Chapter 1

Michelle Bodnar, Andrew Lohr

August 28, 2017

Exercise 1.1-1

An example of a real world situation that would require sorting would be if you wanted to keep track of a bunch of people's file folders and be able to look up a given name quickly. A convex hull might be needed if you needed to secure a wildlife sanctuary with fencing and had to contain a bunch of specific nesting locations.

Exercise 1.1-2

One might measure memory usage of an algorithm, or number of people required to carry out a single task.

Exercise 1.1-3

An array. It has the limitation of requiring a lot of copying when re-sizing, inserting, and removing elements.

Exercise 1.1-4

They are similar since both problems can be modeled by a graph with weighted edges and involve minimizing distance, or weight, of a walk on the graph. They are different because the shortest path problem considers only two vertices, whereas the traveling salesman problem considers minimizing the weight of a path that must include many vertices and end where it began.

Exercise 1.1-5

If you were for example keeping track of terror watch suspects, it would be unacceptable to have it occasionally bringing up a wrong decision as to whether a person is on the list or not. It would be fine to only have an approximate solution to the shortest route on which to drive, an extra little bit of driving is not that bad.

Exercise 1.2-1

A program that would pick out which music a user would like to listen to next. They would need to use a bunch of information from historical and popular preferences in order to maximize.

Exercise 1.2-2

We wish to determine for which values of n the inequality $8n^2 < 64n\log_2(n)$ holds. This happens when $n < 8\log_2(n)$, or when $n \leq 43$. In other words, insertion sort runs faster when we're sorting at most 43 items. Otherwise merge sort is faster.

Exercise 1.2-3

We want that $100n^2 < 2^n$. note that if $n = 14$, this becomes $100(14)^2 = 19600 > 2^14 = 16384$. For $n = 15$ it is $100(15)^2 = 22500 < 2^{15} = 32768$. So, the answer is $n = 15$.

Problem 1-1

We assume a 30 day month and 365 day year.

	1 Second	1 Minute	1 Hour	1 Day	1 Month	1 Year	1 Century
$\lg n$	$2^{1 \times 10^6}$	$2^{6 \times 10^7}$	$2^{3.6 \times 10^9}$	$2^{8.64 \times 10^{10}}$	$2^{2.592 \times 10^{12}}$	$2^{3.1536 \times 10^{13}}$	$2^{3.15576 \times 10^{15}}$
$\sqrt{n}$	1×10^{12}	3.6×10^{15}	1.29×10^{19}	7.46×10^{21}	6.72×10^{24}	9.95×10^{26}	9.96×10^{30}
n	1×10^6	6×10^7	3.6×10^9	8.64×10^{10}	2.59×10^{12}	3.15×10^{13}	3.16×10^{15}
$n \lg n$	62746	2801417	133378058	2755147513	71870856404	797633893349	6.86×10^{13}
n^2	1000	7745	60000	293938	1609968	5615692	56176151
n^3	100	391	1532	4420	13736	31593	146679
2^n	19	25	31	36	41	44	51
$n!$	9	11	12	13	15	16	17

Chapter 2

Michelle Bodnar, Andrew Lohr

January 3, 2018

Exercise 2.1-1

31	41	59	26	41	58
31	41	59	26	41	58
31	41	59	26	41	58
26	31	41	59	41	58
26	31	41	41	59	58
26	31	41	41	58	59

Exercise 2.1-2

Algorithm 1 Non-increasing Insertion-Sort(A)

1: **for** $j = 2$ to $A.length$ **do**
2:     $key = A[j]$
3:     // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.
4:     $i = j - 1$
5:     **while** $i > 0$ and $A[i] < key$ **do**
6:         $A[i+1] = A[i]$
7:         $i = i - 1$
8:     **end while**
9:     $A[i+1] = key$
10: **end for**

Exercise 2.1-3

On each iteration of the loop body, the invariant upon entering is that there is no index $k < j$ so that $A[k] = v$. In order to proceed to the next iteration of the loop, we need that for the current value of j, we do not have $A[j] = v$. If the loop is exited by line 5, then we have just placed an acceptable value in i on the previous line. If the loop is exited by exhausting all possible values of j, then we know that there is no index that has value j, and so leaving NIL in i is correct.

Exercise 2.1-4

Algorithm 2 Linear-Search(A,v)
___
1: $i = NIL$
2: **for** $j = 1$ to $A.length$ **do**
3:     **if** $A[j] = v$ **then**
4:         $i = j$
5:             **return** i
6:     **end if**
7: **end for**
8: **return** i
___

Input: two n-element arrays A and B containing the binary digits of two numbers a and b.

Output: an $(n+1)$-element array C containing the binary digits of $a + b$.

Algorithm 3 Adding n-bit Binary Integers
___
1: carry $= 0$
2: **for** i=n to 1 **do**
3:     $C[i + 1] = (A[i] + B[i] + carry) \pmod 2$
4:     **if** $A[i] + B[i] + carry \geq 2$ **then**
5:         carry $= 1$
6:     **else**
7:         carry $= 0$
8:     **end if**
9: **end for**
10: C[1] = carry
___

Exercise 2.2-1

$$n^3/1000 - 100n^2 - 100n + 3 \in \Theta(n^3)$$

Exercise 2.2-2

Input: An n-element array A.

Output: The array A with its elements rearranged into increasing order.

The loop invariant of selection sort is as follows: At each iteration of the for loop of lines 1 through 10, the subarray $A[1..i-1]$ contains the $i-1$ smallest elements of A in increasing order. After $n-1$ iterations of the loop, the $n-1$ smallest elements of A are in the first $n-1$ positions of A in increasing order, so the nth element is necessarily the largest element. Therefore we do not need to run the loop a final time. The best-case and worst-case running times of selection sort are $\Theta(n^2)$. This is because regardless of how the elements are initially arranged, on the ith iteration of the main for loop the algorithm always inspects each of the remaining $n-i$ elements to find the smallest one remaining.

Algorithm 4 Selection Sort

1: **for** $i = 1$ to $n - 1$ **do**
2:     $min = i$
3:     **for** $j = i + 1$ to n **do**
4:         // Find the index of the ith smallest element
5:         **if** $A[j] < A[min]$ **then**
6:             $min = j$
7:         **end if**
8:     **end for**
9:     Swap $A[min]$ and $A[i]$
10: **end for**

This yields a running time of

$$\sum_{i=1}^{n-1} n - i = n(n-1) - \sum_{i=1}^{n-1} i = n^2 - n - \frac{n^2 - n}{2} = \frac{n^2 - n}{2} = \Theta(n^2).$$

Exercise 2.2-3

Suppose that every entry has a fixed probability p of being the element looked for. A different interpretation of the question is given at the end of this solution. Then, we will only check k elements if the previous $k - 1$ positions were not the element being looked for, and the kth position is the desired value. This means that the probabilty that the number of steps taken is k is $p(1-p)^k$. The last possibility is that none of the elements in the array match what we are looking for, in which case we look at all $A.length$ many positions, and it happens with probability $(1 - p)$.

By multiplying the number of steps in each case by the probability that that case happens, we get the expected value of:

$$\mathbf{E}(steps) = A.length(1 - p)^{A.length} + \sum_{k=1}^{A.length} k(1-p)^{k-1}p$$

The worst case is obviously if you have to check all of the possible positions, in which case, it will take exactly $A.length$ steps, so it is $\Theta(A.length)$.

Now, we analyze the asyptotic behavior of the average-case. Consider the following manipulations, where first, we rewrite the single summation as a double summation, and then use the geometric sum formula twice.

3

$$\mathbf{E}(steps) = A.length(1-p)^{A.length} + \sum_{k=1}^{A.length} k(1-p)^{k-1}p$$

$$= A.length(1-p)^{A.length} + \sum_{s=1}^{A.length}\sum_{k=s}^{A.length} (1-p)^{k-1}p$$

$$= A.length(1-p)^{A.length} + \sum_{s=1}^{A.length} p \sum_{k=s}^{A.length} (1-p)^{k-1}$$

$$= A.length(1-p)^{A.length} + \sum_{s=1}^{A.length} p\frac{(1-p)^{s-1}-(1-p)^{A.length}}{1-(1-p)}$$

$$= A.length(1-p)^{A.length} + \sum_{s=1}^{A.length} (1-p)^{s-1}-(1-p)^{A.length}$$

$$= A.length(1-p)^{A.length} + \sum_{s=1}^{A.length} (1-p)^{s-1} - \sum_{s=1}^{A.length} (1-p)^{A.length}$$

$$= \sum_{s=1}^{A.length} (1-p)^{s-1}$$

$$= \frac{1-(1-p)^{A.length}}{1-(1-p)}$$

$$= \frac{1}{p} - \frac{(1-p)^{A.length}}{p}$$

Since $\frac{(1-p)^{A.length}}{p} > 0$, we have that $\mathbf{E}(steps) < \frac{1}{p}$. Also, since $A.length \geq 1$, we have

$$\mathbf{E}(steps) > \frac{1}{p} - \frac{1-p}{p} = 1$$

Therefore, since we have bounded above and below by a constant, we get the somewhat unintuitive result that the expected runtime as a function of $A.length$, where p is held constant is $\Theta(1)$.

A different way of interpreting this statement of this question is that there is exactly one element in the array that is the one you are looking for, and then each position is equally likely to be the one containing that element. In this case, the worst case behavior is unchanged, and the expected runtime is $\sum_{i=1}^{A.length} \frac{i}{A.length} = \frac{A.length+1}{2}$. This makes the asymptotics for the expected case $\Theta(A.length)$.

Exercise 2.2-4

For a good best-case running time, modify an algorithm to first randomly produce output and then check whether or not it satisfies the goal of the algorithm. If so, produce this output and halt. Otherwise, run the algorithm as usual. It is unlikely that this will be successful, but in the best-case the running time will only be as long as it takes to check a solution. For example, we could modify selection sort to first randomly permute the elements of A, then check if they are in sorted order. If they are, output A. Otherwise run selection sort as usual. In the best case, this modified algorithm will have running time $\Theta(n)$.

Exercise 2.3-1

If we start with reading across the bottom of the tree and then go up level by level.

3	41	52	26	38	57	9	49
3	41	26	52	38	57	9	49
3	26	41	52	9	38	49	57
3	9	26	38	41	49	52	57

Exercise 2.3-2

The following is a rewrite of MERGE which avoids the use of sentinels. Much like MERGE, it begins by copying the subarrays of A to be merged into arrays L and R. At each iteration of the while loop starting on line 13 it selects the next smallest element from either L or R to place into A. It stops if either L or R runs out of elements, at which point it copies the remainder of the other subarray into the remaining spots of A.

Exercise 2.3-3

Since n is a power of two, we may write $n = 2^k$. If $k = 1$, $T(2) = 2 = 2\lg(2)$. Suppose it is true for k, we will show it is true for $k + 1$.

$$T(2^{k+1}) = 2T\left(\frac{2^{k+1}}{2}\right) + 2^{k+1} = 2T\left(2^k\right) + 2^{k+1} = 2(2^k\lg(2^k)) + 2^{k+1}$$

$$= k2^{k+1} + 2^{k+1} = (k+1)2^{k+1} = 2^{k+1}\lg(2^{k+1}) = n\lg(n)$$

Exercise 2.3-4

Let $T(n)$ denote the running time for insertion sort called on an array of size n. We can express $T(n)$ recursively as

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ T(n-1) + I(n) & \text{otherwise} \end{cases}$$

where $I(n)$ denotes the amount of time it takes to insert $A[n]$ into the sorted array $A[1..n-1]$. Since we may have to shift as many as $n-1$ elements once we find the correct place to insert $A[n]$, we have $I(n) = \theta(n)$.

Algorithm 5 $Merge(A, p, q, r)$

1: $n_1 = q - p + 1$
2: $n_2 = r - q$
3: let $L[1, ..n_1]$ and $R[1..n_2]$ be new arrays
4: **for** $i = 1$ to n_1 **do**
5:     $L[i] = A[p + i - 1]$
6: **end for**
7: **for** $j = 1$ to n_2 **do**
8:     $R[j] = A[q + j]$
9: **end for**
10: $i = 1$
11: $j = 1$
12: $k = p$
13: **while** $i \neq n_1 + 1$ and $j \neq n_2 + 1$ **do**
14:     **if** $L[i] \leq R[j]$ **then**
15:         $A[k] = L[i]$
16:         $i = i + 1$
17:     **else** $A[k] = R[j]$
18:         $j = j + 1$
19:     **end if**
20:     $k = k + 1$
21: **end while**
22: **if** $i == n_1 + 1$ **then**
23:     **for** $m = j$ to n_2 **do**
24:         $A[k] = R[m]$
25:         $k = k + 1$
26:     **end for**
27: **end if**
28: **if** $j == n_2 + 1$ **then**
29:     **for** $m = i$ to n_1 **do**
30:         $A[k] = L[m]$
31:         $k = k + 1$
32:     **end for**
33: **end if**

Exercise 2.3-5

The following recursive algorithm gives the desired result when called with
$a = 1$ and $b = n$.

1: BinSearch(a,b,v)
2: **if** **then**$a > b$
3:     return NIL
4: **end if**
5: $m = \lfloor \frac{a+b}{2} \rfloor$
6: **if** **then**$m = v$
7:     return m
8: **end if**
9: **if** **then**$m < v$
10:     return BinSearch(a,m,v)
11: **end if**
12: return BinSearch(m+1,b,v)

Note that the initial call should be $BinSearch(1, n, v)$. Each call results in
a constant number of operations plus a call to a problem instance where the
quantity $b - a$ falls by at least a factor of two. So, the runtime satisfies the
recurrence $T(n) = T(n/2) + c$. So, $T(n) \in \Theta(\lg(n))$

Exercise 2.3-6

A binary search wouldn't improve the worst-case running time. Insertion
sort has to copy each element greater than *key* into its neighboring spot in the
array. Doing a binary search would tell us how many how many elements need
to be copied over, but wouldn't rid us of the copying needed to be done.

Exercise 2.3-7

We can see that the while loop gets run at most $O(n)$ times, as the quantity
$j-i$ starts at $n-1$ and decreases at each step. Also, since the body only consists
of a constant amount of work, all of lines 2-15 takes only $O(n)$ time. So, the
runtime is dominated by the time to perform the sort, which is $\Theta(n \lg(n))$.
We will prove correctness by a mutual induction. Let $m_{i,j}$ be the proposition
$A[i] + A[j] < S$ and $M_{i,j}$ be the proposition $A[i] + A[j] > S$. Note that because
the array is sorted, $m_{i,j} \Rightarrow \forall k < j, m_{i,k}$, and $M_{i,j} \Rightarrow \forall k > i, M_{k,j}$.

Our program will obviously only output true in the case that there is a valid
i and j. Now, suppose that our program output false, even though there were
some i, j that was not considered for which $A[i] + A[j] = S$. If we have $i > j$,
then swap the two, and the sum will not change, so, assume $i \leq j$. we now have
two cases:

Case 1 $\exists k, (i, k)$ was considered and $j < k$. In this case, we take the smallest

```
 1: Use Merge Sort to sort the array $A$ in time $\Theta(n \lg(n))$
 2: $i = 1$
 3: $j = n$
 4: while $i < j$ do
 5:     if $A[j] + A[j] = S$ then
 6:         return true
 7:     end if
 8:     if $A[i] + A[j] < S$ then
 9:         $i = i + 1$
10:     end if
11:     if $A[i] + A[j] > S$ then
12:         $j = j - 1$
13:     end if
14: end while
15: return false
```

such k. The fact that this is nonzero meant that immediately after considering it, we considered (i+1,k) which means $m_{i,k}$ this means $m_{i,j}$

Case 2 $\exists k, (k, j)$ was considered and $k < i$. In this case, we take the largest such k. The fact that this is nonzero meant that immediately after considering it, we considered (k,j-1) which means $M_{k,j}$ this means $M_{i,j}$

Note that one of these two cases must be true since the set of considered points separates $\{(m, m') : m \leq m' < n\}$ into at most two regions. If you are in the region that contains $(1, 1)$(if nonempty) then you are in Case 1. If you are in the region that contains (n, n) (if non-empty) then you are in case 2.

Problem 2-1

a. The time for insertion sort to sort a single list of length k is $\Theta(k^2)$, so, n/k of them will take time $\Theta(\frac{n}{k} k^2) = \Theta(nk)$.

b. Suppose we have coarseness k. This meas we can just start using the usual merging procedure, except starting it at the level in which each array has size at most k. This means that the depth of the merge tree is $\lg(n) - \lg(k) = \lg(n/k)$. Each level of merging is still time cn, so putting it together, the merging takes time $\Theta(n \lg(n/k))$.

c. Viewing k as a function of n, as long as $k(n) \in O(\lg(n))$, it has the same asymptotics. In particular, for any constant choice of k, the asymptotics are the same.

d. If we optimize the previous expression using our calculus 1 skills to get k, we have that $c_1 n - \frac{nc_2}{k} = 0$ where c_1 and c_2 are the coeffients of nk and $n \lg(n/k)$ hidden by the asymptotics notation. In particular, a constant choice of k is optimal. In practice we could find the best choice of this k by just trying and timing for various values for sufficiently large n.

Problem 2-2

1. We need to prove that A' contains the same elements as A, which is easily seen to be true because the only modification we make to A is swapping its elements, so the resulting array must contain a rearrangement of the elements in the original array.

2. The **for** loop in lines 2 through 4 maintains the following loop invariant: At the start of each iteration, the position of the smallest element of $A[i..n]$ is at most j. This is clearly true prior to the first iteration because the position of any element is at most $A.length$. To see that each iteration maintains the loop invariant, suppose that $j = k$ and the position of the smallest element of $A[i..n]$ is at most k. Then we compare $A[k]$ to $A[k-1]$. If $A[k] < A[k-1]$ then $A[k-1]$ is not the smallest element of $A[i..n]$, so when we swap $A[k]$ and $A[k-1]$ we know that the smallest element of $A[i..n]$ must occur in the first $k-1$ positions of the subarray, the maintaining the invariant. On the other hand, if $A[k] \geq A[k-1]$ then the smallest element can't be $A[k]$. Since we do nothing, we conclude that the smallest element has position at most $k-1$. Upon termination, the smallest element of $A[i..n]$ is in position i.

3. The **for** loop in lines 1 through 4 maintain the following loop invariant: At the start of each iteration the subarray $A[1..i-1]$ contains the $i-1$ smallest elements of A in sorted order. Prior to the first iteration $i = 1$, and the first 0 elements of A are trivially sorted. To see that each iteration maintains the loop invariant, fix i and suppose that $A[1..i-1]$ contains the $i-1$ smallest elements of A in sorted order. Then we run the loop in lines 2 through 4. We showed in part b that when this loop terminates, the smallest element of $A[i..n]$ is in position i. Since the $i-1$ smallest elements of A are already in $A[1..i-1]$, $A[i]$ must be the ith smallest element of A. Therefore $A[1..i]$ contains the i smallest elements of A in sorted order, maintaining the loop invariant. Upon termination, $A[1..n]$ contains the n elements of A in sorted order as desired.

4. The ith iteration of the **for** loop of lines 1 through 4 will cause $n - i$ iterations of the **for** loop of lines 2 through 4, each with constant time execution, so the worst-case running time is $\Theta(n^2)$. This is the same as that of insertion sort; however, bubble sort also has best-case running time $\Theta(n^2)$ whereas insertion sort has best-case running time $\Theta(n)$.

Problem 2-3

a. If we assume that the arithmetic can all be done in constant time, then since the loop is being executed n times, it has runtime $\Theta(n)$.

b.
```
1: y = 0
2: for i=0 to n do
3:     y_i = x
4:     for j=1 to n do
5:         y_i = y_i x
6:     end for
7:     y = y + a_i y_i
8: end for
```

This code has runtime $\Theta(n^2)$ because it has to compute each of the powers of x. This is slower than Horner's rule.

c. Initially, $i = n$, so, the upper bound of the summation is -1, so the sum evaluates to 0, which is the value of y. For preservation, suppose it is true for an i, then,

$$y = a_i + x \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k = a_i + x \sum_{k=1}^{n-i} a_{k+i} x^{k-1} = \sum_{k=0}^{n-i} a_{k+i} x^k$$

At termination, $i = 0$, so is summing up to $n - 1$, so executing the body of the loop a last time gets us the desired final result.

d. We just showed that the algorithm evaluated $\Sigma_{k=0}^{n} a_k x^k$. This is the value of the polynomial evaluated at x.

Problem 2-4

a. The five inversions are $(2, 1)$, $(3, 1)$, $(8, 6)$, $(8, 1)$, and $(6, 1)$.

b. The n-element array with the most inversions is $\langle n, n - 1, \ldots, 2, 1 \rangle$. It has $n - 1 + n - 2 + \ldots + 2 + 1 = \frac{n(n-1)}{2}$ inversions.

c. The running time of insertion sort is a constant times the number of inversions. Let $I(i)$ denote the number of $j < i$ such that $A[j] > A[i]$. Then $\sum_{i=1}^{n} I(i)$ equals the number of inversions in A. Now consider the **while** loop on lines 5-7 of the insertion sort algorithm. The loop will execute once for each element of A which has index less than j is larger than $A[j]$. Thus, it will execute $I(j)$ times. We reach this **while** loop once for each iteration of the **for** loop, so the number of constant time steps of insertion sort is $\sum_{j=1}^{n} I(j)$ which is exactly the inversion number of A.

10

d. We'll call our algorithm M.Merge-Sort for Modified Merge Sort. In addition to sorting A, it will also keep track of the number of inversions. The algorithm works as follows. When we call M.Merge-Sort(A,p,q) it sorts $A[p..q]$ and returns the number of inversions in the elements of $A[p..q]$, so $left$ and $right$ track the number of inversions of the form (i, j) where i and j are both in the same half of A. When M.Merge(A,p,q,r) is called, it returns the number of inversions of the form (i, j) where i is in the first half of the array and j is in the second half. Summing these up gives the total number of inversions in A. The runtime is the same as that of Merge-Sort because we only add an additional constant-time operation to some of the iterations of some of the loops. Since Merge is $\Theta(n \log n)$, so is this algorithm.

Algorithm 6 M.Merge-Sort(A, p, r)

---

if $p < r$ **then**
    $q = \lfloor (p + r)/2 \rfloor$
    $left = M.Merge - Sort(A, p, q)$
    $right = M.Merge - Sort(A, q + 1, r)$
    $inv = M.Merge(A, p, q, r) + left + right$
    return inv
end if
return 0

Algorithm 7 M.Merge(A,p,q,r)

---

  inv $= 0$
  $n_1 = q - p + 1$
  $n_2 = r - q$
  let $L[1, ..n_1]$ and $R[1..n_2]$ be new arrays
  for $i = 1$ to n_1 **do**
     $L[i] = A[p + i - 1]$
  end for
  for $j = 1$ to n_2 **do**
     $R[j] = A[q + j]$
  end for
  $i = 1$
  $j = 1$
  $k = p$
  while $i \neq n_1 + 1$ and $j \neq n_2 + 1$ **do**
     if $L[i] \leq R[j]$ **then**
        $A[k] = L[i]$
        $i = i + 1$
     else $A[k] = R[j]$
        $inv = inv + j$ // This keeps track of the number of inversions between
the left and right arrays.
        $j = j + 1$
     end if
     $k = k + 1$
  end while
  if $i == n_1 + 1$ **then**
     for $m = j$ to n_2 **do**
        $A[k] = R[m]$
        $k = k + 1$
     end for
  end if
  if $j == n_2 + 1$ **then**
     for $m = i$ to n_1 **do**
        $A[k] = L[m]$
        $inv = inv + n_2$ // Tracks inversions once we have exhausted the right
array. At this point, every element of the right array contributes an inversion.
        $k = k + 1$
     end for
  end if
  return inv

Chapter 3

Michelle Bodnar, Andrew Lohr

October 25, 2017

Exercise 3.1-1

Since we are requiring both f and g to be aymptotically non-negative, suppose that we are past some n_1 where both are non-negative (take the max of the two bounds on the n corresponding to both f and g). Let $c_1 = .5$ and $c_2 = 1$.

$$0 \leq .5(f(n) + g(n)) \leq .5(\max(f(n), g(n)) + \max(f(n), g(n)))$$

$$= \max(f(n), g(n)) \leq \max(f(n), g(n)) + \min(f(n), g(n)) = (f(n) + g(n))$$

Exercise 3.1-2

Let $c = 2^b$ and $n_0 \geq 2a$. Then for all $n \geq n_0$ we have $(n+a)^b \leq (2n)^b = cn^b$ so $(n+a)^b = O(n^b)$. Now let $n_0 \geq \frac{-a}{1-1/2^{1/b}}$ and $c = \frac{1}{2}$. Then $n \geq n_0 \geq \frac{-a}{1-1/2^{1/b}}$ if and only if $n - \frac{n}{2^{1/b}} \geq -a$ if and only if $n + a \geq (1/2)^{a/b} n$ if and only if $(n+a)^b \geq cn^b$. Therefore $(n+a)^b = \Omega(n^b)$. By Theorem 3.1, $(n+a)^b = \Theta(n^b)$.

Exercise 3.1-3

There are a ton of different funtions that have growth rate less than or equal to n^2. In particular, functions that are constant or shrink to zero arbitrarily fast. Saying that you grow more quickly than a function that shrinks to zero quickly means nothing.

Exercise 3.1-4

$2^{n+1} \geq 2 \cdot 2^n$ for all $n \geq 0$, so $2^{n+1} = O(2^n)$. However, 2^{2n} is not $O(2^n)$. If it were, there would exist n_0 and c such that $n \geq n_0$ implies $2^n \cdot 2^n = 2^{2n} \leq c2^n$, so $2^n \leq c$ for $n \geq n_0$ which is clearly impossible since c is a constant.

Exercise 3.1-5

Suppose $f(n) \in \Theta(g(n))$, then $\exists c_1, c_2, n_0, \forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$, if we just look at these inequalities saparately, we have $c_1 g(n) \leq f(n)$ ($f(n) \in \Omega(g(n))$) and $f(n) \leq c_2 g(n)$ ($f(n) \in O(g(n))$).

Suppose that we had $\exists n_1, c_1, \forall n \geq n_1, c_1 g(n) \leq f(n)$ and $\exists n_2, c_2, \forall n \geq n_2, f(n) \leq c_2 g(n)$. Putting these together, and letting $n_0 = \max(n_1, n_2)$, we have $\forall n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n)$.

Exercise 3.1-6

Suppose the running time is $\Theta(g(n))$. By Theorem 3.1, the running time is $O(g(n))$, which implies that for any input of size $n \geq n_0$ the running time is bounded above by $c_1 g(n)$ for some c_1. This includes the running time on the worst-case input. Theorem 3.1 also imlpies the running time is $\Omega(g(n))$, which implies that for any input of size $n \geq n_0$ the running time is bounded below by $c_2 g(n)$ for some c_2. This includes the running time of the best-case input.

On the other hand, the running time of any input is bounded above by the worst-case running time and bounded below by the best-case running time. If the worst-case and best-case running times are $O(g(n))$ and $\Omega(g(n))$ respectively, then the running time of any input of size n must be $O(g(n))$ and $\Omega(g(n))$. Theorem 3.1 implies that the running time is $\Theta(g(n))$.

Exercise 3.1-7

Suppose we had some $f(n) \in o(g(n)) \cap \omega(g(n))$. Then, we have

$$0 = \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

a contradiction.

Exercise 3.1-8

$\Omega(g(n, m)) = \{f(n, m) : \text{there exist positive constants } c, n_0, \text{ and } m_0 \text{ such that } f(n, m) \geq cg(n, m)$
$\text{for all } n \geq n_0 \text{ or } m \geq m_0\}$

$\Theta(g(n, m)) = \{f(n, m) : \text{there exist positive constants } c_1, c_2, n_0, \text{ and } m_0 \text{ such that } c_1 g(n, m) \leq f(n, m)$
$\leq c_2 g(n, m) \text{ for all } n \geq n_0 \text{ or } m \geq m_0\}$

Exercise 3.2-1

Let $n_1 < n_2$ be arbitrary. From f and g being monatonic increasing, we know $f(n_1) < f(n_2)$ and $g(n_1) < g(n_2)$. So

$$f(n_1) + g(n_1) < f(n_2) + g(n_1) < f(n_2) + g(n_2)$$

Since $g(n_1) < g(n_2)$, we have $f(g(n_1)) < f(g(n_2))$. Lastly, if both are nonnegative, then,

$$f(n_1)g(n_1) = f(n_2)g(n_1) + (f(n_2) - f(n_1))g(n_1)$$

$$= f(n_2)g(n_2) + f(n_2)(g(n_2) - g(n_1)) + (f(n_2) - f(n_1))g(n_1)$$

Since $f(n_1) \geq 0$, $f(n_2) > 0$, so, the second term in this expression is greater than zero. The third term is nonnegative, so, the whole thing is $< f(n_2)g(n_2)$.

Exercise 3.2-2

$$a^{\log_b(c)} = a^{\frac{\log_a(c)}{\log_a(b)}} = c^{\frac{1}{\log_a(b)}} = c^{\log_b(a)}.$$

Exercise 3.2-3

As the hint suggests, we will apply stirling's approximation

$$\lg(n!) = \lg\left(\sqrt{(2\pi n)}\left(\frac{n}{e}\right)^n\left(1 + \Theta\left(\frac{1}{n}\right)\right)\right)$$

$$= \frac{1}{2}\lg(2\pi n) + n\lg(n) - n\lg(e) + \lg\left(\Theta\left(\frac{n+1}{n}\right)\right)$$

Note that this last term is $O(\lg(n))$ if we just add the two expression we get when we break up the lg instead of subtract them. So, the whole expression is dominated by $n\lg(n)$. So, we have that $\lg(n!) = \Theta(n\lg(n))$.

$$\lim_{n\to\infty}\frac{2^n}{n!} = \lim_{n\to\infty}\frac{1}{\sqrt{2\pi n}(1 + \Theta(\frac{1}{n}))}\left(\frac{2e}{n}\right)^n \leq \lim_{n\to\infty}\left(\frac{2e}{n}\right)^n$$

If we restrict to $n > 4e$, then this is

$$\leq \lim_{n\to\infty}\frac{1}{2^n} = 0$$

$$\lim_{n\to\infty}\frac{n^n}{n!} = \lim_{n\to\infty}\frac{1}{\sqrt{2\pi n}(1 + \Theta(\frac{1}{n}))}e^n = \lim_{n\to\infty}O(n^{-.5})e^n \geq \lim_{n\to\infty}\frac{e^n}{c_1\sqrt{n}}$$

$$\geq \lim_{n\to\infty}\frac{e^n}{c_1 n} = \lim_{n\to\infty}\frac{e^n}{c_1} = \infty$$

Exercise 3.2-4

The function $\lceil \log n\rceil!$ is not polynomially bounded. If it were, there would exist constants c, a, and n_0 such that for all $n \geq n_0$ the inequality $\lceil \log n\rceil! \leq cn^a$ would hold. In particular, it would hold when $n = 2^k$ for $k \in \mathbb{N}$. Then

this becomes $k! \leq c(2^a)^k$, a contradiction since the factorial function is not exponentially bounded.

