1/2).

(d) **[5 points]** Consider the following generalization of the $i$th element: the **weighted $i$th element** is the element $x_k$ satisfying

$$\sum_{x_j<x_k} w_j < \frac{i}{n} \text{ and } \sum_{x_j>x_k} w_j \leq \frac{n-i}{n}$$

Modify your algorithms from parts (b) and (c) to compute the weighted $i$th element in, respectively, $\Theta(n \lg n)$ and $\Theta(n)$ worst-cased running time.

**Answer:** We can modify the algorithm in part (b) by simply comparing our sum $\sum_{i=1}^{k-1} w_i$ to $i/n$ instead of $1/2$. The rest of the analysis holds as before. The running time is still dominated by the sort which takes $\Theta(n \lg n)$. The actual selection of the weighted $i^{th}$ element from the sorted array takes time $\Theta(n)$ since it consists of one pass through the sorted array, summing the weights.
In part (c), we can simply call W-SELECT($A$, 1, $n$, $i/n$).

6. **[28 points]** Prisoner's Other Dilemma

An evil computer science professor has kidnapped $n$ of his students and locked them up in a dungeon. Each is kept in a separate cell in complete isolation from others, except that each evening one student is chosen randomly to spend the night in a conference room. The conference room is empty except it contains a switch, which may be toggled ON or OFF. The only way to observe or change the state of the switch is to spend a night in the conference room. Initially, the switch is known to be in the OFF position.

The students are told that once they have all been in the room at least once, one of them needs to tell this to the professor (this can be done when he is transferring the student to the conference room in the evening or back to the cell in the morning). If the claim is true, all students are set free and receive their degrees. If the claim is false, the students will be imprisoned forever and forced to proofread problem sets and papers for food.

The students are given one hour to come up with a joint strategy, after which point they will be taken to their cells and will only be able to communicate by observing and toggling the switch. Luckily, the students have studied algorithms and are able to come up with a solution plan that guarantees that when they do make the claim, it will be correct. Your job will be to analyze the expected time of their solution.

(a) **[10 points]** Consider the following idea: one of the students is designated as the **counter**. His job will be to toggle the switch OFF anytime he enters the room and sees that the switch is set to ON. Everyone else will toggle the switch ON if it's set to OFF, but only once.

Who can make the claim and when? Formally analyze the expected running time of this algorithm. Your answer should be asymptotic in terms of $n$.

**Answer:** $E[time] = \Theta(n^2)$. The **counter** can make the claim that everyone has been in the room once he has toggled the switch to OFF $n-1$ times. Since each student besides the counter sets the switch to ON exactly once, we know that $n-1$ distinct students besides the counter have been in the room. So, the counter

knows that all $n$ students have been in the room exactly once.
We can analyze the expected running time of this algorithm by breaking the sequence of students in the room into periods. We will define two types of periods
$X_i$ - the number of students who enter the room until (and including) the one who sets the switch to ON for the $i^{th}$ time.
$Y_i$ - the number of students who enter the room after the $i^{th}$ student has turned the switch ON until (and including) the counter who turns the switch to OFF.
For example, if student 1 is the counter and there are 4 students, then the following sequence would be broken up into the given periods:
4123431224131
$X_1 = \{4\}$, $Y_1 = \{1\}$, $X_2 = \{2\}$, $Y_2 = \{3, 4, 3, 1\}$, $X_3 = \{2, 2, 4, 1, 3\}$, $Y_3 = \{1\}$
Notice that the periods run $X_1, Y_1, X_2, Y_2, \ldots, X_{n-1}, Y_{n-1}$. After period $Y_{n-1}$ we are finished because the counter has turned off the switch $n - 1$ times.

The expected length of time for the algorithm is $E[time] = E[\sum_{i=1}^{n-1}(X_i + Y_i)]$. By linearity of expectation, we can pull the sum outside of the expectation. $E[time] = \sum_{i=1}^{n-1}(E[X_i] + E[Y_i])$.

$X_i$ is the number of students who enter the room until one can set the switch to ON. After $i - 1$ students have set the switch to ON, there are $(n - 1) - (i - 1)$ students who can set it to ON. Since we choose students uniformly at random, the probability at any point that one of these is selected is $(n - i)/n$. Thus, $Pr\{X_i = k\} = (\frac{n-i}{n})^{j-1}(\frac{i}{n})$. This is the geomtric series, and it's expectation is given by $E[X_i] = \frac{n}{n-i}$.

Similarily, we can find the expected value of $Y_i$, the number of students who enter the room after the $i^{th}$ student turns on the switch until the counter enters. The probability that the counter enters is $1/n$. So, $Pr\{Y_i = k\} = (\frac{n-1}{n})^{j-1}(\frac{1}{n})$. Again, this is a geometric distribution with expected value $E[Y_i] = n$. Using these formulas, we can find the expected length of time.

$$\begin{aligned}
E[time] &= \sum_{i=1}^{n-1}\left(\frac{n}{n-i} + n\right) \\
&= n(n-1) + n\sum_{i=1}^{n-1}\frac{1}{n-i} \\
&= n(n-1) + n\sum_{j=1}^{n-1}\frac{1}{j} \\
&= n(n-1) + n\Theta(\ln(n-1)) \\
&= \Theta(n^2)
\end{aligned}$$

(b) **[10 points]** Now consider the following modification to this idea: instead of deciding on the counter ahead of time, the counter will be the first student to

enter the room twice. Now depending on what day that occurs, the counter will know how many distinct people have been in the room already. Notice that this means the students need different rules for the first $n$ days of the imprisonment - after that, they can continue using the algorithm from part (a).

How must the students behave during the first $n$ days to make this work? Pay particular attention to what happens on day $n$. Analyze the running time of the modified algorithm - we do not expect a formal analysis this time, but your explanation should be complete and mathematical.

**Answer:** For the first $n$ days everyone should act as follows:

i. The switch remains OFF until someone has been in the room twice. That person turns the switch ON and becomes the counter, setting the count to $j-2$, where $j$ is the day when he enters the room the second time.

ii. If someone other than the counter enters the room and finds the switch OFF, the counter has not been determined yet. Thus, that person will be counted in the $j-2$ and does not touch the switch after day $n$.

iii. If someone other than the counter enters the room and finds the switch ON, the counter has been determined. Therefore, that person has not been counted in $j-2$ and must participate in the algorithm after day $n$. Same goes for anyone who has not entered the room at all during days 1 through $n$.

iv. Finally, the person who is in the room on day $n$ turns the switch OFF prior to beginning the second phase. If the person entering the room on day $n$ finds the switch OFF, then nobody has been in the room twice and that person can make the claim.

After the first $n$ days, everyone should act as in part (a) with the counter determined, except that the people who entered the room when the switch was OFF (befor the counter was determined) no longer participate.

The expected running time of this algorithm depends on how many people we counted during the first $n$ days, call that number $j$. We will first analyze the expected time after the first $n$ days based on $j$. As in part (a), we alternate time periods $X_i, Y_i$, but now since $j$ people have already been counted, we start with period $X_{j+1}$. Therefore, the expected time is given below with the sum starting from $j+1$

$$\begin{aligned} E[time|j] &= \sum_{i=j+1}^{n-1} \left( \frac{n}{n-i} + n \right) \\ &= n(n-j-1) + n \sum_{i=j+1}^{n-1} \frac{1}{n-i} \\ &= n(n-j-1) + n\Theta(\ln(n-j-1)) \end{aligned}$$

Now we will bound the probability of $j$, the number of people counted during the

first $n$ days. This is the birthday paradox problem, where the days are the people in the birthday paradox, and the people are the birthdays. We know that when $j = \Theta(\sqrt{n})$, we have $Pr\{\text{someone enters twice in} \leq j \text{ days}\} \geq 1/2$. So, we can now bound our expected time.

$$\begin{aligned}
E[time] &= n + \sum_{j=0}^{n-1} Pr\{j\} E[time|j] \\
&= n + \sum_{j=0}^{c\sqrt{n}} Pr\{j\} E[time|j] + \sum_{j=c\sqrt{n}+1}^{n-1} Pr\{j\} E[time|j] \\
&\geq n + \sum_{j=0}^{c\sqrt{n}} Pr\{j\} E[time|c\sqrt{n}] + 0 \\
&= n + \left(n(n - c\sqrt{n} - 1) + n\Omega(\ln(n - c\sqrt{n} - 1))\right) \sum_{j=0}^{c\sqrt{n}} Pr\{j\} \\
&\geq n + \left(n(n - c\sqrt{n} - 1) + n\Omega(\ln n)\right)(1/2) \\
&\geq n + (1/2)n^2 - (c/2)n^{3/2} - n + \Omega(n \ln n) \\
&= \Omega(n^2)
\end{aligned}$$

This expected time is also $O(n^2)$, as it's bounded above by $n + E[time|0] = \Theta(n^2)$. Thus, our asymptotic expected time remains $\Theta(n^2)$.

(c) **[8 points]** It is possible to do asymptotically better than your answers in parts (a) and (b). However, there are well-defined lower bounds on the performance of **any** correct algorithm for this problem. For instance, since all students have to have been in the room at the time the claim is made, clearly no correct algorithm can have an expected running time asymptotically less than $\Theta(n)$. Can you come up with a tighter lower bound? Argue that your answer is correct; you may use results proven in class or in the textbook.

**Answer:** No correct algorithm can have a better expected running time than $\Omega(n \lg n)$. As we proved in class with the balls and bins question, the expected amount of time until each bin has at least one ball is $\Theta(n \lg n)$. In this question, the days are balls and the students are bins. We want the expected number time until each students has been in the room for at least one day. So, the expected number of days is $\Theta(n \lg n)$. Any algorithm the students come up with can't have an expected running time faster than the expected time for the condition to actually hold, otherwise it would not be correct. Therefore, the expected running time of any algorithm which will only tell the professor they have all been in the room once they have actually all been there must take at least $\Theta(n \lg n)$ days. Hence, $E[time] = \Omega(n \lg n)$.

**CS161: Design and Analysis of Algorithms** **Summer 2004**

# Problem Set #3 Solutions

## General Notes

- **Regrade Policy:** If you believe an error has been made in the grading of your problem set, you may resubmit it for a regrade. If the error consists of more than an error in addition of points, please include with your problem set a detailed explanation of which problems you think you deserve more points on and why. We reserve the right to regrade your entire problem set, so your final grade may either increase or decrease.
- If we ask for an answer in terms of $n$ and $d$, please give the answer in terms of $n$ and $d$ and do not assume that $d$ is a constant. Many people did this on the first problem as well as the skip list problem from the last problem set.
- In part (b) of problem 3, many people only showed that the probability that **any** slot had $k$ elements was $\leq nQ_k$. You had to show the **maximum** number of elements was $k$.

1. **[14 points]** Analysis of $d$-ary Heaps

   Throughout this problem, when asked to give an implementation of a certain operation, please provide pseudocode or a verbal description with the same level of clarity and detail. When asked to analyze running times, your analysis does not need to be formal, but should be justified.

   Note that we are asking you to solve 6-2(e) before 6-2(d). Keep in mind that you may always (not only in this problem) use the results of any previous parts in answering a later part of a multi-part problem.

   (a) **[1 point]** Do problem 6-2(a) on page 143 of CLRS.

   **Answer:** A $d$-ary heap can be represented in a 1-dimensional array by keeping the root of the heap in $A[1]$, its $d$ children in order in $A[2]$ through $A[d+1]$, their children in order in $A[d+2]$ through $A[d^2+d+1]$, and so on. The two procedures that map a node with index $i$ to its parent and to its $j^{th}$ child (for $1 \leq j \leq d$) are

   ```
   D-PARENT(i)
   1    return ⌈(i − 1)/d⌉

   D-CHILD(i, j)
   2    return d(i − 1) + j + 1
   ```

   (b) **[1 point]** Do problem 6-2(b) on page 143 of CLRS.

   **Answer:** Since each node has $d$ children, the total number of nodes in a tree of height $h$ is bounded above by $1 + d + \ldots + d^h$ inclusively and below by $1+d+\ldots+d^{h-1}$ exclusively. This yields $h = \lceil \log_d(1+n(d-1)) \rceil - 1 = \Theta(\log_d n)$.

(c) **[2 points]** Give an efficient implementation of HEAPIFY in a $d$-ary max-heap. Analyze its running time in terms of $d$ and $n$.

**Answer:** First, we will need to find if the root element given to D-HEAPIFY is larger than all its children. If not, we swap it with its largest child and recursively call D-HEAPIFY on that child.

```
D-HEAPIFY(A, i, d)
1   largest ← i
2   for j ← 1 to d
3       j ← D-CHILD(i, j)
4       if j ≤ heapsize[A] and A[j] > A[largest]
5           then largest ← j
6   if largest ≠ i
7       then exchange A[i] ↔ A[largest]
8           D-HEAPIFY(A, largest, d)
```

This algorithm does $\Theta(d)$ comparisons in each call to D-HEAPIFY as well as $O(\log_d n)$ calls to HEAPIFY, one for each level of the tree. Therefore, the total running time for this algorithm is $O(d \log_d n)$.

(d) **[5 points]** Give an efficient implementation of BUILD-HEAP in a $d$-ary max-heap. Analyze its running time in terms of $d$ and $n$.

**Answer:** To build a heap, we can simply assume that our array is already a heap and call D-HEAPIFY in a bottom up manner to convert the array into a d-max-heap. The leaves are already heaps since they have no children, so we start with the lowest non-leaf element. By our procedure for D-PARENT, the parent of the last element is $\lceil (n-1)/d \rceil$. By the time we call D-HEAPIFY on any element, D-HEAPIFY will have already been called on all of its children and therefore the call is valid since the children are all themselves heaps.

```
D-BUILD-HEAP(A, d)
1   heapsize[A] ← length[A]
2   for i ← ⌈(length[A] − 1)/d⌉ downto 1
3       do D-HEAPIFY(A, i, d)
```

We can bound the running time of this algorithm by modifying the analysis in section 6.3 of CLRS. Notice that there are

$$d^{\log_d(n(d-1))-(h+1)} = d^{\log_d\left(\frac{n(d-1)}{d^{h+1}}\right)} = \frac{n(d-1)}{d^{h+1}}$$

nodes at each height $h$. From part (c), the time for each call to HEAPIFY at a node of height $h$ is $O(dh)$. So, the running time

$$
\begin{aligned}
T(n) &\leq \sum_{h=0}^{\lg_d(n(d-1))} dh\frac{n(d-1)}{d^{h+1}} \\
&< n(d-1)\sum_{h=0}^{\infty}\frac{h}{d^h} \\
&\leq n(d-1)\frac{1/d}{(1-1/d)^2} \qquad \text{A.8} \\
&= n\frac{d}{d-1} \\
&= n\left(1+\frac{1}{d-1}\right)
\end{aligned}
$$

For any $d \geq 2$ we have $1 \leq 1 + 1/(d-1) \leq 2$, so $T(n) = \Theta(n)$.

(e) **[2 points]** Do problem 6-2(c) on page 143 of CLRS.

**Answer:** The procedure D-EXTRACT-MAX is the same as for the binary heap. We return the value of $A[1]$, move the element in $A[heapsize]$ into $A[1]$, decrement $heapsize$, and then call D-HEAPIFY on the root of the heap.

```
D-EXTRACT-MAX(A, d)
1   if heapsize[A] < 1
2      then error
3   max ← A[1]
4   A[1] ← A[heapsize]
5   heapsize ← heapsize − 1
6   D-HEAPIFY(A, 1, d)
7   return max
```

All operations besides D-HEAPIFY take $O(1)$ time. D-HEAPIFY takes time $\Theta(d \log_d n)$ so D-EXTRACT-MAX takes time $\Theta(d \log_d n)$.

(f) **[2 points]** Do problem 6-2(e) on page 143 of CLRS.

**Answer:** If $k < A[i]$, then D-INCREASE-KEY does not change the heap. Otherwise, it will set $A[i]$ to $k$ and move the element upwards toward the root until it is smaller than its parent.

```
D-INCREASE-KEY(A, i, k, d)
1   if k > A[i]
2      then A[i] ← k
3         while i > 1 and A[D-PARENT(A, i, d)] < A[i]
4            do exchange A[i] ↔ A[D-PARENT(A, i, d)]
5               i ← A[D-PARENT(A, i, d)]
```

The running time of the algorithm is proportional to the height of the heap, and each level takes a constant amount of work, so the overall time is $\Theta(\log_d n)$.

(g) **[1 point]** Do problem 6-2(d) on page 143 of CLRS.

**Answer:** The procedure D-INSERT is the same as for the binary heap. We add the element to the end of the array and set its key to $-\infty$. Then, we call D-INCREASE-KEY to increase to increase the key to $key$.

D-INCREASE-KEY$(A, key, d)$
1 $\quad heapsize[A] \leftarrow heapsize[A] + 1$
2 $\quad A[heapsize[A]] \leftarrow -\infty$
3 $\quad$ D-INCREASE-KEY$(A, heapsize[A], key, d)$

The first two operations take constant time, and the call to D-INCREASE-KEY takes time $\Theta(\log_d n)$. So, the running time for D-INCREASE-KEY is $\Theta(\log_d n)$.