We'll show that $\lceil \log \log n \rceil ! \leq n$. Without loss of generality assume $n = 2^{2^k}$. Then this becomes equivalent to showing $k! \leq 2^{2^k}$, or $1 \cdot 2 \cdots (k-1) \cdot k \leq 4 \cdot 16 \cdot 2^8 \cdots 2^{2^k}$, which is clearly true for $k \geq 1$. Therefore it is polynomially bounded.

Exercise 3.2-5

Note that $\lg^*(2^n) = 1 + \lg^*(n)$, so,

$$\lim_{n \to \infty} \frac{\lg(\lg^*(n))}{\lg^*(\lg(n))} = \lim_{n \to \infty} \frac{\lg(\lg^*(2^n))}{\lg^*(\lg(2^n))}$$
$$= \lim_{n \to \infty} \frac{\lg(1 + \lg^*(n))}{\lg^*(n)}$$
$$= \lim_{n \to \infty} \frac{\lg(1 + n)}{n}$$
$$= \lim_{n \to \infty} \frac{1}{1 + n}$$
$$= 0$$

So, we have that $\lg^*(\lg(n))$ grows more quickly

Exercise 3.2-6

$$\phi^2 = \left(\frac{1 + \sqrt{5}}{2}\right)^2 = \frac{6 + 2\sqrt{5}}{4} = 1 + \frac{1 + \sqrt{5}}{2} = 1 + \phi$$

$$\hat{\phi}^2 = \left(\frac{1 - \sqrt{5}}{2}\right)^2 = \frac{6 - 2\sqrt{5}}{4} = 1 + \frac{1 - \sqrt{5}}{2} = 1 + \hat{\phi}$$

Exercise 3.2-7

First, we show that $1 + \phi = \frac{6 + 2\sqrt{5}}{4} = \phi^2$. So, for every i, $\phi^{i-1} + \phi^{i-2} = \phi^{i-2}(\phi + 1) = \phi^i$. Similarly for $\hat{\phi}$.

For $i = 0$, $\frac{\phi^0 - \hat{\phi}^0}{\sqrt{5}} = 0$. For $i = 1$, $\frac{\frac{1 + \sqrt{5}}{2} - \frac{1 - \sqrt{5}}{2}}{\sqrt{5}} = \frac{\sqrt{5}}{\sqrt{5}} = 1$. Then, by induction, $F_i = F_{i-1} + F_{i-2} = \frac{\phi^{i-1} + \phi^{i-2} - (\hat{\phi}^{i-1} + \hat{\phi}^{i-2})}{\sqrt{5}} = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}}$.

Exercise 3.2-8

Let c_1 and c_2 be such that $c_1 n \leq k \ln k \leq c_2 n$. Then we have $\ln c_1 + \ln n = \ln(c_1 n) \leq \ln(k \ln k) = \ln k + \ln(\ln k)$ so $\ln n = O(\ln k)$. Let c_3 be such that $\ln n \leq c_3 \ln k$. Then

$$\frac{n}{\ln n} \geq \frac{n}{c_3 \ln k} \geq \frac{k}{c_2 c_3}$$

so that $\frac{n}{\ln n} = \Omega(k)$. Similarly, we have $\ln k + \ln(\ln k) = \ln(k \ln k) \leq \ln(c_2 n) = \ln(c_2) + \ln(n)$ so $\ln(n) = \Omega(\ln k)$. Let c_4 be such that $\ln n \geq c_4 \ln k$. Then

$$\frac{n}{\ln n} \leq \frac{n}{c_4 \ln k} \leq \frac{k}{c_1 c_4}$$

so that $\frac{n}{\ln n} = O(k)$. By Theorem 3.1 this implies $\frac{n}{\ln n} = \Theta(k)$. By symmetry, $k = \Theta\left(\frac{n}{\ln n}\right)$.

Problem 3-1

a. If we pick any $c > 0$, then, the end behavior of $cn^k - p(n)$ is going to infinity, in particular, there is an n_0 so that for every $n \geq n_0$, it is positive, so, we can add $p(n)$ to both sides to get $p(n) < cn^k$.

b. If we pick any $c > 0$, then, the end behavior of $p(n) - cn^k$ is going to infinity, in particular, there is an n_0 so that for every $n \geq n_0$, it is positive, so, we can add cn^k to both sides to get $p(n) > cn^k$.

c. We have by the previous parts that $p(n) = O(n^k)$ and $p(n) = \Omega(n^k)$. So, by Theorem 3.1, we have that $p(n) = \Theta(n^k)$.

d.

$$\lim_{n \to \infty} \frac{p(n)}{n^k} = \lim_{n \to \infty} \frac{n^d (a_d + o(1))}{n^k} < \lim_{n \to \infty} \frac{2 a_d n^d}{n^k} = 2 a_d \lim_{n \to \infty} n^{d-k} = 0$$

e.

$$\lim_{n \to \infty} \frac{n^k}{p(n)} = \lim_{n \to \infty} \frac{n^k}{n^d O(1)} < \lim_{n \to \infty} \frac{n^k}{n^d} = \lim_{n \to \infty} n^{k-d} = 0$$

Problem 3-2

A	B	O	o	Ω	ω	Θ
$\lg^k n$	n^ϵ	yes	yes	no	no	no
n^k	c^n	yes	yes	no	no	no
$\sqrt{n}$	$n^{\sin n}$	no	no	no	no	no
2^n	$2^{n/2}$	no	no	yes	yes	no
$n^{\log c}$	$c^{\log n}$	yes	no	yes	no	yes
$\log(n!)$	$\log(n^n)$	yes	no	yes	no	yes

Problem 3-3

a.

$$2^{2^{n+1}}$$
$$2^{2^n}$$
$$(n+1)!$$
$$n!$$
$$n2^n$$
$$e^n$$
$$2^n$$
$$\left(\tfrac{3}{2}\right)^n$$
$$(\lg(n))!$$
$$n^{\lg(\lg(n))} \qquad \lg(n)^{\lg(n)}$$
$$n^3$$
$$n^2 \qquad 4^{\lg(n)}$$
$$n\lg(n) \qquad \lg(n!)$$
$$2^{\lg(n)} \qquad n$$
$$(\sqrt{2})^{\lg(n)}$$
$$2^{\sqrt{2\lg(n)}}$$
$$\lg^2(n)$$
$$\ln(n)$$
$$\sqrt{\lg(n)}$$
$$\ln(\ln(n))$$
$$2^{\lg^*(n)}$$
$$\lg^*(n) \qquad \lg^*(\lg(n))$$
$$\lg(\lg^*(n)$$
$$1 \qquad n^{1/\lg(n)}$$

The terms are in decreasing growth rate by row. Functions in the same row are Θ of each other.

b. If we define the function

$$f(n) = \begin{cases} g_1(n)! & n \mod 2 = 0 \\ \frac{1}{n} & n \mod 2 = 1 \end{cases}$$

Note that $f(n)$ meets the asymptotically positive requirement that this chapter puts on the functions analyzed.

Then, for even n, we have

$$\lim_{n\to\infty} \frac{f(2n)}{g_i(2n)} \geq \lim_{n\to\infty} \frac{f(2n)}{g_1(2n)}$$
$$= \lim_{n\to\infty} (g_1(2n) - 1)!$$
$$= \infty$$

And for odd n, we have

6

$$\lim_{n\to\infty} \frac{f(2n+1)}{g_i(2n+1)} \leq \lim_{n\to\infty} \frac{f(2n+1)}{1}$$
$$= \lim_{n\to\infty} \frac{1}{2n+1}$$
$$= 0$$

By looking at the even n we have that $f(n)$ is not $O(g_i(n))$ for any n. By looking at the odd n, we have that $f(n)$ is not $\Omega(g_i(n))$ for any n.

Problem 3-4

a. False. Counterexample: $n = O(n^2)$ but $n^2 \neq O(n)$.

b. False. Counterexample: $n + n^2 \neq \Theta(n)$.

c. True. Since $f(n) = O(g(n))$ there exist c and n_0 such that $n \geq n_0$ implies $f(n) \leq cg(n)$ and $f(n) \geq 1$. This means that $\log(f(n)) \leq \log(cg(n)) = \log(c) + \log(g(n))$. Note that the inequality is preserved after taking logs because $f(n) \geq 1$. Now we need to find d such that $\log(f(n)) \leq d\log(g(n))$. It will suffice to make $\log(c) + \log(g(n)) \leq d\log(g(n))$, which is achieved by taking $d = \log(c) + 1$, since $\log(g(n)) \geq 1$.

d. False. Counterexample: $2n = O(n)$ but $2^{2n} \neq 2^n$ as shown in exercise 3.1-4.

e. False. Counterexample: Let $f(n) = \frac{1}{n}$. Suppose that c is such that $\frac{1}{n} \leq c\frac{1}{n^2}$ for $n \geq n_0$. Choose k such that $kc \geq n_0$ and $k > 1$. Then this implies $\frac{1}{kc} \leq \frac{c}{k^2c^2} = \frac{1}{k^2c}$, a contradiction.

f. True. Since $f(n) = O(g(n))$ there exist c and n_0 such that $n \geq n_0$ implies $f(n) \leq cg(n)$. Thus $g(n) \geq \frac{1}{c}f(n)$, so $g(n) = \Omega(f(n))$.

g. False. Counterexample: Let $f(n) = 2^{2n}$. By exercise 3.1-4, $2^{2n} \neq O(2^n)$.

h. True. Let g be any function such that $g(n) = o(f(n))$. Since g is asymptotically positive let n_0 be such that $n \geq n_0$ implies $g(n) \geq 0$. Then $f(n) + g(n) \geq f(n)$ so $f(n) + o(f(n)) = \Omega(f(n))$. Next, choose n_1 such that $n \geq n_1$ implies $g(n) \leq f(n)$. Then $f(n) + g(n) \leq f(n) + f(n) = 2f(n)$ so $f(n) + o(f(n)) = O(f(n))$. By Theorem 3.1, this implies $f(n) + o(f(n)) = \Theta(f(n))$.

Problem 3-5

a. Suppose that we do not have that $f = O(g(n))$. This means that $\forall c > 0, n_0, \exists n \geq n_0, f(n) > cg(n)$. Since this holds for every c, we can let it be arbitrary, say 1. Initially, we set $n_0 = 1$, then, the resulting n we will call a_1. Then, in general, let $n_0 = a_i + 1$ and let a_{i+1} be the resulting value of n. Then, on the infinite set $\{a_1, a_2, \ldots\}$, we have $f(n) > g(n)$, and so, $f = \overset{\infty}{\Omega}(g(n))$

This is not the case for the usual definition of Ω. Suppose we had $f(n) = n^2(n \bmod 2)$ and $g(n) = n$. On all the even values, $g(n)$ is larger, but on all the odd values, $f(n)$ grows more quickly.

b. The advantage is that you get the result of part a which is a nice property. A disadvantage is that the infinite set of points on which you are making claims of the behavior could be very sparse. Also, there is nothing said about the behavior when outside of this infinite set, it can do whatever it wants.

c. A function f can only be in $\Theta(g(n))$ if $f(n)$ has an infinite tail that is non–negative. In this case, the definition of $O(g(n))$ agrees with $O'(g(n))$. Similarly, for a function to be in $\Omega(g(n))$, we need that $f(n)$ is non-negative for some infinite tail, on which $O(g(n))$ is identical to $O'(g(n))$. So, we have that in both directions, changing O to O' does not change anything.

d. Suppose $f(n) \in \widetilde{\Theta}(g(n))$, then $\exists c_1, c_2, k_1, k_2, n_0, \forall n \geq n_0, 0 \leq \frac{c_1 g(n)}{\lg^{k_1}(n)} \leq f(n) \leq c_2 g(n) \lg^{k_2}(n)$, if we just look at these inequalities separately, we have $\frac{c_1 g(n)}{\lg^{k_1}(n)} \leq f(n)$ $(f(n) \in \widetilde{\Omega}(g(n)))$ and $f(n) \leq c_2 g(n) \lg^{k_2}(n)$ $(f(n) \in \widetilde{O}(g(n)))$.

Now for the other direction. Suppose that we had $\exists n_1, c_1, k_1 \forall n \geq n_1, \frac{c_1 g(n)}{\lg^{k_1}(n)} \leq f(n)$ and $\exists n_2, c_2, k_2, \forall n \geq n_2, f(n) \leq c_2 g(n) \lg^{k_2}(n)$. Putting these together, and letting $n_0 = \max(n_1, n_2)$, we have $\forall n \geq n_0, \frac{c_1 g(n)}{\lg^{k_1}(n)} \leq f(n) \leq c_2 g(n) \lg^{k_2}(n)$.

Problem 3-6

$f(n)$	c	$f_c^*(n)$
$n-1$	0	$\lceil n \rceil$
$\log n$	1	$\log^* n$
$n/2$	1	$\lceil \log(n) \rceil$
$n/2$	2	$\lceil \log(n) \rceil - 1$
$\sqrt{n}$	2	$\log \log n$
$\sqrt{n}$	1	undefined
$n^{1/3}$	2	$\log_3 \log_2(n)$
$n/\log n$	2	$\Omega\left(\frac{\log n}{\log(\log n)}\right)$

Chapter 4

Michelle Bodnar, Andrew Lohr

February 5, 2018

Exercise 4.1-1

It will return the least negative position. As each of the cross sums are computed, the most positive one must have the shortest possible lengths. The algorithm doesn't consider length zero sub arrays, so it must have length 1.

Exercise 4.1-2

Algorithm 1 Brute Force Algorithm to Solve Maximum Subarray Problem
---
$left = 1$
$right = 1$
$max = A[1]$
$curSum = 0$
for $i = 1$ to n **do** // Increment left end of subarray
    $curSum = 0$
    for $j = i$ to n **do** // Increment right end of subarray
        $curSum = curSum + A[j]$
        if $curSum > max$ **then**
            $max = curSum$
            $left = i$
            $right = j$
   **end if**
    end for
end for

Exercise 4.1-3

The crossover point is at around a length 20 array, however, the times were incredibly noisy and I think that there was a garbage collection during the run, so it is not reliable. It would probably be more effective to use an actual profiler for measuring runtimes. By switching over the way the recursive algorithm handles the base case, the recursive algorithm is now better for smaller values of n. The chart included has really strange runtimes for the brute force algorithm. These times were obtained on a Core 2 duo P8700 and java 1.8.0.51.

In the chart of runtimes, the x axis is the length of the array input. The y axis is the measured runtime in nanoseconds.
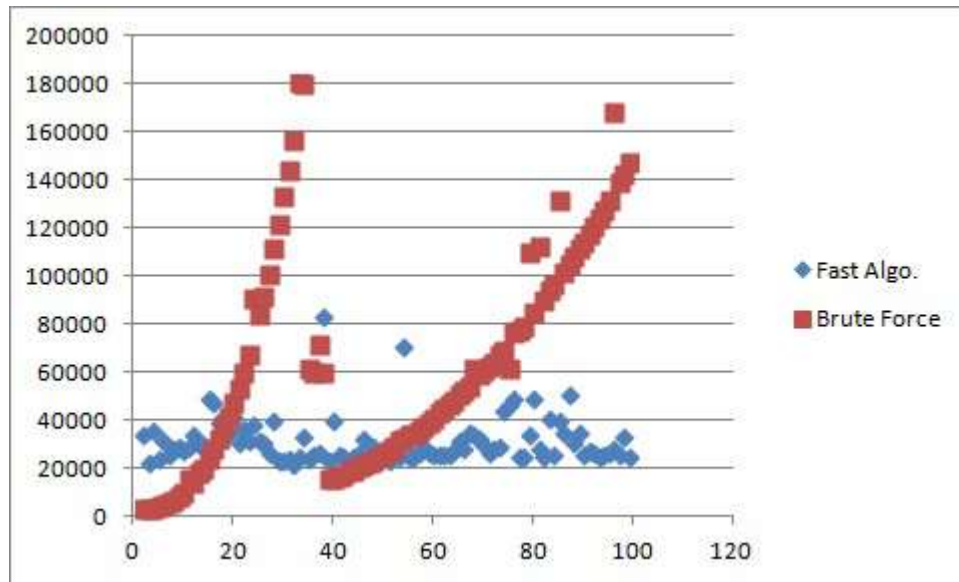
Exercise 4.1-4

First do a linear scan of the input array to see if it contains any positive entries. If it does, run the algorithm as usual. Otherwise, return the empty subarray with sum 0 and terminate the algorithm.

Exercise 4.1-5

See the algorithm labeled linear time maximum subarray.

Exercise 4.2-1

Algorithm 2 linear time maximum subarray(A)

1: $M = -\infty$
2: $low_M, high_M = null$
3: $M_r = 0$
4: $low_r = 1$
5: **for** i from 1 to A.length **do**
6:     $M_r + = A[i]$
7:     **if** $M_r > M$ **then**
8:         $low_M = low_r$
9:         $high_M = i$
10:         $M = M_r$
11:     **end if**
12:     **if** $M_r < 0$ **then**
13:         $M_r = 0$
14:         $low_r = i + 1$
15:     **end if**
16: **end for**
17: **return** $(low_M, high_M, M)$

$$S_1 = 8 - 2 = 6$$
$$S_2 = 1 + 3 = 4$$
$$S_3 = 7 + 5 = 12$$
$$S_4 = 4 - 6 = -2$$
$$S_5 = 1 + 5 = 6$$
$$S_6 = 6 + 2 = 8$$
$$S_7 = 3 - 5 = -2$$
$$S_8 = 4 + 2 = 6$$
$$S_9 = 1 - 7 = -6$$
$$S_{10} = 6 + 8 = 14$$

$$P_1 = 6$$
$$P_2 = 8$$
$$P_3 = 72$$
$$P_4 = -10$$
$$P_5 = 48$$
$$P_6 = -12$$
$$P_7 = -84$$

$$C_{11} = 48 - 10 - 8 - 12 = 18$$
$$C_{12} = 6 + 8 = 14$$
$$C_{21} = 72 - 10 = 62$$
$$C_{22} = 48 + 6 - 72 + 84 = 66$$

So, we get the final result:

$$\begin{pmatrix} 18 & 14 \\ 62 & 66 \end{pmatrix}$$

Exercise 4.2-2

As usual, we will assume that n is an exact power of 2 and A and B are n by n matrices. Let $A[i..j][k..m]$ denote the submatrix of A consisting of rows i through j and columns k through m.

Exercise 4.2-3

You could pad out the input matrices to be powers of two and then run the given algorithm. Padding out the the next largest power of two (call it m) will at most double the value of n because each power of two is off from each other by a factor of two. So, this will have runtime

$$m^{\lg 7} \leq (2n)^{\lg 7} = 7n^{\lg 7} \in O(n^{\lg 7})$$

and

$$m^{\lg 7} \geq n^{\lg 7} \in \Omega(n^{\lg 7})$$

Putting these together, we get the runtime is $\Theta(n^{\lg 7})$.

Exercise 4.2-4

Assume that $n = 3^m$ for some m. Then, using block matrix multiplication, we obtain the recursive running time $T(n) = kT(n/3) + O(1)$. Using the Master theorem, we need the largest integer k such that $\log_3 k < \lg 7$. This is given by $k = 21$.

Exercise 4.2-5

If we take the three algorithms and divide the number of multiplications by the side length of the matrices raised to $\lg(7)$, we approximately get the

Algorithm 3 Strassen(A, B)

if $A.length == 1$ **then**
    return $A[1] \cdot B[1]$
end if
Let C be a new n by n matrix
$A11 = A[1..n/2][1..n/2]$
$A12 = A[1..n/2][n/2 + 1..n]$
$A21 = A[n/2 + 1..n][1..n/2]$
$A22 = A[n/2 + 1..n][n/2 + 1..n]$
$B11 = B[1..n/2][1..n/2]$
$B12 = B[1..n/2][n/2 + 1..n]$
$B21 = B[n/2 + 1..n][1..n/2]$
$B22 = B[n/2 + 1..n][n/2 + 1..n]$
$S_1 = B12 - B22$
$S_2 = A11 + A12$
$S_3 = A21 + A22$
$S_4 = B21 - B11$
$S_5 = A11 + A22$
$S_6 = B11 + B22$
$S_7 = A12 - A22$
$S_8 = B21 + B22$
$S_9 = A11 - A21$
$S_{10} = B11 + B12$
$P_1 = Strassen(A11, S_1)$
$P_2 = Strassen(S_2, B22)$
$P_3 = Strassen(S_3, B11)$
$P_4 = Strassen(A22, S_4)$
$P_5 = Strassen(S_5, S_6)$
$P_6 = Strassen(S_7, S_8)$
$P_7 = Strassen(S_9, S_{10})$
$C[1..n/2][1..n/2] = P_5 + P_4 - P_2 + P_6$
$C[1..n/2][n/2 + 1..n] = P_1 + P_2$
$C[n/2 + 1..n][1..n/2] = P_3 + P_4$
$C[n/2 + 1..n][n/2 + 1..n] = P_5 + P_1 - P_3 - P_7$
return C

following values

$$3745$$
$$3963$$
$$4167$$

This means that, if used as base cases for a Strassen Algorithm, the first one will perform best for very large matrices.

Exercise 4.2-6

By considering block matrix multiplication and using Strassen's algorithm as a subroutine, we can multiply a $kn \times n$ matrix by an $n \times kn$ matrix in $\Theta(k^2 n^{\log 7})$ time. With the order reversed, we can do it in $\Theta(kn^{\log 7})$ time.

Exercise 4.2-7

We can see that the final result should be

$$(a + bi)(c + di) = ac - bd + (cb + ad)i$$

We will be multiplying

$$P_1 = (a + b)c = ac + bc \qquad P_2 = b(c + d) = bc + bd \qquad P_3 = (a - b)d = ad + bd$$

Then, we can recover the real part by taking $P_1 - P_2$ and the imaginary part by taking $P_2 + P_3$.

Exercise 4.3-1

Inductively assume $T(n) \leq cn^2$, were c is taken to be $\max(1, T(1))$ then

$$T(n) = T(n-1) + n \leq c(n-1)^2 + n = cn^2 + (1 - 2c)n + 1 \leq cn^2 + 2 - 2c \leq cn^2$$

The first inequality comes from the inductive hypothesis, the second from the fact that $n \geq 1$ and $1 - 2c < 0$. The last from the fact that $c \geq 1$.

Exercise 4.3-2

We'll show $T(n) \leq 3 \log n - 1$, which will imply $T(n) = O(\log n)$.

$$
\begin{aligned}
T(n) &= T(\lceil n/2 \rceil) + 1 \\
&\leq 3 \log(\lceil n/2 \rceil) - 1 + 1 \\
&\leq 3 \log(3n/4) \\
&= 3 \log n + 3 \log(3/4) \\
&\leq 3 \log n + \log(1/2) \\
&= 3 \log n - 1.
\end{aligned}
$$

Exercise 4.3-3

Inductively assume that $T(n) \le cn \lg n$ where $c = \max(T(2)/2, 1)$. Then,

$$T(n) = 2T(\lfloor n/2 \rfloor) + n \le 2c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor) + n$$

$$\le cn \lg(n/2) + n = cn(\lg(n) - 1) + n = cn(\lg(n) - 1 + \frac{1}{c}) \le cn \lg(n)$$

And so, $T(n) \in O(n \lg(n))$.

Now, inductively assume that $T(n) \ge c'n \lg(n)$ where $c' = \min(1/3, T(2)/2)$.

$$T(n) = 2T(\lfloor n/2 \rfloor) + n \ge 2c' \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor) + n \ge c'(n-1) \lg((n-1)/2) + n$$

$$= c'(n-1)(\lg(n) - 1 - \lg(n/(n-1))) + n$$

$$= c'n(\lg(n) - 1 - \lg(n/(n-1)) + \frac{1}{c'}) - c'(\lg(n) - 1 - \lg(n/(n-1)))$$

$$\ge c'n(\lg(n) - 2 + \frac{1}{c'} - \frac{(\lg(n-1)-1)}{n}) \ge c'n(\lg(n) - 3 + \frac{1}{c'}) \ge c'n \lg(n)$$

So, $T(n) \in \Omega(n)$. Together with the first part of this problem, we get that $T(n) \in \Theta(n)$.

Exercise 4.3-4

We'll use the induction hypothesis $T(n) \le 2n \log n + 1$. First observe that this means $T(1) = 1$, so the base case is satisfied. Then we have

$$\begin{aligned}
T(n) &= 2T(\lfloor n/2 \rfloor) + n \\
&\le 2((2n/2) \log(n/2) + 1) + n \\
&= 2n \log(n) - 2n \log 2 + 2 + n \\
&= 2n \log(n) + 1 + n + 1 - 2n \\
&\le 2n \log(n) + 1.
\end{aligned}$$

Exercise 4.3-5

If n is even, then that step of the induction is the same as the "inexact" recurrence for merge sort. So, suppose that n is odd, then, the recurrence is
$T(n) = T((n+1)/2) + T((n-1)/2) + \Theta(n)$. However, shifting the argument in $n \lg(n)$ by a half will only change the value of the function by at most $\frac{1}{2} \cdot \frac{d}{dn}(n \lg(n)) = \frac{\lg(n)}{2} + 1$, but this is $o(n)$ and so will be absorbed into the $\Theta(n)$ term.

Exercise 4.3-6

Choose n_1 such that $n \geq n_1$ implies $n/2 + 17 \leq 3n/4$. We'll find c and d such that $T(n) \leq cn \log n - d$.

$$
\begin{aligned}
T(n) &= 2T(\lfloor n/2 \rfloor + 17) + n \\
&\leq 2(c(n/2 + 17) \log(n/2 + 17) - d) + n \\
&\leq cn \log(n/2 + 17) + 17c \log(n/2 + 17) - 2d + n \\
&\leq cn \log(3n/4) + 17c \log(3n/4) - 2d + n \\
&= cn \log n - d + cn \log(3/4) + 17c \log(3n/4) - d + n.
\end{aligned}
$$

Take $c = -2/\log(3/4)$ and $d = 34$. Then we have $T(n) \leq cn \log n - d + 17c \log(n) - n$. Since $\log(n) = o(n)$, there exists n_2 such that $n \geq n_2$ implies $n \geq 17c \log(n)$. Letting $n_0 = \max\{n_1, n_2\}$ we have that $n \geq n_0$ implies $T(n) \leq cn \log n - d$. Therefore $T(n) = O(n \log n)$.

Exercise 4.3-7

We first try the substitution proof $T(n) \leq cn^{\log_3 4}$.

$$
T(n) = 4T(n/3) + n \leq 4c(n/3)^{\log_3 4} + n = 4cn^{\log_3 4} + n
$$

This clearly will not be $\leq cn^{\log_3 4}$ as required.

Now, suppose instead that we make our inductive hypothesis $T(n) \leq cn^{\log_3 4} - 3n$.

$$
T(n) = 4T(n/3) + n \leq 4(c(n/3)^{\log_3 4} - n) + n = cn^{\log_3 4} - 4n + n = cn^{\log_3 4} - 3n
$$

as desired.

Exercise 4.3-8

Suppose we want to use substitution to show $T(n) \leq cn^2$ for some c. Then we have

$$
\begin{aligned}
T(n) &= 4T(n/2) + n \\
&\leq 4(c(n/2)^2) + n \\
&= cn^2 + n,
\end{aligned}
$$

which fails to be less than cn^2 for any $c > 0$. Next we'll attempt to show $T(n) \leq cn^2 - n$.

$$
\begin{aligned}
T(n) &= 4T(n/2) + n \\
&\leq 4(c(n/2)^2 - n) + n \\
&= cn^2 - 4cn + n \\
&\leq cn^2
\end{aligned}
$$

provided that $c \geq 1/4$.

Exercise 4.3-9

Consider n of the form 2^k. Then, the recurrence becomes

$$T(2^k) = 3T(2^{(}k/2)) + k$$

We define $S(k) = T(2^k)$. So,

$$S(k) = 3S(k/2) + k$$

We use the inductive hypothesis $S(k) \leq (S(1) + 2)k^{\log_2 3} - 2k$

$$S(k) = 3S(k/2) + k \leq 3(S(1) + 2)(k/2)^{\log_2 3} - 3k + k = (S(1) + 2)k^{\log_2 3} - 2k$$

as desired. Similarly, we show that $S(k) \geq (S(1) + 2)k^{\log_2 3} - 2k$

$$S(k) = 3S(k/2) + k \geq (S(1) + 2)k^{\log_2 3} - 2k$$

So, we have that $S(k) = (S(1) + 2)k^{\log_2 3} - 2k$. Translating this back to T, $T(2^k) = (T(2) + 2)k^{\log_2 3} - 2k$. So, $T(n) = (T(2) + 2)(\lg(n))^{\log_2 3} - 2\lg(n)$.

Exercise 4.4-1

Since in a recursion tree, the depth of the tree will be $\lg(n)$, and the number of nodes that are i levels below the root is 3^i. This means that we would estimate that the runtime is $\sum_{i=0}^{\lg(n)} 3^i(n/2^i) = n \sum_{i=0}^{\lg(n)} (3/2)^i = n\frac{(3/2)^{\lg(n)} - 1}{.5} \approx n^{\lg(3)}$. We can see this by performing a substutiton $T(n) \leq cn^{\lg(3)} - 2n$. Then, we have that

$$\begin{aligned} T(n) &= 3T(n\lfloor n/2 \rfloor) + n \\ &\leq 3cn^{\lg(3)}/2^{\lg(3)} - 3n + n \\ &= cn^{\lg(3)} - n \end{aligned}$$

So, we have that $T(n) \in O(n^{\lg(3)})$.