2. **[23 points]** Average-Case Lower Bounds on Comparison Sorting

Throughout the next two problems, when asked to prove, argue, show or conclude something, please make your explanations as clear and complete as possible.

(a) **[4 points]** Do problem 8-1(a) on page 178 of CLRS.

**Answer:** Since the sort can be run on any input, the algorithm must be able to reach any permutation of the input in order to get the correct sorted order as an answer. Because $A$ is deterministic, any input will always follow exactly one path in $T_A$. There are $n!$ possible permutations of $n$ elements, and each permutation should lead to a distinct leaf in $T_A$ in order to yield the correct sorted order. Since we assume that every permutation of $A$'s input is equally likely and since the probabilities must sum to 1, each leaf must have probability $1/n!$. Any other leaf is never reached and thus must have probability 0, as needed to be shown.

(b) **[2 points]** Do problem 8-1(b) on page 178 of CLRS.

**Answer:** The external path length $T$ is given by the sum of the depths of all the leaves contained in the right subtree plus the sum of the depths of all the leaves contained in the left subtree. The sum of the depths of the leaves in the right subtree is the sum of the depths up to the root of the right subtree plus one for each leaf in the right subtree, and likewise for the sum of the depths in the left subtree. So, the total depth $D(T) = D(RT)+$ number of leaves in $RT + D(LT)$ + number of leaves in $LT$. Each leaf must be in exactly one of the left or right subtrees, so $D(T) = D(RT) + D(LT)+$ number of leaves $= D(RT) + D(LT) + k$.

(c) **[3 points]** Do problem 8-1(c) on page 178 of CLRS.

**Answer:** Consider a decision tree $T$ with $k$ leaves that achieves the minimum. Let $i_0$ be the number of leaves in $LT$ and $k - i_0$ be the number of leaves in $RT$. From part (b), we have $D(T) = D(LT) + D(RT) + k$. Notice that this implies that $LT$ and $RT$ also have the minimum external path length over all subtrees

with $i_0$ and $k - i_0$ leaves respectively, since otherwise we could replace them with better trees and get a lower value of $D(T)$.

Finally, the optimal tree $T$ represents the best possible split of the $k$ leaves into (optimal) subtrees with $i_0$ and $k - i_0$ leaves. Since each subtree must have at least one leaf, we have $d(k) = \min_{1 \le i \le k-1}\{d(i) + d(k-i) + k\}$.

(d) **[4 points]** Do problem 8-1(d) on page 178 of CLRS.

**Answer:** We can find the minimum by setting the derivative with respect to $i$ equal to zero.

$$\begin{aligned}
\frac{d}{di}(i \lg i + (k-i)\lg(k-i)) &= (i)(1/i)(\lg e) + (1)(\lg i) + \\
&\quad (k-i)(1/(k-i))(-\lg e) + (-1)(\lg(k-i)) \\
&= \lg e + \lg i - \lg e - \lg(k-i) \\
&= \lg i - \lg(k-i)
\end{aligned}$$

This is zero when $i = k - i$ or $i = k/2$. We can verify that this is in fact a minimum by taking the second derivative and checking that it is positive.

We can now check that $d(k) = \Omega(k \lg k)$ using the guess-and-check method. Assume that $d(i) \ge ci \lg i$ for $i < k$.

$$\begin{aligned}
d(k) &= \min_{1 \le i \le k-1}(d(i) + d(k-i) + k) \\
&\ge \min_{1 \le i \le k-1}(ci \lg i + c(k-i)\lg(k-i) + k) \\
&= k + c \min_{1 \le i \le k-1}\{i \lg i + (k-i)\lg(k-i)\} \\
&\ge k + c((k/2)\lg(k/2) + (k/2)\lg(k/2)) \\
&= ck \lg(k/2) + k \\
&= ck(\lg k - \lg 2) + k \\
&= ck \lg k + k(1-c) \\
&\ge ck \lg k
\end{aligned}$$

with the last inequality holding as long as $c \le 1$. Therefore, $d(k) = \Omega(k \lg k)$.

(e) **[2 points]** Do problem 8-1(e) on page 178 of CLRS.

**Answer:** In part (a), we showed that any deterministic tree $T_A$ has at least $n!$ leaves. Since $d(k)$ is the minimum value over all decision trees with $k$ leaves, we know that $D(T_A) \ge d(k)$. In part (d), we showed that $d(k) = \Omega(k \lg k)$. Since $T_A$ has $k \ge n!$ leaves, we know $D(T_A) \le d(k) = \Omega(k \lg k) = \Omega(n! \lg(n!))$.

The expected time to sort $n$ elements is the sum over all leaves of the length of the path to that leaf times the probability of reaching that leaf. Since the probability of reaching each leaf is $1/n!$ (ignoring leaves with probability 0 that

contribute nothing to the sum), this is precisely $D(T_A)/n!$. Thus, $E[T(n)] = \Omega(n! \lg(n!))/n! = \Omega(\lg n!) = \Omega(n \lg n)$.

(f) **[8 points]** Do problem 8-1(f) on page 178 of CLRS.

**Answer:** Consider the decision tree $T_B$ for algorithm $B$. We will work from the bottom to the top, replacing each randomization node with the branch that has the smallest external path length. We call the resulting tree $T_A$ and we show that $T_A$ represents a valid deterministic sorting algorithm and $E[T_A(n)] \leq E[T_B(n)]$.

First, consider the issue of correctness. Since the decision tree for $A$ represents a possible execution of $B$ (with appropriate random choices made at each step), and $B$ is correct, the resulting decision tree must represent a valid sorting algorithm. Since all randomization nodes have been removed, this algorithm is deterministic.

Now notice that at each randomization node $N$, the expected execution time after that node is reached is equal to $\sum_{i=1}^{r}(1/r)E[T(S_i)]$, where $S_i$ is the $i^{th}$ subtree of $N$. Clearly we have $E[T(N)] \geq min_{i=1}^{r} E[T(S_i)]$, since all terms in the average must be at least the minimum. Thus, each transformation that replaces a randomization node with its best child does not increase the expected running time.

We formalize these notions slightly. Consider a sequence of decision trees $T_1, \ldots, T_m$ where $T_1 = T_B$ and $T_m = T_A$, and each $T_i$ was obtained from $T_{i-1}$ by choosing a randomization node all of whose descendants are deterministic and replacing it with the subtree with the smallest external path length. Notice that since no descendants of $N$ are randomization nodes, the notion of expected running time and external path length (given random input) is the same for its children.

The correctness argument above applies, as does the running time argument: $E[T(T_i)] \leq E[T(T_{i-1})]$. Finally, when all the randomization nodes are removed, we have a deterministic sorting algorithm with the decision tree $T_m = T_A$ and expected running time no worse than the expected running time of $B$.

3. **[18 points]** Slot-Size Bound for Chaining

(a) **[2 points]** Do problem 11-2(a) on page 250 of CLRS.

**Answer** The probability that exactly $k$ keys hash to the same spot is given by first choosing $k$ keys from the $n$, multiplying by the probability that those $k$ keys hashed to a particular spot, and multiplying by the probability that the remaining $n - k$ keys hashed to a different spot (see appendix $C.4$ for a discussion of the binomial distribution if this is unclear). Thus we have

$$Q_k = \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \binom{n}{k}.$$

(b) **[2 points]** Do problem 11-2(b) on page 250 of CLRS.

**Answer:** The probability of the event $M = k$ is equal to $Pr\{\cup_{i=1}^{n} E_i\}$, where $E_i$ is the event that slot $i$ has the maximum number of elements and that slot $i$ has exactly $k$ elements.

The probability of a union of a set of events is bounded by the sum of the probabilities of those events (with equality if and only if those events are mutually exclusive), so $Pr\{M = k\} \leq \sum_{i=1}^{n} Pr\{E_i\} = nPr\{E_1\}$ due to symmetry.

Finally, observe that the event $E_i$ subsumes the event that exactly $k$ keys hash to slot $i$, because $E_i$ represents the event { $k$ keys hash to $i$ **and** $i$ has the maximum number of elements}. Thus, $Pr\{E_i\} \leq Q_k$. Combining this with the previous equation yields

$$P_k = Pr\{M = k\} \leq nPr\{E_1\} \leq nQ_k.$$

(c) **[2 points]** Do problem 11-2(c) on page 250 of CLRS.

**Answer:**

$$\begin{aligned} Q_k &= \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \binom{n}{k} \\ &\leq \left(\frac{1}{n}\right)^k \binom{n}{k} && \text{since } \left(1 - \frac{1}{n}\right)^{n-k} \leq 1 \\ &\leq \left(\frac{1}{n}\right)^k \left(\frac{en}{k}\right)^k && \text{equation C.5} \\ &= \left(\frac{e}{k}\right)^k \end{aligned}$$

(d) **[8 points]** Do problem 11-2(d) on page 250 of CLRS.

**Answer:** We will find a $c > 1$ such that $(e/k_0)^{k_0} < 1/n^3$. Since $(e/k)^k$ is a decreasing function for $k > e$ (when $e/k < 1$), this means that $Q_k < 1/n^3$ for $k \geq k_0$. Finally, combining this with the result of part (b) will yield the desired bound on $P_k$.

Thus, we want $(e/k_0)^{k_0} < 1/n^3$. Taking lg of both sides yields

$$\begin{aligned} k_0 \lg\left(\frac{e}{k_0}\right) &< -3 \lg n \\ k_0(\lg e - \lg k_0) &< -3 \lg n \\ \frac{c \lg n}{\lg \lg n}\left(\lg e - \lg \frac{c \lg n}{\lg \lg n}\right) &< -3 \lg n \\ \frac{c}{\lg \lg n}(\lg e - \lg c - \lg \lg n + \lg \lg \lg n) &< -3 \\ \frac{c}{\lg \lg n}(-\lg e + \lg c + \lg \lg n - \lg \lg \lg n) &> 3 \end{aligned}$$

For $c > e$, the $\lg c$ term is larger than the $\lg e$ term, so we need to make sure the rest of the left side is greater than the right side, i.e.

$$\begin{aligned}
\frac{c}{\lg\lg n}(\lg\lg n - \lg\lg\lg n) &> 3 \\
c &> \frac{3\lg\lg n}{\lg\lg n - \lg\lg\lg n} \\
c &> \frac{3}{1 - \frac{\lg\lg\lg n}{\lg\lg n}}
\end{aligned}$$

As $n$ gets large, the second term in the denominator will go to zero; in fact for $n \geq 16$ we have $\frac{\lg\lg\lg n}{\lg\lg n} \leq 1/2$ so the right hand side is $\leq 6$.

Thus, we can choose an appropriate constant $c$ to guarantee that $(e/k_0)^{k_0} < 1/n^3$ and therefore $Q_k < 1/n^3$ for $k \geq k_0$. Thus, $P_k \leq nQ_k < 1/n^2$ for $k \geq k_0$.

(e) **[4 points]** Do problem 11-2(e) on page 250 of CLRS.
**Answer:**

$$\begin{aligned}
E[M] &= \sum_{i=1}^{n} i \cdot Pr\{M = i\} \\
&= \sum_{i=1}^{k_0} i \cdot Pr\{M = i\} + \sum_{i=k_0+1}^{n} i \cdot Pr\{M = i\} \\
&\leq k_0 \sum_{i=0}^{k_0} Pr\{M = i\} + n \sum_{i=k_0+1}^{n} Pr\{M = i\} \\
&= k_0 Pr\{M \leq k_0\} + nPr\{M > k_0\} \\
&= n \cdot Pr\left\{M > \frac{c\lg n}{\lg\lg n}\right\} + \frac{c\lg n}{\lg\lg n} \cdot Pr\left\{M \leq \frac{c\lg n}{\lg\lg n}\right\}
\end{aligned}$$

Plugging in the values from part (d), we have

$$\begin{aligned}
E[M] &\leq (n)(n - k_0)(1/n^2) + \frac{c\lg n}{\lg\lg n} \cdot 1 \\
&\leq (n)(n)(1/n^2) + \frac{c\lg n}{\lg\lg n}
\end{aligned}$$

Thus, $E[M] \leq 1 + c\lg n/\lg\lg n = O(\lg n/\lg\lg n)$.

**CS161: Design and Analysis of Algorithms** **Summer 2004**

# Problem Set #4 Solutions

## General Notes

- **Regrade Policy:** If you believe an error has been made in the grading of your problem set, you may resubmit it for a regrade. If the error consists of more than an error in addition of points, please include with your problem set a detailed explanation of which problems you think you deserve more points on and why. We reserve the right to regrade your entire problem set, so your final grade may either increase or decrease.
- On problem 1(f), many people assumed that if the $key[y] < key[x]$ then $y$ was in the left subtree of $x$ or that if $priority[y] < priority[x]$ then $y$ was a child of $x$. These stipulations are not true, since in either case you can also have $y$ not be a descendant of $x$ and the condition may still hold. Also, many people did not prove both the if and the only if parts of the statement.
- For augmented red-black trees in general, if the augmentation does not satisfy Theorem 14.1, you should show that it can be maintained efficiently through insertion, deletion, *and rotations*.
- On problem 5(b), a common mistake was to sort by the $x$ indices, but to not take into account the order for equal $x$ index. For equal $x_i$'s, you need to sort by decreasing index $y_j$ so that you can match at most one $y_j$ to each $x_i$.

1. **[34 points]** Treaps

   Throughout this problem, please be as formal as possible in your arguments whenever asked to show or conclude some result.

   (a) **[4 points]** Do problem 13-4(a) on page 297 of CLRS.

   **Answer:** Our proof is inductive and constructive. The induction is on the number of elements in the treap. The base case of 0 nodes is the empty treap. We now suppose that the claim is true for $0 \leq k < n$ nodes, and show it is true for $n$. Given a set of $n$ nodes with distinct keys and priorities, the root is uniquely determined - it must be the node with the smallest priority in order for the min-heap property to hold. Now suppose this root node has key $r$. All elements with key $< r$ must be in the left subtree, and all elements with key $> r$ must be in the right subtree. Notice that each of these sets has at most $n-1$ keys, and therefore has a unique treap by the inductive assumption. Thus, for the $n$-node treap, the root is uniquely determined, and so are its subtrees. Therefore, the treap exists and is unique.

   Notice this suggests an algorithm for finding the unique treap: choose the element with the smallest priority, partition the remaining elements around its key, and recursively build the unique treaps from the elements smaller than the root key

(these will go in the left subtree) and the elements larger than the root key (right subtree).

(b) **[2 points]** Do problem 13-4(b) on page 298 of CLRS.

**Answer:** Recall from lecture and section 12.4 of CLRS that the expected height of a randomly built binary search tree is $O(\lg n)$. We show that constructing the unique treap is equivalent to inserting the elements into a binary search tree in random order.

Suppose we insert the elements into a BST in order of increasing priority. Normal BST insertion will maintain the binary search tree property, so we only need to show that the min-heap property is maintained on the priorities. Suppose $v$ is a child of $u$. The insertion procedure from section 12.3 of CLRS always inserts each element at the bottom of the tree, therefore $v$ must have been inserted after $u$. Thus, the priority of $v$ is greater than the priority of $u$, and we have built a treap.

The priorites are assigned randomly, which means any permutation of the priorities is equally likely. When considering this as BST insertion, it translates into any ordering being equally likely, and therefore the expected height of a treap is equal to the expected height of a randomly built BST, which is $O(\lg n)$.

(c) **[6 points]** Do problem 13-4(c) on page 298 of CLRS.

**Answer:** Treap-Insert works by first inserting the node according to its key and the normal BST insertion procedure. This is guaranteed to maintain the first two conditions of the treap, since those correspond to BST properties. However, we may have violated the third condition, the min-heap ordering on the priorities. Note that (initially) this violation is localized to the node we're inserting and its parent, since we always insert at the leaves. Rotations preserve BST properties, so we will use rotations to move our node $x$ up as long as the min-heap ordering is violated, i.e. as long as $priority[x] < priority[parent[x]]$. If $x$ is a left child we will right-rotate it, and if it's a right child we will left-rotate it. Notice that this preserves the heap property elsewhere in the treap, assuming that the children of $x$ had higher priority than $parent[x]$ prior to the rotation (to be completely formal we would have to prove this using a loop invariant).

The pseudocode is as follows:

```
TREAP-INSERT(T, x, priority)
1   TREE-INSERT(T, x)
2   while parent[x] ≠ NIL ∧ priority[x] < priority[parent[x]]
3       if left[parent[x]] == x
4           Right-Rotate(T, parent[x])
5       else
6           Left-Rotate(T, parent[x])
```

(d) **[2 points]** Do problem 13-4(d) on page 298 of CLRS.

**Answer:** TREAP-INSERT first performs a BST insert procedure which runs in time proportional to the height of the treap. Then, it rotates the node up one level until the min-heap property is satisfied. Thus, the number of rotations we perform is bounded by the height of the treap. Each rotation takes constant

time, so the total running time is proportional to the height of the treap, which we showed in (b) to be expected $\Theta(\lg n)$.

(e) **[4 points]** Do problem 13-4(e) on page 298 of CLRS.

**Answer:** We will prove this using a loop invariant: after having performed $k$ rotations during the insertion of $x$, $C_k + D_k = k$.
Initialization: After we insert $x$ using BST-INSERT but before performing any rotations, $x$ is a leaf and its subtrees are empty, so $C_0 = D_0 = 0$.
Maintenance: Assume that we have performed $k$ rotations on $x$ and that $C_k + D_k = k$. Now, if we perform a right-rotation on $x$, the left child of $x$ remains the same, so $C_{k+1} = C_k$. The right child of $x$ changes from $\beta$ to $y$ with subtrees $\beta$ (left) and $\gamma$ (right) using the notation from page 278. The left spine of the right child used to be the left spine of $\beta$ and now it is $y$ plus the left spine of $\beta$. Therefore, $D_{k+1} = D_k + 1$. Thus, $C_{k+1} + D_{k+1} = C_k + D_k + 1 = k + 1$, which is precisely the number of rotations performed. The same holds for left-rotations, where we can show that $D_{k+1} = D_k$ and $C_{k+1} = C_k + 1$.
Termination: After TREAP-INSERT is finished, the number of rotations we performed is equal to $C + D$, precisely the condition that needed to be shown.

(f) **[5 points]** Do problem 13-4(f) on page 298 of CLRS.

**Answer:** First, assume $X_{i,k} = 1$. We will prove $priority[y] > priority[x]$, $key[y] < key[x]$, and $\forall z$ such that $key[y] < key[z] < key[x]$, we have $priority[y] < priority[z]$. The first property follows from the min-heap property on priorities, since $y$ is a descendant of $x$. The second follows from the BST property, since $y$ is in the left subtree of $x$. Finally, consider any node $z$ satisfying $key[y] < key[z] < key[x]$, and imagine an inorder tree walk on the treap. After $y$ is printed, we will print the right subtree of $y$, at which point the entire left subtree of $x$ will have been printed since $y$ is in the right spine of that subtree. Thus, the only nodes printed after $y$ but before $x$ are in the right subtree of $y$; therefore, $z$ must be in the right subtree of $y$ and by the min-heap property $priority[y] < priority[z]$.

Now, we will assume the three properties and prove that $X_{i,k} = 1$. First consider the possibility that $y$ is in the left subtree of $x$ but not in the right spine of that subtree. Then there exists some node $z$ in the spine such that going left from $z$ will lead to $y$. Note that this $z$ satisfies $key[y] < key[z] < key[x]$, but $priority[z] < priority[y]$ which violates the third property. Clearly $y$ cannot be in the right subtree of $x$ without violating the second property, and $x$ cannot be a descendant of $y$ without violating the first property. Suppose that $x$ and $y$ are not descendants of each other, and let $z$ be their common ancestor. Again we have $key[y] < key[z] < key[x]$ but $priority[z] < priority[y]$, violating the third property. The only remaining possibility is that $y$ is in the right spine of the left subtree of $x$, i.e. $X_{i,k} = 1$.

(g) **[4 points]** Do problem 13-4(g) on page 300 of CLRS.

**Answer:** Assume that $k > i$. $Pr\{X_{i,k} = 1\}$ is the probability that all the conditions in part (f) hold. Consider all the elements with keys $\{i, i+1, \ldots, k\}$.

There are $k-i+1$ such elements. The values of their priorities can be in any order, so there are $(k-i+1)!$ possible (and equally likely) permutations. In order to satisfy our conditions, we need $priority[z] > priority[y] > priority[x]$ for all $z \in \{i+1, i+2, \ldots, k-1\}$. This fixes the priorities of $x$ and $y$ to be the lowest two priorities, allowing $(k-i-1)!$ permutations of the remaining priorities among the elements with keys in $\{i+1, \ldots, k-1\}$. The probability of $X_{i,k} = 1$ is the ratio of these two, which gives $Pr\{X_{i,k} = 1\} = \frac{(k-i-1)!}{(k-i+1)!}$. Most of the terms in the factorials will cancel, except for the first two terms in the denominator. This leaves us with $Pr\{X_{i,k} = 1\} = \frac{1}{(k-i+1)(k-i)}$.

(h) **[3 points]** Do problem 13-4(h) on page 300 of CLRS.

**Answer:** The expected value for $C$, which is the number of elements in the right spine of the left subtree of $x$ is simply the expectation of the sum of $X_{i,k}$ over all $i < k$. This is

$$\begin{aligned}
E[C] &= E[\sum_{i=1}^{k-1} X_{i,k}] \\
&= \sum_{i=1}^{k-1} E[X_{i,k}] \\
&= \sum_{i=1}^{k-1} Pr[X_{i,k} = 1] \\
&= \sum_{i=1}^{k-1} \frac{1}{(k-i+1)(k-i)} \\
&= \frac{1}{(k)(k-1)} + \frac{1}{(k-1)(k-2)} + \cdots + \frac{1}{(2)(1)} \\
&= \sum_{j=1}^{k-1} \frac{1}{(j+1)(j)} \\
&= \sum_{j=1}^{k-1} \left(\frac{1}{j} - \frac{1}{j+1}\right) \\
&= \frac{1}{1} - \frac{1}{k} \\
&= 1 - \frac{1}{k}
\end{aligned}$$

(i) **[2 points]** Do problem 13-4(i) on page 300 of CLRS.

**Answer:** Note that the expected length of the right spine of the left subtree of $x$ depends only on the rank $k$ of the element $x$. The expected length of the left spine of the right subtree will have the same expected value with respect to the rank of $x$ in the reverse ordering, which is $n-k+1$. Thus, $E[D] = 1 - \frac{1}{n-k+1}$.

(j) **[2 points]** Do problem 13-4(j) on page 300 of CLRS.

**Answer:** Since the number of rotations equals $C + D$, the expected number of rotations equals $E[C + D] = E[C] + E[D] = 1 - \frac{1}{k} + 1 - \frac{1}{n-k+1} \leq 2$. So, the expected number of rotations is less than 2.

2. **[32 points]** Augmenting Data Structures

Throughout this problem, your explanations should be as complete as possible. In particular, if you use an augmented data structure that is not presented in the textbook, you should explain how that data structure can be efficiently maintained, explain how new operations can be implemented, and analyze the running time of those operations. If the problem asks you to give an algorithm, you do not need to give pseudocode or formally prove its correctness, but your explanation should be detailed and complete, and should include a running time analysis.

(a) **[9 points]** Do problem 14.1-8 on page 308 of CLRS.

**Answer:** To solve this problem, we will first define some kind of coordinate system. This can be done by picking an arbitrary nonendpoint starting point on the circle and setting it to zero and defining clockwise as positive values (for example, number of degrees from the starting point). We can label each chord's endpoints as $S_i, E_i$ where the start endpoint has lower value. We then notice that to count the chords which intersect a particular chord $i$, we can count the chords which have only one endpoint between $S_i$ and $E_i$. Since we'll then double-count the number of intersections, we'll just count those intersections that are in the *forward* direction, namely the chord has its start endpoint in $S_i, E_i$, but not its other endpoint. The chords which have their end endpoints in the interval will count $i$ as intersecting them.

We can then use an order-statistic red-black tree to help us count the number of intersections. We will first sort the endpoints according to our coordinate system. Since there are $n$ chords each with 2 endpoints, this step will take $\Theta(2n \lg 2n) = \Theta(n \lg n)$. We then process the endpoints according to their sorted values, starting with the smallest. If it is a start endpoint, we insert it into the order-statistic tree ($\Theta(\lg n)$). If it is an end endpoint, we want to count the number of start endpoints we have seen without matching end endpoints. If we delete start endpoints as we see their corresponding end endpoints, we will maintain the property that all the nodes in our tree are *open* chords, ones which we can still intersect. So, for each end endpoint, we delete the corresponding start endpoint and then count the number of open endpoints left in the tree - this will be precisely the number of chords that have their starting endpoint between $S_i$ and $E_i$ but have their end endpoint after $E_i$. Thus, we add this number to a counter and delete the start endpoint of that chord (each end endpoint can have a pointer to its start). This step takes $\Theta(\lg n)$ for the deletion and $\Theta(1)$ to figure out the number of forward intersections for the chord. We proceed like this through all the endpoints. Since each endpoint takes time $\Theta(\lg n)$ to insert or delete from the tree, the total running time of this step is ($\Theta(n \lg n)$.

After processing all endpoints, we will have a count of the number of intersections inside the circle. The total running time is the time to sort plus the time to insert and delete from the OS-tree, both of which are $\Theta(n \lg n)$. Therefore, the total running time is $\Theta(n \lg n)$.

(b) **[4 points]** Do problem 14.2-1 on page 310 of CLRS.

**Answer:** We can find the SUCCESSOR and PREDECESSOR of any node $x$ in time $\Theta(1)$ by storing pointers to the successor and predecessor of $x$ inside the node structure. Note that these values do not change during rotations, so we only need to show that we can maintain them during insertions and deletions.

As soon as we insert a node $x$, we can call the regular TREE-SUCCESSOR($x$) and TREE-PREDECESSOR($x$) functions (that run in $O(\lg n)$) to set its fields. Once the successor and predecessor are determined, we can set the appropriate fields in $x$, and then update the SUCC field of the predecessor of $x$ and the PRED field of the successor of $x$ to point to $x$. When we delete a node, we update its successor and predecessor to point to each other rather than to $x$. To simplify the handling of border cases (e.g. calling SUCCESSOR on the largest element in the tree), it may be useful to keep a dummy node with key $-\infty$ and a dummy node with key $+\infty$.

Finally, MINIMUM and MAXIMUM (which are global properties of the tree) may be maintained directly. When we insert a node, we check if its key is smaller than the current minimum or larger than the current maximum, and update the MINIMUM and/or MAXIMUM pointer if necessary. When we delete a node, we may need to set MINIMUM and/or MAXIMUM to its successor and/or predecessor, respectively, if we are deleting the minimum or maximum element in the tree.

Thus, these operations run in $\Theta(1)$ and may be maintained without changing the asymptotic running time of other tree operations.

(c) **[5 points]** Do problem 14.2-5 on page 311 of CLRS.

**Answer:** Using our answer from the previous part where we showed how to maintain SUCCESSOR and PREDECESSOR pointers with $O(1)$ running time, we can easily perform RB-ENUMERATE.

```
RB-ENUMERATE(x, a, b)
1   start ← TREE-SEARCH(x, a)
2   if start < a
3      then start ← SUCC[start]
4   while start ≤ b
5      print start
6      start ← SUCC[start]
```

We must modify TREE-SEARCH to return the smallest element greater than or equal to $a$ rather than $NIL$ if $a$ is not in the tree.

This algorithm finds and outputs all the elements between $a$ and $b$ in $x$. The

TREE-SEARCH operation takes time $\lg n$ where $n$ is the number of nodes in the tree. The while loop runs $m$ times where $m$ is the number of keys which are output. Each iteration takes constant time since finding the SUCCESSOR of a node now takes constant time. Therefore, the total running time of the algorithm is $\Theta(m + \lg n)$.

We can also answer this question without resorting to augmenting the data structure: we perform an inorder tree walk, except we ignore (do not recurse on) the left children of nodes with key $< a$ and the right children of nodes with key $> b$. Thus, if we consider both children of a node we will output at least one of them, therefore, all nodes not counted by $m$ will lie on two paths from the root to a leaf in the tree. This yields a total running time of $\Theta(m + \lg n)$ as before.

(d) **[6 points]** Do problem 14.3-6 on page 317 of CLRS. You may assume that all elements of $Q$ will always be distinct.

**Answer:** We can keep an augmented red-black tree of the different elements in $Q$. Each node $x$ in the tree will be augmented with the following three fields:

i. $mingap[x]$ is the minimum different between two nodes in the subtree rooted at $x$. If $x$ is a leaf, $mingap[x]$ is $\infty$.

ii. $max[x]$ is the maximum key value in the subtree rooted at $x$.

iii. $min[x]$ is the minimum key value in the subtree rooted at $x$.

We can efficiently calculate these fields based simply on the node $x$ and its children.

$$min[x] = \begin{cases} min[left[x]] & \text{if left child exists} \\ key[x] & \text{otherwise} \end{cases}$$

$$max[x] = \begin{cases} max[right[x]] & \text{if right child exists} \\ key[x] & \text{otherwise} \end{cases}$$

$$mingap[x] = min \begin{cases} mingap[left[x]] & \text{if left child exists} \\ mingap[right[x]] & \text{if right child exists} \\ key[x] - max[left[x]] & \text{if left child exists} \\ min[right[x]] - key[x] & \text{if right child exists} \\ \infty & \end{cases}$$

Notice that to calculate the *mingap* of a node $x$, we compare the *mingap*s of the subtree's rooted at the children as well as the two possible minimum gaps that $x$ can have, namely where you subtract $x$ from the smallest element largest than it and where you subtract the largest element smaller than $x$ from $x$.

All three fields depend only on the node $x$ and its children so we can efficiently maintain them through insertion, deletion, and rotation according to theorem 14.1 in CLRS. Notice that we keep the *min* and *max* values so that we can calculate *mingap* using only information from the node and its children.

The running time of the insert and delete operations are the same as for the red-black tree and are $\Theta(\lg n)$. The procedure $MinGap(Q)$ can simply look in the root node and return the *mingap* field which holds the minimum gap in the tree. Thus, the running time for $MinGap$ is $O(1)$.

(e) **[8 points]** Do problem 14-1(b) on page 318 of CLRS. You may assume the result of 14-1(a) without proof.

**Answer:** We know from part (a) that the point of maximum overlap is the endpoint of one of the intervals. So, we wish to find the endpoint which is the point of maximum overlap. To do this, we will follow the hint and keep a red-black tree of the endpoints of the intervals. With each node $x$, we will associate a field $v[x]$ which equals $+1$ if $x$ is a left endpoint and $-1$ if $x$ is a left endpoint. We will also augment each node with some additional information to allow us to find the point of maximum overlap efficiently.

First, we will provide some intuition. If $x_1, x_2, \ldots, x_n$ are the sorted sequence of endpoints of the intervals, then if we sum from $i = 1$ to $j$ of $v[x_i]$, we find precisely the number of intervals which overlap endpoint $j$ (this assumes our intervals are half-open of the form $[a, b)$). So, the point of maximum overlap will be the value of $j$ which has the maximum value of $\sum_{i=1}^{j} v[x_i]$.

We will augment our red-black tree with enough information to calculate the point of maximum overlap for the subtrees rooted at each node. The point of maximum overlap depends on the cumulative sum of the values of $v[x]$. So, in addition to each $v[x]$, we will keep a variable $SUM[x]$ at each node which is the sum of the $v[i]$ values of all the nodes in $x$'s subtree. Also, we will keep a variable $MAX[x]$ which is the maximum possible cumulative sum of $v[i]$'s in the subtree rooted at $x$. Finally, we will keep $POM[x]$ which is the endpoint $x_i$ which maximizes the $MAX$ expression above. Clearly, $POM[root]$ will be the point of maximum overlap on the entire set of intervals.

We will demonstrate how to compute these fields using only information at each node and its children. Therefore, by theorem 14.1, we can maintain this information efficiently through insertions, deletions, and rotations. The sum of the $v[i]$ of the subtree rooted at node $x$ will simply be
$SUM[x] = SUM[left[x]] + v[x] + SUM[right[x]]$.
The maximum cumulative sum can either be in the left subtree, $x$ itself, or in the right subtree. If it is in the left subtree, it is simply $MAX[left[x]]$. If it is $x$, the cumulative sum till $x$ is $SUM[left[x]] + v[x]$. If it is in the right subtree, we have to add the cumulative sum up till the right subtree to the max value of the right subtree, $SUM[left[x]] + v[x] + MAX[right[x]]$.
$MAX[x] = \max(MAX[left[x]], SUM[left[x]] + v[x], SUM[left[x]] + v[x] + MAX[right[x]])$
The point of maximum overlap is the value of the node which maximized the above expression. ($l[x]$ and $r[x]$ refer to the left and right children of $x$ respectively)

$$PMO[x] = \begin{cases} PMO[l[x]] & \text{if } MAX[x] = MAX[l[x]] \\ x & \text{if } MAX[x] = SUM[l[x]] + x \\ PMO[r[x]] & \text{if } MAX[x] = SUM[l[x]] + x + MAX[r[x]] \end{cases}$$

So, since each just depends on itself and its children, Interval-Insert and Interval-Delete will run in time $\Theta(\lg n)$ since we either insert or delete 2 nodes from the tree. At any time, $PMO(root)$ will hold the point of maximum overlap and

$MAX(root)$ will return how many intervals overlap it. Thus, the operation Find-PMO can be performed in $\Theta(1)$ time.