Exercise 4.4-2

As we construct the tree, there is only one node at depth d, and its weight is $n^2/(2^d)^2$. Since the tree has $\log(n)$ levels, we guess that the solution is roughly $\sum_{i=0}^{\log n} \frac{n^2}{4^i} = O(n^2)$. Next we use the substitution method to verify that $T(n) \leq cn^2$.

$$\begin{aligned} T(n) &= T(n/2) + n^2 \\ &\leq c(n/2)^2 + n^2 \\ &= (\frac{c}{4} + 1)n^2 \\ &\leq cn^2 \end{aligned}$$

9

provided that $c \geq 4/3$.

Exercise 4.4-3

Again, we notice that the depth of the tree is around $\lg(n)$, and there are 4^i vertices on the ith level below the root, so, we have that our guess is n^2. We show this fact by the substitution method. We show that $T(n) \leq cn^2 - 6n$

$$T(n) = 4T(n/2 + 2) + n$$
$$\leq 4c(n^2/4 + 2n + 4 - 3n - 12) + n$$
$$= cn^2 - 4cn - 32c + n$$

Which will be $\leq cn^2 - 6$ so long as we have $-4c + 1 \leq -6$ and $c \geq 0$. These can both be satisfied so long as $c \geq \frac{7}{4}$.

Exercise 4.4-4

The recursion tree looks like a complete binary tree of height n with cost 1 at each node. Thus we guess that the solution is $O(2^n)$. We'll use the substitution method to verify that $T(n) \leq 2^n - 1$.

$$T(n) = 2T(n-1) + 1$$
$$\leq 2(2^{n-1} - 1) + 1$$
$$= 2^n - 1.$$

Exercise 4.4-5

The recursion tree looks like one long branch and off of it, branches that jump all the way down to half. This seems like a pretty full tree, we'll we'll guess that the runtime is $O(n^2)$. To see this by the substitution method, we try to show that $T(n) \leq 2^n$ by the substitution method.

$$T(n) = T(n-1) + T(n/2) + n$$
$$\leq 2^{n-1} + \sqrt{2^n} + n$$
$$\leq 2^n$$

Now, to justify that this is actually a pretty tight bound, we'll show that we can't have any polynomial upper bound. That is, if we have that $T(n) \leq cn^k$ then, when we substitute into the recurrence, we get that the new coefficient for n^k can be as high as $c(1 + \frac{1}{2^k})$ which is bigger than c regardless of how we choose c.

Exercise 4.4-6

Examining the tree in figure 4.6 we observe that the cost at each level of the tree is exactly cn. To find a lower bound on the cost of the algorithm, we need a lower bound on the height of the tree. The shortest simple path from root to leaf is found by following the left child at each node. Since we divide by 3 at each step, we see that this path has length $\log_3 n$, so the cost of the algorithm is $cn(\log_3 n + 1) \geq cn \log_3 n = \frac{c}{\log 3} n \log n = \Omega(n \log n)$.

Exercise 4.4-7

Here is an example for $n = 4$.
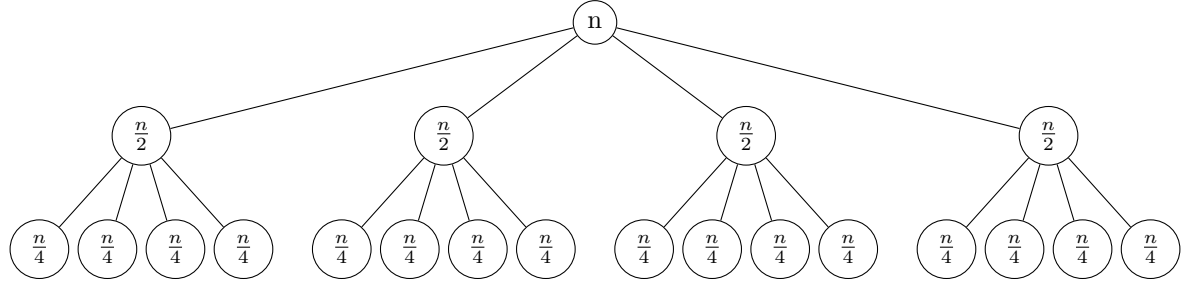
We can see by an easy substitution that the answer is $\Theta(n^2)$. Suppose that $T(n) \leq c'n^2$ then

$$T(n) = 4T(\lfloor n/2 \rfloor) + cn$$
$$\leq c'n^2 + cn$$

which is $\leq c'n^2$ whenever we have that $c' + \frac{c}{n} \leq 1$, which, for large enough n is true so long as $c' < 1$. We can do a similar thing to show that it is also bounded below by n^2.

Exercise 4.4-8

$$T(a) + cn$$

$$\vert$$

$$T(a) + c(n - a)$$

$$\vert$$

$$T(a) + c(n - 2a)$$

$$\vdots$$

$$T(1)$$

Since each node of the recursion tree has only one child, the cost at each level is just the cost of the node. Moreover, there are $\lceil n/a \rceil$ levels in the tree. Summing the cost at each level we see that the total cost of the algorithm is

$$\sum_{i=0}^{\lceil n/a \rceil - 1} T(a) + c(n - ia) = \lceil n/a \rceil T(a) + c\lceil n/a \rceil n - ca\frac{\lceil n/a \rceil (\lceil n/a \rceil - 1)}{2}.$$

To compute the asymptotoics we can assume n is divisible by a and ignore the ceiling functions. Then this becomes

$$\frac{c}{2a}n^2 + (T(a)/a + c/2)n = \Theta(n^2).$$

Exercise 4.4-9

Since the sum of the sizes of the two children is $\alpha n + (1 - \alpha)n = n$, we would guess that this behaves the same way as in the analysis of merge sort, so, we'll try to show that it has a solution that is $T(n) \le c'n\lg(n) - cn$.

$$
\begin{aligned}
T(n) &= T(\alpha n) + T((1 - \alpha)n) + cn \\
&\le c'\alpha n(\lg(\alpha) + \lg(n)) - c\alpha n + c'(1 - \alpha)n(\lg(1 - \alpha) + \lg(n)) - c(1 - \alpha)n + cn \\
&= c'n\lg(n) + c'n(\alpha\lg(\alpha) + (1 - \alpha)\lg(1 - \alpha)) \\
&\le c'n\lg(n) - c'n
\end{aligned}
$$

Where we use the fact that $x\lg(x)$ is convex for the last inequality. This then completes the induction if we have $c' \ge c$ which is easy to do.

Exercise 4.5-1

a. $\Theta(\sqrt{n})$

b. $\Theta(\sqrt{n}\lg(n))$

c. $\Theta(n)$

d. $\Theta(n^2)$

Exercise 4.5-2

Recall that Strassen's algorithm has running time $\Theta(n^{\lg 7})$. We'll choose $a = 48$. This is the largest integer such that $\log_4(a) < \lg 7$. Moreover, $2 < \log_4(48)$ so there exists $\epsilon > 0$ such that $n^2 < n^{\log_4(48) - \epsilon}$. By case 1 of the Master theorem, $T(n) = \Theta(n^{\log_4(48)})$ which is asymptotically better than $\Theta(n^{\lg 7})$.

Exercise 4.5-3

Applying the method with $a = 1, b = 2$, we have that $\Theta(n^{\log_2 1}) = \Theta(1)$. So, we are in the second case, so, we have a final result of $\Theta(n^{\log_2 1} \lg(n)) = \Theta(\lg(n))$.

Exercise 4.5-4

The master method cannot be applied here. Observe that $\log_b a = \log_2 4 = 2$ and $f(n) = n^2 \lg n$. It is clear that cases 1 and 2 do not apply. Furthermore, although f is asymptotically larger than n^2, it is not polynomially larger, so case 3 does not apply either. We'll show $T(n) = O(n^2 \lg^2 n)$. To do this, we'll prove inductively that $T(n) \le n^2 \lg^2 n$.

$$
\begin{aligned}
T(n) &= 4T(n/2) + n^2 \lg n \\
&\le 4((n/2)^2 \lg^2(n/2)) + n^2 \lg n \\
&= n^2 (\lg n - \lg 2)^2 + n^2 \lg n \\
&= n^2 \lg^2 n - n^2 (2 \lg n - 1 - \lg n) \\
&= n^2 \lg^2 n - n^2 (\lg n - 1) \\
&\le n^2 \lg^2 n
\end{aligned}
$$

provided that $n \ge 2$.

Exercise 4.5-5

Let $\epsilon = a = 1, b = 3$, and $f = 3n + 2^{3n}\chi_{\{2^i : i \in \mathbb{N}\}}$ where χ_A is the indicator function of the set A. Then, we have that for any number n which is three times a power of 2, we know that

$$
f(n) = 3n < 2^n + n = f(n/3)
$$

And so, it fails the regularity condition, even though $f \in \Omega(n) = \Omega(n^{\log_b(a)+\epsilon})$.

Exercise 4.6-1

n_j is obtained by shifting the base b representation j positions to the right, and adding 1 if any of the j least significant positions are non-zero.

Exercise 4.6-2

Assume that $n = b^m$ for some m. Let c_1 and c_2 be such that $c_2 n^{\log_b a} \lg^k n \le f(n) \le c_1 n^{\log_b a} \lg^k n$. We'll first prove by strong induction that $T(n) \le n^{\log_b a} \lg^{k+1} n - dn^{\log_b a} \lg^k n$ for some choice of $d \ge 0$. Equivalently, that $T(b^m) \le a^m \ln^{k+1}(b^m) -$

$da^m \lg^k(b^m)$.

$$T(b^m) = aT(b^m/b) + f(b^m)$$
$$\leq a(a^{m-1}\lg^{k+1}(b^{m-1}) - da^{m-1}\lg^k b^{m-1}) + c_1 a^m \lg^k(b^m)$$
$$=\leq a^m \lg^{k+1}(b^{m-1}) - da^m \lg^k b^{m-1} + c_1 a^m \lg^k(b^m)$$
$$=\leq a^m[\lg(b^m) - \lg b]^{k+1} - da^m[\lg b^m - \lg b]^k + c_1 a^m \lg^k(b^m)$$
$$= a^m \lg^{k+1}(b^m) - da^m d\lg^k b^m$$
$$\ldots - a^m\left(d\sum_{r=0}^{k-1}\binom{k}{r}\lg^r(b^m)(-\lg b)^{k-r} + \sum_{r=0}^{k}\binom{k+1}{r}\lg^r(b^m)(-\lg(b))^{k+1-r} + c_1\lg^k(b^m)\right)$$
$$= a^m \lg^{k+1}(b^m) - da^m \lg^k b^m$$
$$\ldots - a^m\left((c_1 - k\lg b)\lg^k(b^m) + \sum_{r=0}^{k-1}\binom{k+1}{r}\lg^r(b^m)(-\lg(b))^{k+1-r} + d\sum_{r=0}^{k-1}\binom{k}{r}\lg^r(b^m)(-\lg b)^{k-r}\right)$$
$$\leq a^m \lg^{k+1}(b^m) - da^m \lg^k b^m$$

for $c_1 \geq k\lg b$. Thus $T(n) = O(n^{\log_b a}\lg^{k+1} n)$. A similar analysis shows $T(n) = \Omega(n^{\log_b a}\lg^{k+1} n)$.

Exercise 4.6-3

Suppose that f satisfies the regularity condition, we want that $\exists \epsilon, d, k, \forall n \geq k$, we have $f(n) \geq dn^{\log_b a + \epsilon}$. By the regularity condition, we have that for sufficiently large n, $af(n/b) \leq cf(n)$. In particular, it is true for all $n \geq bk$. Let this be our k from above, also, $\epsilon = -\log_b(c)$. Finally let d be the largest value of $f(n)/n^{\log_b(a)+\epsilon}$ between bk and b^2k. Then, we will prove by induction on the highest i so that $b^i k$ is less than n that for every $n \geq k$, $f(n) \geq dn^{\log_b a + \epsilon}$. By our definition of d, we have it is true for $i = 1$. So, suppose we have $b^{i-1}k < n \leq b^i k$. Then, by regularity and the inductive hypothesis, $cf(n) \geq af(n/b) \geq ad\left(\frac{n}{b}\right)^{\log_b(a)+\epsilon}$. Solving for $f(n)$, we have

$$f(n) \geq \frac{ad}{c}\left(\frac{n}{b}\right)^{\log_b a/c} = \frac{a/c}{b^{\log_b(a/c)}}dn^{\log_b(a)+\epsilon} = dn^{\log_b(a)+\epsilon}$$

Completing the induction.

Problem 4-1

a. By Master Theorem, $T(n) \in \Theta(n^4)$

b. By Master Theorem, $T(n) \in \Theta(n)$

c. By Master Theorem, $T(n) \in \Theta(n^2\lg(n))$

d. By Master Theorem, $T(n) \in \Theta(n^2)$

e. By Master Theorem, $T(n) \in \Theta(n^{\lg(7)})$

f. By Master Theorem, $T(n) \in \Theta(n^{1/2} \lg(n))$

g. Let $d = m \mod 2$, we can easily see that the exact value of $T(n)$ is

$$\sum_{j=1}^{j=n/2} (2j+d)^2 = \sum_{j=1}^{n/2} 4j^2 + 4jd + d^2 = \frac{n(n+2)(n+1)}{6} + \frac{n(n+2)d}{2} + \frac{d^2 n}{2}$$

This has a leading term of $n^3/6$, and so $T(n) \in \Theta(n^3)$

Problem 4-2

a. 1. $T(n) = T(n/2) + \Theta(1)$. Solving the recursion we have $T(N) = \Theta(\lg N)$.
   2. $T(n) = T(n/2) + \Theta(N)$. Solving the recursion we have $T(N) = \Theta(N \lg N)$.
   3. $T(n) = T(n/2) + \Theta(n/2)$. Solving the recursion we have $T(N) = \Theta(N)$.

b. 1. $T(n) = 2T(n/2) + cn$. Solving the recursion we have $T(N) = \Theta(N \lg N)$.
   2. $T(n) = 2T(n/2) + cn + 2\Theta(N)$. Solving the recursion we have $T(N) = \Theta(N \lg N) + \Theta(N^2) = \Theta(N^2)$.
   3. $T(n) = 2T(n/2) + cn + 2c'n/2$. Solving the recursion we have $T(N) = \Theta(N \ln N)$.

Problem 4-3

a. By Master Theorem, $T(n) \in \Theta(n^{\log_3(4)})$

b. We first show by substitution that $T(n) \le n \lg(n)$.

$$T(n) = 3T(n/3) + n/\lg(n) \le cn \lg(n) - cn \lg(3) + n/\lg(n) = cn \lg(n) + n(\frac{1}{\lg(n)} - c \lg(3)) \le cn \lg(n)$$

now, we show that $T(n) \ge cn^{1-\epsilon}$ for every $\epsilon > 0$.

$$T(n) = 3T(n/3) + n/\lg(n) \ge 3c/3^{1-\epsilon} n^{1-\epsilon} + n/\lg(n) = 3^\epsilon cn^{1-\epsilon} + n/\lg(n)$$

showing that this is $\le cn^{1-\epsilon}$ is the same as showing

$$3^\epsilon + n^\epsilon/(c \lg(n)) \ge 1$$

Since $\lg(n) \in o(n^\epsilon)$ this inequality holds. So, we have that The function is soft Theta of n, see problem 3-5.

c. By Master Theorem, $T(n) \in \Theta(n^{2.5})$

d. it is $\Theta(n \lg(n))$. The subtraction occurring inside the argument to T won't change the asymptotics of the solution, that is, for large n the division is so much more of a change than the subtraction that it is the only part that matters. once we drop that subtraction, the solution comes by the master theorem.

e. By the same reasoning as part 2, the function is $O(n \lg(n))$ and $\Omega(n^{1-\epsilon})$ for every ϵ and so is soft theta of n, see problem 3-5.

f. We will show that this is $O(n)$ by substitution. We want that $T(k) \le ck$ for $k < n$, then,

$$T(n) = T(n/2) + T(n/4) + T(n/8) + n \le \frac{7}{8}cn + n$$

So, this is $\le cn$ so long as $\frac{7}{8}c + 1 \le c$ which happens whenever $c \ge 8$.

g. Recall that χ_A denotes the indicator function of A, then, we see that the sum is

$$T(0) + \sum_{j=1}^{n} \frac{1}{j} = T(0) + \int_{1}^{n+1} \sum_{j=1}^{n+1} \frac{\chi_{(j,j+1)}(x)}{j} dx$$

However, since $\frac{1}{x}$ is monatonically decreasing, we have that for every $i \in \mathbb{Z}^+$,

$$\sup_{x \in (i,i+1)} \sum_{j=1}^{n+1} \frac{\chi_{(j,j+1)}(x)}{j} - \frac{1}{x} = \frac{1}{i} - \frac{1}{i+1} = \frac{1}{i(i+1)}$$

So, our expression for $T(n)$ becomes

$$T(N) = T(0) + \int_{1}^{n+1} \left(\frac{1}{x} + O(\frac{1}{\lfloor x \rfloor (\lfloor x \rfloor + 1)}) \right) dx$$

We deal with the error term by first chopping out the constant amount between 1 and 2 and then bound the error term by $O(\frac{1}{x(x-1)})$ which has an anti-derivative (by method of partial fractions) that is $O(\frac{1}{n})$. so,

$$T(N) = \int_{1}^{n+1} \frac{dx}{x} + O(\frac{1}{n}) = \lg(n) + T(0) + \frac{1}{2} + O(\frac{1}{n})$$

This gets us our final answer of $T(n) \in \Theta(\lg(n))$

h. we see that we explicity have

$$T(n) = T(0) + \sum_{j=1}^{n} \lg(j) = T(0) + \int_{1}^{n+1} \sum_{j=1}^{n+1} \chi_{(j,j+1)}(x) \lg(j) dx$$

16

Similarly to above, we will relate this sum to the integral of $\lg(x)$.

$$\sup_{x \in (i,i+1)} \left| \sum_{j=1}^{n+1} \chi_{(j,j+1)}(x) \lg(j) - \lg(x) \right| = \lg(j+1) - \lg(j) = \lg\left(\frac{j+1}{j}\right)$$

So,

$$T(n) \leq \int_i^n \lg(x+2) + \lg(x) - \lg(x+1) dx = (1 + O(\frac{1}{\lg(n)}))\Theta(n \lg(n))$$

i. See the approach used in the previous two parts, we will get $T(n) \in \Theta(li(n)) = \Theta(\frac{n}{\lg(n)})$

j. Let i be the smallest i so that $n^{\frac{1}{2^i}} < 2$. We recall from a previous problem (3-6.e) that this is $\lg(\lg(n))$ Expanding the recurrence, we have that it is

$$T(n) = n^{1-\frac{1}{2^i}} T(2) + n + n \sum_{j=1}^{i} 1 \in \Theta(n \lg(\lg(n)))$$

Problem 4-4

a. Recall that $F_0 = 0$, $F_1 = 1$, and $F_i = F_{i-1} + F_{i-2}$ for $i \geq 2$. Then we have

$$\mathcal{F}(z) = \sum_{i=0}^{\infty} F_i z^i$$

$$= F_0 + F_1 z + \sum_{i=2}^{\infty} (F_{i-1} + F_{i-2}) z^i$$

$$= z + z \sum_{i=2}^{\infty} F_{i-1} z^{i-1} + z^2 \sum_{i=2}^{\infty} F_{i-2} z^{i-2}$$

$$= z + z \sum_{i=1}^{\infty} F_i z^i + z^2 \sum_{i=0}^{\infty} F_i z^i$$

$$= z + z\mathcal{F}(z) + z^2 \mathcal{F}(z).$$

b. Manipulating the equation given in part (a) we have $\mathcal{F}(z) - z\mathcal{F}(z) - z^2 \mathcal{F}(z) = z$, so factoring and dividing gives

$$\mathcal{F}(z) = \frac{z}{1 - z - z^2}.$$

Factoring the denominator with the quadratic formula shows $1 - z - z^2 = (1 - \phi z)(1 - \hat{\phi} z)$, and the final equality comes from a partial fraction decomposition.

17

c. From part (b) and our knowledge of geometric series we have

$$\mathcal{F}(z) = \frac{1}{\sqrt{5}} \left(\frac{1}{1 - \phi z} - \frac{1}{1 - \hat{\phi} z} \right)$$

$$= \frac{1}{\sqrt{5}} \left(\sum_{i=0}^{\infty} (\phi z)^i - \sum_{i=0}^{\infty} (\hat{\phi} z)^i \right)$$

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

d. From the definition of the generating function, F_i is the coefficient of z^i in $\mathcal{F}(z)$. By part (c) this is given by $\frac{1}{\sqrt{5}} (\phi^i - \hat{\phi}^i)$. Since $|\hat{\phi}| < 1$ we must have $|\frac{\hat{\phi}^i}{\sqrt{5}}| < |\frac{\hat{\phi}}{\sqrt{5}}| < \frac{1}{2}$. Finally, since the Fibonacci numbers are integers we see that the exact solution must be the approximated solution $\frac{\phi^i}{\sqrt{5}}$ rounded to the nearest integer.

Problem 4-5

a. The strategy for the bad chips is to always say that other bad chips are good and other good chips are bad. This mirrors the strategy used by the good chips, and so, it would be impossible to distinguish

b. Arbitrarily pair up the chips. Look only at the pairs for which both chips said the other was good. Since we have at least half of the chips being good, we know that there will be at least one such pair which claims the other is good. We also know that at least half of the pairs which claim both are good are actually good. Then, just arbitrarily pick a chip from each pair and let these be the chips that make up the sub-instance of the problem

c. Once we have identified a single good chip, we can just use it to query every other chip. The recurrence from before for the number of tests to find a good chip was

$$T(n) \leq T(n/2) + n/2$$

This has solution $\Theta(n)$ by the Master Theorem. So, we have the problem can be solved in $O(n)$ pairwise tests. Since we also necessarily need to look at at least half of the chips, we know that the problem is also $\Omega(n)$.

Problem 4-6

a. If an array A is Monge then trivially it must satisfy the inequality by taking $k = i + 1$ and $l = j + 1$. Now suppose $A[i, j] + A[i + 1, j + 1] \leq A[i, j + 1] + A[i + 1, j]$. We'll use induction on rows and columns to show that the array

is Monge. The base cases are each covered by the given inequality. Now fix i and j, let $r \geq 1$, and suppose that $A[i,j] + A[i+1, j+r] \leq A[i, j+r] + A[i+1, j]$. By applying the induction hypothesis and given inequality we have

$$
\begin{aligned}
A[i,j] + A[i+1, j+r+1] &\leq A[i, j+r] + A[i+1, j] - A[i+1, j+r] \\
&\quad + A[i, j+r+1] + A[i+1, j+r] - A[i, j+r] \\
&= A[i+1, j] + A[i, j+r+1]
\end{aligned}
$$

so it follows that we can extend columns and preserve the Monge property. Next we induct on rows. Suppose that $A[i,j] + A[k,l] \leq A[i,l] + A[k,j]$. Then we have

$$
\begin{aligned}
A[i,j] + A[k+1, l] &\leq A[i,l] + A[k,j] - A[k,l] + A[k+1, l] && \text{by assumption} \\
&\leq A[i,l] + A[k,j] - A[k,l] + A[k,l] + A[k+1, l-1] - A[k, l-1] && \text{by given inequality} \\
&= A[i,l] + (A[k,j] + A[k+1, l-1]) - A[k, l-1] \\
&\leq A[i,l] + A[k, l-1] + A[k+1, j] - A[k, l-1] && \text{by row proof} \\
&= A[i,l] + A[k+1, j].
\end{aligned}
$$

b. Change the 7 to a 5.

c. Suppose that there exist i and k such that $i < k$ but $f(i) > f(k)$. Since A is Monge we must have $A[i, f(k)] + A[k, f(i)] \leq A[k, f(k)] + A[i, f(i)]$. Since $f(i)$ gives the position of the leftmost minimum element in row i, this implies that $A[i, f(k)] > A[i, f(i)]$. Moreover, $A[k, f(k)] \leq A[k, f(i)]$. Combining these with the Monge inequality implies $A[i, f(i)] + A[k, f(i)] < A[k, f(i)] + A[i, f(i)]$, which is impossible since the two sides are equal. Therefore no such i and k can exist.

d. Linearly scan row 1 indices 1 through $f(2)$ for the minimum element of row 1 and record as $f(1)$. Next linearly scan indices $f(2)$ through $f(4)$ of row 3 for the minimum element of row 3. In general, we need only scan indices $f(2k)$ through $f(2k+2)$ of row $2k+1$ to find the leftmost minimum element of row $2k+1$. If m is odd, we'll need to search indices $f(m-1)$ through n to find the leftmost minimum in row m. By part (c) we know that the indices of the leftmost minimum elements are increasing, so we are guaranteed to find the desired minimum from among the indices scanned. An element of column j will be scanned $N_j + 1$ times, where N_j is the number of i such that $f(i) = j$. Since $\sum_{j=1}^{n} N_j = n$, the total number of comparisons is $m + n$, giving a running time of $O(m+n)$.

e. Let $T(m, n)$ denote the running time of the algorithm applied to an m by n matrix. $T(m, n) = T(m/2, n) + c(m+n)$ for some constant c. We'll show

$T(m, n) \leq c(m + n \log m) - 2cm.$

$$\begin{aligned}
T(m, n) &= T(m/2, n) + c(m + n) \\
&\leq c(m/2 + n \log(m/2)) - 2cm + c(m + n) \\
&= c(m/2 + n \log m) - cn + cn - cm \\
&\leq c(m + n \log m) - cm
\end{aligned}$$

so by induction we have $T(m, n) = O(m + n \log m)$.

Chapter 5

Michelle Bodnar, Andrew Lohr

February 16, 2018

Exercise 5.1-1

We may have been presented the candidates in increasing order of goodness. This would mean that we can apply transitivity to determine our preference between any two candidates

Exercise 5.1-2

Algorithm 1 RANDOM(a,b)

1: $n = \lceil \lg(b - a + 1) \rceil$
2: Initialize an array A of length n
3: **while** true **do**
4:     **for** $i = 1$ to n **do**
5:         $A[i] = RANDOM(0, 1)$
6:     **end for**
7:     **if** A holds the binary representation of one of the numbers in a through b **then**
8:         **return** number represented by A
9:     **end if**
10: **end while**

Each iteration of the while loop takes n time to run. The probability that the while loop stops on a given iteration is $(b - a + 1)/2^n$. Thus the expected running time is the expected number of times run times n. This is given by:

$$n \sum_{i \geq 1} i \left(1 - \frac{b - a + 1}{2^n}\right)^{i-1} \left(\frac{b - a + 1}{2^n}\right) = n \left(\frac{b - a + 1}{2^n}\right) \left(\frac{2^n}{b - a + 1}\right)^2 = n \frac{2^n}{b - a + 1} = O(\lg(b - a)).$$

Exercise 5.1-3

Clearly since a and b are IID, the probability this algorithm returns one is equal to the probability it returns 0. Also, since there is a constant positive probability $(2p(p - 1))$ that the algorithm returns on each iteration of the for

```
 1: for all eternity do
 2:     a = BiasedRandom
 3:     b = BiasedRandom
 4:     if a > b then
 5:         return 1
 6:     end if
 7:     if a < b then
 8:         return 0
 9:     end if
10: end for
```

loop. This program will expect to go through the loop a number of times equal to:

$$\sum_{j=0}^{\infty} j(1 - 2p(p-1))^j (2p(p-1)) = \frac{2p(p-1)(1 - 2p(p-1))}{(2p(p-1))^2} = \frac{1 - 2p(p-1)}{2p(p-1)}$$

Note that the formula used for the sum of $j\alpha^j$ can be obtained by differentiawting both sides of the geometric sum formula for α^j with respect to α

Exercise 5.2-1

You will hire exactly one time if the best candidate is presented first. There are $(n-1)!$ orderings with the best candidate first, so, it is with probability $\frac{(n-1)!}{n!} = \frac{1}{n}$ that you only hire once. You will hire exactly n times if the candidates are presented in increasing order. This fixes the ordering to a single one, and so this will occur with probability $\frac{1}{n!}$.

Exercise 5.2-2

Since the first candidate is always hired, we need to compute the probability that that exactly one additional candidate is hired. Since we view the candidate ranking as reading an random permutation, this is equivalent to the probability that a random permutation is a decreasing sequence followed by an increase, followed by another decreasing sequence. Such a permutation can be thought of as a partition of $[n]$ into 2 parts. One of size k and the other of size $n-k$, where $1 \leq k \leq n-1$. For each such partition, we obtain a permutation with a single increase by ordering the numbers each partition in decreasing order, then concatenating these sequences. The only thing that can go wrong is if the numbers n through $n-k+1$ are in the first partition. Thus there are $\binom{n}{k} - 1$ permutations which correspond to hiring the second and final person on step $k+1$. Summing, we see that the probability you hire exactly twice is

$$\frac{\sum_{k=1}^{n-1} \binom{n}{k} - 1}{n!} = \frac{2^n - 2 - (n-1)}{n!} = \frac{2^n - n - 1}{n!}.$$

2

Exercise 5.2-3

Let X_j be the indicator of a dice coming up j. So, the expected value of a single dice roll X is

$$E[X] = \sum_{j=1}^{6} j \Pr(X_j) = \frac{1}{6} \sum_{j=1}^{6} j$$

So, the sum of n dice has probability

$$E[nX] = nE[X] = \frac{n}{6} \sum_{j=1}^{6} j = \frac{n6(6+1)}{12} = 3.5n$$

Exercise 5.2-5

Let $X_{i,j}$ for $i < j$ be the indicator of $A[i] > A[j]$. Then, we have that the expected number of inversions is

$$E\left[\sum_{i<j} X_{i,j} \right] = \sum_{i<j} E[X_{i,j}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr(A[i] > A[j]) = \frac{1}{2} \sum_{i=1}^{n-1} n - i$$

$$= \frac{n(n-1)}{2} - \frac{n(n-1)}{4} = \frac{n(n-1)}{4}.$$

Exercise 5.2-4

Let X be the number of customers who get back their own hat and X_i be the indicator random variable that customer i gets his hat back. The probability that an individual gets his hat back is $\frac{1}{n}$. Then we have

$$E[X] = E\left[\sum_{i=1}^{n} X_i \right] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} \frac{1}{n} = 1.$$

Exercise 5.3-1

We modify the algorithm by unrolling the $i = 1$ case.

1: swap $A[1]$ with $A[\text{Random}(1,n)]$
2: **for** i from 2 to n **do**
3:     swap $A[i]$ with $A[\text{Random}(i,n)]$
4: **end for**

Modify the proof of the lemma by starting with $i = 2$ instead of $i = 1$. This entirely sidesteps the issue of talking about 0- permutations.

Exercise 5.3-2

The code does not do what he intends. Suppose $A = [1, 2, 3]$. If the algorithm worked as proposed, then with nonzero probability the algorithm should output $[3, 2, 1]$. On the first iteration we swap $A[1]$ with either $A[2]$ or $A[3]$. Since we want $[3, 2, 1]$ and will never again alter $A[1]$, we must necessarily swap with $A[3]$. Now the current array is $[3, 2, 1]$. On the second (and final) iteration, we have no choice but to swap $A[2]$ with $A[3]$, so the resulting array is $[3, 1, 2]$. Thus, the procedure cannot possibly be producing random non-identity permutations.

Exercise 5.3-3

Consider the case of $n = 3$ in running the algorithm, three IID choices will be made, and so you'll end up having 27 possible end states each with equal probability. There are $3! = 6$ possible orderings, these shuld appear equally often, but this can't happen because 6 does not divide 27

Exercise 5.3-4

Fix a position j and an index i. We'll show that the probability that $A[i]$ winds up in position j is $1/n$. The probability $B[j] = A[i]$ is the probability that $dest = j$, which is the probability that $i + offset$ or $i + offset - n$ is equal to j, which is $1/n$. This algorithm can't possibly return a random permutation because it doesn't change the relative positions of the elements; it merely cyclically permutes the whole permutation. For instance, suppose $A = [1, 2, 3]$. If $offset = 1$, $B = [3, 1, 2]$. If $offset = 2$, $B = [2, 3, 1]$. If $v = 3$, $B = [1, 2, 3]$. Thus, the algorithm will never produce $B = [1, 3, 2]$, so the resulting permutation cannot be uniformly random.

Exercise 5.3-5

Let $X_{i,j}$ be the event that $P[i] = P[j]$. Then, the event that all are unique is the complement of there being some pair that are equal, so, we must show that $\Pr(\cup_{i,j} X_{i,j}) \leq 1/n$. We start by applying a union bound

$$\Pr(\cup_{i<k} X_{i,j}) \leq \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr(X_{i,j}) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{n^3}$$

Where we use the fact that any two indices will be equal with probability equal to one over the size of the probability space being drawn from, which is n^3.

$$= \sum_{i=1}^{n-1} \frac{n-i}{n^3} = \frac{n(n-1)}{n^3} - \frac{n(n-1)}{2n^3} = \frac{n-1}{2n^2} < \frac{1}{n}$$

4

Exercise 5.3-6

After the completion of the for loop, check whether or not two or more priorities are identical. Suppose that k of them are identical. Then call $RANDOM(1, k^3)$ k times to introduce a relative ordering on the k identical entries. Some entries m may still be identical, so simply call $RANDOM(1, m^3)$ that many times to introduce a relative ordering on these identical entries. For example, if the first sequence of numbers we get is $1, 1, 1, 3, 4, 5, 5$, we'll need to order the 1's and the 5's. For the 1's, we'll call $RANDOM(1, 27)$ 3 times. Suppose it produces $23, 14, 23$. Then we'll call $RANDOM(1, 8)$ twice to sort the 23's. Suppose it gives $3, 5$. Then the relative ordering of $23, 14, 23$ becomes $2, 1, 3$, so the new relative ordering on the original array is $2, 1, 3, 4, 5, 6, 6$. Now we call $RANDOM(1, 8)$ twice to sort the 6's. If the first number is larger than the second, our final array will be $2, 1, 3, 4, 5, 7, 6$.

Exercise 5.3-7

We prove that it produces a random m subset by induction on m. It is obviously true if $m = 0$ as there is only one size m subset of $[n]$. Suppose S is a uniform $m - 1$ subset of $n - 1$, that is, $\forall j \in [n - 1], \Pr[j \in S] = \frac{m-1}{n-1}$. Then, if we let S' denote the returned set, suppose first $j \in [n - 1]$, $\Pr[j \in S'] = \Pr[j \in S] + \Pr[j \notin S \wedge i = j] = \frac{m-1}{n-1} + \Pr[j \notin S] \Pr[i = j] = \frac{m-1}{n-1} + \left(1 - \frac{m-1}{n-1}\right) \frac{1}{n} = \frac{n(m-1)+n-m}{(n-1)n} = \frac{nm-m}{(n-1)n} = \frac{m}{n}$. Since the constructed subset contains each of $[n - 1]$ with the correct probability, it must also contain n with the correct probability because the probabilities sum to 1.

Exercise 5.4-1

The probability that none of n people have the same birthday as you is $(1 - \frac{1}{365})^n = \frac{364^n}{365^n}$. This falls below zero when $n \geq \log_{\frac{364}{365}}(.5) \approx 252.6$ so, when $n = 253$. Since you are also a person in the room, we add one to get the final answer of 254.

The probability that k of the n people have july 4 as a birthday is $\binom{n}{k} \frac{364^{n-k}}{365}$. In particular, for $k = 0$ it is $\frac{364^n}{365^n}$ and for $k = 1$ it is $\frac{n364^{n-1}}{365^n}$. Adding these up and solving for n to have the sum drop less than a half, we get

$$\left(\frac{364}{365}\right)^n \left(1 + \frac{n}{364}\right) < .5$$

This is difficult to solve analytically, but because of the LHS's monotonicity, the answer can be found rather quickly by using a gallop search to be $n = 612$.

Exercise 5.4-2

We compute directly from the definition of expectation. Let X denote the

number of balls tossed until some bin contains two balls.

$$E[X] = \sum_{i=2}^{b+1} iP(X = i).$$

The probability that i tosses are required is the probability that the first $i - 1$ tosses go into unique bins, so we have

$$P(X = i) = 1 \cdot \left(\frac{b-1}{b}\right) \left(\frac{b-2}{b}\right) \cdots \left(\frac{b-i+2}{b}\right) \left(\frac{i-1}{b}\right).$$

Thus

$$E[X] = \sum_{i=2}^{b+1} i \left(\frac{b-1}{b}\right) \left(\frac{b-2}{b}\right) \cdots \left(\frac{b-i+2}{b}\right) \left(\frac{i-1}{b}\right)$$

$$= \sum_{i=2}^{b+1} i \left(1 - \frac{1}{b}\right) \left(1 - \frac{2}{b}\right) \cdots \left(1 - \frac{i-2}{b}\right) \left(\frac{i-1}{b}\right).$$

Exercise 5.4-3

Pairwise independence is sufficient. All the independence is used for is to show that $\Pr(b_i = r \wedge b_j = r) = \Pr(b_i = r)\Pr(b_j = r)$. This is a result of pairwise independence.

Exercise 5.4-4

We can compute "likely" in two ways. The probability that at least three people share the same birthday is 1 minus the probability that none share the same birthday, minus the probability that any number of pairs of people share the same birthday. Let $n = 365$ denote the number of days in a year and k be the number of people at the party. As computed earlier in the section, we have

$$P(\text{all unique birthdays}) = 1 \cdot \left(\frac{n-1}{n}\right) \left(\frac{n-2}{n}\right) \cdots \left(\frac{n-k+1}{n}\right) \leq e^{-k(k-1)/2n}.$$

Next we compute the probability that exactly i pairs of people share a birthday. There are $\binom{k}{2}\binom{k-2}{2}\cdots\binom{k-2i+2}{2}$ ways to choose an ordered collection of i pairs from the k people, $\binom{n}{i}$ ways to select the set of birthdays the pairs will share, and the probability that any such ordered subset has these precise birthdays is $(1/n^2)^i$. Multiplying by the probability that the rest of the birthdays

are different and unique we have

$$P(\text{exactly } i \text{ pairs of people same}) = \frac{\binom{k}{2}\binom{k-2}{2}\cdots\binom{k-2i+2}{2}\binom{n}{i}}{n^{2i}}\left(\frac{n-i}{n}\right)\left(\frac{n-i-1}{n}\right)\cdots\left(\frac{n-k+i+1}{n}\right)$$

$$\leq \frac{k!\binom{n}{i}}{2^i(k-2i)!n^{2i}}\left(1-\frac{i}{n}\right)\left(1-\frac{i+1}{n}\right)\cdots\left(1-\frac{k-i-1}{n}\right)$$

$$\leq \frac{k!\binom{n}{i}}{2^i(k-2i)!n^{2i}}e^{-(k-1)(k-2i-1)/2n}$$

Thus,

$$P(\geq 3 \text{ people share a birthday}) \leq 1-e^{-k(k-1)/2n}-\frac{k!\binom{n}{i}}{2^i(k-2i)!n^{2i}}e^{-(k-1)(k-2i-1)/2n}.$$

This is pretty messy, even with the simplifying inequality $1 + x \leq e^x$, so we'll do another analysis, this time on expectation. We'll determine the value of k required such that the expected number of triples (i, j, m) where person i, person j, and person m share a birthday is at least 1. Let X_{ijm} be the indicator variable that this triple of people share a birthday and X denote the total number of triples of birthday-sharers. Then we have

$$E[X] = \sum_{\text{distinct triples } (i,j,m)} E[X_{ijm}] = \binom{k}{3}\frac{1}{n^2}.$$

To make $E[X]$ exceed 1 we need to find the smallest k such that $k(k-1)(k-2) \geq 6(365)^2$, which happens when $k = 94$.

Exercise 5.4-5

Since to be a k-permutation, we need that no letter appears repeated, it is equivalent to the birthday problem with k people and n days. So, this probability is given on the top of page 132 to be:

$$\prod_{i=1}^{k-1}\left(1-\frac{i}{n}\right)$$

Problem 5.4-6

Let X_i be the indicator variable that bin i is empty after all balls are tossed and X be the random variable that gives the number of empty bins. Then we have

$$E[X] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n}\left(\frac{n-1}{n}\right)^n = n\left(\frac{n-1}{n}\right)^n.$$

7

Now let X_i be the indicator variable that bin i contains exactly 1 ball after all balls are tossed and X be the random variable that gives the number of bins containing exactly 1 ball. Then we have

$$E[X] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} \binom{n}{1} \left(\frac{n-1}{n}\right)^{n-1} \frac{1}{n} = n \left(\frac{n-1}{n}\right)^{n-1}$$

because we need to choose which toss will go into bin i, then multiply by the probability that that toss goes into that bin and the remaining $n-1$ tosses avoid it.

Exercise 5.4-7

We split up the n flips into n/s groups where we pick $s = \lg(n) - 2\lg(\lg(n))$. We will show that at least one of these groups comes up all heads with probability at least $\frac{n-1}{n}$. So, the probability the group starting in position i comes up all heads is:

$$\Pr(A_{i,\lg(n)-2\lg(\lg(n))}) = \frac{1}{2^{\lg(n)-2\lg(\lg(n))}} = \frac{\lg(n)^2}{n}$$

Since the groups are based of of disjoint sets of IID coin flips, these probabilities are independent. so,

$$\Pr(\bigwedge_i \neg A_{i,\lg(n)-2\lg(\lg(n))}) = \prod_i \Pr(\neg A_{i,\lg(n)-2\lg(\lg(n))})$$

$$= \left(1 - \frac{\lg(n)^2}{n}\right)^{\frac{n}{\lg(n)-2\lg(\lg(n))}} \le e^{-\frac{\lg(n)^2}{\lg(n)-2\lg(\lg(n))}} = \frac{1}{n} e^{\frac{-2\lg(\lg(n))\lg(n)}{\lg(n)-2\lg(\lg(n))}}$$

$$= n^{-1-\frac{2\lg(\lg(n))}{\lg(n)-2\lg(\lg(n))}} < n^{-1}$$

Showing that the probability that there is no run of length at least $\lg(n) - 2\lg(\lg(n))$ to be $< \frac{1}{n}$.

Problem 5-1

a. We will show that the expected increase from each increment operation is equal to one. Suppose that the value of the counter is currently i. Then, we will increase the number represented from n_i to n_{i+1} with a probability of $\frac{1}{n_{i+1}-n_i}$, leaving the value alone otherwise. Multiplying these together, we get that the expected increase is $\frac{n_{i+1}-n_i}{n_{i+1}-n_i} - 1$.

b. For this choice of n_i, we have that at each increment operation, the probability that we change the value of the counter is $\frac{1}{100}$. Since this is a constant with respect to the current value of the counter i, we can view the final result as a binomial distribution with a p value of .01. Since the variance of a binomial distribution is $np(1-p)$, and we have that each success is worth 100 instead, the variance is going to be equal to .99n.

Problem 5-2

a. Assume that A has n elements. Our algorithm will use an array P to track the elements which have been seen, and add to a counter c each time a new element is checked. Once this counter reaches n, we will know that every element has been checked. Let $RI(A)$ be the function that returns a random index of A.

Algorithm 2 RANDOM-SEARCH

    Initialize an array P of size n containing all zeros
    Initialize integers c and i to 0
    while $c \neq n$ **do**
        $i = RI(A)$
        if $A[i] == x$ **then**
            return i
        end if
        if $P[i] == 0$ **then**
            $P[i] = 1$
            $c = c + 1$
        end if
    end while
    return A does not contain x

---

b. Let N be the random variable for the number of searches required. Then

$$E[N] = \sum_{i \geq 1} i P(i \text{ iterations are required})$$

$$= \sum_{i \geq 1} i \left(\frac{n-1}{n}\right)^{i-1} \left(\frac{1}{n}\right)$$

$$= \frac{1}{n} \frac{1}{\left(1 - \frac{n-1}{n}\right)^2}$$

$$= n.$$

c. Let N be the random variable for the number of searches required. Then

$$E[N] = \sum_{i \geq 1} i P(i \text{ iterations are required})$$

$$= \sum_{i \geq 1} i \left(\frac{n-k}{n}\right)^{i-1} \left(\frac{k}{n}\right)$$

$$= \frac{k}{n} \frac{1}{\left(1 - \frac{n-k}{n}\right)^2}$$

$$= \frac{n}{k}.$$

9

d. This is identical to the "How many balls must we toss until every bin contains at least one ball?" problem solved in section 5.4.2, whose solution is $b(\ln b + O(1))$.

e. The average case running time is $(n+1)/2$ and the worst case running time is n.

f. Let X be the random variable which gives the number of elements examined before the algorithm terminates. Let X_i be the indicator variable that the ith element of the array is examined. If i is an index such that $A[i] \neq x$ then $P(X_i) = \frac{1}{k+1}$ since we examine it only if it occurs before every one of the k indices that contains x. If i is an index such that $A[i] = x$ then $P(X_i) = \frac{1}{k}$ since only one of the indices corresponding to a solution will be examined. Let $S = \{i | A[i] = x\}$ and $S' = \{i | A[i] \neq x\}$. Then we have

$$E[X] = \sum_{i=1}^{n} E[X_i] = \sum_{i \in S} P(X_i) + \sum_{i \in S'} P(X_i) = \frac{k}{k} + \frac{n-k}{k+1} = \frac{n+1}{k+1}.$$

Thus the average case running time is $\frac{n+1}{k+1}$. The worst case happens when every occurrence of x is in the last k positions of the array. This has running time $n - k + 1$.

g. The average and worst case running times are both n.

h. SCRAMBLE-SEARCH works identically to DETERMINISTIC-SEARCH, except that we add to the running time the time it takes to randomize the input array.

i. I would use DETERMINISTIC-SEARCH since it has the best expected runtime and is guaranteed to terminate after n steps, unlike RANDOM-SEARCH. Moreover, in the time it takes SCRAMBLE-SEARCH to randomly permute the input array we could have performed a linear search anyway.

Chapter 6

Michelle Bodnar, Andrew Lohr

May 22, 2017

Exercise 6.1-1

At least 2^h and at most $2^{h+1} - 1$. Can be seen because a complete binary tree of depth $h - 1$ has $\Sigma_{i=0}^{h-1} 2^i = 2^h - 1$ elements, and the number of elements in a heap of depth h is between the number for a complete binary tree of depth $h - 1$ exclusive and the number in a complete binary tree of depth h inclusive.

Exercise 6.1-2

Write $n = 2^m - 1 + k$ where m is as large as possible. Then the heap consists of a complete binary tree of height $m - 1$, along with k additional leaves along the bottom. The height of the root is the length of the longest simple path to one of these k leaves, which must have length m. It is clear from the way we defined m that $m = \lfloor \lg n \rfloor$.

Exercise 6.1-3

If there largest element in the subtee were somewhere other than the root, it has a parent that is in the subtree. So, it is larger than it's parent, so, the heap property is violated at the parent of the maximum element in the subtree

Exercise 6.1-4

The smallest element must be a a leaf node. Suppose that node x contains the smallest element and x is not a leaf. Let y denote a child node of x. By the max-heap property, the value of x is greater than or equal to the value of y. Since the elements of the heap are distinct, the inequality is strict. This contradicts the assumption that x contains the smallest element in the heap.

Exercise 6.1-5

Yes, it is. The index of a child is always greater than the index of the parent, so the heap property is satisfied at each vertex.

Exercise 6.1-6

No, the array is not a max-heap. 7 is contained in position 9 of the array, so its parent must be in position 4, which contains 6. This violates the max-heap property.

Exercise 6.1-7

It suffices to show that the elements with no children are exactly indexed by $\{\lfloor n/2 \rfloor + 1, \ldots, n\}$. Suppose that we had an i in this range. It's childeren would be located at $2i$ and $2i+1$ but both of these are $\geq 2\lfloor n/2 \rfloor + 2 > n$ and so are not in the array. Now, suppose we had an element with no kids, this means that $2i$ and $2i+1$ are both $> n$, however, this means that $i > n/2$. This means that $i \in \{\lfloor n/2 \rfloor + 1, \ldots, n\}$.

Exercise 6.2-1

27	17	3	16	13	10	1	5	7	12	4	8	9	0
27	17	10	16	13	3	1	5	7	12	4	8	9	0
27	17	10	16	13	9	1	5	7	12	4	8	3	0

Exercise 6.2-2

Algorithm 1 MIN-HEAPIFY(A,i)

1: $l = LEFT(i)$
2: $r = RIGHT(i)$
3: **if** $l \leq A.heap - size$ and $A[l] < A[i]$ **then**
4:     $smallest = l$
5: **else** $smallest = i$
6: **end if**
7: **if** $r \leq A.heap - size$ and $A[r] < A[smallest]$ **then**
8:     $smallest = r$
9: **end if**
10: **if** $smallest \neq i$ **then**
11:     exchange $A[i]$ with $A[smallest]$
12:     MIN-HEAPIFY$(A, smallest)$
13: **end if**

The running time of MIN-HEAPIFY is the same as that of MAX-HEAPIFY.

Exercise 6.2-3

The array remains unchanged since the if statement on line line 8 will be false.

Exercise 6.2-4

If $i > A.heap - size/2$ then l and r will both exceed $A.heap - size$ so the if statement conditions on lines 3 and 6 of the algorithm will never be satisfied. Therefore $largest = i$ so the recursive call will never be made and nothing will happen. This makes sense because i necessarily corresponds to a leaf node, so MAX–HEAPIFY shouldn't alter the heap.

Exercise 6.2-5
Iterative Max Heapify(A, i)

```
while i < A.heap-size do
    l =LEFT(i)
    r =LEFT(i)
    largest = i
    if l ≤ A.heap-size and A[l] > A[i] then
        largest = l
    end if
    if l ≤ A.heap-size and A[r] > A[i] then
        largest = r
    end if
    if largest ≠ i then
        exchange A[i] and A[largest]
    elsereturn A
    end if
end while
return A
```

Exercise 6.2-6

Consider the heap resulting from A where $A[1] = 1$ and $A[i] = 2$ for $2 \leq i \leq n$. Since 1 is the smallest element of the heap, it must be swapped through each level of the heap until it is a leaf node. Since the heap has height $\lfloor \lg n \rfloor$, MAX-HEAPIFY has worst-case time $\Omega(\lg n)$.

Exercise 6.3-1

5	3	17	10	84	19	6	22	9
5	3	17	22	84	19	6	10	9
5	3	19	22	84	17	6	10	9
5	84	19	22	3	17	6	10	9
84	5	19	22	3	17	6	10	9
84	22	19	5	3	17	6	10	9
84	22	19	10	3	17	6	5	9

Exercise 6.3-2

If we had started at 1, we wouldn't be able to guarantee that the max-heap property is maintained. For example, if the array A is given by [2,1,1,3] then MAX-HEAPIFY won't exchange 2 with either of it's children, both 1's. However, when MAX-HEAPIFY is called on the left child, 1, it will swap 1 with 3. This violates the max-heap property because now 2 is the parent of 3.

Exercise 6.3-3

All the nodes of height h partition the set of leaves into sets of size between $2^{h-1} + 1$ and 2^h, where all but one is size 2^h. Just by putting all the children of each in their own part of trhe partition. Recall from 6.1-2 that the heap has height $\lfloor \lg(n) \rfloor$, so, by looking at the one element of this height (the root), we get that there are at most $2^{\lfloor \lg(n) \rfloor}$ leaves. Since each of the vertices of height h partitions this into parts of size at least $2^{h-1} + 1$, and all but one corresponds to a part of size 2^h, we can let k denote the quantity we wish to bound, so,

$$(k-1)2^h + k(2^{h-1} + 1) \leq 2^{\lfloor \lg(n) \rfloor} \leq n/2$$

so

$$k \leq \frac{n + 2^h}{2^{h+1} + 2^h + 1} \leq \frac{n}{2^{h+1}} \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$$

Exercise 6.4-1

4

5	13	2	25	7	17	20	8	4
5	13	20	25	7	17	2	8	4
5	25	20	13	7	17	2	8	4
25	5	20	13	7	17	2	8	4
25	13	20	5	7	17	2	8	4
25	13	20	8	7	17	2	5	4
4	13	20	8	7	17	2	5	25
20	13	4	8	7	17	2	5	25
20	13	17	8	7	4	2	5	25
5	13	17	8	7	4	2	20	25
17	13	5	8	7	4	2	20	25
2	13	5	8	7	4	17	20	25
13	2	5	8	7	4	17	20	25
13	8	5	2	7	4	17	20	25
4	8	5	2	7	13	17	20	25
8	4	5	2	7	13	17	20	25
8	7	5	2	4	13	17	20	25
4	7	5	2	8	13	17	20	25
7	4	5	2	8	13	17	20	25
2	4	5	7	8	13	17	20	25
5	4	2	7	8	13	17	20	25
2	4	5	7	8	13	17	20	25
4	2	5	7	8	13	17	20	25
2	4	5	7	8	13	17	20	25

Exercise 6.4-2

We'll prove the loop invariant of HEAPSORT by induction:

Base case: At the start of the first iteration of the for loop of lines 2-5 we have $i = A.length$. The subarray $A[1..n]$ is a max-heap since BUILD-MAX-HEAP(A) was just called. It contains the n smallest elements, and the empty subarray $A[n+1..n]$ trivially contains the 0 largest elements of A in sorted order.

Suppose that at the start of the ith iteration of 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 $n - i$ largest elements of $A[1..n]$ in sorted order. Since $A[1..i]$ is a max-heap, $A[1]$ is the largest element in $A[1..i]$. Thus it is the $(n - (i - 1))^{th}$ largest element from the original array since the $n - i$ largest elements are assumed to be at the end of the array. Line 3 swaps $A[1]$ with $A[i]$, so $A[i..n]$ contain the $n - i + 1$ largest elements of the array, and $A[1..i - i]$ contains the $i - 1$ smallest elements. Finally, MAX-HEAPIFY is called on $A, 1$. Since $A[1..i]$ was a max-heap prior to the iteration and only the elements in positions 1 and i were swapped, the left and right subtrees of node 1, up to node $i - 1$, will be max-heaps. The call to MAX-HEAPIFY will place the element now located at node 1 into the correct position and restore the

5

max-heap property so that $A[1..i-1]$ is a max-heap. This concludes the next iteration, and we have verified each part of the loop invariant. By induction, the loop invariant holds for all iterations.

After the final iteration, the loop invariant says that the subarray $A[2..n]$ contains the $n-1$ largest elements of $A[1..n]$, sorted. Since $A[1]$ must be the nth largest element, the whole array must be sorted as desired.

Exercise 6.4-3

If it's already sorted in increasing order, doing the BUILD-MAX-HEAP call on line 1 will take $\Theta(n \lg(n))$ time. As we call MAX-HEAPIFY(A,i) in BUILD-HEAP, we know that $A[i]$ is larger than all elements to its right. This means that the time it takes will be $\lfloor \lg(n) - \lg(i) \rfloor$. So the total time to build the heap will be

$$\sum_{i=1}^{i=\lfloor n/2 \rfloor} \lg(n) - \lg(i) \in \Theta(n \lg(n))$$

There will be n iterations of the for loop in HEAPSORT, each taking $\Theta(\lg(n))$ time because the element that was at position i was the smallest and so will have $\lfloor \lg(n) \rfloor$ steps when doing MAX-HEAPIFY on line 5. So, it will be $\Theta(n \lg(n))$ time.

If it's already sorted in decreasing order, then the call on line one will only take $\Theta(n)$ time, since it was already a max heap to begin with. This is because elements in earlier positions correspond to begin further up in the heap, and since the array is descending, those have larger values. So, each of the MAX-HEAPIFY calls in the BUILD-MAX-HEAP procedure will only take constant time since the check on line 8 of MAX-HEAPIFY will always be false. It will still take $n \lg(n)$ peel off the elements from the heap and re-heapify. This is because each time we swap the head of the heap with the last element, we now have the smallest element at the head, so it will have to have as many recursive calls of MAX-HEAPIFY as the depth of the heap, which is about $\lg(n)$.

Exercise 6.4-4

Consider calling HEAPSORT on an array which is sorted in decreasing order. Every time $A[1]$ is swapped with $A[i]$, MAX-HEAPIFY will be recursively called a number of times equal to the height h of the max-heap containing the elements of positions 1 through $i-1$, and has runtime $O(h)$. Since there are 2^k nodes at height k, the runtime is bounded below by

$$\sum_{i=1}^{\lfloor \lg n \rfloor} 2^i \log(2^i) = \sum_{i=1}^{\lfloor \lg n \rfloor} i 2^i = 2 + (\lfloor \lg n \rfloor - 1) 2^{\lfloor \lg n \rfloor} = \Omega(n \lg n).$$
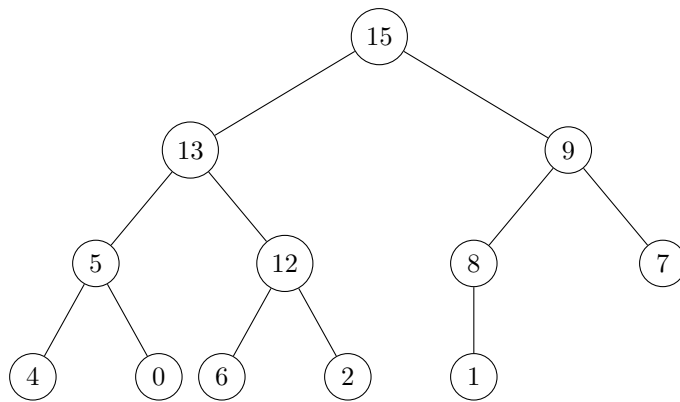
Exercise 6.4-5

Since the call on line one could possibly take only linear time (if the input was already a max-heap for example), we will focus on showing that the for loop takes n log n time. This is the case because each time that the last element is placed at the beginning to replace the max element being removed, it has to go through every layer, because it was already very small since it was at the bottom level of the heap before.
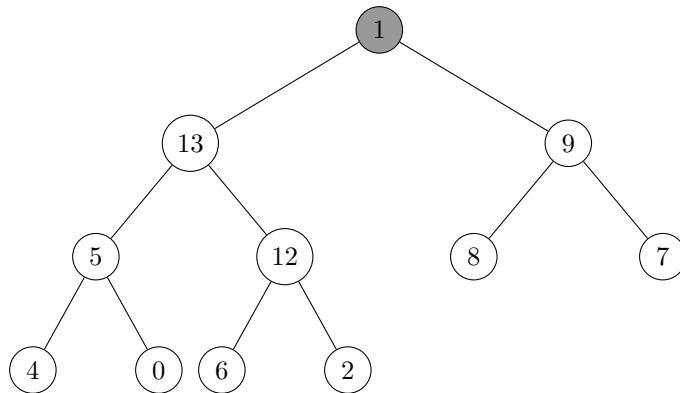
Exercise 6.5-1

The following sequence of pictures shows how the max is extracted from the heap.