3. **[15 points]** Dynamic Programming: Optimization

   (a) **[7 points]** You are planning a cross-country trip along a straight highway with $n+1$ gas stations. You start at gas station 0, and the distance to gas station $i$ is $d_i$ miles ($0 = d_0 < d_1 < \ldots < d_n$). Your car's gas tank holds $G > 0$ gallons, and your car travels $m > 0$ miles on each gallon of gas. Assume that $G$, $m$, and all $d_i$ are integers. You start with an empty tank of gas (at gas station 0), and your destination is gas station $n$. You may not run out of gas, although you may arrive at a gas station with an empty tank. The price of gas at gas station $i$ is $p_i$ dollars per gallon (not necessarily an integer). You cannot sell gas.

   Give an algorithm to calculate the cost of the cheapest possible trip. State the running time and space usage of your algorithm in terms of the relevant parameters.

   **Answer:** We will solve this problem using dynamic programming by noting that if we knew the optimal way to leave gas station $i-1$ with any amount of gas in the tank, then we could easily calculate the optimal way to leave gas station $i$ with any amount of gas in the tank. We could do this by taking the minimum over the cost of leaving gas station $i-1$ plus the cost of however much gas we'd need to put in the tank at gas station $i$. Notice that the amount of gas in the tank times $m$ must be an integer in the range from 0 to $mG$ since the distances we travel are given by $gas/m$ and are all integers. From now on, we will call these values of $gas \times m$ units of gas. So, we can build a table $T$ which is $mG$ by $n$.

   We initialize the first row of the table by noticing that the only way to leave gas station 0 with $k$ units of gas is to fill up $k$ units of gas at gas station 0. This has a cost of $(p_0 \times k)/m$. So, our base case of the table $T$ is

   $$T[0,k] = (p_0 \times k)/m.$$

   We can then define each additional row in terms of a minimum over the previous row. For example, let's say we want to get to gas station $i$ with no units of gas in the tank. There is only one way to do that since we can't sell gas. We must leave gas station $i-1$ with exactly enough gas to get to gas station $i$. This distance is $d_i - d_{i-1}$, and the amount of gas necessary is $(d_i - d_{i-1}) \times m$, so our units of gas are exactly the distance, $d_i - d_{i-1}$. So, the value of $T[i,0]$ is equal to $T[i-1, d_i - d_{i-1}]$. Let's now consider the case where we want to get to gas station $i$ with 1 unit of gas in the tank. There are two ways we can acheive this. Either we can leave gas station $i-1$ with one more unit of gas in the tank or we can leave with exactly enough gas to get to $i$ and buy one unit of gas at gas station $i$. $T[i,1] = \min(T[i-1, d_i - d_{i-1} + 1], T[i-1, d_i - d_{i-1}] + (p_i \times 1)/m)$. In general, we take the minimum over many cells in the previous row. First, define $\Delta_i$ as the distance between gas station $i$ and $i-1$ so $\Delta_i = d_i - d_{i-1}$. We can calculate

$T[i,k] = \min_{\Delta_i \le k' \le \Delta_i + k} T[i-1,k'] + (p_i \times (k' - \Delta_i))/m$

Some times we will look in the table for values that don't exists, for example when $k$ equals $mG$. Assume that $T[i-1, k' > mG]$ will return infinity.

After filling the table, we will search in the row $T[n,k]$ for the lowest value and return that as the cheapest possible trip. Notice that this cheapest cell will always be at $T[n,0]$ since any extra gas we have will have cost us something, and we may as well not have bought it. Notice also that we can return the actual trip by keeping pointers for each cell which point to the cell which caused the minimum.

For each cell in the table, we have to take a minimum over at most $mG$ items. Since there are $n \times mG$ cells in the table, the running time for this algorithm is $\Theta(n(mG)^2)$. The space requirement is the size of the table which is $\Theta(nmG)$. Since calculating each row in the table only depends on the previous row, we can delete rows as we are finished processing them. So, we can reduce the space requirement to $\Theta(mG)$. However, if we wish to calculate the actual trip, we will need to maintain the $\Theta(nmG)$ table of pointers, which we aren't able to delete, since we don't know what our optimal trip is until after calculating the whole table. So, in that case the space requirement remains at $\Theta(nmG)$.

If we are clever in the way we update cells, we can actually reduce the running time of the algorithm to $\Theta(nmG)$ as well. Let's say we break the update to the cells into two phases, before adding gas at station $i$ and after adding gas at station $i$. For the pre-fillup phase, we only have one way to arrive at a gas station with a given amount of gas $k$, so we simply set each

$T[i,k] = T[i-1, \Delta_i + k]$

where again we assume that $T[i-1, k' > mG] = \infty$. For the post-fillup stage, we can update each $T[i,k]$ by looking at the cost of adding one unit of gas to $T[i,k-1]$, assuming $T[i,k-1]$ already holds the minimum value to get there.

$T[i,k] = T[i,k-1] + (p_i \times 1)/m$ if that value is less than the current value in the cell.

Since each cell now only looks back at a constant number of cells instead of $O(mG)$ cells, the running time is reduced to $\Theta(nmG)$.

(b) **[8 points]** Do problem 15-7 on page 369 of CLRS.

**Answer:** Since there are so many variables in this problem, the first thing we need to figure out is what our optimal subproblems are. We do this by considering an optimal schedule over jobs $a_1, \ldots, a_j$ that runs until time $t$. Assume that we know which subset of jobs we perform to get the highest profit. What order do we need to perform those jobs in? Notice that the job with the latest deadline should be performed last. If we don't perform that job last, then we will waste the time between the second last deadline and the last deadline. The same thing

then applies for the second last deadline job being performed second last. We can argue that the subset of jobs that we perform will be done in increasing order of deadline.

This gives us our optimal subproblems. Namely, we order the jobs by increasing deadline. When we consider job $a_i$ finishing at any time $t$ (we assume these are now in sorted order so it has the $i$th deadline) we can simply look back at the optimal way to schedule the $i-1$ jobs and whether or not we add $a_i$ to the schedule. We will also make the additional assumption that we leave no time gaps between the jobs. It is easy to argue that if we have a schedule with time gaps between jobs, we can also do the jobs in the same order with no time gaps and receive the same profit and possibly more.

The actual algorithm is as follows. We keep a grid of the $n$ jobs versus the time, which can run up till $d_n$ (since the jobs are now sorted by deadline). Notice that since the processing times are integers between 1 and $n$, the maximum time taken to complete all the jobs is at most $n^2$. So, if we have any deadline which exceeds $n^2$, we can simply replace it by $n^2$. Thus, our table is at most $n \times n^2$. Each cell in the table $i, j$ will represent the maximum profit possible for scheduling the first $i$ jobs in time *exactly* $j$, assuming that we have no gaps between our jobs.

We initialize the first row in the table by figuring out our possible profit based on completing the first job at exactly time $t$. Since we assume no gaps in our schedule, we must start the first job at time 0. Therefore, we will have a profit of $p_1$ in the cell at $t_1$ if $t_1 \leq d_1$ and a profit of zero otherwise. We'll initialize all other cells to zero. So, if we call our table $T$, we have

$$T[1,t] = \begin{cases} 0 & \text{if } t \neq t_1 \\ p_1 & \text{if } t = t_1 \leq d_1 \\ 0 & \text{if } t = t_1 > d_1 \end{cases}$$

Then, we can set the remaining cells in our table based simply on the previous row. At each cell $T[i,t]$, we have the choice of whether or not to perform job $i$. If we decide not to perform job $i$, then our profit is simply $T[i-1,t]$. If we decide to perform job $i$, then we know it takes $t_i$ units of time to complete it. So, we must finish the previous jobs exactly at time $t - t_i$. We will get profit for the job if $t < d_i$. We will pick the maximum profit from these two choices.

$$T[i,t] = \max \begin{cases} T[i-1,t] & \\ T[i-1,t-t_i] + p_i & \text{if } t \leq d_i \\ T[i-1,t-t_i] & \text{if } t > d_i \end{cases}$$

At each cell, we will also keep a pointer to the cell which maximized the above expression. After filling in the whole table, we can search through the $T[n,t]$ row for the highest profit. Then, we can follow the pointers back to find the schedule. If in row $i$ we take the pointer above us, we add $i$ to the end of the schedule. Otherwise, if we take the pointer that is diagonal, we say we complete $i$ at that time.

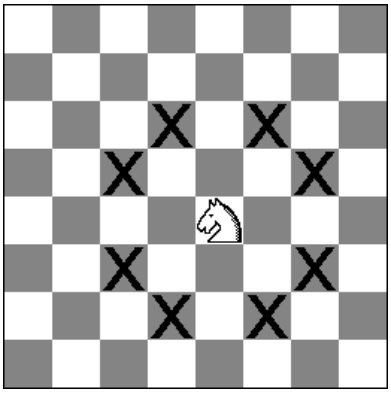
The running time of this algorithm is the number of cells in the table since each cell takes a constant amount of time to evaluate. The search through the last row

and following of the pointers both take time $\Theta(n)$. Therefore, the running time is $\Theta(n^3)$ if $d_n \geq n^2$ and $\Theta(nd_n)$ if $d_n < n^2$.

The space requirement for this algorithm is the size of the table, which is the same as the running time. Notice that since we want the actual schedule, we can't compress the size requirement.

4. **[15 points]** Dynamic Programming: Counting and Games

   (a) **[7 points]** Consider a chess knight on an $n \times n$ chessboard. The knight may move from any square $(i, j)$ to $(i', j')$ if and only if $(|i' - i|, |j' - j|) \in \{(1, 2), (2, 1)\}$ and that square is on the board; in other words, the knight may move two squares in one direction and one square in a perpendicular direction (see diagram).



Give an algorithm to calculate the number of ways in which the knight can travel from square $(i_s, j_s)$ to $(i_t, j_t)$ in exactly $k \geq 0$ moves. State the running time and space usage of your algorithm in terms of the relevant parameters.

**Answer:** We can calculate the solution to this problem by noticing that if we knew the number of ways to get to every square on the board in $k-1$ moves from the starting location, we could easily calculate the number of ways to get to a square in $k$ moves by simply summing over the atmost 8 squares that the knight could move from. Each way to get to the predecessor in $k-1$ moves contributes one way to get to the square in $k$ moves.

Our DP matrix will be $n \times n \times k$. We initialize for all $k = 0$ the number of ways to get to that square in 0 moves. Namely,
$Ways(a, b, 0) = 1$ if $a = i_s$ AND $b = j_s$ and zero otherwise. We can build up our DP matrix for each $0 < i \leq k$ from these values. For each value of $i = 1 \ldots k$, and for each square $(a, b)$ on the $n \times n$ board, we set the value of $Ways(a, b, i) = \sum_{(u,v) \in neighbors(a,b)} Ways(u, v, i - 1)$. The neighbors of a cell are simply those that follow the condition given above for the legal knight's moves. At the end, we look at $Ways(i_t, j_t, k)$ to find the number of ways to get to $(i_t, j_t)$ from $(i_s, j_s)$ in exactly $k$ moves.

Notice here a trend common to DP problems. In order to solve how many ways there are to get from a certain cell to another cell in $k$ moves, we actually solve

more than asked for. We figure out how many ways there are to get from the start cell to **every** cell in $k$ moves. While it seems like we do more computation, this actually makes solving the next layer more efficient.

The running time of this algorithm will be proportional to the number of cells in the matrix, since each cell takes a constant amount of time to calculate. Thus, the running time is $\Theta(n^2k)$. The amount of space is the same at $\Theta(n^2k)$. Notice, however, that to calculate the matrix for a given value of $i$ we only use the values at $i-1$. So, at any time, we don't need to store the entire $n^2k$ matrix, we only need two levels of it. If we delete levels (in $k$) as we finish using them, we only need space $\Theta(n^2)$.

(b) [**8 points**] The cat and mouse game is played as follows: there is an $n \times n$ board with a cat in (initial) location $(i_c, j_c)$, a mouse in (initial) location $(i_m, j_m)$, and a piece of swiss cheese in location $(i_s, j_s)$.

Each cell $(i, j)$ on the board has up to four neighbors: north $(i, j+1)$, south $(i, j-1)$, east $(i+1, j)$, and west $(i-1, j)$. All four cells adjacent in this manner are considered neighbors, except for the edges of the board and where there are **walls blocking some of the directions.** Therefore, assume that for any given cell $(i, j)$ you have a list of zero to four neighboring cells $NBR(i, j)$ specified as you like (linked list, for example). You may assume the neighbor relationship is **symmetric**, i.e. if $(i', j') \in NBR(i, j)$, then $(i, j) \in NBR(i', j')$ and vice versa.

The game alternates between the moves of the mouse, who moves first, and the cat. The cheese does not move. If the mouse reaches the cheese before the cat catches it (i.e. without ever entering the same cell as the cat), then the mouse wins (the cheese endows the mouse with super powers). If the cat catches the mouse before the mouse can reach the cheese, the cat wins. If both conditions are satisfied simultaneously, i.e. the mouse enters a cell with the cat and the cheese in it, the game is considered a draw. You may assume that if none of this occurs within $2n^2$ moves, the game ends in a draw. Also, assume the cat and the mouse **must** move every round.

Assume that the cat and mouse are infinitely intelligent, can see the entire board and location of each other, and always make optimal decisions.

Give a dynamic programming algorithm to predict who will win the game. State its running time (a good upper bound is sufficient) and space usage in terms of $n$.

**Answer:** We can solve this problem by noticing that if the mouse knew the outcome for each of his at most four possible moves, he would know which way to go. If there was a move for which the mouse wins, he'll choose that move and win. If not, but there's a move for which the game is a draw, he'll choose that and draw. Otherwise, the cat wins for every move of the mouse, so we know the cat wins. Similar reasoning holds for the cat's strategy.

From this, we can see that our subproblems are who wins for each possible board configuration, whose move it is, and which move we're on. Since after $2n^2$ moves the game ends in a draw, we know the base case of whether the cat wins, mouse

wins, or a draw for the $2n^2$ move. We can define a set of boards for $2n^2+1$ which have the cat winning if the cat and the mouse are on the same square not on the cheese, the mouse winning if the mouse is on the cheese and the cat is elsewhere, and a draw otherwise.

We need to keep track of all possible boards for each move $k$. A board position consists of a cat position ($n^2$) and a mouse position ($n^2$), as well as whose turn it is. Since the mouse goes first, we know that for odd $k$ it is the mouse's turn and for even $k$ it is the cat's turn. We keep $2n^2$ possible sets of boards, each corresponding to which move number we are on. We initialize the $k = 2n^2$ board as above, and work backwards.

For each $0 \leq k < 2n^2$ and each of the $n^4$ possible board configurations, we wish to calculate who wins or if it is a draw. Our base cases are if the cat and the mouse are on the same square not on the cheese, then the cat wins. If the cat and the mouse are both on the same square as the cheese, then it's a draw. And if the mouse is on the cheese and the cat is elsewhere, then the mouse wins. Otherwise, we look at the at most four possible board configurations in the $k+1$ level corresponding to the possible moves. If $k$ is odd (mouse's turn) and there is a possible board where the mouse wins, then we label this board as mouse, else if there is a possible board where there is a draw, we label this board draw, else we label this board cat. The symmetric case holds for $k$ even (cat's turn).

After filling out our $(2n^2) \cdot (n^4)$ table, we look at the board in the $k = 0$ level corresponding to the starting locations of the cat and mouse and give the label of that board (cat, mouse, draw) as who wins the game.

Each cell looks at a constant number of other cells, so the running time of this algorithm is $\Theta(n^6)$. The memory requirement is also $\Theta(n^6)$. Notice however that we only need the $k+1$ boards to calculate the $k$ boards. So, we need only store two sets of board configurations at a time, leading to a space requirement of $\Theta(n^4)$.

We can actually achieve a running time of $\Theta(n^4)$ if we do our updates in an intelligent manner. Rather than doing update move by move, we will initailiaze our board ($n^2 \times n^2 \times 2$ states) with the known terminal states (cat win, mouse win, draw), and the rest will start out unknown. As some state is updated, it may trigger up to four updates for its neighboring states. If we keep the states that need to be updated in a data structure such as a queue, we can process them one by one, doing a constant amount of work per update. We begin with neighbors of the terminal states in the queue, and continue adding neighbors to the queue for each update. The total number of updates done is at most the number of states on the board, which yields a running time of $O(n^4)$. Any state that fails to be updated when the queue is empty is then labeled as a draw state due to the assumption that any game that lasts at least $2n^2$ moves will end in a draw.

5. **[14 points]** Dynamic Programming: Sequences and Sparse DP

   (a) **[6 points]** Let $X$ be a sequence $x_1, \ldots, x_n$ with associated weights $w_1, \ldots, w_n$. An increasing subsequence of $t$ elements satisfies $x_{j_1} < \ldots < x_{j_t}$ and its weight

is $\sum_{i=1}^{t} w_i$. Describe a $\Theta(n \lg n)$ algorithm for finding the heaviest increasing subsequence.

**Answer:** This problem is similar to the longest increasing subsequence problem, where in that problem all elements had weight equal to 1. In that problem, we were able to store the lowest value of the last element of a subsequence of length $i$ in an array $L[i]$. We showed in class that the $L[i]$ array remained sorted. For each element $j$ in our sequence, we were able to perform a binary search in the $L$ array for the largest value smaller than $j$ and update our array accordingly. This led to a total running time of $\Theta(n \lg n)$ since each binary search took time $\Theta(\lg n)$.

In this case, we can't keep an array since the weights may not be integral values. However, we can keep a red-black tree which will allow us to find elements in time $\Theta(\lg n)$. Also notice that adding a single element may cause **many** previous subsequences to never be used again. For example, if we have the sequence of $(key, weight)$ of $(3,3), (2,2), (1,1), (0,5)$, we would find for the first three elements a subsequence of weight 1 ending in key 1, one of weight 2 ending in key 2 and one of weight 3 ending in key 3. Notice that all three of these subsequences may still be part of our heaviest subsequence. But, when we add the fourth element which gives a subsequence with weight 5 ending in key 0, none of the previous 3 subsequences can be part of our heaviest subsequence, since any time we could append elements to those, we could append them to the subsequence ending in 0 and achieve a higher weight. Thus, in order to maintain only the useless sequences we may now have to delete more than one element at a time.

Notice that the total number of elements deleted over the course of the algorithm is $n$, which yields running time of $O(n \lg n)$. However, each particular update may now take $\Theta(n)$, so without this realization (which is a mild form of amortized analysis) we do not have the desired bound with this approach.

The following is an alternative solution that uses augmented red-black trees. If you're comfortable with counting the deletion time separate from insertion time for the solution above, feel free to skip it. If you fear amortization like the plague, read on.

So, for each key, we will wish to compute the heaviest subsequence ending in that key. We will keep an augmented red-black tree of these ($key$, weight of subsequence) pairs keyed on the $key$. We will refer to the weight of the subsequence as $w_s$, not to be confused with the weight of the element. We will also keep pointers for each element of which element is its predecessor in the heaviest subsequence ending in that element. At the end, we will search through the tree for the highest weight and follow the pointers to find our heaviest subsequence.

For each element $i$, we will need to find in the tree the node $x$ with the highest weight such that $key[x] < key[i]$. $x$ is the node that we should append $i$ onto to get the heaviest increasing subsequence ending in $i$. To do this, we will augment each node with $W[x]$, the maximum weight of subsequences ending in elements less than or equal to $x$ in the subtree rooted at $x$. We can efficiently calculate $W[x]$ as $\max(W[left[x]], w_s[x])$. Since this property only depends on the values stored

at node $x$ and its children, we can maintain it during insertion, deletion, and rotation. We will also keep at each node a pointer $P[x]$ which points to the node which attains that maximum weight. $P[x] = P[left[x]]$ if $W[left[x]] > weight[x]$ and $P[x] = x$ if $weight[x] \geq W[left[x]]$.

When we look at a new element $i$ , we will insert it into the red-black tree. We initialize $totalweight = -\infty$ and $pointer = NIL$. We'll search in the tree for where $i$ should go. Everytime we follow a left child, we do nothing. Everytime we follow a right child of some parent $x$, we set $totalweight = \max(totalweight, W[x])$ and $pointer = P[x] if W[x] > totalweight$. So, we essentially keep track of the maximum weight and the pointer to the node that had it for all keys that are less than $i$. When we find where to insert $x$, we set the weight of the subsequence ending in $x$ to be $w_s[x] = totalweight + weight[x]$. We also set $W[x] = w_s[x]$ and $P[x] = x$. In addition, in our original array, we add a pointer from $x$ to $pointer$, telling us how to backtrack to find the heaviest increasing subsequence.

We do this for all elements in the sequence (starting with the dummy element $-\infty$ with weight 0). Each insert into the tree takes time $\lg n$ and we have to perform $n$ of them. At the end, we search through the tree for the highest weight ($\Theta(n)$) and follow the pointers back from that element to find the actual heaviest increasing sequence ($O(n)$). So, the total running time is $\Theta(n \lg n)$.

(b) **[8 points]** Let $X = x_1, \ldots, x_m$ and $Y = y_1, \ldots, y_n$ be two sequences. Assume that the number of pairs $(x_i, y_j)$ that match (i.e. $x_i = y_j$) is small, i.e. $\emptyset(k)$ for some $k(m, n)$. Assume the set of matching pairs is given to you as part of the input. Give an $O(k \lg k)$ algorithm for finding the longest common subsequence of $X$ and $Y$ under these conditions.

**Hint:** reduce this to another sparse DP problem we have seen.

**Answer:** We will try to reduce our problem to the longest increasing subsequence problem. Notice that if we try to read through the entire input arrays, we will take time $\Theta(n + m)$ which could be bigger than $k$, so we are restricted to simply looking through the pairs of elements given. The longest common subsequence of $X$ and $Y$ must have increasing indexes in both $X$ and $Y$. Thus, if we sort the pairs according to their index in $X$ and feed the corresponding $Y$ indexes into the longest increasing subsequence algorithm, we will obtain a common subsequence that is nondecreasing in the $X$ index and increasing in the $Y$ index. To avoid the problem of potentially having several pairs $(x_i, y_j)$ and $(x_i, y_{j'})$ chosen by the LIS algorithm, we place the pairs with equal values of $x_i$ in decreasing order of $y_j$, so that two pairs with the same $X$ index can never be returned as part of a subsequence with increasing $Y$ index. Sorting can be done using mergesort in $\Theta(k \lg k)$ time, and the sparse version of LIS runs in $\Theta(k \lg k)$ time, so the overall running time is $\Theta(k \lg k)$.

**CS161: Design and Analysis of Algorithms** **Summer 2004**

# Problem Set #5 Solutions

## General Notes

- **Regrade Policy:** If you believe an error has been made in the grading of your problem set, you may resubmit it for a regrade. If the error consists of more than an error in addition of points, please include with your problem set a detailed explanation of which problems you think you deserve more points on and why. We reserve the right to regrade your entire problem set, so your final grade may either increase or decrease.

1. **[25 points]** Coin Changing

   Consider a currency system with $k \geq 1$ distinct (integer) denominations $d_1 < \ldots < d_k$. Assume that $d_1 = 1$ so that any integer amount greater than 0 can be formed. Throughout the problem, assume there are always coins of every denomination available when needed (i.e. you cannot "run out" of any denomination). Our goal will be to determine how to make change using the fewest number of coins.

   Some parts of this problem are similar to problem 16-1 on page 402 of CLRS. You may wish to read the statement of that problem to see how similar questions are presented.

   (a) **[2 points]** Describe a greedy strategy for making change using as few coins as possible. This strategy should work for the US coin system which uses denominations $\{1, 5, 10, 25\}$. You do not need to prove the correctness of this strategy for this set of denominations.

   **Answer:** The greedy strategy consists of always taking the largest denomination that we can at the time. We repeat this until we have the correct amount of change.

   MAKE-CHANGE($M$)
   1 **for** $i = k$ **downto** 1
   2 $\quad count[i] \leftarrow \lfloor M/d_i \rfloor$
   3 $\quad M \leftarrow M - count[i] \cdot d_i$

   Since for each denomination we calculate the number of coins we can take until we would make more than the amount of change asked for, this operation takes time $O(k)$.

   We can also write this in a slightly less efficient manner that will be useful when analyzing the correctness of the greedy strategy. In this case, we simply take one coin at a time of the largest denomination possible.

   MAKE-CHANGE-SLOW($M$)
   1 **for** $i = k$ **downto** 1
   2 $\quad$ **while** $M \geq d_i$
   3 $\quad\quad count[i] \leftarrow count[i] + 1$
   4 $\quad\quad M \leftarrow M - d_i$

   This runs in time $O(C + k)$ where $C$ is the number of coins in the greedy solution.

(b) **[5 points]** Suppose that $d_i = c^{i-1}$ for some integer $c > 1$, i.e. the denominations are $1, c, c^2, \ldots, c^{k-1}$. Prove that the greedy strategy from part (a) is optimal.

**Answer:** We'll represent a change solution by a series of counts, one for each coin denomination. The counts must obey the property $\sum_{i=1}^{k} count[i] \cdot d_i = M$. The number of coins in this solution is $\sum_{i=1}^{k} count[i]$. Note that the counts for each coin denomination $d_i < d_k$ must be strictly less than $c$ in an optimal solution. Otherwise, if we have $count[i] \geq c$, we can replace $c$ of the $d_i = c^{i-1}$ coins which have value $c * c^{i-1} = c^i$ with one $d_{i+1} = c^i$ coin and get a smaller count.

We will prove by contradiction that the greedy solution is optimal. Let the optimal count values be represented by $optimal[i]$. Let $j$ be the first index (highest) for which $count[j] \neq optimal[j]$. We know that $count[j] > optimal[j]$ because the greedy solution always takes the maximum number of any coin and this is the first value on which they disagree. If the optimal solution chose more of coin $d_j$, then it would have made too much change. Let $B = \sum_{i=1}^{j-1} count[i] \cdot d_i$, the amount of change that the greedy solution makes with the coins $1 \ldots j-1$. Since the optimal solution used fewer coins of denomination $d_j$, we know that the amount of change that the optimal solution makes with coins $1 \ldots j-1$ is $D = B + (count[j] - optimal[j]) \cdot d_j = B + (count[j] - optimal[j]) \cdot c^{j-1}$. Since this value is greater than $c^{j-1}$, we will show that the optimal solution must take at least $c$ coins for some coin $d_1 \ldots d_{j-1}$.

Assume that the optimal solution has $optimal[i] < c$ for $i = 1 \ldots j-1$. Then, the amount of change that the optimal solution makes with the first $j-1$ coins is

$$\begin{aligned} D &= \sum_{i=1}^{j-1} optimal[i] \cdot d_i \\ &\leq \sum_{i=1}^{j-1} (c-1) \cdot c^{i-1} \\ &= (c-1) \sum_{i=1}^{j-1} c^{i-1} \\ &= (c-1) \sum_{i=0}^{j-2} c^i \\ &= (c-1) \frac{c^{j-1} - 1}{c-1} \\ &= c^{j-1} - 1 \end{aligned}$$

However, this contradicts the fact that $D$ was greater than $c^{j-1}$. Thus, the optimal solution must take at least $c$ coins for some denomination less than $d_j$. But, as we showed above, this solution can't be optimal. This is a contradiction and therefore the greedy solution must take the same number of coins for all coins as the optimal solution, and thus must itself be optimal.

(c) **[2 points]** Give a set of coin denominations for which the greedy algorithm will

not always yield the optimal solution. Your set should include $d_1 = 1$ so that every positive integer amount can be formed.

**Answer:** A simple example is the set of US denominations without the nickel. If you simply have the values $\{1, 10, 25\}$, the greedy solution will fail. For example, if you are trying to make change for 30 cents, the greedy solution will first take a quarter, followed by 5 pennies, a total of six coins. However, the optimal solution actually consists of three dimes.

(d) **[5 points]** Suppose you wish to make change for the amount $M$ using an arbitrary set of denominations (although we still make the assumption that $d_i$ are distinct integers and $d_1 = 1$). Give an $O(Mk)$ time algorithm to calculate the minimum number of coins needed. State the (asymptotic) space usage of your algorithm in terms of the relevant parameters.

**Answer:** We can solve this problem using dynamic programming. Notice that for any value $M$, we can calculate the optimal solution by looking back at the number of coins needed to make $M - d_i$ coins for all $i$ and choosing the smallest number. We fill an array $T$ which ranges from $1 \ldots M$. At each point in the array, we store the optimal number of coins needed to make change. We initialize the cell $T[0]$ with the value 0. Then, for every other cell up to $T[M]$ we calculate

$$T[i] = 1 + \min_{1 \le j \le k} T[i - d_j]$$

where we assume that for $x < 0$, $T[x] = \infty$. We fill in the table up till $T[M]$ and then simply look in that cell for the optimal number of coins. Since we have to fill in $M$ values in the table and each value looks back at $k$ different cells and takes the minimum, the running time of this algorithm is $O(Mk)$. At any time, we can look back until the $i - d_k$ cell, so the space usage is $O(\min(d_k, M))$.

(e) **[5 points]** Suppose Eugene owes Nina an amount $M$. Give an $O(Mkd_k)$ time algorithm to determine the minimum number of coins that need to exchange hands in order to settle the debt (Nina can give change). Assume that neither Eugene nor Nina will run out of any denomination. State the (asymptotic) space usage of your algorithm in terms of the relevant parameters.

**Answer:** We can define an upper bound on the amount of money that Eugene gives Nina that can be part of an optimal solution. Let's say that Eugene gives Nina an amount more than $Md_k$ and Nina gives change. Eugene gives at least $M$ coins, and Nina still gives back coins. But, we know that Eugene could simply have given Nina $M$ coins of denomination $d_1 = 1$ to make a total value of $M$ without Nina having to give any change. So, we know that Eugene giving an amount more than $Md_k$ can't be an optimal solution.

We can therefore use our solution from part (d) to find the optimal ways of giving change for all values $1 \ldots Md_k$. Then, for each value $i$ that Eugene can give in the range $M \ldots Md_k$, Nina will need to give $i - M$ change. We simply sum the number of coins that Eugene gives plus the number of coins that Nina gives: $T[i] + T[i - M]$. The minimum over all $i$ is the optimal way to settle the debt.

To find the optimal ways to make change for all values up till $Md_k$ takes time $O(Mkd_k)$ using the algorithm in part (d). Notice that in the process of computing the optimum for $Md_k$ the algorithm finds the optimal ways to make change for all the values $1 \ldots Md_k$ so we only need to run it once. Summing and taking the minimum will take time $O(Md_k)$, so the running time is bounded by the dynamic programming part. The algorithm will need to store the previous $d_k$ solutions for the dynamic programming and the previous $M$ solutions for the summation of $T[i] + T[i - M]$. So, the space usage of the algorithm is $O(d_k + M)$.

(f) **[6 points]** Consider again the greedy algorithm stated in part (a) of the problem. We want to decide, in a given currency system $d_1 < \ldots < d_k$, whether the greedy algorithm is optimal. Give a dynamic programming algorithm to determine the smallest amount $M$ for which the greedy algorithm fails to produce the optimal solution. If the greedy approach is always optimal your algorithm should detect and report this. State the (asymptotic) running time and space usage of your algorithm in terms of the relevant parameters. You may assume that $k \geq 2$ so that the greedy algorithm is not trivial.

**Hint:** Can you find an upper bound $B$ such that if greedy is optimal for all $M \leq B$, then greedy must be optimal for all $M$?

**Answer:** We claim that if a counterexample to greedy's optimality exists, it must exist for some $M < d_k + d_{k-1}$. The main idea is that for such $M$, the greedy choice is to choose $d_k$, but we will show that doing so either yields an optimal solution or a smaller counterexample to greedy's optimality.

Thus, suppose (for a contradiction) that $M \geq d_k + d_{k-1}$ is the smallest amount such that greedy is not optimal for this amount. This means that $OPT[M] > OPT[M - d_k] + 1$. However, there must be some denomination $d_j \neq d_k$ such that $OPT[M] = OPT[M - d_j] + 1$ (recall how we computed $OPT[i]$ in part (d)). By assumption, greedy must be optimal for this amount, otherwise we have a smaller counterexample. However, since $d_j \leq d_{k-1}$ (since $j \neq k$), this means the optimal greedy choice for $M - d_j \geq d_k$ is in fact $d_k$; therefore, $OPT[M - d_j] = OPT[M - d_j - d_k] + 1$. Thus,

$$OPT[M] = OPT[M - d_j - d_k] + 2$$

However, notice that we can go from $M - d_k - d_j$ to $M - d_k$ using only the coin $d_j$; therefore, $OPT[M - d_k] \leq OPT[M - d_k - d_j] + 1$. Combining this with the first inequality we established yields

$$OPT[M] > OPT[M - d_k - d_j] + 2$$

for the desired contradiction.

Thus, we simply need to compute the optimal solutions for $M = 1 \ldots d_{k-1} + d_k$, using the algorithm from part (d). As we compute optimum, we must check that greedy is still optimal by checking whether $OPT[M] = OPT[M - d_j] + 1$, where $d_j$ is the largest denomination $\leq M$. If we fail to find a counterexample before $M = d_k + d_{k-1}$, we may stop and report that greedy is always optimal.

The space requirement $O(d_k)$ and the running time of $O(kd_k)$ are obtained by letting $M = O(d_k)$ in the results from part (d).

2. **[16 points]** Amortized Weight-Balanced Trees

   Throughout this problem, your explanations should be as complete as possible, but they do not need to be formal. In particular, rather than proving tree properties by induction you may simply argue why they should hold.

   (a) **[3 points]** Do problem 17-3(a) on page 427 of CLRS.
   **Answer:** We first perform an in-order walk starting from $x$ and store the sorted output in an array. This will require space $\Theta(size[x])$. Then, to rebuild the subtree rooted at $x$, we start by taking the median of the array which we can find in constant time (by calculating its address in the array) and put it at the root of the subtree. This guarantees that the root of the subtree is 1/2-balanced. We recursively repeat this for the two halves of the array to rebuild the subtrees. The total time for the algorithm follows the recurrence $T(size[x]) = 2T(size[x]/2) + 1 = \Theta(size[x])$.