1. Original heap:

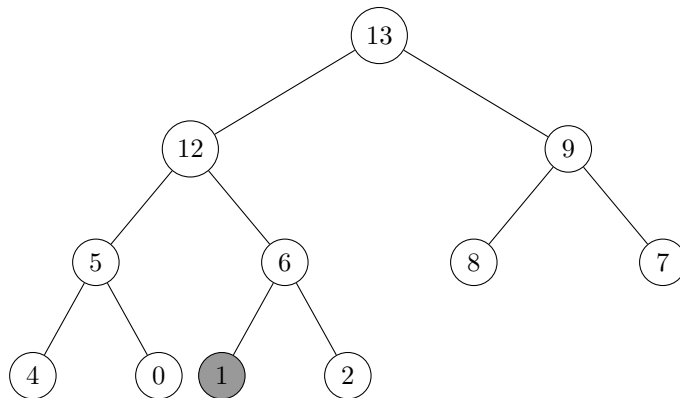2. we move the last element to the top of the heap

3. $13 > 9 > 1$ so, we swap 1 and 13.
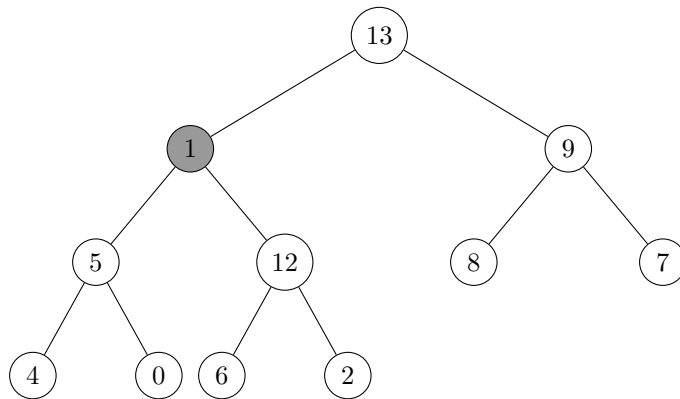
4. Since $12 > 5 > 1$, we swap 1 and 12.
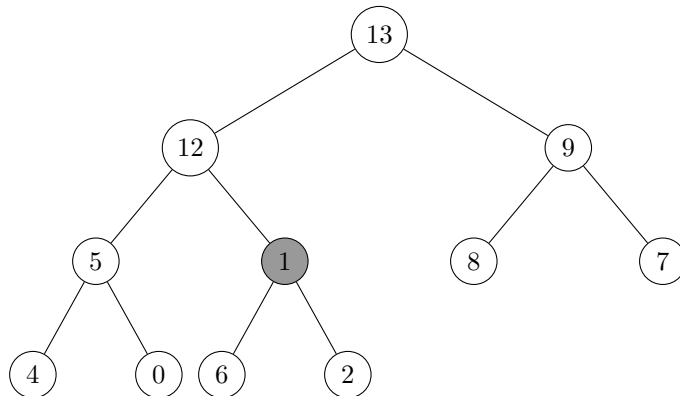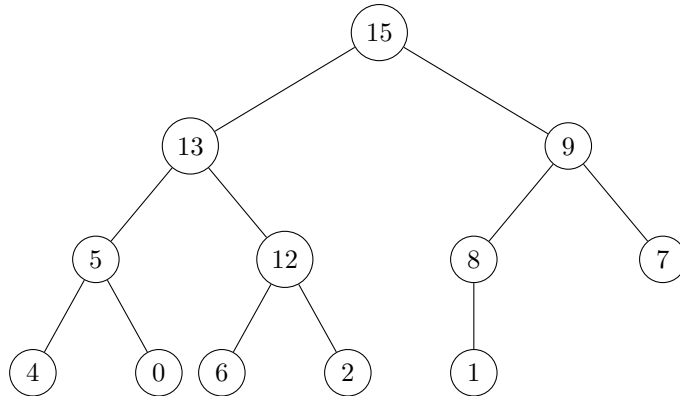

5. Since $6 > 2 > 1$, we swap 1 and 6.

Exercise 6.5-2

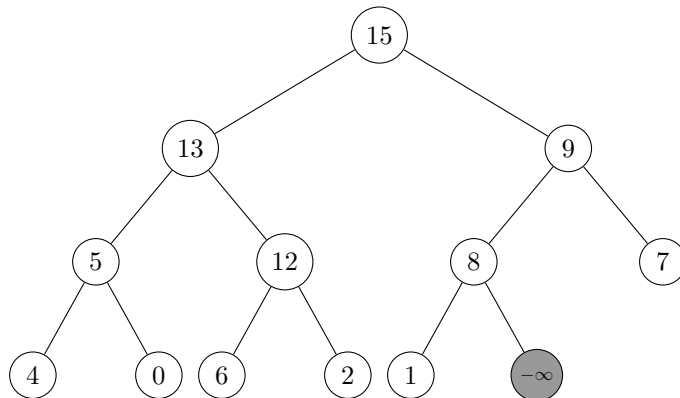The following sequence of pictures shows how 10 is inserted into the heap,

then swapped with parent nodes until the max-heap property is restored. The node containing the new key is heavily shaded.
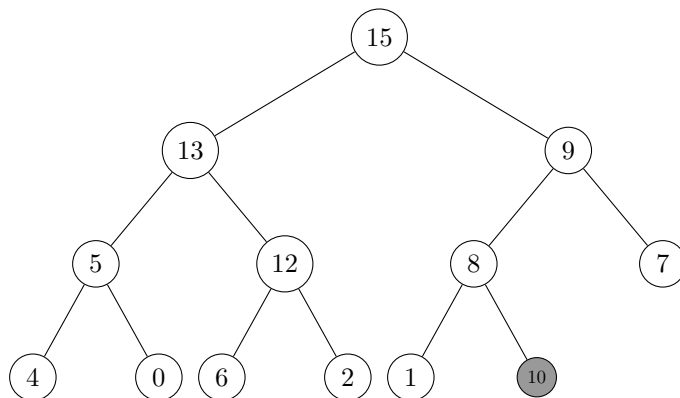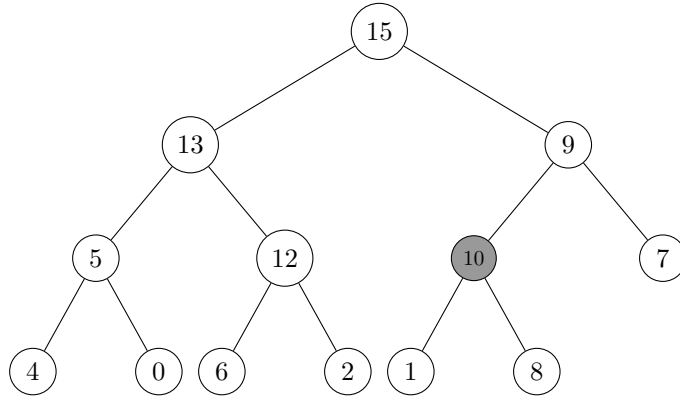
1. Original heap:

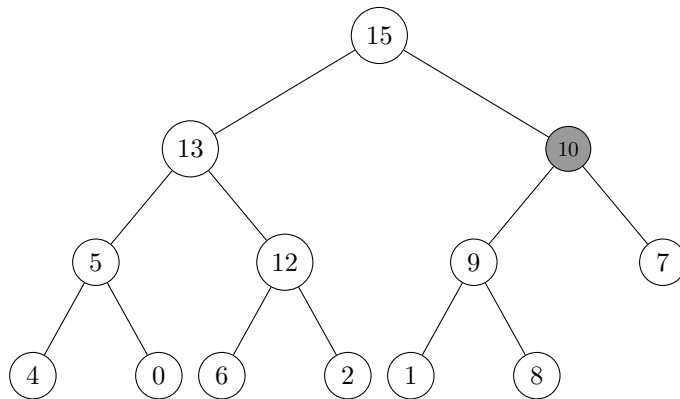2. MAX-HEAP-INSERT(A,10) is called, so we first append a node assigned value $-\infty$:

3. The key value of the new node is updated:

4. Since the parent key is smaller than 10, the nodes are swapped:

5. Since the parent node is smaller than 10, the nodes are swapped:

Exercise 6.5-3

Heap-Minimum(A)

1: **return** A[1]

Heap-Extract-Min(A)
Heap-decrease-key(A,i,key)
Min-Heap-Insert(A,key)

Exercise 6.5-4

If we don't make an assignment to $A[A.heap-size]$ then it could contain any value. In particular, when we call HEAP-INCREASE-KEY, it might be the case that $A[A.heap-size]$ initially contains a value larger than key, causing an error. By assigning $-\infty$ to $A[A.heap-size]$ we guarantee that no error will

```
1:  if A.heap-size < 1 then
2:      Error "heap underflow"
3:  end if
4:  min = A[1]
5:  A[1] = A[A.heap − size]
6:  A.heap − size − −
7:  Min-heapify(A,1)
8:  return min
```

```
1:  if key ¿ A[i] then
2:      Error "new key larger than old key"
3:  end if
4:  A[i] = key
5:  while i > 1 and A[Parent(i)] < A[i] do
6:      exchange A[i] with A[Parent(i)]
7:      i = Parent(i)
8:  end while
```

occur. However, we could have assigned any value less than or equal to *key* to $A[A.heap − size]$ and the algorithm would still work.

Exercise 6.5-5

Initially, we have a heap and then only change the value at i to make it larger. This can't invalidate the ordering between i and it's children, the only other thing that needs to be related to i is that i is less than it's parent, which may be false. Thus we have the invariant is true at initialization. Then, when we swap i with its parent if it is larger, since it is larger than it's parent, it must also be larger than it's sibling, also, since it's parent was initially above its kids in the heap, we know that it's parent is larger than it's kids. The only relation in question is then the new i and it's parent. At termination, i is the root, so it has no parent, so the heap property must be satisfied everywhere.

Exercise 6.5-6

Replace $A[i]$ by *key* in the while condition, and replace line 5 by "$A[i] = A[PARENT(i)]$." After the end of the while loop, add the line $A[i] = key$. Since the key value doesn't change, there's no sense in assigning it until we know where it belongs in the heap. Instead, we only make the assignment of

```
1:  A.heap − size + +
2:  A[A.heap − size] = ∞
3:  Heap-Decrease-Key(A,A.heap-size,key)
```

the parent to the child node. At the end of the while loop, i is equal to the position where *key* belongs since it is either the root, or the parent is at least *key*, so we make the assignment.

Exercise 6.5-7

Have a field in the structure that is just a count of the total number of elements ever added. When adding an element, use the current value of that counter as the key.

Exercise 6.5-8

The algorithm works as follows: Replace the key of the node to be deleted by ∞, ie a value which will be interpreted as greater than all other keys stored in the max-heap. Calling HEAP-INCREASE-KEY will float that node to the top of the max-heap. We then replace the value of the root node with the value of the last element in the heap, known to be smaller than $A[1]$ by the max-heap property. We update the size of the heap, then call MAX-HEAPIFY to restore the max-heap property. This has running time $O(\lg n)$ since INCREASE-KEY runs in $O(\lg n)$ and the number of times MAX-HEAPIFY is recursively called is as most the height of the heap, which is $\lfloor \lg n \rfloor$.

Algorithm 2 HEAP-DELETE(A,i)

1: HEAP-INCREASE-KEY(A, i, ∞)
2: A[1] = A[A.heap-size]
3: A.heap-size = A.heap-size - 1
4: MAX-HEAPIFY(A,1)

Exercise 6.5-9

Construct a min heap from the heads of each of the k lists. Then, to find the next element in the sorted array, extract the minimum element (in $O \lg(k)$ time). Then, add to the heap the next element from the shorter list from which the extracted element originally came (also $O(\lg(k))$ time). Since finding the next element in the sorted list takes only at most $O(\lg(k))$ time, to find the whole list, you need $O(n \lg(k))$ total steps.

Problem 6-1

a. They do not. Consider the array $A = \langle 3, 2, 1, 4, 5 \rangle$. If we run Build-Max-Heap, we get $\langle 5, 4, 1, 3, 2 \rangle$. However, if we run Build-Max-Heap', we will get $\langle 5, 4, 1, 2, 3 \rangle$ instead.

b. Each insert step takes at most $O(\lg(n))$, since we are doing it n times, we get a bound on the runtime of $O(n \lg(n))$.

Problem 6-2

a. It will suffice to show how to access parent and child nodes. In a d-ary array, $\text{PARENT}(i) = \lfloor i/d \rfloor$, and $\text{CHILD}(k,i) = di - d + 1 + k$, where $\text{CHILD}(k,i)$ gives the kth child of the node indexed by i.

b. The height of a d-ary heap of n elements is with 1 of $\log_d n$.

c. The following is an implementation of HEAP-EXTRACT-MAX for a d-ary heap. An implementation of DMAX-HEAPIFY is also given, which is the analog of MAX-HEAPIFY for d-ary heap. HEAP-EXTRACT-MAX consists of constant time operations, followed by a call to DMAX-HEAPIFY. The number of times this recursively calls itself is bounded by the height of the d-ary heap, so the running time is $O(d \log_d n)$. Note that the CHILD function is meant to be the one described in part (a).

Algorithm 3 HEAP-EXTRACT-MAX(A) for a d-ary heap

1: **if** $A.heap - size < 1$ **then**
2:     **error** "heap underflow"
3: **end if**
4: $max = A[1]$
5: $A[1] = A[A.heap - size]$
6: $A.heap - size = A.heap - size - 1$
7: DMAX-HEAPIFY(A,1)

Algorithm 4 DMAX-HEAPIFY(A,i)

1: $largest = i$
2: **for** $k = 1$ to d **do**
3:     **if** $CHILD(k,i) \leq A.heap - size$ and $A[CHILD(k,i)] > A[i]$ **then**
4:         **if** $A[CHILD(k,i)] > largest$ **then**
5:             $largest = A[CHILD(k,i)]$
6:         **end if**
7:     **end if**
8: **end for**
9: **if** $largest \neq i$ **then**
10:     exchange $A[i]$ with $A[largest]$
11:     DMAX-HEAPIFY($A, largest$)
12: **end if**

---

d. The runtime of this implementation of INSERT is $O(\log_d n)$ since the while loop runs at most as many times as the height of the d-ary array. Note that

when we call PARENT, we mean it as defined in part (a).

Algorithm 5 INSERT(A,key)

1: $A.heap - size = A.heap - size + 1$
2: $A[A.heap - size] = key$
3: $i = A.heap - size$
4: **while** $i > 1$ and $A[PARENT(i) < A[i]$ **do**
5:     exchange $A[i]$ with $A[PARENT(i)]$
6:     $i = PARENT(i)$
7: **end while**

---

e. This is identical to the implementation of HEAP-INCREASE-KEY for 2-ary heaps, but with the PARENT function interpreted as in part (a). The runtime is $O(\log_d n)$ since the while loop runs at most as many times as the height of the d-ary array.

Algorithm 6 INCREASE-KEY(A,i,key)

1: **if** $key < A[i]$ **then**
2:     **error** "new key is smaller than current key "
3: **end if**
4: $A[i] = key$
5: **while** $i > 1$ and $A[PARENT(i) < A[i]$ **do**
6:     exchange $A[i]$ with $A[PARENT(i)]$
7:     $i = PARENT(i)$
8: **end while**

Problem 6-3

a.

2	3	4	5
8	9	12	14
16	∞	∞	∞
∞	∞	∞	∞

b. For every i, j, $Y[1,1] \leq Y[i,1] \leq Y[i,j]$. So, if $Y[1,1] = \infty$, we know that $Y[i,j] = \infty$ for every i, j. This means that no elements exist. If Y is full, it has no elements labeled ∞, in particular, the element $Y[m,n]$ is not labeled ∞.

c. Extract-Min(Y,i,j), extracts the minimum value from the young tableau Y' obtained by $Y'[i',j'] = Y[i' + i - 1, j' + j - 1]$. Note that in running this algorithm, several accesses may be made out of bounds for Y, define these to return ∞. No store operations will be made on out of bounds locations. Since the largest value of $i + j$ that this can be called with is $n + m$, and

14

```
1:  min = Y[i, j]
2:  if  Y[i, j + 1] = Y[i + 1, j] = ∞ then
3:      Y[i, j] = ∞
4:      return min
5:  end if
6:  if Y[i, j + 1] < Y[i + 1, j]  then
7:      Y[i, j] = Y[i, j + 1]
8:      Y[i, j + 1] = min
9:      return Extract-min(y,i,j+1)
10: else
11:     Y[i, j] = Y[i + 1, j]
12:     Y[i + 1, j] = min
13:     return Extract-min(y,i+1,j)
14: end if
```

this quantity must increase by one for each call, we have that the runtime is bounded by $n + m$.

d. Insert(Y,key) Since $i + j$ is decreasing at each step, starts as $n + m$ and is

```
1:  i = m, j = n
2:  Y[i, j] = key
3:  while Y[i − 1, j] > Y[i, j] or Y[i, j − 1] > Y[i, j] do
4:      if Y[i − 1, j] < Y[i, j − 1] then
5:          Swap Y[i, j] and Y[i, j − 1]
6:          j − −
7:      else
8:          Swap Y[i, j] and Y[i − 1, j]
9:          i − −
10:     end if
11: end while
```

bounded by 2 below, we know that this program has runtime $O(n + m)$.

e. Place the n^2 elements into a Young Tableau by calling the algorithm from part d on each. Then, call the algorithm from part c n^2 to obtain the numbers in increasing order. Both of these operations take time at most $2n \in O(n)$, and are done n^2 times, so, the total runtime is $O(n^3)$

f. Find(Y,key). Let Check(y,key,i,j) mean to return true if $Y[i, j] = key$, otherwise do nothing

```
i = j = 1
while Y[i, j] < key and i < m do
    Check(Y,key,i,j)
    i++
end while
while i > 1 and j < n do
    Check(i,j)
    if Y[i, j] < key then
        j++
    else
        i–
    end if
end while
return false
```

Chapter 7

Michelle Bodnar, Andrew Lohr

September 17, 2017

Exercise 7.1-1

13	19	9	5	12	8	7	4	21	2	6	11
13	19	9	5	12	8	7	4	21	2	6	11
13	19	9	5	12	8	7	4	21	2	6	11
9	19	13	5	12	8	7	4	21	2	6	11
9	5	13	19	12	8	7	4	21	2	6	11
9	5	13	19	12	8	7	4	21	2	6	11
9	5	8	19	12	13	7	4	21	2	6	11
9	5	8	7	12	13	19	4	21	2	6	11
9	5	8	7	4	13	19	12	21	2	6	11
9	5	8	7	4	13	19	12	21	2	6	11
9	5	8	7	4	2	19	12	21	13	6	11
9	5	8	7	4	2	6	12	21	13	19	11
9	5	8	7	4	2	6	11	21	13	19	12

Exercise 7.1-2

If all elements in the array have the same value, PARTITION returns r. To make PARTITION return $q = \lfloor (p + r)/2 \rfloor$ when all elements have the same value, modify line 4 of the algorithm to say this: if $A[j] \leq x$ and $j(mod2) = (p + 1)(mod2)$. This causes the algorithm to treat half of the instances of the same value to count as less than, and the other half to count as greater than.

Exercise 7.1-3

The for loop makes exactly $r - p$ iterations, each of which takes at most constant time. The part outside the for loop takes at most constant time. Since $r - p$ is the size of the subarray, PARTITION takes at most time proportional to the size of the subarray it is called on.

Exercise 7.1-4

To modify QUICKSORT to run in non-increasing order we need only modify line 4 of PARTITION, changing $\leq$ to $\geq$.

1

Exercise 7.2-1

By definition of Θ, we know that there exists c_1, c_2 so that the $\Theta(n)$ term is between $c_1 n$ and $c_2 n$. We make that inductive hypothesis be that $c_1 m^2 \leq T(m) \leq c_2 m^2$ for all $m < n$, then, for large enough n,

$$c_1 n^2 \leq c_1 (n-1)^2 + c_1 n \leq T(n-1) + \Theta(n)$$

$$= T(n) = T(n-1) + \Theta(n) \leq c_2 (n-1)^2 + c_2 n \leq c_2 n^2$$

Exercise 7.2-2

The running time of QUICKSORT on an array in which evey element has the same value is n^2. This is because the partition will always occur at the last position of the array (Exercise 7.1-2) so the algorithm exhibits worst-case behavior.

Exercise 7.2-3

If the array is already sorted in decreasing order, then, the pivot element is less than all the other elements. The partition step takes $\Theta(n)$ time, and then leaves you with a subproblem of size $n-1$ and a subproblem of size 0. This gives us the recurrence considered in 7.2-1. Which we showed has a solution that is $\Theta(n^2)$.

Exercise 7.2-4

Let's say that by "almost sorted" we mean that $A[i]$ is at most c positions from its correct place in the sorted array, for some constant c. For INSERTION-SORT, we run the inner-while loop at most c times before we find where to insert $A[j]$ for any particular iteration of the outer for loop. Thus the running time time is $O(cn) = O(n)$, since c is fixed in advance. Now suppose we run QUICK-SORT. The split of PARTITION will be *at best $n - c$ to c*, which leads to $O(n^2)$ running time.

Exercise 7.2-5

The minimum depth corresponds to repeatedly taking the smaller subproblem, that is, the branch whose size is proportional to α. Then, this will fall to 1 in k steps where $1 \approx (\alpha)^k n$. So, $k \approx \log_\alpha(1/n) = -\frac{\lg(n)}{\lg(\alpha)}$. The longest depth corresponds to always taking the larger subproblem. we then have and identical expression, replacing α with $1 - \alpha$.

Exercise 7.2-6

Without loss of generality, assume that the entries of the input array are distinct. Since only the relative sizes of the entries matter, we may assume that A contains a random permutation of the numbers 1 through n. Now fix $0 < \alpha \le 1/2$. Let k denote the number of entries of A which are less than $A[n]$. PARTITION produces a split more balanced than $1 - \alpha$ to α if and only if $\alpha n \le k \le (1 - \alpha)n$. This happens with probability $\frac{(1-\alpha)n - \alpha n + 1}{n} = 1 - 2\alpha + 1/n \approx 1 - 2\alpha$.

Exercise 7.3-1

We analyze the expected run time because it represents the more typical time cost. Also, we are doing the expected run time over the possible randomness used during computation because it can't be produced adversarially, unlike when doing expected run time over all possible inputs to the algorithm.

Exercise 7.3-2

In either case, $\Theta(n)$ calls are made to RANDOM. PARTITION will run faster in the best case because the inputs will generally be smaller, but RANDOM is called every single time RANDOMIZED-PARTITION is called, which happens $\Theta(n)$ times.

Exercise 7.4-1

By definition of Θ, we know that there exists c_1, c_2 so that the $\Theta(n)$ term is between $c_1 n$ and $c_2 n$. We make that inductive hypothesis be that $c_1 m^2 \le T(m) \le c_2 m^2$ for all $m < n$, then, for large enough n,

$$
\begin{aligned}
c_1 n^2 &\le c_1 \max_{q \in [n]} n^2 - 2n(q+2) + (q+1)^2 + (q+1)^2 + n \\
&= \max_{q \in [n]} c_1(n - q - 2)^2 + c_1(q+1)^2 + c_1 n \\
&\le \max_{q \in [n]} T(n - q - 2) + T(q+1) + \Theta(n) \\
&= T(n)
\end{aligned}
$$

Similarly for the other direction

Exercise 7.4-2

We'll use the substitution method to show that the best-case running time is $\Omega(n \lg n)$. Let $T(n)$ be the best-case time for the procedure QUICKSORT on an input of size n. We have the recurrence

$$
T(n) = \min_{1 \le q \le n-1} (T(q) + T(n - q - 1)) + \Theta(n).
$$

Suppose that $T(n) \ge c(n \lg n + 2n)$ for some constant c. Substituting this guess

into the recurrence gives

$$T(n) \geq \min_{1 \leq q \leq n-1} (cq \lg q + 2cq + c(n-q-1)\lg(n-q-1) + 2c(n-q-1)) + \Theta(n)$$

$$= \frac{cn}{2} \lg(n/2) + cn + c(n/2-1)\lg(n/2-1) + cn - 2c + \Theta(n)$$

$$\geq (cn/2)\lg n - cn/2 + c(n/2-1)(\lg n - 2) + 2cn - 2c\Theta(n)$$

$$= (cn/2)\lg n - cn/2 + (cn/2)\lg n - cn - \lg n + 2 + 2cn - 2c\Theta(n)$$

$$= cn \lg n + cn/2 - \lg n + 2 - 2c + \Theta(n)$$

Taking a derivative with respect to q shows that the minimum is obtained when $q = n/2$. Taking c large enough to dominate the $-\lg n + 2 - 2c + \Theta(n)$ term makes this greater than $cn \lg n$, proving the bound.

Exercise 7.4-3

We will treat the given expression to be continuous in q, and then, any extremal values must be either adjacent to a critical point, or one of the endpoints. The second derivative with respect to q is 4, So, we have that any critical points we find will be minima. The expression has a derivative with respect to q of $2q - 2(n-q-2) = -2n + 4q + 4$ which is zero when we have $2q + 2 = n$. So, there will be a minima at $q = \frac{n-2}{2}$. So, the maximal values must only be the endpoints. We can see that the endpoints are equally large because at $q = 0$, it is $(n-1)^2$, and at $q = n-1$, it is $(n-1)^2 + (n-n+1-1)^2 = (n-1)^2$.

Exercise 7.4-4

We'll use the lower bound (A.13) for the expected running time given in the section:

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k}$$

$$\geq \sum_{i=1}^{n-1} 2\ln(n-i+1)$$

$$= 2\ln\left(\prod_{i=1}^{n-1} n-i+1\right)$$

$$= 2\ln(n!)$$

$$= \frac{2}{\lg e} \lg(n!) \qquad\qquad \geq cn \lg n$$

for some constant c since $\lg(n!) = \Theta(n \lg n)$ by Exercise 3.2-3. Therefore RANDOMIZED-QUICKSORT's expected running time is $\Omega(n \lg n)$.

Exercise 7.4-5

If we are only doing quick-sort until the problem size becomes $\leq k$, then, we will have to take $\lg(n/k)$ steps, since, as in the original analysis of randomized quick sort, we expect there to be $\lg(n)$ levels to the recursion tree. Since we then just call quicksort on the entire array, we know that each element is within k of its final position. This means that an insertion sort will take the shifting of at most k elements for every element that needed to change position. This gets us the running time described.

In theory, we should pick k to minimize this expression, that is, taking a derivative with respect to k, we want it to be evaluating to zero. So, $n - \frac{n}{k} = 0$, so $k \sim \frac{1}{n^2}$. The constant of proportionality will depend on the relative size of the constants in the nk term and in the $n\lg(n/k)$ term. In practice, we would try it with a large number of input sizes for various values of k, because there are gritty properties of the machine not considered here such as cache line size.

Exercise 7.4-6

For simplicity of analysis, we will assume that the elements of the array A are the numbers 1 to n. If we let k denote the median value, the probability of getting at worst an α to $1 - \alpha$ split is the probability that $\alpha n \leq k \leq (1 - \alpha)n$. The number of "bad" triples is equal to the number of triples such that at least two of the numbers come from $[1, \alpha n]$ or at least two of the numbers come from $[(1 - \alpha)n, n]$. Since both intervals have the same size, the probability of a bad triple is $2(\alpha^3 + 3\alpha^2(1 - \alpha))$. Thus the probability of selecting a "good" triple, and thus getting at worst an α to $1 - \alpha$ split, is $1 - 2(\alpha^3 + 3\alpha^2(1 - \alpha)) = 1 + 4\alpha^3 - 6\alpha^2$.

Problem 7-1

a. We will be calling with the parameters $p = 1$, $r = |A| = 12$. So, throughout, $x = 13$.

												i	j
13	19	9	5	12	8	7	4	11	2	6	21	0	13
6	19	9	5	12	8	7	4	11	2	13	21	1	11
6	13	9	5	12	8	7	4	11	2	19	21	2	10
6	13	9	5	12	8	7	4	11	2	19	21	10	2

And we do indeed see that partition has moved the two elements that are bigger than the pivot, 19 and 21, to the two final positions in the array.

b. We know that at the beginning of the loop, we have that $i < j$, because it is true initially so long as $|A| \geq 2$. And if it were to be untrue at some iteration, then we would of left the loop in the prior iteration. To show that we won't access outside of the array, we need to show that at the beginning of every run of the loop, there is a $k > i$ so that $A[k] \geq x$, and a $k' < j$ so that $A[j'] \leq x$. This is clearly true because initially i and j are outside the bounds of the array, and so the element x must be between the two. Since

5

$i < j$, we can pick $k = j$, and $k' = i$. The elements k satisfies the desired relation to x, because the element at position j was the element at position i in the prior iteration of the loop, prior to doing the exchange on line 12. Similarly, for k'.

c. If there is more than one run of the main loop, we have that $j < r$ because it decreases by at least one with every iteration.