   (b) **[2 points]** Do problem 17-3(b) on page 427 of CLRS.
   **Answer:** If we perform a search in an $n$-node $\alpha$-balanced binary search tree, the worst case split at any point in a subtree with $i$ nodes is recursing to a subtree with $\alpha i$ nodes since we know that the number of nodes in any subtree is bounded by this value. So, the recurrence is $T[n] = T[\alpha n] + 1 = \Theta(\log_{1/\alpha} n)$, which since $\alpha$ is a constant is simply $\Theta(\log n)$.

   (c) **[3 points]** Do problem 17-3(c) on page 427 of CLRS.
   **Answer:** If we define the potential of the tree as

$$\Phi(T) = c \sum_{x \in T : \Delta(x) \geq 2} \Delta(x)$$

   where $\Delta(x) = |size[left[x]] - size[right[x]]|$, then the potential for any BST is non-negative. This is easy to see as every term in the summation is the absolute value of the difference in the subtree sizes and therefore is non-negative. Also, we assume that the constant $c$ is positive, thus giving a positive potential $\Phi(T)$.

   For a 1/2-balanced BST, we know that $size[left[x]] \leq (1/2)size[x]$ and $size[right[x]] \leq (1/2)size[x]$ for all nodes $x$ in the tree. We will prove by contradiction that $\Delta(x) < 2$ for all $x$ in the tree. Assume that $\Delta(x) \geq 2$. Then, $|size[left[x]] - size[right[x]]| \geq 2$. Without loss of generality, assume that the left subtree is larger, so we can drop the absolute value signs. We also know that the sum of the sizes of the left subtree and the right subtree must be $size[x] - 1$. Substituting this in for $size[right[x]]$, we find

$$\begin{aligned} size[left[x]] - size[right[x]] &\geq 2 \\ size[left[x]] - (size[x] - size[left[x]] - 1) &\geq 2 \\ 2 \cdot size[left[x]] &\geq 1 + size[x] \\ size[left[x]] &\geq 1/2 + (1/2)size[x] \end{aligned}$$

But, we know from the $\alpha$-balanced condition that $size[left[x]] \leq (1/2)size[x]$, which is a contradiction. Therefore, for all nodes $x$, $\Delta(x) < 2$. The summation in the potential equation is only over nodes with $\Delta(x) \geq 2$. Since no such nodes exist, the summation is 0, and thus $\Phi(T) = 0$ for a 1/2-balanced tree.

(d) **[4 points]** Do problem 17-3(d) on page 427 of CLRS.

**Answer:** We need to pick the constant $c$ such that at any time that we need to rebuild a subtree of size $m$, we have that the potential $\Phi(T)$ is at least $m$. This is because the amortized cost of rebuilding the subtree is equal to the actual cost plus the difference in potential. If we want a constant amortized cost, we need

$$\begin{aligned} \hat{c}_{rebuild} &= c_{rebuild} + \Phi_i - \Phi_{i-1} \\ O(1) &= m + \Phi_i - \Phi_{i-1} \\ \Phi_{i-1} &\geq m \end{aligned}$$

since we know that the end potential $\Phi_i$ is always greater than zero.

We need to figure out the minimum possible potential in the tree that would cause us to rebuild a subtree of size $m$ rooted at $x$. $x$ must not be $\alpha$-balanced, or we wouldn't need to rebuild the subtree. Say the left subtree is larger. Then, to violate the $\alpha$-balanced criteria, we must have $size[left[x]] > \alpha \cdot m$. Since the size of the subtrees must equal $m-1$, we know that $size[right[x]] = m-1-size[left[x]] < m - 1 - \alpha \cdot m = (1-\alpha)m - 1$.

We have

$$\begin{aligned} \Delta(x) &= size[left[x]] - size[right[x]] \\ &> \alpha \cdot m - ((1-\alpha)m - 1) \\ &= (2\alpha - 1)m + 1 \end{aligned}$$

We can then bound the total tree potential $\Phi(T) > c((2\alpha-1)m+1)$. We need the potential larger than $m$ to pay the cost of rebuilding the subtree, so

$$\begin{aligned} m &\leq c((2\alpha-1)m+1) \\ c &\geq \frac{m}{(2\alpha-1)m+1} \\ &= \frac{1}{2\alpha - 1 + 1/m} \\ &\geq \frac{1}{2\alpha} \end{aligned}$$

Therefore, if we pick $c$ larger than this constant based on $\alpha$, we can rebuild the subtree of size $m$ in amortized cost $O(1)$.

(e) **[4 points]** Do problem 17-3(e) on page 427 of CLRS.

**Answer:** The amortized cost of the insert or delete operation in an $n$-node $\alpha$-balanced tree is the actual cost plus the difference in potential between the two states. From part (b), we showed that search took time $O(\lg n)$ in an $\alpha$-balanced tree, so the actual time to insert or delete will be $O(\lg n)$. When we insert or

delete a node $x$, we can only change the $\Delta(i)$ for nodes $i$ that are on the path from the node $x$ to the root. All other $\Delta(i)$ will remain the same since we don't change their subtree sizes. At worst, we will increase each of the $\Delta(i)$ for $i$ in the path by 1 since we may add the node $x$ to the larger subtree in every case. Again, as we showed in part (b), there are $O(\lg n)$ such nodes. The potential $\Phi(T)$ can therefore increase by at most $c\sum_{i \in path} 1 = O(c \lg n) = O(\lg n)$. So, the amortized cost for insertion and deletion is $O(\lg n) + O(\lg n) = O(\lg n)$.

3. **[10 points]** Off-Line Minimum

   Throughout this problem, your explanations should be as complete as possible, but they do not need to be formal. In particular, when asked to prove the correctness of the algorithm given in the problem, you do not need to use loop invariants; you may simply argue why a certain property should hold.

   (a) **[1 point]** Do problem 21-1(a) on page 519 of CLRS.

   **Answer:** For the sequence $4, 8, E, 3, E, 9, 2, 6, E, E, E, 1, 7, E, 5$ the *extracted* array will hold $4, 3, 2, 6, 8, 1$.

   (b) **[5 points]** Do problem 21-1(b) on page 519 of CLRS.

   **Answer:** We will show that the *extracted* array is correct by contradiction. Assume that the *extracted* array is not correct. Let $x = extracted[j]$ be the smallest value $extracted[j]$ for which the *extracted* array is incorrect. Let the correct solution reside in the array *correct*. Call $y$ the value of $correct[j]$. We have two cases, one where $x > y$ and one where $x < y$. We will prove that neither case can occur.

   Assume that $x > y$. Then the element $y$ can't appear in the *extracted* array, or it would have been the smallest value for which the *extracted* array was incorrect. Since we've already processed $y$ before processing $x$, it must have had a set value of $m + 1$. But, if $correct[j]$ was set to $y$, then $y$ must initially have been in some $K_i$ with $i \leq j$. Since $extracted[j]$ had no value yet when we processed $y$, then we couldn't have put $y$ in set $K_{m+1}$ since we only union with the set above us and we haven't unioned $K_j$ yet. Therefore, we can't have $x > y$.

   Assume that $x < y$. We argue that the element $x$ must appear in the *correct* array. Obviously $x$ must appear before the $j$th extraction in the original sequence since the OFF-LINE-MINIMUM algorithm never moves sets of elements backwards, it only unions with sets later than it. If we hadn't extracted $x$ by the $j$th extraction, then the optimal solutions should have chosen $x$ instead of $y$ for the $j$th extraction since $x$ is smaller. Therefore, the optimal solution must have extracted $x$ for some $i < j$. But, that means that $extracted[i]$ holds some $z > x$. By similar reasoning as above, we couldn't have moved $x$ past set $K_i$ since $extracted[i]$ would have been empty at the time $x$ was chosen. So, since we only union with sets above us and $K_i$ hasn't been unioned yet, we can't put $x$ in $extracted[j]$ before $extracted[i]$. Therefore, we can't have $x < y$.

   Since we have shown that we must have $extracted[j] = correct[j]$ for all $j$, the OFF-LINE-MINIMUM algorithm returns the correct array.

(c) **[4 points]** Do problem 21-1(c) on page 519 of CLRS.

**Answer:** We can efficiently implement OFF-LINE-MINIMUM using a disjoint-set forest. We begin by creating $n$ sets, one for each number, using the MAKE-SET operation. Then, we call UNION $n - m$ times to create the $K_j$ sets, the sequences of insertions between extractions. For each set $K_j$, we will also maintain 3 additional pieces of information, *number*, *prev*, and *next* in the representative element of the set (actually this information will be at all the elements, but it will only be maintained for the representative). *number* will correspond to $j$ and can easily be set with the initial creating of the sets. *prev* will point to the representative element of $K_{j-1}$ and *next* will point to the representative element of $K_{j+1}$. We can maintain these three properties through unions as follows: when we union two sets $j$ and $l$ where $l$ is the next set that still exists after $j$, we set *number* of the new representative equal to the maximum of the two representative *number*s, in this case, $l$. We set *prev* of the new set equal to *prev* of set $j$ and we set *next* of *prev* of $j$ equal to the new representative element. Similarly, we set *next* of the new set equal to *next* of set $l$ and we set *prev* of *next* of $l$ equal to the new representative element.

In the OFF-LINE-MINIMUM algorithm, each iteration of the for loop ($n$ iterations) will first call FIND-SET on $i$ (line 2). We use the *number* field of the returned representative as $j$. Then, in line 5, to determine the smallest $l$ greater than $j$ for which set $K_l$ exists, we can simply follow the *next* pointer from $j$, which takes constant time. In line 6, we call UNION again, at most $m$ times throughout the $n$ iterations. For each UNION call, we follow the procedure above for updating the *number*, *prev* and *next*, all of which take constant time. Therefore, we have a total of $n$ MAKE-SET operations, $n$ FIND-SET operations, and $n$ UNION operations. Since we have $3n$ total operations and $n$ MAKE-SET operations, we know by Theorem 21.13 that the worst-case running time is $O(3n\alpha(n)) = O(n\alpha(n))$.

4. **[23 points]** Articulation Points and Bridges

Throughout this problem, your explanations should be as complete as possible. You do not need to formally prove the correctness of any algorithm you are asked to give; however, when asked to prove some property of articulation points or bridges, your argument should be as formal as possible.

Notice that only parts (a)-(f) of problem 22-2 are assigned. Thus, do not worry about biconnected components for this problem.

If you are looking for the definition of a simple cycle, check appendix $B.4$.

(a) **[3 points]** Do problem 22-2(a) on page 558 of CLRS.

**Answer:** First we will prove the forward direction: If the root of $G_\pi$ is an articulation point, then it has at least 2 children in $G_\pi$. We will prove the contrapositive, the if the root has less than 2 children in $G_\pi$, then it is not an articulation point. If $r$ is the root and has no children, then it has no edges adjacent to it. Thus removing $r$ from $G$ can't disconnect the graph and $r$ is not

an articulation point. If $r$ has one child, then all other nodes in $G_\pi$ are reachable through the child. Therefore, removing $r$ from $G$ will not disconnect $G$ and $r$ is not an articulation point.

For the reverse direction, we need to show that if the root has at least 2 children in $G_\pi$ then the root is an articulation point. We will prove this by contradiction. Suppose the root has the two children $C_1$ and $C_2$ and that $C_1$ was explored first in the DFS. If the root is not an articulation point, then there exists a path between a node in $C_1$ and one in $C_2$ that does not include the root $r$. But, while exploring $C_1$, we should have explored this path and thus the nodes of $C_2$ should be children of some node of $C_1$. But, since they are in a separate subtree of the root, we know that no path can exist between them and thus $r$ is an articulation point.

(b) **[4 points]** Do problem 22-2(b) on page 558 of CLRS.

**Answer:** First we will prove the forward direction. Assume that $v$ is an articulation point of $G$. Let $C_r$ be the connected component of $G - v$ containing the root of the tree $G_\pi$. Let $s$ be a neighbor of $v$ that is not in $C_r$ (this neighbor must exist since removing $v$ must create at least two connected components) and $C_s$ be the connected component containing $s$. In $G_\pi$, all the proper ancestors of $v$ are in $C_r$, and all the descendants of $s$ are in $C_s$. Thus, there can be no edges between descendants of $s$ and proper ancestors of $v$.

To prove the backward direction, if such a vertex $s$ exists and there is no back edge from $s$ or any of it's descendants to a proper ancestor of $v$, then, we know that the only path from the root of the tree to $s$ goes through $v$. Therefore, if we remove $v$ from the graph, we have no path from the root of the tree $G_\pi$ to $s$, and we have disconnected the graph. Thus, $v$ is an articulation point.

(c) **[4 points]** Do problem 22-2(c) on page 559 of CLRS.

**Answer:** We can compute $low[v]$ for all vertices $v$ by starting at the leaves of the tree $G_\pi$. We compute $low[v]$ as follows:

$$low[v] = min(d[v], \min_{y \in children(v)} low[y], \min_{backedge(v,w)} d[w])$$

For leaves $v$, there are no descendants $u$ of $v$, so this returns either $d[v]$ or $d[w]$ is there is a back edge $(v, w)$. For vertices $v$ in the tree, if $low[v] = d[w]$, then either there is a back edge $(v, w)$, or there is a back edge $(u, w)$ for some descendant $u$. The last term in the min expression handles the case where $(v, w)$ is a back edge. If $u$ is a descendant of $v$ in $G_\pi$, we know that $d[u] > d[v]$ since $u$ is visited after $v$ in the depth first search. Therefore, if $d[w] < d[v]$, we also have $d[w] < d[u]$, so we will have set $low[u] = d[w]$. The middle term in the min expression therefore handles the case where $(u, w)$ is a back edge for some descendant $u$. Since we start at the leaves of the tree and work our way up, we will have computed everything we need when computing $low[v]$.

For each node $v$, we look at $d[v]$ and something related to all the edges leading from $v$, either tree edges leading to the children or back edges. So, the total running time is linear in the number of edges in $G$, $O(E)$.

(d) **[4 points]** Do problem 22-2(d) on page 559 of CLRS.

**Answer:** To compute the articulation points of $G$, we first run the depth first search and the algorithm from part (c). Depth first search runs in time $O(V+E)$ and since the graph is connected, we know $E > |V| - 1$ so this is simply $O(E)$. We also showed that the algorithm from (c) runs in time $O(E)$. Thus, we have calculated $low[v]$ for all $v \in V$. By part (a), we can test whether the root is an articulation point in O(1) time. By part (b), any non-root vertex $v$ is an articulation point if and only if it has a child $s$ in $G_\pi$ with no back edge to a proper ancestor of $v$. If $low[s] > d[v]$, then there must be a back edge to a proper ancestor of $v$. Otherwise, if there is an edge between a node $u$ that is not a proper ancestor of $v$ and $s$, and $u$ was visited before $v$, then we should have explored $s$ before visiting $v$.

So, if $v$ has a child $s$ in $G_\pi$ such that $low[s] \leq d[v]$, then $s$ has no back edge to a proper ancestor of $v$ and thus $v$ is an articulation point. We can check this in time proportional to the number of children of $v$ in $G_\pi$, so over all non-root vertices, this takes $O(V)$ time. Thus, the total time to find all the articulation points is $O(E)$.

(e) **[4 points]** Do problem 22-2(e) on page 559 of CLRS.

**Answer:** We will first prove the forward direction: if an edge $(u, v)$ is a bridge then it can not lie on a simple cycle. We will prove this by proving the contrapositive, if $(u, v)$ is on a simple cycle, then it is not a bridge. We know that if $(u, v)$ is a simple cycle, then there is a cycle $u \to v \to x_1 \to x_2 \cdots \to x_n \to u$, such that all of $u, v, x_i$ are distinct. If we remove the edge $(u, v)$, then any path which used to exist in the graph $G$ also exists in $G'$. We can prove this because any path which didn't include the edge $(u, v)$ obviously still exists. Any path which did include the edge $(u, v)$ can be modified to eliminate the edge $u \to v$ and include the path $u \to x_n \cdots \to x_1 \to v$ and similarly for the edge $v \to u$. Thus, $(u, v)$ is not a bridge. So, if $(u, v)$ is a bridge, then it is not on a simple cycle.

We will now prove the reverse direction and show that if an edge $(u, v)$ is not on a simple cycle, then it is a bridge. We will prove this by contradiction. Assume $(u, v)$ is not on a simply cycle but it is not a bridge. Let's say that we remove the edge $(u, v)$. Since $(u, v)$ is not a bridge, there is still a path connecting $u$ and $v$, $u \to x_1 \to x_2 \cdots \to x_n \to v$. Then, the edges $v \to u \to x_1 \cdots \to x_n \to v$ form a simple cycle. But, we assumed that $(u, v)$ wasn't on a simple cycle. So, $(u, v)$ must be a bridge.

(f) **[4 points]** Do problem 22-2(f) on page 559 of CLRS.