Note that at line 11 in the first run of the loop, we have that $i = 1$ because $A[p] = x \geq x$. So, if we were to terminate after a single iteration of the main loop, we must also have that $j = 1 < p$.

d. We will show the loop invariant that all the elements in $A[p..i]$ are less than or equal to x which is less than or equal to all the elements of $A[j..r]$. It is trivially true prior to the first iteration because both of these sets of elements are empty. Suppose we just finished an iteration of the loop during which j went from j_1 to j_2, and i went from i_1 to i_2. All the elements in $A[i_1+1..i_2-1]$ were $< x$, because they didn't cause the loop on lines $8 - 10$ to terminate. Similarly, we have that all the elements in $A[j_2 + 1..j_1 - 1]$ were $> x$. We have also that $A[i_2] \leq x \leq A[j_2]$ after the exchange on line 12. Lastly, by induction, we had that all the elements in $A[p..i_1]$ are less than or equal to x, and all the elements in $A[j_1..r]$ are greater than or equal to x. Then, putting it all together, since $A[p..i_2] = A[p..i_1] \cup A[i_1+1..i_2-1] \cup \{A[i_2]\}$ and $A[j_2..r] = \cup\{A[j_2]\} \cup A[j_2+1..j_1-1] \cup A[j_1..r]$, we have the desired inequality. Since at termination, we have that $i \geq j$, we know that $A[p..j] \subseteq A[p..i]$, and so, every element of $A[p..j]$ is less than or equal to x, which is less than or equal to every element of $A[j + 1..r] \subseteq A[j..r]$.

e. After running Hoare-partition, we don't have the guarantee that the pivot value will be in the position j, so, we will scan through the list to find the pivot value, place it between the two subarrays, and recurse

Problem 7-2

a. Since all elements are the same, the initial random choice of index and swap change nothing. Thus, randomized quicksort's running time will be the same as that of quicksort. Since all elements are equal, PARTITION(A, P, r) will always return $r - 1$. This is worst-case partitioning, so the runtime is $\Theta(n^2)$.

b. See the PARTITION' algorithm for modifications.

c. See the RANDOMIZED-PARTITION' algorithm for modifications.

d. Let d be the number of distinct elements in A. The running time is dominated by the time spent in the PARTITION procedure, and there can be at most d calls to PARTITION. If X is the number of comparisons performed in line 4 of PARTITION over the entire execution of QUICKSORT', then the running

Quicksort(A,p,r)

1: **if** $p < r$ **then**
2:     $x = A[p]$
3:     $q = HoarePartition(A, p, r)$
4:     $i = 0$
5:     **while** $A[i] \neq x$ **do**
6:         $i = i + 1$
7:     **end while**
8:     **if** $i \leq q$ **then**
9:         exchange $A[i]$ and $A[q]$
10:    **else**
11:        exchange $A[i]$ and $A[q + 1]$
12:        $q = q + 1$
13:    **end if**
14:    Quicksort(A,p,q-1)
15:    Quicksort(A,q+1,r)
16: **end if**

Algorithm 1 PARTITION'(A,p,r)

1: $x = A[r]$
2: exchange $A[r]$ with $A[p]$
3: $i = p - 1$
4: $k = p$
5: **for** $j = p + 1$ to $r - 1$ **do**
6:     **if** $A[j] < x$ **then**
7:         $i = i + 1$
8:         $k = i + 2$
9:         exchange $A[i]$ with $A[j]$
10:        exchange $A[k]$ with $A[j]$
11:    **end if**
12:    **if** $A[j] = x$ **then**
13:        $k = k + 1$
14:        exchange $A[k]$ with $A[j]$
15:    **end if**
16: **end for**
17: exchange $A[i + 1]$ with $A[r]$
18: **return** $i + 1$ and $k + 1$

Algorithm 2 RANDOMIZED-PARTITION'

1: $i = RANDOM(p, r)$
2: exchange $A[r]$ with $A[i]$
3: **return** PARTITION'(A,p,r)

Algorithm 3 QUICKSORT'(A,p,r)

1: **if** $p < r$ **then**
2:     $(q,t) = RANDOMIZED - PARTITION'$
3:     QUICKSORT'(A,p,q-1)
4:     QUICKSORT'(A,t+1,r)
5: **end if**

---

time is $O(d + X)$. It remains true that each pair of elements is compared at most once. If z_i is the ith smallest element, we need to compute the probability that z_i is compared to z_j. This time, once a pivot x is chosen with $z_i \leq x \leq z_j$, we know that z_i and z_j cannot be compared at any subsequent time. This is where the analysis differs, because there could be many elements of the array equal to z_i or z_j, so the probability that z_i and z_j are compared decreases. However, the expected percentage of distinct elements in a random array tends to $1 - \frac{1}{e}$, so asymptotically the expected number of comparisons is the same.

Problem 7-3

a. Since the pivot is selected as a random element in the array, which has size n, the probabilities of any particular element being selected are all equal, and add to one, so, are all $\frac{1}{n}$. As such, $E[X_i] = \Pr[\text{i smallest is picked}] = \frac{1}{n}$.

b. We can apply linearity of expectation over all of the events X_i. Suppose we have a particular X_i be true, then, we will have one of the sub arrays be length $i - 1$, and the other be $n - i$, and will of course still need linear time to run the partition procedure. This corresponds exactly to the summand in equation (7.5).

c.

$$E\left[\sum_{q=1}^{n} X_q(T(q-1) + T(n-q) + \Theta(n))\right] = \sum_{q=1}^{n} E\left[X_q(T(q-1) + T(n-q) + \Theta(n))\right]$$

$$= \sum_{q=1}^{n}(T(q-1) + T(n-q) + \Theta(n))/n = \Theta(n) + \frac{1}{n}\sum_{q=1}^{n}(T(q-1) + T(n-q))$$

$$= \Theta(n) + \frac{1}{n}\left(\sum_{q=1}^{n} T(q-1) + \sum_{q=1}^{n} T(n-q)\right)$$

$$= \Theta(n) + \frac{1}{n}\left(\sum_{q=1}^{n} T(q-1) + \sum_{q=1}^{n} T(q-1)\right) = \Theta(n) + \frac{2}{n}\sum_{q=1}^{n} T(q-1)$$

$$= \Theta(n) + \frac{2}{n}\sum_{q=0}^{n-1} T(q) = \Theta(n) + \frac{2}{n}\sum_{q=2}^{n-1} T(q)$$

8

d. We will prove this inequality in a different way than suggested by the hint. If we let $f(k) = k \lg(k)$ treated as a continuous function, then $f'(k) = \lg(k)+1$. Note now that the summation written out is the left hand approximation of the integral of $f(k)$ from 2 to n with step size 1. By integration by parts, the anti-derivative of $k \lg k$ is

$$\frac{1}{\ln(2)} \left(\frac{k^2}{2} \ln(k) - \frac{k^2}{4} \right)$$

So, plugging in the bounds and subtracting, we get $\frac{n^2 \lg(n)}{2} - \frac{n^2}{4\ln(2)} - 1$. Since f has a positive derivative over the entire interval that the integral is being evaluated over, the left hand rule provides a underapproximation of the integral, so, we have that

$$\sum_{k=2}^{n-1} k \lg(k) \leq \frac{n^2 \lg(n)}{2} - \frac{n^2}{4\ln(2)} - 1 \leq \frac{n^2 \lg(n)}{2} - \frac{n^2}{8}$$

where the last inequality uses the fact that $\ln(2) > 1/2$.

e. Assume by induction that $T(q) \leq q \lg(q) + \Theta(n)$. Combining (7.6) and (7.7), we have

$$E[T(n)] = \frac{2}{n} \sum_{q=2}^{n-1} E[T(q)] + \Theta(n) \leq \frac{2}{n} \sum_{q=2}^{n-1} (q \lg(q) + Theta(n)) + \Theta(n)$$

$$\leq \frac{2}{n} \sum_{q=2}^{n-1} (q \lg(q)) + \frac{2}{n}(n\Theta(n)) + \Theta(n)$$

$$\leq \frac{2}{n}(\frac{1}{2}n^2 \lg(n) - \frac{1}{8}n^2) + \Theta(n) = n \lg(n) - \frac{1}{4}n + \Theta(n) = n \lg(n) + \Theta(n)$$

Problem 7-4

a. We'll proceed by induction. For the base case, if A contains 1 element then $p = r$ so the algorithm terminates immediately, leave a single sorted element. Now suppose that for $1 \leq k \leq n-1$, TAIL-RECURSIVE-QUICKSORT correctly sorts an array A containing k elements. Let A have size n. We set q equal to the pivot and by the induction hypothesis, TAIL-RECURSIVE-QUICKSORT correctly sorts the left subarray which is of strictly smaller size. Next, p is updated to $q+1$ and the exact same sequence of steps follows as if we had originally called TAIL-RECURSIVE-QUICKSORT(A,q+1,n). Again, this array is of strictly smaller size, so by the induction hypothesis it correctly sorts $A[q + 1...n]$ as desired.

b. The stack depth will be $\Theta(n)$ if the input array is already sorted. The right subarray will always have size 0 so there will be $n - 1$ recursive calls before the while-condition $p < r$ is violated.

c. We modify the algorithm to make the recursive call on the smaller subarray to avoid building pushing too much on the stack:

Algorithm 4 MODIFIED-TAIL-RECURSIVE-QUICKSORT(A,p,r)

1: **while** $p < r$ **do**
2:   $q =$ PARTITION(A, p, r)
3:   **if** $q < \lfloor (r - p)/2 \rfloor$ **then**
4:     MODIFIED-TAIL-RECURSIVE-QUICKSORT$(A, p, q - 1)$
5:     $p = q + 1$
6:   **else**
7:     MODIFIED-TAIL-RECURSIVE-QUICKSORT$(A, q + 1, r)$
8:     $r = q - 1$
9:   **end if**
10: **end while**

Problem 7-5

a. p_i is the probability that a randomly selected subset of size three has the $A'[i]$ as it's middle element. There are 6 possible orderings of the three elements selected. So, suppose that S' is the set of three elements selected. We will compute the probability that the second element of S' is $A'[i]$ among all possible 3-sets we can pick, since there are exactly six ordered 3-sets corresponding to each 3-set, these probabilities will be equal. We will compute the probability that $S'[2] = A'[i]$. For any such S', we would need to select the first element from $[i - 1]$ and the third from $\{i + 1, \ldots, n\}$. So, there are $(i - 1)(n - i)$ such 3-sets. The total number of 3-sets is $\binom{n}{3} = \frac{n(n-1)(n-2)}{6}$. So,

$$p_i = \frac{6(n - i)(i - 1)}{n(n - 1)(n - 2)}$$

b. If we let $i = \lfloor \frac{n+1}{2} \rfloor$, the previous result gets us an increase of

$$\frac{6(\lfloor \frac{n-1}{2} \rfloor)(n - \lfloor \frac{n+1}{2} \rfloor)}{n(n - 1)(n - 2)} - \frac{1}{n}$$

in the limit n going to infinity, we get

$$\lim_{n \to \infty} \frac{\frac{6(\lfloor \frac{n-1}{2} \rfloor)(n - \lfloor \frac{n+1}{2} \rfloor)}{n(n-1)(n-2)}}{\frac{1}{n}} = \frac{3}{2}$$

c. To save the messiness, suppose n is a multiple of 3. We will approximate the sum as an integral, so,

$$\sum_{i=n/3}^{2n/3} p_i \approx \int_{n/3}^{2n/3} \frac{6(-x^2 + nx + x - n)}{n(n-1)(n-2)} dx = \frac{6(-7n^3/81 + 3n^3/18 + 3n^2/18 - n^2/3)}{n(n-1)(n-2)}$$

Which, in the limit n goes to infinity, is $\frac{13}{27}$ which is a constant that is larger than $1/3$ as it was in the original randomized quicksort implementation.

d. Since the new algorithm always has a "bad" choice that is within a constant factor of the original quicksort, it will still have a reasonable probability that the randomness leads us into a bad situation, so, it will still be $n \lg n$.

Problem 7-6

a. Our algorithm will be essentially the same as the modified randomized quicksort written in problem 2. We sort the a_i's, but we replace the comparison operators to check for overlapping intervals. The only modifications needed are made in PARTITION, so we will just rewrite that here. For a given element x in position i, we'll use $x.a$ and $x.b$ to denote a_i and b_i respectively.

Algorithm 5 FUZZY-PARTITION(A,p,r)

1: $x = A[r]$
2: exchange $A[r]$ with $A[p]$
3: $i = p - 1$
4: $k = p$
5: **for** $j = p + 1$ to $r - 1$ **do**
6:     **if** $b_j < x.a$ **then**
7:         $i = i + 1$
8:         $k = i + 2$
9:         exchange $A[i]$ with $A[j]$
10:         exchange $A[k]$ with $A[j]$
11:     **end if**
12:     **if** $b_j \geq x.a$ or $a_j \leq x.b$ **then**
13:         $x.a = \max(a_j, x.a)$ and $x.b = \min(b_j, x.b)$
14:         $k = k + 1$
15:         exchange $A[k]$ with $A[j]$
16:     **end if**
17: **end for**
18: exchange $A[i + 1]$ with $A[r]$
19: **return** $i + 1$ and $k + 1$

When intervals overlap we treat them as equal elements, thus cutting down on the time required to sort.

b. For distinct intervals the algorithm runs exactly as regular quicksort does, so its expected runtime will be $\Theta(n \lg n)$ in general. If all of the intervals overlap then the condition on line 12 will be satisfied for every iteration of the for loop. Thus the algorithm returns p and r, so only empty arrays remain to be sorted. FUZZY-PARTITION will only be called a single time, and since its runtime remains $\Theta(n)$, the total expected runtime is $\Theta(n)$.

Chapter 8

Michelle Bodnar, Andrew Lohr

May 5, 2017

Exercise 8.1-1

We can construct the graph whose vertex set is the indices, and we place an edge between any two indices that are compared on the shortest path. We need this graph to be connected, because otherwise we could run the algorithm twice, once with everything in one component less than the other component, and a second time with the everything in the second component larger. As long as we maintain the same relative ordering of the elements in each component, the algorithm will take exactly the same path, and so produce the same result. This means that there will be no difference in the output, even though there should be. For a graph on n vertices, it is a well known that at least $n-1$ edges are necessary for it to be connected, as the addition of an edge can reduce the number of connected components by at least one, and the graph with no edges has n connected components.

So, it will have depth at least $n-1$.

Exercise 8.1-2

Since $\lg(k)$ is monotonically increasing, we use formula $A.11$ to approximate the sum:

$$\int_0^n \lg(x)dx \le \sum_{k=1}^n \lg(k) \le \int_1^{n+1} \lg(x)dx.$$

From this we obtain the inequality

$$\frac{n\ln(n)-n}{\ln 2} \le \sum_{k=1}^n \lg(k) \le \frac{(n+1)\ln(n+1)-n}{\ln 2}$$

which is $\Theta(n\lg n)$.

Exercise 8.1-3
Suppose to a contradiction that there is a c_1 so that for every $n \ge k$, at least half of the inputs of length n have depth at most $c_1 n$. However, there are less than $2^{c_1 n+1}$ elements in the tree of depth at most $c_1 n$. However, $1/2n! > 1/2(n/e)^n > 2^{c_1 n+1}$ so long as $n > e2^{c_1}$. This is a contradiction.

To have a $1/n$ fraction of them with small depth, similarly, we get a contradiction because $1/nn! > 2^{c_1 n+1}$ for large enough n.

To make an algorithm that is linear for a $1/2^n$ fraction of inputs, we yet again get a contradiction because $2^{-n}n! > (n/2e)^n > 2^{c_1 n+1}$ for large enough n.

The moral of the story is that $n!$ grows very quickly.

Exercise 8.1-4

We assume as in the section that we need to construct a binary decision tree to represent comparisons. Since each subsequence is of length k, there are $k!^{n/k}$ possible output permutations. To compute the height h of the decision tree we must have $k!^{n/k} \leq 2^h$. Taking logs on both sides and using exercise 2 this gives

$$h \geq (n/k)\lg(k!) \geq (n/k)\left(\frac{k\ln k - k}{\ln 2}\right) = \frac{n\ln(k) - n}{\ln 2} = \Omega(n\lg k).$$

Exercise 8.2-1

We have that $C = \langle 2, 4, 6, 8, 9, 9, 11\rangle$. Then, after successive iterations of the loop on lines 10-12, we have $B = \langle , , , , , 2, , , , , \rangle$, $B = \langle , , , , , 2, , 3, , , \rangle$, $B = \langle , , , 1, , 2, , 3, , , \rangle$, and at the end, $B = \langle 0, 0, 1, 1, 2, 2, 3, 3, 4, 6, 6\rangle$

Exercise 8.2-2

Suppose positions i and j with $i < j$ both contain some element k. We consider lines 10 through 12 of COUNTING-SORT, where we construct the output array. Since $j > i$, the loop will examine $A[j]$ before examining $A[i]$. When it does so, the algorithm correctly places $A[j]$ in position $m = C[k]$ of B. Since $C[k]$ is decremented in line 12, and is never again incremented, we are guaranteed that when the for loop examines $A[i]$ we will have $C[k] < m$. Therefore $A[i]$ will be placed in an earlier position of the output array, proving stability.

Exercise 8.2-3

The algorithm still works correctly. The order that elements are taken out of C and put into B doesn't affect the placement of elements with the same key. It will still fill the interval $(C[k-1], C[k]]$ with elements of key k. The question of whether it is stable or not is not well phrased. In order for stability to make sense, we would need to be sorting items which have information other than their key, and the sort as written is just for integers, which don't. We could think of extending this algorithm by placing the elements of A into a collection of elements for each cell in array C. Then, if we use a FIFO collection, the modification of line 10 will make it stable, if we use LILO, it will be anti-stable.

Exercise 8.2-4

The algorithm will begin by preprocessing exactly as COUNTING-SORT does in lines 1 through 9, so that $C[i]$ contains the number of elements less than or equal to i in the array. When queried about how many integers fall into a range $[a..b]$, simply compute $C[b] - C[a-1]$. This takes $O(1)$ times and yields the desired output.

Exercise 8.3-1

Starting with the unsorted words on the left, and stable sorting by progressively more important positions.

COW	SEA	TAB	BAR
DOG	TEA	BAR	BIG
SEA	MOB	EAR	BOX
RUG	TAB	TAR	COW
ROW	RUG	SEA	DIG
MOB	DOG	TEA	DOG
BOX	DIG	DIG	EAR
TAB	BIG	BIG	FOX
BAR	BAR	MOB	MOB
EAR	EAR	DOG	NOW
TAR	TAR	COW	ROW
DIG	COW	ROW	RUG
BIG	ROW	NOW	SEA
TEA	NOW	BOX	TAB
NOW	BOX	FOX	TAR
FOX	FOX	RUG	TEA

Exercise 8.3-2

Insertion sort and merge sort are stable. Heapsort and quicksort are not. To make any sorting algorithm stable we can preprocess, replacing each element of an array with an ordered pair. The first entry will be the value of the element, and the second value will be the index of the element. For example, the array $[2, 1, 1, 3, 4, 4, 4]$ would become $[(2,1),(1,2),(1,3),(3,4),(4,5),(4,6),(4,7)]$. We now interpret $(i,j) < (k,m)$ if $i < k$ or $i = k$ and $j < m$. Under this definition of less-than, the algorithm is guaranteed to be stable because each of our new elements is distinct and the index comparison ensures that if a repeat element appeared later in the original array, it must appear later in the sorted array. This doubles the space requirement, but the running time will be asymptotically unchanged.

Exercise 8.3-3

After sorting on digit i, we will show that if we restrict to just the last i digits, the list is in sorted order. This is trivial for $i = 1$, because it is just claiming that the digits we just sorted were in sorted order. Now, suppose it's

true for $i - 1$, we show it for i. Suppose there are two elements, who, when restricted to the i last digits, are not in sorted order after the i'th step. Then, we must have that they have the same i'th digit because otherwise the sort of digit i would put them in the right order. Since they have the same first digit, their relative order is determined by their restrictions to their last $i - 1$ digits. However, these were placed in the correct order by the $i - 1$'st step. Since the sort on the i'th digit was stable, their relative order is unchanged from the previous step. This means that they are in the correct order still. We use stability to show that being in the correct order prior to doing the sort is preserved.

Exercise 8.3-4

First run through the list of integers and convert each one to base n, then radix sort them. Each number will have at most $\log_n(n^3) = 3$ digits so there will only need to be 3 passes. For each pass, there are n possible values which can be taken on, so we can use counting sort to sort each digit in $O(n)$ time.

Exercise 8.3-5

Since a pass consists of one iteration of the loop on line $1 - 2$, only d passes are needed. Since each of the digits can be one of ten decimal numbers, the most number of piles that would be needed to be kept track of is 10.

Exercise 8.4-1

The sublists formed are $\langle .13, .16 \rangle$, $\langle .20 \rangle, \langle .39 \rangle$, $\langle .42 \rangle$, $\langle .53 \rangle$, $\langle .64 \rangle$, $\langle .71, .79 \rangle$, $\langle .89 \rangle$. Putting them together, we get $\langle .13, .16, .20, .39, .42, .53, .64, .71, .79, .89 \rangle$

Exercise 8.4-2

In the worst case, we could have a bucket which contains all n values of the array. Since insertion sort has worst case running time $O(n^2)$, so does Bucket sort. We can avoid this by using merge sort to sort each bucket instead, which has worst case running time $O(n \lg n)$.

Exercise 8.4-3

X is 0 or 2 with probability a quarter each, and 1 with probability 2. Note that $E[X] = 1$, so, $E^2[x] = 1$. Also, X^2 takes 0 or 4 with probability a quarter each, and 1 with probability a half. So, $E[X^2] = 1.5$.

Exercise 8.4-4

Define $r_i = \sqrt{\frac{i}{n}}$ and $c_i = \{(x, y) | r_{i-1} \leq x^2 + y^2 \leq r_i\}$ for $i = 1, 2, \ldots, n$. The c_i regions partition the unit disk into n parts of equal area, which we will

use as the buckets. Since the points are uniformly distributed on the disk and each region has equal area, bucket sort will run in expected time $\Theta(n)$.

Exercise 8.4-5

We have to pick our bounds for our buckets in bucket sort in such a way that there is approximately equal probability that an element drawn from the distribution will be any one of the buckets. To do this, we can perform a binary search to find the Dyadic rationals with denominator at least a constant fraction of n. Finding each of these takes only $\lg(n)$ time. Then, these form the bounds for a bucket sort. There are an expected constant number of elements in any one of these buckets, so just use any sort you want at this point.

Problem 8-1

a. There are $n!$ possible permutations of the input array because the input elements are all distinct. Since each is equally likely, the distribution is uniformly supported on this set. So, each occurs with probability $\frac{1}{n!}$ and corresponds to a different leaf because the program needs to be able to distinguish between them.

b. The depths of particular elements of LT(resp. RT) are all one less than their depths when considered elements of T. In particular, this is true for the leaves of the two subtrees. Also, $\{LT, RT\}$ form a partition of all the leaves of T. So, if we let $L(T)$ denote the leaves of T,

$$D(T) = \sum_{\ell \in L(T)} D_T(\ell) = \sum_{\ell \in L(LT)} D_T(\ell) + \sum_{\ell \in L(RT)} D_T(\ell) =$$

$$\sum_{\ell \in L(LT)} (D_{LT}(\ell) + 1) + \sum_{\ell \in L(RT)} (D_{RT}(\ell) + 1) =$$

$$\sum_{\ell \in L(LT)} D_{LT}(\ell) + \sum_{\ell \in L(RT)} D_{RT}(\ell) + k = D(LT) + D(RT) + k$$

c. Suppose we have a T with k leaves so that $D(T) = d(k)$. Let i_0 be the number of leaves in LT. Then, $d(k) = D(T) = D(LT) + D(RT) + k$ but, we can pick LT and RT to minimize the external path length

d. We treat i as a continuous variable, and take a derivative to find critical points. The given expression has the following as a derivative with respect to i

$$\frac{1}{\ln(2)} + \lg(i) + \frac{1}{\ln(2)} - \lg(k - i) = \frac{2}{\ln(2)} + \lg\left(\frac{i}{k - i}\right)$$

which is zero when we have $\frac{i}{k-i} = 2^{-\frac{2}{\ln(2)}} = 2^{-\lg(e^2)} = e^{-2}$. So, $(1 + e^{-2})i = k$, so, $i = \frac{k}{1 + e^{-2}}$.

Since we are picking the two subtrees to be roughly equal size, the total depth will be order $\lg(k)$, with each level contributing k, so the total external path length is at least $k \lg(k)$.

e. Since before we that a tree with k leaves needs to have external length $k \lg(k)$, and that a sorting tree needs at least $n!$ trees, a sorting tree must have external tree length at least $n! \lg(n!)$. Since the average case run time is the depth of a leaf weighted by the probability of that leaf being the one that occurs, we have that the run time is at least $\frac{n! \lg(n!)}{n!} = \lg(n!) \in \Omega(n \lg(n))$

f. Since the expected runtime is the average over all possible results from the random bits, if every possible fixing of the randomness resulted in a higher runtime, the average would have to be higher as well.

Problem 8-2

Algorithm 1 O(n) and Stable(A)

a.   Create a new array C
    $index = 1$
    for $i = 1$ to n **do**
       if $A[i] == 0$ **then**
          $C[index] = A[i]$
          $index = index + 1$
       end if
    end for
    for $i = 1$ to n **do**
       if $A[i] == 1$ **then**
          $C[index] = A[i]$
          $index = index + 1$
       end if
    end for

This algorithm first selects all the elements with key 0 and puts them in the new array C in the order in which they appeared in A, then selects all the elements with key 1 and puts them in C in the order in which they appeared in A. Thus, it is stable. Since it only makes two passes through the array, it is $O(n)$.

The algorithm maintains the 0's seen so far at the start of the array. Each time an element with key 0 is encountered, the algorithm swaps it with the position following the last 0 already seen. This is in-place and $O(n)$, but not stable.

c. Simply run BubbleSort on the array.

Algorithm 2 O(n) and in place(A)

---
b. $index = 1$
    for $i = 1$ to n **do**
        if $A[i] == 0$ **then**
            Swap $A[i]$ with $A[index]$
            $index = index + 1$
        end if
    end for

---

d. Use the algorithm given in part a. For each of the b-bit keys it takes $O(n)$ and is stable, as is required by Radix-Sort. Thus, the total running time will be $O(bn)$.

e. Create the array C as done in lines 1 through 5 of counting sort. Create an array B which is a copy of C, then run lines 7 through 9 on B. In other words, $C[i]$ gives the number of elements in A which are equal to i, and $B[i]$ gives the number of elements in A which are less than or equal to i, so given an element, we can correctly identify where it belongs in the array. While the element in position i doesn't belong there, swap it with the element in the place it belongs. Once an element which belongs in position i appears, increment i. The preprocessing takes $O(n + k)$ time, the total number of swaps is at most n, and we iterate through the whole array once, so the overall runtime is $O(n + k)$. This algorithm has the advantage of sorting in place, however it is no longer stable like counting sort.

Problem 8-3

a. First, sort the integer by their lengths. This can be done efficiently using a bucket sort, where we make a bucket for each possible number of digits. We sort each these uniform length sets of integers using radix sort. Then, we just concatenate the sorted lists obtained from each bucket.

b. Make a bucket for every letter in the alphabet, each containing the words that start with that letter. Then, forget about the first letter of each of the words in the bucket, concatenate the empty word (if it's in this new set of words) with the result of recursing on these words of length one less. Since each word is processed a number of times equal to it's length, the runtime will be linear in the total number of letters.

Problem 8-4

a. Select a red jug. Compare it to blue jugs until you find one which matches. Set that pair aside, and repeat for the next red jug. This will use at most $\sum_{i=1}^{n-1} i = n(n - 1)/2 = O(n^2)$ comparisons.

7

b. We can imagine first lining up the red jugs in some order. Then a solution to this problem becomes a permutation of the blue jugs such that the ith blue jug is the same size as the ith red jug. As in section 8.1, we can make a decision tree which represents comparisons made between blue jugs and red jugs. An internal node represents a comparison between a specific pair of red and blue jugs, and a leaf node represents a permutation of the blue jugs based on the results of the comparison. We are interested in when one jug is greater than, less than, or equal in size to another jug, so the tree should have 3 children per node. Since there must be at least $n!$ leaf nodes, the decision tree must have height at least $log_3(n!)$. Since a solution corresponds to a simple path from root to leaf, an algorithm must make at least $\Omega(n \lg n)$ comparisons to reach any leaf.

c. We use an algorithm analogous to randomized quicksort. Select a blue jug at random. Partition the red jugs into those which are smaller than the blue jug, and those which are larger. At some point in the comparisons, you will find the red jug which is of equal size. Once the red jugs have been divided by size, use the red jug of equal size to partition the blue jugs into those which are smaller and those which are larger. If k red jugs are smaller than the originally chosen jug, we need to solve the original problem on input of size $k-1$ and size $n-k$, which we will do in the same manner. A subproblem of size 1 is trivially solved because if there is only one red jug and one blue jug, they must be the same size. The analysis of expected number of comparisons is exactly the same as that of randomized-quicksort given on pages 181-184. We are running the procedure twice so the expected number of comparisons is doubled, but this is absorbed by the big-O notation. In the worst case, we pick the largest jug each time, which results in $\sum_{i=2}^{n} i + i - 1 = n^2$ comparisons.

Problem 8-5

a. Since each of the averages will be of sets of a single element, so, each element will be $\leq$ than the next element. So, this is the usual definition of sorted

b. $\langle 1, 6, 2, 7, 3, 8, 4, 9, 10 \rangle$

c. Suppose we have $A[i] \leq A[i+k]$ for all appropriate i. Then, every element in the sum $\sum_{j=i}^{i+k-1} A[j]$ has a corresponding element in the sum $\sum_{j=i}^{i+k-1} A[j]$ that it is less than. This means the sums must have the desired inequality.

Now, suppose that we had the array was k-sorted. This means that

$$\sum_{j=i}^{i+k-1} A[j] \leq \sum_{j=i+1}^{i+k} A[j]$$

but, if we subtract off all the terms that both sums have in common, we get:

$$A[i] \leq A[i+k]$$

d. We can do quicksort until we are down to subarrays of size at most k. In exercise 7.4-5, this was shown to take time $O(n \lg(n/k))$. Note that this much work is all that is needed because we only need show that elements separated by k positions are in different blocks which is true because they are $\leq k$ positions wide.

e. Consider the lists, which, for every $i \in \{0, \ldots, k-1\}$, are $L_i = \langle A[i], A[i+k], \ldots, A[i+\lfloor n/k \rfloor k] \rangle$. Note that by part c, if A is $k-$ sorted, then we must have each L_i is sorted. Then, by 6.5-9, we can merge them all into a single sorted array in the desired time.

f. Let $t(k, n)$ be the time required to k-sort arrays of length n. Then, we have that, for every k, we can sort an array by first k sorting it and then applying the previous part in time $t(n, k) + n \lg(k)$. If $t(n, k)$ were $o(n \lg(n))$, then we would have a contradiction to the comparison based sorting lower bounds.

Problem 8-6

a. There are $\binom{2n}{n}$ ways to divide $2n$ numbers into two sorted lists, each with n numbers.

b. Any decision tree to merge two sorted lists must have at least $\binom{2n}{n}$ leaf nodes, so it has height at least $\lg\left(\frac{2n!}{n!n!}\right)$. We'll use the inequalities derived in Exercise 8.1-2 to bound this:

$$\lg\left(\frac{2n!}{n!n!}\right) = \lg((2n)!) - 2\lg(n!)$$

$$= \sum_{k=1}^{2n} \lg(k) - 2\sum_{k=1}^{n} \lg(k)$$

$$\geq \frac{2n\ln(2n) - 2n}{\ln 2} - 2\frac{(n+1)\ln(n+1) - n}{\ln 2}$$

$$= 2n + 2n\lg(n) - 2n\lg(n+1) - 2\lg(n+1)$$

$$= 2n + 2n\lg(n/(n+1)) - 2\lg(n+1)$$

$$= 2n - o(n).$$

c. If we don't compare them, then there is no way to distinguish between the original unmerged lists and the unmerged lists obtained by swapping those two items between lists. In the case where the two elements are different we require a different action to be taken in order to correctly merge the lists, so the two must be compared.

d. Let list $A = 1, 3, 5, ..., 2n-1$ and $B = 2, 4, 6, \ldots, 2n$. By part c, we must compare 1 with 2, 2 with 3, 3 with 4, and so on up until we compare $2n-1$ with $2n$. This amounts to a total of $2n-1$ comparisons.