**Answer:** Any bridge in the graph $G$ must exist in the graph $G_\pi$. Otherwise, assume that $(u, v)$ is a bridge and that we explore $u$ first. Since removing $(u, v)$ disconnects $G$, the only way to explore $v$ is through the edge $(u, v)$. So, we only need to consider the edges in $G_\pi$ as bridges. If there are no simple cycles in the graph that contain the edge $(u, v)$ and we explore $u$ first, then we know that there are no back edges between $v$ and anything else. Also, we know that anything

in the subtree of $v$ can only have back edges to other nodes in the subtree of $v$. Therefore, we will have $low[v] = d[v]$ since $v$ is the first node visited in the subtree rooted at $v$. Thus, we can look over all the edges of $G_\pi$ and see whether $low[v] = d[v]$. If so, then we will output that $(parent[v]_{G_\pi}, v)$ is a bridge, i.e. that $v$ and its parent in $G_\pi$ form a bridge. Computing $low[v]$ for all vertices $v$ takes time $O(E)$ as we showed in part (c). Looping over all the edges takes time $O(V)$ since there are $|V| - 1$ edges in $G_\pi$. Thus the total time to calculate the bridges in $G$ is $O(E)$.

5. **[18 points]** Assorted Graph Problems

   Throughout this problem, your explanations should be as complete as possible, but they do not need to be formal. In particular, when asked to prove the correctness of the algorithm given in the problem, you do not need to use loop invariants; you may simply argue why a certain property should hold.

   (a) **[5 points]** Do problem 22.1-5 on page 530 of CLRS.

   **Answer:** The edge $(u, w)$ exists in the square of a graph $G$ if there exists a vertex $v$ such that the edges $(u, v)$ and $(v, w)$ exist in $G$. To calculate this efficiently from an adjacency matrix, we notice that this condition is exactly what we get when we square the matrix. The cell $M^2[u, w] = \sum_v M[u, v] \cdot M[v, w]$ when we multiply two matrices. So, if we represent edges present in $G$ with ones and all other entries as zeroes, we will get the square of the matrix with zeroes when edges aren't in the graph $G^2$ and positive integers representing the number of paths of length exactly two for edges that are in $G^2$. Using Strassen's algorithm or other more sophisticated matrix multiplication algorithms, we can compute this in $O(V^{2.376})$.

   Using adjacency lists, we need to loop over all edges in the graph $G$. For each edge $(u, v)$, we will look at the adjacency list of $v$ for all edges $(v, w)$ and add the edge $(u, w)$ to the adjacency lists for $G^2$. The maximum number of edges in the adjacency list for $v$ is $V$, so the total running time is $O(VE)$. This assumes that we can add and resolve conflicts when inserting into the adjacency lists for $G^2$ in constant time. We can do this by having hash tables for each vertex instead of linked lists.

   (b) **[4 points]** Use your result from part (a) to give an algorithm for computing $G^k$ for some integer $k \geq 1$. Try to make your algorithm as efficient as possible. For which $k$ is it asymptotically better to convert $G$ from one representation to another prior to computing $G^k$?

   **Answer:** To calculate the $G^k$ graph using the adjacency matrix representation, we can use the trick of repeated squaring. Thus, by first calculating $G^{k/2}$ for even $k$, and $G^{(k-1)/2}$ for odd $k$, we can solve the problem in time $O(V^{2.376} \lg k)$. For adjacency lists, we can do the same thing. If we calculate the $G^2$ graph, we can calculate the $G^4$ graph by running our algorithm from part (a) on the $G^2$ graph. Thus, to calculate $G^k$ using adjacency lists takes time $O(VE \lg k)$. Converting between the two representations takes time $O(V^2)$ which is asymptotically less than calculating $G^2$ in either case. Converting between the two represenations

therefore depends on how many edges you have relative to the number of vertices. If $E = o(V^{1.376})$ then you should convert to the adjacency list representation and otherwise you should convert to the matrix. Notice that the number of edges will keep changing for each $G^i$ so you may need to convert back and forth when calculating $G^k$.

(c) **[4 points]** Do problem 22.3-11 on page 549 of CLRS. How does your algorithm compare (in the asymptotic running time sense) with the algorithm given in class and in section 21.1 of CLRS for determining the connected components of $G$ using disjoint-set data structures?

**Answer:** We will first modify DFS to label the connected components.

```
DFS(G)
1   for each vertex u ∈ V[G]
2     do color[u] ← WHITE
3        π[u] ← NIL
4   time ← 0
5   k ← 0
6   for each vertex u ∈ V[G]
7     do if color[u] = WHITE
8        k ← k + 1
9        DFS-VISIT(u, k)
```

Lines 5, 8, and 9 are the ones which were added or changed. In DFS-VISIT, we will always call DFS-VISIT with the same value of $k$. In addition, after setting the color of $u$ to BLACK, we will set the connected component, $cc[u]$ to $k$.

Since the graph $G$ is undirected, two nodes $u$ and $v$ will only get the same connected component label if there is a path between them in the graph $G$.

The running time of this algorithm is the same as for DFS which is $O(V + E)$. The algorithm given in section 21.1 of CLRS runs in time $O((V+E)\alpha(V))$, which is asymptotically slower. However, $\alpha(V)$ is very small ($\leq 4$) for any reasonable size $V$, so the running times are comparable.

(d) **[5 points]** Do problem 22.4-2 on page 552 of CLRS.

**Answer:** We first run topological sort on the graph $G$. This takes time $O(V+E)$. We know that any path that runs between $s$ and $t$ must use only the vertices located between $s$ and $t$ in the topological sort. If there was a vertex $a < s$ in the topological sort, then there can't be a path from $s \rightarrow a$ in $G$. Likewise there can be no vertex $b > t$ on a path from $s$ to $t$. So, we can ignore all vertices $< s$ or $> t$ in the topological sort. Then, we can use dynamic programming to calculate the number of paths from $s$ to $t$. We will label each node from $s$ to $t$ with the number of paths from $s$ to that node. We start by labelling the node $s$ with a 1 since there is one path from $s$ to $s$ and by labelling all other nodes with 0. Then, for each node $i$ starting from $s$ in the sort, we calculate the number of paths from $s$ as

$$paths[i] = \sum_{(j,i)\in E} paths[j]$$

We finish when we calculate $paths[t]$ which we output as the answer. This algorithm is correct since all paths from $s$ to $i$ must only contain vertices between $s$ and $i$ in the topological sort. These we have calculated by the time we calculate $paths[i]$. Also, the predecessor to $i$ on any path from $s$ to $i$ must be such that there is an edge from $(pred, i)$. We sum over all possible predecessors to calculate $paths[i]$. For at most each vertex, we sum over the number of incoming edges. So, in total, we look at each edge once. Therefore, the running time of this step is $O(V + E)$. Thus, the total time for this algorithm is $O(V + E)$ which is linear in the size of the input.

6. **[18 points]** Minimal Spanning Trees

Throughout this problem, if you are arguing about the correctness of a minimal spanning tree algorithm, please be as formal as possible in proving the property the algorithm relies on for its correctness; however, you do not need to resort to loop invariants or similar formal methods to prove the correctness of the actual algorithm.

(a) **[6 points]** Do problem 23.1-11 on page 567 of CLRS.

**Answer:** If we decrease the weight of an edge $(u, v)$ not in $T$, then the new minimal spanning tree may now contain that edge. The algorithm to compute the new minimum spanning tree is to find the heaviest weight edge on the path from $u \to v$. If this edge weight is higher than the weight of edge $(u, v)$, then we delete this edge and add $(u, v)$. Else, we do nothing. The running time of this algorithm is the time taken to find the heaviest weight edge on the path $u \to v$. We can do this by running DFS from from $u$ till we hit $v$ since there is only one path from $u$ to $v$ in a tree. The running time is $O(V + E) = O(V)$ since this is a tree. The result is obviously a spanning tree. We will show that it is a minimum spanning tree.

We will do this by considering Kruskal's algorithm over the new graph. Kruskal's algorithm grows an MST by always adding the lowest weight edge that does not create a cycle. For all edges in the MST with weight less than $w'(u, v)$, we will take the exact same edges as we did before. When we consider the edge $(u, v)$, we have two choices, either we take the edge or we don't. If we don't take the edge, then $(u, v)$ must have created a cycle in the MST. All edges in the cycle must have weight less than $w'(u, v)$ or they wouldn't be in the MST already. The algorithm will then proceed as before. This corresponds to the above case where we don't modify the MST. If we take the edge $(u, v)$ then all further edges will be added in the same manner until we reach the one which creates a cycle with the edge $(u, v)$. This will be the highest weight edge in the original path from $u \to v$, which we won't add. All other edges will be added as before. This corresponds to the case where our algorithm deletes the highest weight edge on the path from

$u \to v$ and adds the edge $(u, v)$. Our algorithm produces the same spanning tree that Kruskal's algorithm produces, and it is therefore a minimum spanning tree.

(b) **[6 points]** Do problem 23.2-7 on page 574 of CLRS.

**Answer:** Let's assume that we add the new vertex and incident edges and initialize all the edge weights to $\infty$. Then, we can take any of the edges and add it to the original spanning tree to give a minimum spanning tree. Using the answer from part (a), we know that we can reduce the weight of this edge and the other edges one at a time, add the edge to the MST, and remove the edge with the highest weight in the newly created cycle. This will run in time $O(V^2)$ since there may be at most $V$ edges from the new vertex. However, we can do better by noticing that the only possible edges in the new MST are the ones in the old MST or the new edges we just added. There are a total of $|V| - 1$ edges in the old MST and a total of at most $|V|$ edges added. So, if we simply run either Kruskal's or Prim's algorithm on the graph with all the vertices but only these $|E| = 2|V| - 1 = O(V)$ possible edges, we will create the new MST in time $O(V \lg V)$, which is better than the $O(V^2)$ time.

(c) **[6 points]** Do problem 23.2-8 on page 574 of CLRS.

**Answer:** This algorithm does not compute the minimum spanning tree correctly. Suppose we have a graph with three nodes, $A, B, C$. Suppose also that the graph has three edges with the following weights: $w(A, B) = 1, w(B, C) = 2, w(C, A) = 3$. Let's say we partition the graph into the two sets $V_1 = \{A, C\}$ and $V_2 = \{B\}$. This partition satisfies the condition that $|V_1|$ and $|V_2|$ differ by at most 1. The edges sets will be $E_1 = \{(A, C)i\}$ and $E_2 = \emptyset$. So, when we recursively solve the subproblems, we will add the edge $(A, C)$. Then, when we select the minimum edge that crosses the partition, we will select the edge $(A, B)$ with weight 1. The total weight of our MST is $1 + 3 = 4$. However, the actual minimum spanning tree has edges $(A, B)$ and $(B, C)$ and weight $1 + 2 = 3$. Therefore, this algorithm fails to produce the correct minimum spanning tree.

**CS161: Design and Analysis of Algorithms** **Summer 2004**

# Problem Set #6 Solutions

## General Notes

- **Regrade Policy:** If you believe an error has been made in the grading of your problem set, you may resubmit it for a regrade. If the error consists of more than an error in addition of points, please include with your problem set a detailed explanation of which problems you think you deserve more points on and why. We reserve the right to regrade your entire problem set, so your final grade may either increase or decrease.

Throughout this entire problem set, your proofs should be as formal as possible. However, when asked to state an algorithm, you do not need to give pseudocode or prove correctness at the level of loop invariants. Your running times should always be given in terms of $|V|$ and/or $|E|$. If representation of the graph is not specified, you may assume whichever is convenient.

1. **[19 points]** Gabow's Scaling Algorithm for Single-Source Shortest Paths

   (a) **[5 points]** Do problem 24-4(a) on page 616 of CLRS.

   **Answer:** Since the edge weights are all positive, we can use Dijkstra's algorithm to find the shortest paths from the start vertex to all other vertices. The running time of Dijkstra's algorithm using a binary heap is $O(E \lg V)$. The $\lg V$ term comes from the $V$ calls to extract-min and the $E$ calls to decrease-key.

   Because of the constraints that the edge weights are integers and that the shortest path distances are bounded by $|E|$, we can do better. We can use a method similar to counting sort to maintain the list of vertices. We know that the weights of the path to each vertex will always be an integer between 0 and $|E|$, so we can keep an array SHORTEST of linked lists for each possible value. For Dijkstra's algorithm, we need to implement the INSERT, DECREASE-KEY, and EXTRACT-MIN functions. INSERT is easy. If we want to insert a vertex that is reachable in length $i$, we simply add it to the beginning of the linked list in SHORTEST[$i$], which is $O(1)$. To call DECREASE-KEY on a vertex $v$, we remove $v$ from it's current linked list ($O(1)$), decrease it's key to $i$, and insert it into SHORTEST[$i$], for a total time of $O(1)$. To EXTRACT-MIN, we notice that throughout the running of Dijkstra's algorithm, we always extract vertices with shortest paths of non-decreasing value. So, if the previous vertex was extracted at value $i$, we know that there can be no vertices in the array at values less than $i$. So, we start at SHORTEST[$i$] and if it is non-empty, we remove the first element from the linked list. If it is empty, we move up to SHORTEST[$i + 1$] and repeat. The actual extraction takes time $O(1)$. Notice that the scanning for a non-empty list could take time $O(E)$ since there are a total of $E$ lists, but since we never backtrack,

this $O(E)$ scan of the lists is averaged over the $O(V)$ calls to EXTRACT-MIN, for an amoritized cost of $O(E/V)$ each.

Our total running time therefore includes $V$ calls to INSERT ($O(V)$), $E$ calls to DECREASE-KEY ($O(E)$) and $V$ calls to EXTRACT-MIN ($O(V+E)$) for a total running time of $O(V+E)$. Since we know $|E| > |V| - 1$, the $O(E)$ term dominates for a running time of $O(E)$.

(b) **[1 point]** Do problem 24-4(b) on page 616 of CLRS.

**Answer:** Using part (a), we can show how to compute $\delta_1(s, v)$ for all $v \in V$ in $O(E)$ time. We know that $\delta_1(s, v)$ is computed using the edge weight function $w_1$, which only uses the first significant bit of the actual edge weights. Therefore, the weights $w_1$ are always either 0 or 1. The maximum number of edges on a shortest path is $|V| - 1$ since it can have no cycles. Since the maximum edge weight $w_1$ is 1, the maximum weight of a shortest path is $|V| - 1$. Therefore, we have the condition that for all vertices $v \in V$, we have $\delta_1(s, v) \leq |V| - 1 \leq |E|$ by assumption. From part (a), we know that we can compute $\delta_1(s, v)$ for all $v \in V$ in $O(E)$ time.

(c) **[3 points]** Do problem 24-4(c) on page 616 of CLRS.

**Answer:** We know that $w_i$ is defined as $\lfloor w(u, v)/2^{k-i} \rfloor$. We will show that either $w_i(u, v) = 2w_{i-1}(u, v)$ or $w_i(u, v) = 2w_{i-1}(u, v) + 1$. Weight $w_i$ is the $i$ most significant bits of $w$. We can get $w_i$ from $w_{i-1}$ by shifting $w_{i-1}$ to the left by one space and adding the $i$th significant bit. Shifting to the left is equivalent to multiplying by 2 and the $i$th significant bit can be either 0 or 1. So, we have that $w_i(u, v) = 2w_{i-1}(u, v)$ if the $i$th significant bit is a zero or $w_i(u, v) = 2w_{i-1}(u, v)+1$ if the $i$th significant bit is a one.

We now want to prove that $2\delta_{i-1}(s, v) \leq \delta_i(s, v) \leq 2\delta_{i-1}(s, v) + |V| - 1$. The shortest path $\delta_i(s, v)$ has weight

$$\begin{aligned}
\delta_i(s, v) &= \min \sum_{e \in Path(s,v)} w_i(e) \\
&\geq \min \sum_{e \in Path(s,v)} 2w_{i-1}(e) \\
&= 2 \cdot \min \sum_{e \in Path(s,v)} w_{i-1}(e) \\
&= 2\delta_{i-1}(s, v)
\end{aligned}$$

with the inequality holding because the $w_i(e)$ is greater than or equal to $2w_{i-1}(e)$ since it equals either that or that plus one. The last equality holds because the minumum weight of any path from $s$ to $v$ using the weight function $w_{i-1}$ is equal to the shortest path distance from $s$ to $v$ using $w_{i-1}$.

Similarly,

$$\begin{aligned}
\delta_i(s,v) &= \min \sum_{e \in Path(s,v)} w_i(e) \\
&\leq \min \sum_{e \in Path(s,v)} (2w_{i-1}(e) + 1) \\
&= \min \left( 2 \sum_{e \in Path(s,v)} w_{i-1}(e) + \sum_{e \in Path(s,v)} 1 \right) \\
&\leq \min \left( 2 \sum_{e \in Path(s,v)} w_{i-1}(e) + |V| - 1 \right) \\
&= 2\delta_{i-1}(s,v) + |V| - 1
\end{aligned}$$

The first inequality holds because $w_i(e)$ is less than or equal to $2w_{i-1}(e) + 1$ by similar reasoning as above. We then set the minimum weight of any paths from $s$ to $v$ using the weight function $w_{i-1}$ equal to the shortest path as above. The second inequality holds because the minimum path length is bounded by $|V| - 1$ because it can have no cycles.