9

Problem 8-7

a. Since we claim that $A[p]$ is the smallest value placed at a wrong location, everything that has value less than or equal to $A[p]$ is in it's correct spot, and so, cannot be placed where $A[p]$ should of been. This means that $A[q] \geq A[p]$. Also, if we had that $A[q] = A[p]$. Then it wouldn't be an error to have $A[q]$ in the spot that should of had $A[q]$. This gets us $A[q] > A[p]$ so $1 = B[q]$. Since equality puts us in the 0 case, we have $0 = B[p]$.

b. Let A_i be A after the first i operations, and B_i being B after the first i operations. So, $A = A_0$ and $B = B_0$. We are able to maintain the following invariant for all j and i

$$A_i[j] > A[p] \Leftrightarrow B_i[j] > 0 = B[p] \qquad (*)$$

Initially (at $i = 0$) it is true because we defined those elements larger the $A[p]$ to make a 1 at the corresponding position in B, similarly, those that aren't larger have a 0 at the corresponding position in B. Now, suppose at step $i + 1$ that we perform some compare swap operation between positions j_1 and j_2 with $j_1 < j_2$. By induction, it is not possible that $A_i[j_1] \leq A_i[j_2]$ while $B_i[j_1] > B_i[j_2]$ because this relation in B would mean that $A_i[j_1] > A[p] \geq A_i[j_2]$, a contradiction to our assumption on A. This leaves us with four cases:

Case 1: $A_i[j_1] \leq A_i[j_2]$ and $B_i[j_1] \leq B_i[j_2]$. Then no swap occurs in either array, that is $A_{i+1} = A_i$ and $B_{i+1} = B_i$. Clearly, the desired property continues to hold.

Case 2: $A_i[j_1] > A_i[j_2]$ and $B_i[j_1] \leq B_i[j_2]$. Then, a swap will occur in A but not in B. We also know that either $A_i[j_1]$ and $A_i[j_2]$ are both larger or both at most $A[p]$. Otherwise we would have $B_i[j_1] \neq B_i[j_2]$, which would mean $0 = B_i[j_1] < B_i[j_2] = 1$, which means $A_i[j_1] \leq A[p] < A_i[j_2]$, a contradiction to the case that we're in. Suppose we are in the both of $A_i[j_1]$ and $A_i[j_2]$ larger than $A[p]$ case, the both smaller case will be identical except for having the positions in the B array equal to 0 instead of 1. Since both are larger, we have $B_i[j_1] = B_i[j_2] = 1$, which means $A_{i+1}[j_2] = A_i[j_1] > A[p]$ and $A_{i+1}[j_2] = A_i[j_1] > A[p]$ check out.

Case 3: $A_i[j_1] > A_i[j_2]$ and $B_i[j_1] > B_i[j_2]$. A swap occurs in both A and B. If j is neither j_1 nor j_2, it is unchanged, so we don't need to consider it. So, for j_2 we have

$$A_i[j_1] > A[p] \Leftrightarrow B_i[j_1] > 0$$
$$A_{i+1}[j_2] > A[p] \Leftrightarrow B_{i+1}[j_2] > 0$$

With the j_1 case almost identical. Completing our proof of claim (*).

To put it all together, let i be the total number of operations, and let j be the position that $A[p]$ ends up at, and k be the position that it should of

ended up at. We know that $k < j$ because $A[p]$ was the smallest incorrectly sorted element. If it were in a lower position than it should of been it would be taking the spot belonging to a smaller value. If it were in a position that should belong to something with an equal value than it wouldn't be incorrectly sorted. By definition $A[q] = A_i[k]$, $A[p] = A_i[j]$. By part a, we have $A_i[j] > A[p]$. By the invariant we just proved, $B_i[j] > 0$, and $B[k] = 0$. Therefore, B is not sorted.

c. All of the even steps in the algorithm do not even look at the values of the cells, they just push them around. The odd steps are also oblivious because we can just use the oblivious compare exchange version of insertion sort giving at the beginning of the problem.

d. After step 1, we know that each column looks like some number of zeros followed by some number of ones. Suppose that column i had z_i zeros in it. Then, after the reshaping step, we know that each of the columns will contribute $\lceil z_i/(r/s) \rceil$ zeros to the first $z_i \mod s$ columns, and one less to the rest. In particular, since the number of zeros contributed to each of the columns after step 2 only has a single jump by one, the sum over each must only change by at most s. Then, after sorting, this means that there will only be s dirty rows.

e. From the prior part, we know that before step 4, there are at most s dirty rows, with a total of s^2 elements in them. So, after step 4, these dirty elements will map to some bottom part of a column, some of another column, and then some top part of a column. with everything to the left being zero, and everything to the right being one.

f. From the previous part, we have that there are at most s^2 values that may be bad. Also, $r \geq 2s^2$. Then, there are two cases. The first is that the dirty region spans two columns, in this case, we have that after step 6, the dirty region is entirely in one column. So, after doing step 7, we have that the column that was dirty has a clean top half, and the array is sorted, so, it remains sorted when applying step 8. The second case is that the dirty region is entirely in one column. If this is the case, then, after sorting in step 5, the array is already sorted, and the only dirty column has a clean first half. So, it will maintain it's being sorted after performing steps 6-8.

g. The proof is very similar to the way that part e was shown. There must be some care taken with how the transformation in steps 2 and 4 are changed. We would need that $r \geq 2s^2 - 2s$, since a dirty region of $2s - 1$ rows corresponds to a dirty region (when read column major) of size $2s^2 - s$. we could simply pad the array with a number of rows with all zero, to bring the number of rows up to a multiple of s. Then, take away that many zeroes from the final answer.

Chapter 9

Michelle Bodnar, Andrew Lohr

April 12, 2016

Exercise 9.1-1

In this problem, we will be recursing by dividing the array into two equal size sets of elements, we will neglect taking floors and ceilings. The analysis will be the same but just a bit uglier if we don't assume that n is a power of 2.

Break up the elements into disjoint pairs. Then, compare each pair, consider only the smallest elements from each. From among this set of elements, the result to the original problem will be either what had been paired with the smallest element, or what what is the second smallest element of the sub-problem. Doing this will get both the smallest and second smallest element. So, we get the recurrence $T(n) = T(n/2) + n/2 + 1$ and $T(2) = 1$. So, solving this recurrence, we will use the substitution method with $T(n) \leq n + \lceil \lg(n) \rceil - 2$, it agrees with the base case, and, $T(n) = n/2 + T(n/2) + 1 \leq n/2 + n/2 + \lceil \lg(n/2) \rceil - 2 + 1 = n + \lceil \lg(n) \rceil - 2$ as desired.

Exercise 9.1-2

Initially, all n numbers are potentially either the maximum or minimum. Let MAX be the set of numbers which are potentially the maximum and MIN denote the set of numbers which are potentially the minimum. Say we compare two elements a and b. If $a \leq b$, we can remove a from MAX and remove b from MIN, so we reduce the counts of both sets by 1. If we compare two elements in MIN, we reduce the size of MIN by 1, and if we compare two elements in MAX, we can reduce the size of MAX by 1. Thus, the first type of comparison is optimal. There are $\lceil n/2 \rceil$ such comparisons which can be made until MIN and MAX are disjoint, and the sets will have sizes $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$. Within each of these, only the second and third types of comparisons can be made, each reducing the set size by 1, so we require a total of $\lceil n/2 \rceil - 1 + \lfloor n/2 \rfloor - 1 = n - 2$ comparisons. Adding this to initial $\lceil n/2 \rceil$ comparisons gives $\lceil 3n/2 \rceil - 2$.

Exercise 9.2-1

Calling a zero length array would mean that the second and third arguments are equal. So, if the call is made on line 8, we would need that $p = q - 1$. which

means that $q - p + 1 = 0$ .However, i is assumed to be a nonnegative number, and to be executing line 8, we would need that $i < k = q - p + 1 = 0$, a contradiction. The other possibility is that the bad recursive call occurs on line 9. This would mean that $q + 1 = r$. To be executing line 9, we need that $i > k = q - p + 1 = r - p$. This would be a nonsensical original call to the array though because we are asking for the ith element from an array of strictly less size.

Exercise 9.2-2

The probability that X_k is equal to 1 is unchanged when we know the max of $k-1$ and $n-k$. In other words, $P(X_k = a | \max(k-1, n-k) = m) = P(X_k = a)$ for $a = 0, 1$ and $m = k - 1, n - k$ so X_k and $\max(k-1, n-k)$ are independent. By C.3-5, so are X_k and $T(\max(k-1, n-k))$.

Exercise 9.2-3

Algorithm 1 ITERATIVE-RANDOMIZED-SELECT

---
while $p < r$ **do**
    $q = RANDOMIZED - PARTITION(A, p, r)$
    $k = q - p + 1$
    if i=k **then**
        return $A[q]$
    end if
    if $i < k$ **then**
        $r = q - 1$
    else
        $p = q$
        $i = i - k$
    end if
end while
return $A[p]$

Exercise 9.2-4

When the partition selected is always the maximum element of the array we get worst-case performance. In the example, the sequence would be 9, 8, 7, 6, 5, 4, 3, 2, 1, 0.

Exercise 9.3-1

It will still work if they are divided into groups of 7, because we will still know that the median of medians is less than at least 4 elements from half of the $\lceil n/7 \rceil$ groups, so, it is greater than roughly $4n/14$ of the elements. Similarly, it

is less than roughly $4n/14$ of the elements. So, we are never calling it recursively on more than $10n/14$ elements. So, $T(n) \leq T(n/7) + T(10n/14) + O(n)$. So, we can show by substitution this is linear. Suppose $T(n) < cn$ for $n < k$, then, for $m \geq k$, $T(m) \leq T(m/7) + T(10m/14) + O(m) \leq cm(1/7 + 10/14) + O(m)$. So, as long as we have that the constant hidden in the big-Oh notation is less than $c/7$, we have the desired result.

Suppose now that we use groups of size 3 instead. So, For similar reasons, we have that the recurrence we are able to get is $T(n) = T(\lceil n/3 \rceil) + T(4n/6) + O(n) \geq T(n/3) + T(2n/3) + O(n)$. So, we will show it is $\geq cn \lg(n)$.
$\quad T(m) \geq c(m/3)\lg(m/3) + c(2m/3)\lg(2m/3) + O(m) \geq cm \lg(m) + O(m)$. So, we have that it grows more quickly than linear.

Exercise 9.3-2

We know that the number of elements greater than or equal to x and the number of elements less than or equal to x is at least $3n/10 - 6$. Then for $n \geq 140$ we have

$$3n/10 - 6 = \frac{n}{4} + \frac{n}{20} - 6 \geq n/4 + 140/20 - 6 = n/4 + 1 \geq \lceil n/4 \rceil.$$

Exercise 9.3-3

We can modify quicksort to run in worst case $n \lg(n)$ time by choosing our pivot element to be the exact median by using quick select. Then, we are guaranteed that our pivot will be good, and the time taken to find the median is on the same order of the rest of the partitioning.

Exercise 9.3-4

Create a graph with n vertices and draw a directed edge from vertex i to vertex j if the ith and j^{th} elements of the array are compared in the algorithm and we discover that $A[i] \geq A[j]$. Observe that $A[i]$ is one of the $i - 1$ smaller elements if there exists a path from x to i in the graph, and $A[i]$ is one of the $n-i$ larger elements if there exists a path from i to x in the graph. Every vertex i must either lie on a path to or from x because otherwise the algorithm can't distinguish between $i \leq x$ and $i \geq x$. Moreover, if a vertex i lies on both a path to x and a path from x then it must be such that $x \leq A[i] \leq x$, so $x = A[i]$. In this case, we can break ties arbitrarily.

Exercise 9.3-5

To use it, just find the median, partition the array based on that median. If i is less than half the length of the original array, recurse on the first half, if i is half the length of the array, return the element coming from the median finding

black box. Lastly, if i is more than half the length of the array, subtract half the length of the array, and then recurse on the second half of the array.

Exercise 9.3-6

Without loss of generality assume that n and k are powers of 2. We first find the $n/2^{th}$ order statistic, in time $O(n)$ using SELECT, then reduce the problem to finding the $k/2^{th}$ quantiles of the smaller $n/2$ elements and the $k/2^{th}$ quantiles of the larger $n/2$ elements. Let $T(n)$ denote the time it takes the algorithm to run on input of size n. Then $T(n) = cn + 2T(n/2)$ for some constant c, and the base case is $T(n/k) = O(1)$. Then we have:

$$
\begin{aligned}
T(n) &\leq cn + 2T(n/2) \\
&\leq 2cn + 4T(n/4) \\
&\leq 3cn + 8T(n/8) \\
&\vdots \\
&\leq \log(k)cn + kT(n/k) \\
&\leq \log(k)cn + O(k) \\
&= O(n\log k).
\end{aligned}
$$

Exercise 9.3-7

Find the $n/2 - k/2$ largest element in linear time. Partition on that element. Then, find the k largest element in the bigger subarray formed from the partition. Then, the elements in the smaller subarray from partitioning on this element are the desired k numbers.

Exercise 9.3-8

Without loss of generality, assume n is a power of 2.

Algorithm 2 Median(X,Y,n)

---
if n==1 **then**
    return $\min(X[1], Y[1])$
end if
if $X[n/2] < Y[n/2]$ **then**
    return $Median(X[n/2+1...n], Y[1...n/2], n/2)$
else
    if $X[n/2] \geq Y[n/2]$ **then**
        return $Median(X[1...n/2], Y[n/2+1...n], n/2)$
    end if
end if

Exercise 9.3-9

If n is odd, then, we pick the y coordinate of the main pipeline to be equal to the median of all the y coordinates of the wells. if n is even, then, we can pick the y coordinate of the pipeline to be anything between the y coordinates of the wells with y-coordinates which have order statistics $\lfloor (n+1)/2 \rfloor$ and the $\lceil (n+1)/2 \rceil$. These can all be found in linear time using the algorithm from this section.

Problem 9-1

a. Sorting takes time $n \lg(n)$, and listing them out takes time i, so the total runtime is $O(n \lg(n) + i)$

b. Heapifying takes time $n \lg(n)$, and each extraction can take time $\lg(n)$, so, the total runtime is $O((n + i) \lg(n))$

c. Finding and partitioning around the ith largest takes time n. Then, sorting the subarray of length i coming from the partition takes time $i \lg(i)$. So, the total runtime is $O(n + i \lg(i))$.

Problem 9-2

a. Let m_k be the number of x_i smaller than x_k. When weights of $1/n$ are assigned to each x_i, we have $\sum_{x_i < x_k} w_i = m_k/n$ and $\sum_{x_i > x_k} w_i = (n - m_k - 1)/2$. The only value of m_k which makes these sums $< 1/2$ and $\le 1/2$ respectively is when $\lceil n/2 \rceil - 1$, and this value of x must be the median since it has equal numbers of $x_i's$ which are larger and smaller than it.

b. First use mergesort to sort the x_i's by value in $O(n \log n)$ time. Let S_i be the sum of the weights of the first i elements of this sorted array and note that it is $O(1)$ to update S_i. Compute $S_1, S_2, \dots$ until you reach k such that $S_{k-1} < 1/2$ and $S_k \ge 1/2$. The weighted median is x_k.

c. We modify SELECT to do this in linear time. Let x be the median of medians. Compute $\sum_{x_i < x} w_i$ and $\sum_{x_i > x} w_i$ and check if either of these is larger than $1/2$. If not, stop. If so, recurse on the collection of smaller or larger elements known to contain the weighted median. This doesn't change the runtime, so it is $\Theta(n)$.

d. Let p be the minimizer, and suppose that p is not the weighted median. Let ϵ be small enough such that $\epsilon < \min_i(|p - p_i|)$, where we don't include k if $p = p_k$. If p_m is the weighted median and $p < p_m$, choose $\epsilon > 0$. Otherwise

choose $\epsilon < 0$. Then we have

$$\sum_{i=1}^{n} w_i d(p + \epsilon, p_i) = \sum_{i=1}^{n} w_i d(p, p_i) + \epsilon \left(\sum_{p_i < p} w_i - \sum_{p_i > p} w_i \right) < \sum_{i=1}^{n} w_i d(p, p_i)$$

since the difference in sums will take the opposite sign of epsilon.

e. Observe that

$$\sum_{i=1}^{n} w_i d(p, p_i) = \sum_{i=1}^{n} w_i |p_x - (p_i)_x| + \sum_{i=1}^{n} w_i |p_y - (p_i)_y|.$$

It will suffice to minimize each sum separately, which we can do since we choose p_x and p_y individually. By part e, we simply take $p = (p_x, p_y)$ to be such that p_x is the weighted median of the x-coordinates of the p_i's and p_y is the weighted medain of the y-coordiantes of the p_i's.

Problem 9-3

a. If $i \geq n/2$, then just use the algorithm from this chapter to get the answer in time $T(n)$. If $i < n/2$, then, we can compare disjoint pairs of elements from the list, and then we know that the ith smallest is in the set of elements that are smaller in each pair. So, we can recurse, this gets us runtime in this case of $\lfloor n/2 \rfloor + U_i(\lceil n/2 \rceil) + T(2i)$. Note that the last term comes from the fact that the ith smallest could of also been any of the elements paired with the ith smallest elements from the subproblem.

b. By the Substitution method, suppose that $U_i(n) = n + cT(2i)\lg(n/i)$ for smaller n, then, there are two cases based on whether or not $i < n/4$. If it is, $U_i(n) = \lfloor n/2 \rfloor + U_i(\lceil n/2 \rceil) + T(2i) \leq n/2 + n/2 + cT(2i)\lg(n/2i) + T(2i)$. This then satisfies the recurrence if we have that $c \geq 1$. The other case is that $n/4 \leq i < n/2$. In this case, we have that $U_i(n) = n/2 + T(\lceil n/2 \rceil) + T(2i) \leq n/2 + 2T(2i)$, which works if we have $c \geq 2$. So, we can just pick $c = 2$, and both cases of the recurrence go through.

c. From the previous part, if i is a constant, the $O(T(2i)\lg(n/i))$ becomes $T(\lg(n))$. So, $U_i(n) = n + O(T(2i)\lg(n/i)) = n + O(\lg(n))$.

d. From part c, we just substitute in n/k for i to get $U_i(n) = n + O(T(2i)\lg(n/i)) = n + O(T(2n/k)\lg k)$.

Problem 9-4

a. We only need to worry about what happens when we select a pivot element which is between $\min(z_i, z_j, z_k)$ and $\max(z_i, z_j, z_k)$, since at this point we will either select z_i or z_j and compare them, select an element between z_i and z_j and never compare them, or select an element between z_k and the $[z_i, z_j]$ interval, so that we will will never again be selecting pivots from a range containing z_i and z_j. We split into three cases. If $z_k \leq z_i < z_j$ then $E(X_{ijk}) = \frac{2}{j-k+1}$. If $z_i \leq z_k < z_j$ then $E(X_{ijk}) = \frac{2}{j-i+1}$. If $z_i < z_j \leq z_k$ then $E(X_{ijk}) = \frac{2}{k-i+1}$.

b.

$$E[X_k] \leq \sum_{i=1}^{n} \sum_{j=1}^{n} E[X_{ijk}]$$

$$= \sum_{i=1}^{k} \sum_{j=k}^{n} \frac{2}{j-i+1} + \sum_{i=k+1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-k+1} + \sum_{i=1}^{k-2} \sum_{j=i+1}^{k-1} \frac{2}{k-i+1}.$$

For the second term, fix some value m. The term $\frac{2}{m-k+1}$ will appear once for every time j doesn't exceed m, so $m - (k+1)$ times in the sum. For the third term, each term in the sum doesn't depend on j so we can rewrite it as $2\sum_{i=1}^{k-2} \frac{k-i-1}{k-i+1}$. This gives

$$E[X_k] = 2 \left(\sum_{i=1}^{k} \sum_{j=k}^{n} \frac{1}{j-i+1} + \sum_{j=k+1}^{n} \frac{j-k-1}{j-k+1} + \sum_{i=1}^{k-2} \frac{k-i-1}{k-i+1} \right).$$

c. We can bound the summands of the second and third terms from (b) by 1, so the total contribution of these terms is $n - (k+1) + 1 + k - 2 = n - 2$. For the first double sum, consider a term of the form $\frac{1}{c}$ for some c. There are at most c of these, because we must have $j - i = c - 1$. The largest term which appears is 1 and the smallest term which appears is $\frac{1}{n}$, so the double sum contributes a total of at most n. Thus, $E[X_k] \leq 2(n + n - 2) \leq 4n$.

d. The running time of RANDOMIZED-SELECT is dominated by the time spent in RANDOMIZED-PARTITION, whose running time is dominated by comparisons. By part (c) we know that the expected number of comparisons over the entire algorithm, including recursions, is $O(n)$.

Chapter 10

Michelle Bodnar, Andrew Lohr

April 12, 2016

Exercise 10.1-1

4		
4	1	
4	1	3
4	1	
4	1	8
4	1	

Exercise 10.1-2

We will call the stacks T and R. Initially, set $T.top = 0$ and $R.top = n+1$. Essentially, stack T uses the first part of the array and stack R uses the last part of the array. In stack T, the top is the rightmost element of T. In stack R, the top is the leftmost element of R.

Algorithm 1 PUSH(S,x)

1: **if** $S == T$ **then**
2:     **if** $T.top + 1 == R.top$ **then**
3:         **error** "overflow"
4:     **else**
5:         $T.top = T.top + 1$
6:         $T[T.top] = x$
7:     **end if**
8: **end if**
9: **if** $S == R$ **then**
10:     **if** $R.top - 1 == T.top$ **then**
11:         **error** "overflow"
12:     **else**
13:         $R.top = R.top - 1$
14:         $T[T.top] = x$
15:     **end if**
16: **end if**

Algorithm 2 POP(S)

> **if** $S == T$ **then**
> > **if** $T.top == 0$ **then**
> > > **error** "underflow"
> >
> > **else**
> > > $T.top = T.top - 1.$
> > > **return** $T[T.top + 1]$
> >
> > **end if**
>
> **end if**
> **if** $S == R$ **then**
> > **if** $R.top == n + 1$ **then**
> > > **error** "underflow"
> >
> > **else**
> > > $R.top = R.top + 1.$
> > > **return** $R[R.top - 1]$
> >
> > **end if**
>
> **end if**

Exercise 10.1-3

4			
4	1		
4	1	3	
	1	3	
	1	3	8
		3	8

Exercise 10.1-4

Algorithm 3 ENQUEUE

> **if** $Q.head == Q.tail + 1$, or $Q.head == 1$ and $Q.tail == Q.length$ **then**
> > **error** "overflow"
>
> **end if**
> $Q[Q.tail] = x$
> **if** $Q.tail == Q.length$ **then**
> > $Q.tail = 1$
>
> **else**
> > $Q.tail = Q.head + 1$
>
> **end if**

Exercise 10.1-5

As in the example code given in the section, we will neglect to check for overflow and underflow errors.

Algorithm 4 DEQUEUE

if $Q.tail == Q.head$ **then**
    error "underflow"
end if
$x = Q[Q.head]$
if $Q.head == Q.length$ **then**
    $Q.head = 1$
else
    $Q.head = Q.head + 1$
end if
return x

Algorithm 5 HEAD-ENQUEUE(Q,x)

$Q[Q.head] = x$
if $Q.head == 1$ **then**
    $Q.head = Q.length$
else
    $Q.head = Q.head - 1$
end if

Algorithm 6 TAIL-ENQUEUE(Q,x)

$Q[Q.tail] = x$
if $Q.tail == Q.length$ **then**
    $Q.tail = 1$
else
    $Q.tail = Q.tail + 1$
end if

Algorithm 7 HEAD-DEQUEUE(Q,x)

$x = Q[Q.head]$
if $Q.head == Q.length$ **then**
    $Q.head = 1$
else
    $Q.head = Q.head + 1$
end if

Algorithm 8 TAIL-DEQUEUE(Q,x)

$x = Q[Q.tail]$
if $Q.tail == 1$ **then**
    $Q.tail = Q.length$
else
    $Q.tail = Q.tail - 1$
end if

Exercise 10.1-6

The operation enqueue will be the same as pushing an element on to stack 1. This operation is $O(1)$. To dequeue, we pop an element from stack 2. If stack 2 is empty, for each element in stack 1 we pop it off, then push it on to stack 2. Finally, pop the top item from stack 2. This operation is $O(n)$ in the worst case.

Exercise 10.1-7

The following is a way of implementing a stack using two queues, where pop takes linear time, and push takes constant time. The first of these ways, consists of just enqueueing each element as you push it. Then, to do a pop, you dequque each element from one of the queues and place it in the other, but stopping just before the last element. Then, return the single element left in the original queue.

Exercise 10.2-1

To insert an element in constant time, just add it to the head by making it point to the old head and have it be the head. To delete an element, it needs linear time because there is no way to get a pointer to the previous element in the list without starting at the head and scanning along.

Exercise 10.2-2

The PUSH(L,x) operation is exactly the same as LIST-INSERT(L,x). The POP operation sets x equal to $L.head$, calls LIST-DELETE(L,L.head), then returns x.

Exercise 10.2-3

In addition to the head, also keep a pointer to the last element in the linked list. To enqueue, insert the element after the last element of the list, and set it to be the new last element. To dequeue, delete the first element of the list and return it.

Exercise 10.2-4

First let $L.nil.key = k$. Then run LIST-SEARCH' as usual, but remove the check that $x \neq L.nil$.

Exercise 10.2-5

To insert, just do list insert before the current head, in constant time. To search, start at the head, check if the element is the current node being inspected, check the next element, and so on until at the end of the list or you

found the element. This can take linear time in the worst case. To delete, again linear time is used because there is no way to get to the element immediately before the current element without starting at the head and going along the list.

Exercise 10.2-6

Let L_1 be a doubly linked list containing the elements of S_1 and L_2 be a doubly linked list containing the elements of S_2. We implement UNION as follows: Set $L_1.nil.prev.next = L_2.nil.next$ and $L_2.nil.next.prev = L_1.nil.prev$ so that the last element of L_1 is followed by the first element of L_2. Then set $L_1.nil.prev = L_2.nil.prev$ and $L_2.nil.prev.next = L_1.nil$, so that $L_1.nil$ is the sentinel for the doubly linked list containing all the elements of L_1 and L_2.

Exercise 10.2-7

Algorithm 9 REVERSE(L)

$a = L.head.next$
$b = L.head$
while $a \neq NIL$ **do**
    $tmp = a.next$
    $a.next = b$
    $b = a$
    $a = tmp$
end while
$L.head = b$

Exercise 10.2-8

We will store the pointer value for $L.head$ separately, for convenience. In general, A XOR $(A$ XOR $C) = C$, so once we know one pointer's true value we can recover all the others (namely $L.head$) by applying this rule. Assuming there are at least two elements in the list, the first element will contain exactly the address of the second.

Algorithm 10 LISTnp-SEARCH(L,k)

$p = NIL$
$x = L.head$
while $x \neq NIL$ and $x.key \neq k$ **do**
    $temp = x$
    $x = pXORx.np$
    $p = temp$
end while

To reverse the list, we simply need to make the head be the "last" ele-

---

Algorithm 11 LISTnp-INSERT(L,x)

$x.np = L.head$
$L.nil.np = xXOR(L.nil.npXORL.head)$
$L.head = x$

Algorithm 12 LISTnp-Delete(L,x)

$L.nil.np = L.nil.npXORL.headXORL.head.np$
$L.head.np.np = L.head.np.npXORL.head$

---

ment before $L.nil$ instead of the first one after this. This is done by setting $L.head = L.nil.npXORL.head.$

Exercise 10.3-1

A multiple array version could be $L = 2$,

/	3	4	5	6	7	/
	12	4	8	19	5	11
	/	2	3	4	5	6

A single array version could be $L = 4$,

			12	7	/	4	10	4	8	13	7	19	16	10	5	19	13	11	/	16

Exercise 10.3-2

Algorithm 13 Allocate-Object()

if $free == NIL$ **then**
    error "out of space"
else
    $x = free$
    $free = A[x + 1]$
end if

Exercise 10.3-3

Allocate object just returns the index of some cells that it's guaranteed to not give out again until they've been freed. The prev attribute is not modified because only the next attribute is used by the memory manager, it's up to the code that calls allocate to use the prev and key attributes as it sees fit.

Exercise 10.3-4

For ALLOCATE-OBJECT, we will keep track of the next available spot in the array, and it will always be one greater than the number of elements being stored. For FREE-OBJECT(x), when a space is freed, we will decrement the

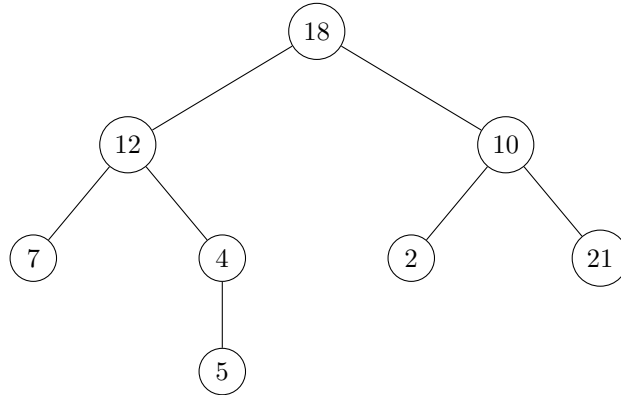| **Algorithm 14** Free-Object(x) |
|---|
| $A[x + 1] = free$ |
| $free = x$ |

position of each element in a position greater than that of x by 1 and update pointers accordingly. This takes linear time.

Exercise 10.3-5

See the algorithm $COMPACTIFY - LIST(L, F)$

Exercise 10.4-1

Note that indices 8 and 2 in the array do not appear, and, in fact do not represent a valid tree.

Exercise 10.4-2

See the algorithm PRINT-TREE.

Exercise 10.4-3

Exercise 10.4-4

See the algorithm PRINT-TREE.