(d) **[3 points]** Do problem 24-4(d) on page 616 of CLRS.
**Answer:** We define for $i = 2, 3, \ldots k$ and all $(u, v) \in E$,

$$\hat{w}_i(u,v) = w_i(u,v) + 2\delta_{i-1}(s,u) - 2\delta_{i-1}(s,v)$$

We wish to prove for all $i = 2, 3, \ldots, k$ and all $u, v \in V$ that $\hat{w}_i$ of edge $(u, v)$ is a nonnegative integer. We can prove this starting from the triangle inequality.

$$\begin{aligned}
\delta_{i-1}(s,v) &\leq \delta_{i-1}(s,u) + w_{i-1}(u,v) \\
2\delta_{i-1}(s,v) &\leq 2\delta_{i-1}(s,u) + 2w_{i-1}(u,v) \\
2\delta_{i-1}(s,v) &\leq 2\delta_{i-1}(s,u) + w_i(u,v) \\
0 &\leq w_i(u,v) + 2\delta_{i-1}(s,u) - 2\delta_{i-1}(s,v) \\
0 &\leq \hat{w}_i
\end{aligned}$$

Thus, $\hat{w}_i$ is never negative. It is an integer since all the edge weights are always integers and thus all the shortest path distances are always integers.

(e) **[4 points]** Do problem 24-4(e) on page 617 of CLRS.
**Answer:** We define $\hat{\delta}_i(s, v)$ as the shortest path weight from $s$ to $v$ using $\hat{w}_i$. We want to prove for $i = 2, 3, \ldots k$ and all $v \in V$ that

$$\delta_i(s,v) = \hat{\delta}_i(s,v) + 2\delta_{i-1}(s,v)$$

and that $\hat{\delta}_i \leq |E|$.

We prove this by expanding the $\hat{\delta}_i(s,v)$ term.

$$\begin{aligned}
\hat{\delta}_i(s,v) &= \min \sum_{e \in Path(s,v)} \hat{w}_i(e) \\
&= \min(\hat{w}_i(s,x_1) + \hat{w}_i(x_1,x_2) + \cdots + \hat{w}_i(x_n,v)) \\
&= \min(w_i(s,x_1) + 2\delta_{i-1}(s,s) - 2\delta_{i-1}(s,x_1)) + \\
&\quad (w_i(x_1,x_2) + 2\delta_{i-1}(s,x_1) - 2\delta_{i-1}(s,x_2)) + \cdots + \\
&\quad (w_i(x_n,v) + 2\delta_{i-1}(s,x_n) - 2\delta_{i-1}(s,v)) \\
&= \min(2\delta_{i-1}(s,s) - 2\delta_{i-1}(s,v) + \sum_{e \in Path(s,v)} w_i(e)) \\
&= -2\delta_{i-1}(s,v) + \min \sum_{e \in Path(s,v)} w_i(e) \\
&= -2\delta_{i-1}(s,v) + \delta_i(s,v) \\
\delta_i(s,v) &= \hat{\delta}_i(s,v) + 2\delta_{i-1}(s,v)
\end{aligned}$$

We expand $\hat{\delta}_i(s,v)$ in terms of the weights of edges along the path. We see that if we also expand the $\hat{w}_i$ terms, many things cancel. The term $\delta_{i-1}(s,s) = 0$ since the shortest path from a node to itself is zero regardless of the edge weight function. Also, we know that the minimum path length from $s$ to $v$ using the $w_i$ function is $\delta_i(s,v)$.

From this, we can easily show that $\hat{\delta}_i \leq |E|$ using the results from part (c).

$$\begin{aligned}
\delta_i(s,v) &= \hat{\delta}_i(s,v) + 2\delta_{i-1}(s,v) \\
2\delta_{i-1}(s,v) + |V| - 1 &\geq \hat{\delta}_i(s,v) + 2\delta_{i-1}(s,v) \\
\hat{\delta}_i(s,v) &\leq 2\delta_{i-1}(s,v) + |V| - 1 - 2\delta_{i-1}(s,v) \\
\hat{\delta}_i(s,v) &\leq |V| - 1 \\
\hat{\delta}_i(s,v) &\leq |E|
\end{aligned}$$

(f) **[3 points]** Do problem 24-4(f) on page 617 of CLRS.

**Answer:** If we are given $\delta_{i-1}(s,v)$, we can compute $\hat{w}_i(u,v)$ using the equation in part (d) in time $O(E)$ since we can easily calculate $w_i(u,v)$ and we need to compute it for each edge. In part (e), we showed that $\hat{\delta}_i(s,v)$ is bounded by $|E|$. So, we can use the results from part (a) to compute the shortest path distances $\hat{\delta}_i(s,v)$ in time $O(E)$. From these results, we can use the equation in part (e) to calculate the shortest path distances $\delta_i(s,v)$ in time $O(V)$, once for each vertex.

Therefore, we can compute $\delta(s,v)$ in time $O(E \lg W)$. We will simply start by calculating $\delta_1(s,v)$ in time $O(E)$ as we showed in part (b). Then, as we just explained, we can calculate each $\delta_{i+1}$ from $\delta_i$ in time $O(E)$. In total, we need to calculate up till $\delta_k$. Since $k = O(\lg W)$, the running time of this is $O(E \lg W)$.

2. **[12 points]** Transitive Closure of a Dynamic Graph

   (a) **[3 points]** Do problem 25-1(a) on page 641 of CLRS.

   **Answer:** When we add a new edge to a graph, we need to update the transitive closure. The only new paths we will add are the ones which use the new edge $(u, v)$. These paths will be of the form $a \rightsquigarrow u \rightarrow v \rightsquigarrow b$. We can find all these new paths by looking in the old transitive closure for the vertices $a \in A$ which had paths to $u$ and for vertices $b \in B$ which $v$ had paths to. The new edges in the transitive closure will then be $(a, b)$, where $a \in A$ and $b \in B$. Since the number of vertices in each set $A$ or $B$ is bounded by $V$, the total number of edges we'll need to add is $O(V^2)$. If we represent the transitive closure graph $G^*$ with an adjacency matrix, we can very easily find the sets $A$ and $B$. $A$ will be all the vertices which have a one in the column $u$ and $B$ will be all the vertices which have a one in the row $v$.

   (b) **[1 point]** Do problem 25-1(b) on page 641 of CLRS.

   **Answer:** Suppose our graph $G$ consists of two disjoint graphs, each of size $V/2$. Assume that each subgraph is complete, that there are edges between each vertex in the subgraph to all other vertices in the same subgraph. If we then add an edge from a vertex in the first subgraph to one in the second subgraph, we will need to add edges in the transitive closure from every vertex in the first subgraph to every vertex in the second subgraph. This is a total of $V/2 \times V/2 = V^2/4$ edges. Regardless of what our algorithm is for calculating the transitive closure, we will always need to add $\Omega(V^2)$ edges, and thus the running time must be $\Omega(V^2)$.

   (c) **[8 points]** Do problem 25-1(c) on page 641 of CLRS.

   **Answer:** We will give you an outline of the solution to this problem on the final exam. You will then complete it and analyze its correctness and running time.

3. **[24 points]** Assorted Graph Problems

   (a) **[8 points]** Do problem 22.2-7 on page 539 of CLRS. Assume the edges of $T$ are undirected and weighted, and give your running time in terms of $|V|$ (since $|E| = |V| - 1$).

   **Answer:** First notice that the shortest path distance between two vertices is given by the length of the path between the two vertices, since there is only one path between any pair of vertices in a tree. So, we are basically asking for the longest simple path between any two leaves in a tree.

   We will keep two items of information at each edge $(u, v) \in T$. We will keep $d[u \rightarrow v]$, which represents the longest simple path distance in the tree from $u$ to $v$ to any leaf. Also, we keep $d[v \rightarrow u]$, defined symmetrically.

   First we arbitrarily root the tree at some vertex $r$ by pointing all the edges outward from $r$. We will compute the downward distances bottom-up, and then the upward distance top-down.

COMPUTE-DOWN($x$)
1 **if** IsLeaf($x$)
2 **return**
3 **for** $y \in children(x)$
4 COMPUTE-DOWN($y$)
5 $d[x \to y] \leftarrow w(x, y) + \max_{z \in children(y)} d[y \to z]$

After calling COMPUTE-DOWN on the root of the tree $r$, every edge will have its downward field set. Notice that because of the recursion, the information is actually first computed at the leaves and then propagated upward.

The top-down pass is somewhat tricker, because now we must integrate information from different branches as we descend down the tree. There are two differences: first, the information is now computed at the top first, and at the bottom at the end; second, updating $d[y \to x]$ where $x$ is the parent of $y$ now depends not only on $d[x \to p(x)]$ but also on $d[x \to z]$, where $z$ are other children of $x$. However, these values were computed on the downward pass; therefore, as long as we compute $d[x \to p(x)]$ before $d[y \to x]$, we have all the information we need.

COMPUTE-UP($x$)
1 $p \leftarrow parent(x)$
2 **for** $y \in children(x)$
3 $d[y \to x] \leftarrow w(x, y) + \max_{z \in children(x) \cup \{p\}, z \neq y} d[x \to z]$
4 COMPUTE-UP($y$)

Thus, to calculate the diameter, we root the tree arbitrarily at a vertex $r$, call COMPUTE-DOWN($r$) and then COMPUTE-UP($r$). The largest value $d[u \to v]$ computed during these procedures is indeed the diameter of the tree.

The first procedure clearly runs in $O(V)$, since every edge $u \to v$ is examined at most twice - once when we $d[u \to v]$, and once when we set $d[p(u) \to u]$.

The second procedure as it is written runs in worst-case time $O(V^2)$, since if we have a node with $\Theta(V)$ chilren, we will be looking through all the downward distance values $V$ times for each maximization step in line 3. However, we notice that the only useful values from that step are the two largest downward $x \to z$ values, since at most one such $z$ can be the $y$ we are currently looking at. Thus, we precompute at each node $x$ the two largest $x \to z$ values, and then use the largest unless $z = y$. Thus, each edge is considered at most a constant number of times, and the running time is $O(V)$ as well.

(b) **[8 points]** Do problem 22-3(b) on page 559 of CLRS. You may assume the result of 22-3(a) without proof.

**Answer:** From part (a), we know that if there is an Euler tour in the graph $G$, then the in-degree equals the out-degree for every vertex $v$. Because of this fact, we know that if we pick a path with unique edges, any time we reach a vertex (use one of the in-degree edges), there must still be a way to leave the vertex (use one of the out-degree edges) except for the first vertex in the path.

Our algorithm therefore is to pick a random starting vertex $v$. We pick an outgoing

edge from $v$, $(v, u)$, at random. We move to vertex $u$ and delete edge $(v, u)$ from the graph and repeat. If we get to a vertex with no outgoing edges and it is not equal to $v$, we report that no Euler tour exists. If the vertex with no outgoing edges is $v$, we have a cycle, which we represent as $v \to u \to \cdots \to v$.

The problem is that we may not have visited all the edges in the graph yet. This may only occur if there are vertices in our cycle which have outgoing edges that we didn't traverse. Otherwise, our graph would not be connected. If we can efficiently pick one of these vertices $u$ and start a cycle from that point, we know that the new cycle from $u$ will end at $u$ by the same reasoning as above. Therefore, we can connect the two cycles $\{\ldots a \to u \to b \ldots\}$ and $\{u \to c \cdots \to d \to u\}$ by making one big cycle $\{\ldots a \to u \to c \cdots \to d \to u \to b \ldots\}$. This is a valid cycle since we removed all the edges in the first cycle before computing the second cycle, so we still have the property that each edge is used at most once. We assume that when we start the new cycle from $u$, we keep a pointer to the position of $u$ in the old cycle so that we know where to break the old cycle in constant time.

If we repeat this process until there are no more edges in the graph, we will have an Euler tour. The only problem is how to pick the vertex $u$ that still has outgoing edges from the current cycle. Our original vertex was $v$. For the first cycle created, we simply walk along the cycle from $v$ and let $u$ equal the first vertex which still has outgoing edges. Then, after adding the next cycle, we start walking from $u$ along the newly merged cycles and so on. Once we have traversed the path from $v \rightsquigarrow u$, we know that none of the vertices in that section can ever have outgoing edges, so we don't need to consider them the second time. We will need to walk over the entire Euler tour at some point in the algorithm, so the total running time of finding the vertex $u$ for all the cycles is $O(E)$. Also, the cost of actually finding the cycles traverses each edge exactly once, so it is also $O(E)$. Therefore, the total running time of the algorithm is $O(E)$.

(c) **[3 points]** Suppose we have a single-source shortest path algorithm $A$ that runs in time $f(|V|, |E|)$ when all edge weights are nonnegative. Give an algorithm to solve problem 24.3-4 on page 600 of CLRS in time $f(|V|, |E|) + O(E)$ by using $A$ as a subroutine.

**Answer:** The reliability of a path from $u$ to $v$ is the product of the reliabilities of each edge in the path. This is because the probability that the channel does not fail is the probability that all individual edges do not fail. Since the failures or successes of the individual edges are independent, we simply multiply to get the total probability of success.

But, our shortest path algorithm finds the shortest path by summing the edge weights. We need some transformation of our reliabilities so that we can use the shortest path algorithm given. One function we have seen that converts a product of terms to a sum of terms is the log function.

If we define a new weight function $w(u, v) = -\lg(r(u, v))$, then all the edge weights are positive since the log of a positive number less than 1 is negative. As the reliability decreases, the value of the weight of the edge will increase. Since we are trying to find the most reliable path, this will correspond to the shortest

weight path in the graph with the new edge weights.

We can show this more formally, where RPATH represents the reliability of a path.

$$\begin{aligned} RPATH(s,t) &= \prod_{e \in Path(s,t)} r(e) \\ \log RPATH(s,t) &= \sum_{e \in Path(s,t)} \log r(e) \\ -\log RPATH(s,t) &= \sum_{e \in Path(s,t)} -\log r(e) \end{aligned}$$

Since the log function is monotonic as long as the base is greater than 1, if we find the minimum path with edge weights $w$, we will also find the most reliable path.

The time for the conversion to the weight function $w$ is constant for each edge, so the total time is $O(E)$. We then call the single-source shortest path algorithm which runs in time $f(|V|, |E|)$ and returns our most reliable path. Therefore, the total running time is $f(|V|, |E|) + O(E)$.

(d) **[5 points]** Do problem 25.2-9 on page 635 of CLRS.

**Answer:** If we have an algorithm to compute the transitive closure of a directed acyclic graph, we can use it to calculate the transitive closure of any directed graph. We first decompose the directed graph $G$ into its strongly connected component graph $G_{SCC}$. This runs in time $O(V + E)$ and results in a graph with at most $V$ vertices and $E$ edges. We keep track of which set of vertices in $G$ map to each vertex in $G_{SCC}$. Call this $g(v)$ which returns a stronly connected component in $G_{SCC}$. We then call the transitive closure algorithm on the graph $G_{SCC}$. This runs in time $O(f(|V|, |E|))$. All that is left is to convert back from the strongly connected component graph's transitive closure to $G$'s transitive closure. There is an edge $(u, v)$ in the transitive closure of $G$ if and only if there is an edge $(g(u), g(v))$ in the graph $G_{SCC}$. The forward direction is easy to see. If there is a path from $u$ to $v$ in $G$, then there must be a path between the component that $u$ belongs to and the one that $v$ belongs to in $G_{SCC}$. The reverse direction is equally easy. If there is a path between $u$'s strongly connected component and $v$'s strongly connected component in $G_{SCC}$, then there is a path between some vertex $a$ in $g(u)$ and some vertex $b$ in $g(v)$ in $G$. But, since $u$ and $a$ are in the same SCC, there is a path from $u \rightsquigarrow a$ and since $v$ and $b$ are in the same SCC, there is a path from $v \rightsquigarrow b$. Therefore, there is a path $u \rightsquigarrow a \rightsquigarrow b \rightsquigarrow v$ in $G$.

To actually calculate the transitive closure of $G$ this way would require time $V^2$ since we'd need to look at all pairs. Instead, we will look at the transitive closure of $G_{SCC}$. For each edge $(a, b)$ in the transitive closure of $G_{SCC}$, we assume that we know the set of vertices in $G$ that map to $a$ and $b$, call these sets $g^{-1}(a)$ and $g^{-1}(b)$ respectively. We will add the edge $(x, y)$ for all vertices $x \in g^{-1}(a)$ and

$y \in g^{-1}(b)$. This will take time equal to the number of edges in the transitive closure of $G$, $O(E^*)$.

Therefore the total running time for the algorithm is the time to build the strongly connected components ($O(V+E)$), the time to run the transitive closure algorithm on $G_{SCC}$ ($O(f(|V|, |E|))$ and the time to convert back to the transitive closure of $G$ ($O(E^*)$). Since we know that $E < E^*$, the total time is $O(f(|V|, |E|)+V+E^*)$.