Exercise 10.4-5

See the algorithm INORDER-PRINT'(T)

Exercise 10.4-6

Our two pointers will be $left$ and $right$. For a node x, $x.left$ will point to the leftmost child of x and $x.right$ will point to the sibling of x immediately to its right, if it has one, and the parent of x otherwise. Our boolean value b, stored at x, will be such that $b = depth(x) \mod 2$. To reach the parent of a node, simply keep following the "right" pointers until the parity of the boolean value changes. To find all the children of a node, start by finding $x.left$, then follow

Algorithm 15 COMPACTIFY-LIST(L,F)

---

  if n=m **then**
    return
  end if
  $e = \max\{\max_{i \in [m]}\{|key[i]|\}, \max_{i \in L}\{|key[i]|\}\}$
  increase every element of $key[1..m]$ by $2e$
  for every element of L, if its key is greater than e, reduce it by $2e$
  $f = 1$
  while $key[f] < e$ **do**
    $f + +$
  end while
  $a = L.head$
  if $a > m$ **then**
    $next[prev[f]] = next[f]$
    $prev[next[f]] = prev[f]$
    $next[f] = next[a]$
    $key[f] = key[a]$
    $prev[f] = prev[a]$
    $FREE - OBJECT(a)$
    $f + +$
    while $key[f] < e$ **do**
      $f + +$
    end while
  end if
  while $a \neq L.head$ **do**
    if $a > m$ **then**
      $next[prev[f]] = next[f]$
      $prev[next[f]] = prev[f]$
      $next[f] = next[a]$
      $key[f] = key[a]$
      $prev[f] = prev[a]$
      $FREE - OBJECT(a)$
      $f + +$
      while $key[f] < e$ **do**
        $f + +$
      end while
    end if
  end while

Algorithm 16 PRINT-TREE(T.root)

---
if $T.root == NIL$ **then**
    return
else
    Print $T.root.key$
    PRINT-TREE(T.root.left)
    PRINT-TREE(T.root.right)
end if

Algorithm 17 INORDER-PRINT(T)

---
let S be an empty stack
$push(S, T)$
while S is not empty **do**
    $U = pop(S)$
    if $U \neq NIL$ **then**
        print $U.key$
        $push(S, U.left)$
        $push(S, U.right)$
    end if
end while

Algorithm 18 PRINT-TREE(T.root)

---
if $T.root == NIL$ **then**
    return
else
    Print $T.root.key$
    $x = T.root.left - child$
    while $x \neq NIL$ **do**
        PRINT-TREE(x)
        $x = x.right - sibling$
    end while
end if

Algorithm 19 INORDER-PRINT'(T)

$a = T.left$
$prev = T$
while $a \neq T$ **do**
    if $prev = a.left$ **then**
        print $a.key$
        $prev = a$
        $a = a.right$
    else if $prev = a.right$ **then**
        $prev = a$
        $a = a.p$
    else if $prev = a.p$ **then**
        $prev = a$
        $a = a.left$
    end if
end while
print $T.key$
$a = T.right$
while $a \neq T$ **do**
    if $prev = a.left$ **then**
        print $a.key$
        $prev = a$
        $a = a.right$
    else if $prev = a.right$ **then**
        $prev = a$
        $a = a.p$
    else if $prev = a.p$ **then**
        $prev = a$
        $a = a.left$
    end if
end while

---

the "right" pointers until the parity of the boolean value changes, ignoring this last node since it will be x.

Problem 10-1

For each, we assume sorted means sorted in ascending order

	$unsorted, single$	$sorted, single$	$unsorted, double$	$sorted, double$
$SEARCH(L, k)$	n	n	n	n
$INSERT(L, x)$	1	1	1	1
$DELETE(L, x)$	n	n	1	1
$SUCCESSOR(L, x)$	n	1	n	1
$PREDECESSOR(L, x)$	n	n	n	1
$MINIMUM(L, x)$	n	1	n	1
$MAXIMUM(L, x)$	n	n	n	1

Problem 10-2

In all three cases, MAKE-HEAP simply creates a new list L, sets $L.head = NIL$, and returns L in constant time. Assume lists are doubly linked. To realize a linked list as a heap, we imagine the usual array implementation of a binary heap, where the children of the ith element are $2i$ and $2i + 1$.

a. To insert, we perform a linear scan to see where to insert an element such that the list remains sorted. This takes linear time. The first element in the list is the minimum element, and we can find it in constant time. Extract-min returns the first element of the list, then deletes it. Union performs a merge operation between the two sorted lists, interleaving their entries such that the resulting list is sorted. This takes time linear in the sum of the lengths of the two lists.

b. To insert an element x into the heap, begin linearly scanning the list until the first instance of an element y which is strictly larger than x. If no such larger element exists, simply insert x at the end of the list. If y does exist, replace yt by x. This maintains the min-heap property because $x \leq y$ and y was smaller than each of its children, so x must be as well. Moreover, x is larger than its parent because y was the first element in the list to exceed x. Now insert y, starting the scan at the node following x. Since we check each node at most once, the time is linear in the size of the list. To get the minimum element, return the key of the head of the list in constant time.

To extract the minimum element, we first call MINIMUM. Next, we'll replace the key of the head of the list by the key of the second smallest element y in the list. We'll take the key stored at the end of the list and use it to replace the key of y. Finally, we'll delete the last element of the list, and call MIN-HEAPIFY on the list. To implement this with linked lists, we need to step through the list to get from element i to element $2i$. We omit this detail from the code, but we'll consider it for runtime analysis. Since the value of i on which MIN-HEAPIFY is called is always increasing and we never need

to step through elements multiple times, the runtime is linear in the length of the list.

Algorithm 20 EXTRACT-MIN(L)

$min = MINIMUM(L)$
Linearly scan for the second smallest element, located in position i.
$L.head.key = L[i]$
$L[i].key = L[L.length].key$
DELETE(L, L[L.length])
MIN-HEAPIFY($L[i], i$)
return min

Algorithm 21 MIN-HEAPIFY(L[i],i)

1: $l = L[2i].key$
2: $r = L[2i + 1].key$
3: $p = L[i].key$
4: $smallest = i$
5: **if** $L[2i] \neq NIL$ and $l < p$ **then**
6:     $smallest = 2i$
7: **end if**
8: **if** $L[2i + 1] \neq NIL$ and $r < L[smallest]$ **then**
9:     $smallest = 2i + 1$
10: **end if**
11: **if** $smallest \neq i$ **then**
12:     exchange $L[i]$ with $L[smallest]$
13:     MIN-HEAPIFY(L[smallest],smallest)
14: **end if**

Union is implemented below, where we assume A and B are the two list representations of heaps to be merged. The runtime is again linear in the lengths of the lists to be merged.

c. Since the algorithms in part b didn't depend on the elements being distinct, we can use the same ones.

Problem 10-3

a. If the original version of the algorithm takes only t iterations, then, we have that it was only at most t random skips though the list to get to the desired value, since each iteration of the original while loop is a possible random jump followed by a normal step through the linked list.

b. The for loop on lines 2-7 will get run exactly t times, each of which is constant runtime. After that, the while loop on lines 8-9 will be run exactly X_t times. So, the total runtime is $O(t + E[X_t])$.

Algorithm 22 UNION(A,B)

1: **if** $A.head = NIL$ **then**
2:     **return** B
3: **end if**
4: $i = 1$
5: $x = A.head$
6: **while** $B.head \neq NIL$ **do**
7:     **if** $B.head.key \leq x.key$ **then**
8:         Insert a node at the end of list B with key $x.key$
9:         $x.key = B.head.key$
10:         $Delete(B, B.head)$
11:     **end if**$x = x.next$
12: **end while**
13: **return** A

---

c. Using equation C.25, we have that $E[X_t] = \sum_{i=1}^{\infty} Pr(X_t \geq i)$. So, we need to show that $Pr(X_t \geq i) \leq (1 - i/n)^t$. This can be seen because having X_t being greater than i means that each random choice will result in an element that is either at least i steps before the desired element, or is after the desired element. There are $n - i$ such elements, out of the total n elements that we were pricking from. So, for a single one of the choices to be from such a range, we have a probability of $(n - i)/n = (1 - i/n)$. Since each of the selections was independent, the total probability that all of them were is $(1 - i/n)^t$, as desired. Lastly, we can note that since the linked list has length n, the probability that X_t is greater than n is equal to zero.

d. Since we have that $t > 0$, we know that the function $f(x) = x^t$ is increasing, so, that means that $\lfloor x \rfloor^t \leq f(x)$. So,

$$\sum_{r=0}^{n-1} r^t = \int_0^n \lfloor r \rfloor^t dr \leq \int_0^n f(r)dr = \frac{n^{t+1}}{t+1}$$

e.

$$E[X_t] \leq \sum_{r=1}^{n}(1 - r/n)^t = \sum_{r=1}^{n}\sum_{i=0}^{t}\binom{t}{i}(-r/n)^i = \sum_{i=0}^{t}\sum_{r=1}^{n}\binom{t}{i}(-r/n)^i$$

$$= \sum_{i=0}^{t}\binom{t}{i}(-1)^i\left(n^i - 1 + \sum_{r=0}^{n-1}(r)^t\right)/n \leq \sum_{i=0}^{t}\binom{t}{i}(-1)^i\left(n^i - 1 + \frac{n^{i+1}}{i+1}\right)/n$$

$$\leq \sum_{i=0}^{t}\binom{t}{i}(-1)^i\frac{n^i}{i+1} = \frac{1}{t+1}\sum_{i=0}^{t}\binom{t+1}{i+1}(-n)^i \leq \frac{(1-n)^{t+1}}{t+1}$$

f. We just put together parts b and e to get that it runs in time $O(t+n/(t+1))$. But, this is the same as $O(t + n/t)$.

g. Since we have that for any number of iterations t that the first algorithm takes to find its answer, the second algorithm will return it in time $O(t + n/t)$. In particular, if we just have that $t = \sqrt{n}$. The second algorithm takes time only $O(\sqrt{n})$. This means that tihe first list search algorithm is $O(\sqrt{n})$ as well.

h. If we don't have distinct key values, then, we may randomly select an element that is further along than we had been before, but not jump to it because it has the same key as what we were currently at. The analysis will break when we try to bound the probability that $X_t \geq i$.

Chapter 11

Michelle Bodnar, Andrew Lohr

May 5, 2017

Exercise 11.1-1

Starting from the first index in T, keep track of the highest index so far that has a non NIL entry. This takes time $O(m)$.

Exercise 11.1-2

Start with a bit vector b which contains a 1 in position k if k is in the dynamic set, and a 0 otherwise. To search, we return true if $b[x] == 1$. To insert x, set $b[x] = 1$. To delete x, set $b[x] = 0$. Each of these takes $O(1)$ time.

Exercise 11.1-3

You could have each entry in the table be either a pointer to a doubly linked list containing all the objects with that key, or NIL if there are none. search just returns the first element in the list corresponding to the given key. Since all the elements in the list have that same key, it doesn't matter which search returns. Insert just adds to the start of the doubly linked list. Finally, deletion can be done in constant time in a doubly linked list, see problem 10-1

Exercise 11.1-4

The additional data structure will be a doubly linked list S which will behave in many ways like a stack. Initially, set S to be empty, and do nothing to initialize the huge array. Each object stored in the huge array will have two parts: the key value, and a pointer to an element of S, which contains a pointer back to the object in the huge array. To insert x, add an element y to the stack which contains a pointer to position x in the huge array. Update position $A[x]$ in the huge array A to contain a pointer to y in S. To search for x, go to position x of A and go to the location stored there. If that location is an element of S which contains a pointer to $A[x]$, then we know x is in A. Otherwise, $x \notin A$. To delete x, delete the element of S which is pointed to by $A[x]$. Each of these takes $O(1)$ and there are at most as many elements in S as there are valid elements in A.

Exercise 11.2-1

Under the assumption of simple uniform hashing, we will use linearity of expectation to compute this. Suppose that all the keys are totally ordered $\{k_1, \ldots, k_n\}$. Let X_i be the number of $\ell > k_i$ so that $h(\ell) = h(k_i)$. Note, that this is the same thing as $\sum_{j>i} Pr(h(k_j) = h(k_i)) = \sum_{j>i} 1/m = (n-i)/m$. Then, by linearity of expectation, the number of collisions is the sum of the number of collisions for each possible smallest element in the collision. The expected number of collisions is $\sum_{i=1}^{n} \frac{n-i}{m} = \frac{n^2 - \frac{n(n+1)}{2}}{m} = \frac{n^2 - n}{2m}$

Exercise 11.2-2

Label the slots of our table be $0, 1, 2, \ldots, 8$. Numbers which appear to the left in the table have been inserted later.

0	$\emptyset$
1	10, 19, 28
2	20
3	12
4	$\emptyset$
5	5
6	33, 15
7	$\emptyset$
8	17

Exercise 11.2-3

Both kinds of searches become expected runtime of $\Theta(1 + \lg(\alpha))$. Insertions and deletions stay $\Theta(1 + \alpha)$ because the time to insert into or delete from a sorted list is linear.

Exercise 11.2-4

The flag in each slot of the hash table will be 1 if the element contains a value, and 0 if it is free. The free list must be doubly linked. Search is unmodified, so it has expected time $O(1)$. To insert an element x, first check if $T[h(x.key)]$ is free. If it is, delete $T[h(x.key)]$ and change the flag of $T[h(x.key)]$ to 1. If it wasn't free to begin with, simply insert $x.key$ at the start of the list stored there. To delete, first check if $x.prev$ and $x.next$ are NIL. If they are, then the list will be empty upon deletion of x, so insert $T[h(x.key)]$ into the free list, update the flag of $T[h(x.key)]$ to 0, and delete x from the list it's stored in. Since deletion of an element from a singly linked list isn't $O(1)$, we must use a doubly linked list. All other operations are $O(1)$.

Exercise 11.2-5

There is a subset of size n hashing to the same spot, because if each spot only

had $n-1$ elements hashing to it, then the universe could only be size $(n-1)m$. The worst case searching time would be if all of the elements that we put in the hashtable were this subset of size n all going to the same spot, which is linear.

Exercise 11.2-6

Choose one of the m spots in the hash table at random. Let n_k denote the number of elements stored at $T[k]$. Next pick a number x from 1 to L uniformly at random. If $x < n_j$, then return the xth element on the list. Otherwise, repeat this process. Any element in the hash table will be selected with probability $1/mL$, so we return any key with equal probability. Let X be the random variable which counts the number of times we must repeat this process before we stop and p be the probability that we return on a given attempt. Then $E[X] = p(1+\alpha) + (1-p)(1+E[X])$ since we'd expect to take $1+\alpha$ steps to reach an element on the list, and since we know how many elements are on each list, if the element doesn't exist we'll know right away. Then we have $E[X] = \alpha + 1/p$. The probability of picking a particular element is $n/mL = \alpha/L$, so we have $E[X] = \alpha + L/\alpha = L(\alpha/L + 1/\alpha) = O(L(1 + 1/\alpha))$ since $\alpha \leq L$.

Exercise 11.3-1

If every element also contained a hash of the long character string, when we are searching for the desired element, we'll first check if the hashvalue of the node in the linked list, and move on if it disagrees. This can increase the runtime by a factor proportional to the length of the long character strings.

Exercise 11.3-2

Compute the value of the first character mod m, add the value of the second character mod m, add the value of the third character mod m to that, and so on, until all r characters have been taken care of.

Exercise 11.3-3

We will show that each string hashes to the sum of it's digits mod $2^p - 1$. We will do this by induction on the length of the string. As a base case, suppose the string is a single character, then the value of that character is the value of k which is then taken mod m. Now, for an inductive step, let $w = w_1 w_2$ where $|w_1| \geq 1$ and $|w_2| = 1$. Suppose, $h(w_1) = k_1$. Then, $h(w) = h(w_1)2^p + h(w_2)$ mod $2^p - 1 = h(w_1) + h(w_2) \mod 2^p - 1$. So, since $h(w_1)$ was the sum of all but the last digit mod m, and we are adding the last digit mod m, we have the desired conclusion.

Exercise 11.3-4

The keys 61, 62, 63, 64, and 65 are mapped to locations 700, 318, 936, 554,

and 172 respectively.

Exercise 11.3-5

As a simplifying assumption, assume that $|B|$ divides $|U|$. It's just a bit messier if it doesn't divide evenly.

Suppose to a contradiction that $\epsilon > \frac{1}{|B|} - \frac{1}{|U|}$. This means that for all pairs k, ℓ in U, we have that the number $n_{k,\ell}$ of hash functions in $\mathcal{H}$ that have a collision on those two elements satisfies $n_{k,\ell} \leq \frac{|\mathcal{H}|}{|B|} - \frac{|\mathcal{H}|}{|U|}$. So, summing over all pairs of elements in U, we have that the total number is less than$\leq \frac{|\mathcal{H}||U|^2}{2|B|} - \frac{|\mathcal{H}||U|}{2}$.

Any particular hash function must have that there are at least $|B|\binom{|U|/|B|}{2} = |B|\frac{|U|^2 - |U||B|}{2|B|^2} = \frac{|U|^2}{2|B|} - \frac{|U|}{2}$ colliding pairs for that hash function, summing over all hash functions, we get that there are at least $|\mathcal{H}|\left(\frac{|U|^2}{2|B|} - \frac{|U|}{2}\right)$ colliding pairs total. Since we have that there are at most some number less than this many, we have a contradiction, and so must have the desired restriction on ϵ.

Exercise 11.3-6

Fix $b \in \mathbb{Z}_p$. By exercise 31.4-4, $h_b(x)$ collides with $h_b(y)$ for at most $n - 1$ other $y \in U$. Since there are a total of p possible values that h_b takes on, the probability that $h_b(x) = h_b(y)$ is bounded from above by $\frac{n-1}{p}$. Since this holds for any value of b, $\mathcal{H}$ is ((n-1)/p)-universal.

Exercise 11.4-1

This is what the array will look like after each insertion when using linear probing:

										10
22										10
22									31	10
22				4					31	10
22				4	15				31	10
22				4	15	28			31	10
22				4	15	28	17		31	10
22	88			4	15	28	17		31	10
22	88			4	15	28	17	59	31	10

For quadradic probing, it will look identical until there is a collision on inserting the fifth element. Then, it is

									10
22									10
22								31	10
22			4					31	10
22			4				15	31	10
22			4		28		15	31	10
22		17	4		28		15	31	10
22	88	17	4		28		15	31	10
22	88	17	4		28	59	15	31	10

Lastly, for double hashing, we have:

									10
22									10
22								31	10
22			4					31	10
22			4	15				31	10
22			4	15	28			31	10
22		17	4	15	28			31	10
22		17	4	15	28	88		31	10
22	59	17	4	15	28	88		31	10

Exercise 11.4-2

Algorithm 1 HASH-DELETE(T,k)

$i = HASH - SEARCH(T, k)$

$T[i] = DELETED$

We modify INSERT by changing line 4 to: **if** $T[j] == NIL$ or $T[j] == DELETED$.

Exercise 11.4-3

For the $\alpha = 3/4$ case, we just plug into theorems 11.6 and 11.8 respectively to get that for a unsuccessful search, the expected number of probes is bounded by 4. And for a successful search, the expected number of probes is bounded by $\frac{4}{3} \ln(4)$.

For $\alpha = 7/8$. The bound for expected number of probes of unsuccessful is 8, and for successful is $\frac{8}{7} \ln(8)$.

Exercise 11.4-4

Write $h_2(k) = dc_1$ and $m = dc_2$ for constants c_1 and c_2. When we have examined $(1/d)^{th}$ of the table entries, this corresponds to $m/d = c_2$ of them. The entry that we check at this point is $[h_1(k) + (m/d)h_2(k)] \mod m = [h_1(k) + (m/d)(dc_1)] \mod m \equiv h_1(k)$. When $d = 1$, we may have to examine the entire

hash table.

Exercise 11.4-5

We will try to find a rational solution, let $\alpha = m/n$. Using theorems 11.6 and 11.8, we have to solve the equation

$$\frac{1}{1-\alpha} = \frac{2}{\alpha}\ln(\frac{1}{1-\alpha})$$

Unfortunately, this transcendental equation cannot be solved using simple techniques. There is an exact solution using the Lambert W function of

$$\frac{1}{2}\frac{1 + 2LambertW(-1, -(\frac{1}{2}) * exp(-\frac{1}{2}))}{LambertW(-1, -(\frac{1}{2}) * exp(-\frac{1}{2}))}$$

Which evaluates to approximately $\alpha = .7153$.

Exercise 11.5-1

Let $A_{j,k}$ be the event that j and k hash to different things. Due to uniform hashing, $Pr(A_{j,k}) = \frac{m-1}{m}$. Also, we can say that there is a negative correlation between the events. That is, if we know that several pairs of elements hashed to the same thing, then we can only decrease the likelihood that some other pair hashed to different things. This gets us that the probability that all events happen is $\leq$ the probability that the all happened if they were independent, so,

$$Pr(\cap_{j,k}A_{j,k}) \leq \left(\frac{m-1}{m}\right)^{\binom{n}{2}} \leq \left(e^{-1/m}\right)^{\frac{n(n-1)}{2}} = e^{\frac{-n(n-1)}{2m}}$$

So, if we have that that n exceeds m, the $-n^2/(2m)$ term is much bigger than the $n/2m$ term, so, the exponent is going to $-\infty$, which means the probability is going to 0.

Problem 11-1

a. The index for each probe is computed uniformly from among all the possible indices. Since we have $n \leq m/2$, we know that there are at least half of the indices empty at any stage. So, for more than k probes to be required, we would need that in each of k first probes, we probed a vertex that already had an entry, this has probability less than $1/2$, so the probability of it happening each time is $< 1/(2^k)$.

b. Using the result from the previous part with $k = 2\lg(n)$, we have that the probability that so many probes will be required is

$$< 2^{-2\lg(n)} = 2^{\lg(n^{-2})} = n^{-2} = \frac{1}{n^2}$$

6

c. We apply a union bound followed by the results of the previous part

$$Pr\{X > 2\lg(n)\} = Pr\{\vee_i X_i > 2\lg(n)\} \le \sum_i Pr\{X_i > 2\lg(n)\} \le \sum_i \frac{1}{n^2} = \frac{n}{n^2} = \frac{1}{n}$$

d. The longest possible length of a probe sequence is n, as we would try checking every single entry already placed in the array. We also know that the probability that a sequence of length more than $2\lg(n)$ is required is $\le 1/n$. So, we have that the largest the expected value can be is

$$E[X] \le Pr\{X \le 2\lg(n)\}2\lg(n) + Pr\{X > \lg(n)\}n = \frac{n-1}{n}2\lg(n) + \frac{1}{n}n = 2\lg(n) + 1 - 2\frac{\lg(n)}{n} \in O(\lg(n))$$

Problem 11-2

a. Let Q_k denote the probability that exactly k keys hash to a particular slot. There are $\binom{n}{k}$ ways to select the k keys, they hash to that spot with probability $\left(\frac{1}{n}\right)^k$, and the remaining $n-k$ keys hash to the remaining $n-1$ slots with probability $\left(\frac{n-1}{n}\right)^{n-k}$. Thus,

$$Q_k = \left(\frac{1}{n}\right)^k \left(\frac{n-1}{n}\right)^{n-k} \binom{n}{k}.$$

b. The probability that the slot containing the most keys contains exactly k is bounded above by the probability that some slot contains k keys. There are n slots which we could select to contain those k, so $P_k \le nQ_k$.

c. Using the fact that $\binom{n}{k} \le \left(\frac{ne}{k}\right)^k$ we have $Q_k \le \frac{(n-1)^{n-k}}{n^n}\left(\frac{ne}{k}\right)^k \le e^k/k^k$.

d. We'd like to make $Q_{k_0} < n^3$. Taking the log of both sides and expanding using log rules this becomes

$$\frac{c\lg n}{\lg\lg n}(\lg e + \lg\lg\lg n - \lg c - \lg\lg n) < -3\lg n.$$

dividing both sides by $-\lg\lg n$ this is equivalent to

$$3 < -\frac{c\lg e}{\lg\lg n} - \frac{\lg\lg\lg n}{\lg\lg n} + \frac{\lg c}{\lg\lg n} + c.$$

As $n \to \infty$, all terms except the last go to 0, so choosing $c = 4$ works for all $n > N$ for some fixed value of N. Let c_i denote the value that works for $n = i$. Then by letting $c = \max_i(c_i, 4)$ we obtain the desired value of c. By part (b), we have $P_{k_0} \le nQ_{k_0} < n/n^3 = 1/n^2$. Since Q_k is a decreasing function of k, we conclude that $P_k < 1/n^2$ for all $k \ge k_0$.

e. This comes from computing expectation by conditioning, and bounding M from above by n in the first term. From the bound in (d) we have $nP(M > \frac{c \lg n}{\lg \lg n}) < 1$. For the second term, $\sum_{i=1}^{d} P(M = i) \leq 1$ where $d = c \lg n / \lg \lg n$, so the asymptotic expectation follows.

Problem 11-3

a. At each step, we are increasing the amount we increase by by 1, so, this leads to the "Gauss numbers" which have formula $\frac{i^2 + i}{2}$. So, we have $c_1 = c_2 = \frac{1}{2}$.

b. To show that this algorithm examines every number, we will show that every number that it examines is distinct. Then, since it examines m numbers total, this will imply that every number is visited. Suppose that we visited the same position on two distinct rounds $i < i' < m$, then,

$$h(k) + \frac{i + i^2}{2} \equiv h(k) + \frac{i' + i'^2}{2} \quad \mod m$$

Which means

$$\frac{i + i^2}{2} \equiv \frac{i' + i'^2}{2} \quad \mod m$$

So,

$$i + i^2 \equiv i' + i'^2 \quad \mod 2m$$

Rearranging,

$$i'^2 - i^2 = (i + i')(i' - i) \equiv i' - i \quad \mod 2m$$

So,

$$(i' - i)(i' + i - 1) \equiv 0 \quad \mod 2m$$

Now, we split into two cases based on whether $i - i'$ is even or odd. If $i' - i$ is odd, then we can cancel it on both sides, because it has no common factor with 2m (because m is a power of 2). This leaves us with

$$i + i' + 1 \equiv 0 \quad \mod 2m$$

Since $i < i'$, we have $i + i' + 1 \leq 2i' < 2m$, so, we must have $i + i' + 1 = 0$, a contradiction to our assumption that such i and i' existed.

If we have that $i' - i$ is even, then $i + i' + 1$ is odd, so it has no factors in common with 2m. Similarly, this means that we can cancel out that factor to get $i' - i \equiv 0 \mod 2m$. However, we have that $|i' - i| \leq i' + i < 2i' < 2m$. So, we must have $i' - i = 0$, a contradiction to how we picked i and i'.

Problem 11-4

8

a. Let $k, l \in U$ be arbitrary. Then $P(h(k) = h(l)) = P(< h(k), h(l) > = < x, x >)$ for some $x \in [m]$. Since h comes from a set of 2-universal hash functions, this is equally likely to be any of the m^2 sequences. There are m possible values of x which would cause a collision, so the probability of collision is $\frac{m}{m^2} = \frac{1}{m}$.

b. The probability of collision is $\frac{1}{p}$, so $\mathcal{H}$ is universal. Now consider the tuple $z = < 0, 0, \ldots, 0 >$. Then $h_a(x) = h_b(x) = 0$ for any $a, b \in U$, so the sequence $< z, x^{(2)} >$ is equally likely to be any of p sequences starting with 0, but can't be any of the other $p^2 - p$ sequences, so $\mathcal{H}$ is not 2-universal.

c. Let $x, y \in U$ be fixed, distinct n-tuples. As a_i and b range over $\mathbb{Z}_p$, $h'_{ab}(x)$ is equally likely to achieve every value from 1 to p since for any sequence a, we can let b vary from 1 to $p - 1$. Thus, $< h'_{ab}(x), h'_{ab}(y) >$ is equally likely to be any of the p^2 sequences, so $\mathcal{H}$ is 2-universal.

d. Since $\mathcal{H}$ is 2-universal, there are p other functions which map m to $h(m)$, and the adversary has no way of knowing which one of these Alice and Bob have agreed on in advance, so the best he can do is try one of them, which will succeed in fooling Bob with probability $1/p$.
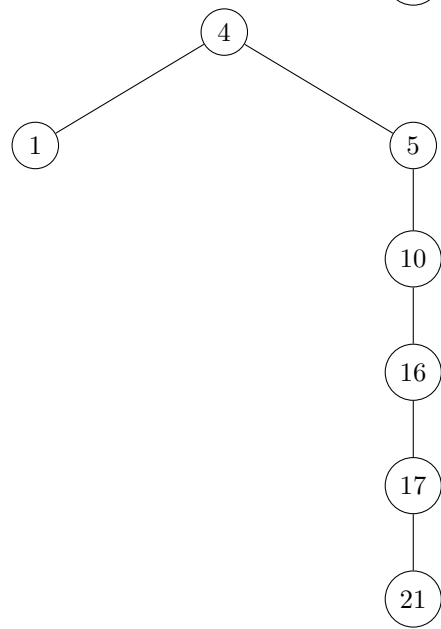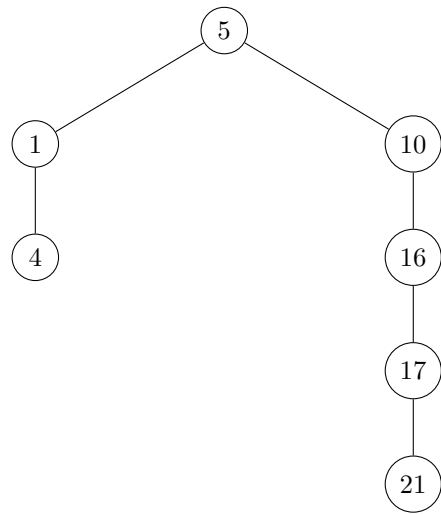
Chapter 12

Michelle Bodnar, Andrew Lohr
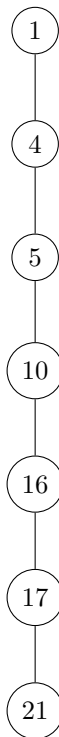
May 5, 2017

Exercise 12.1-1

Anytime that a node has a single child, treat it as the right child, with the left child being NIL
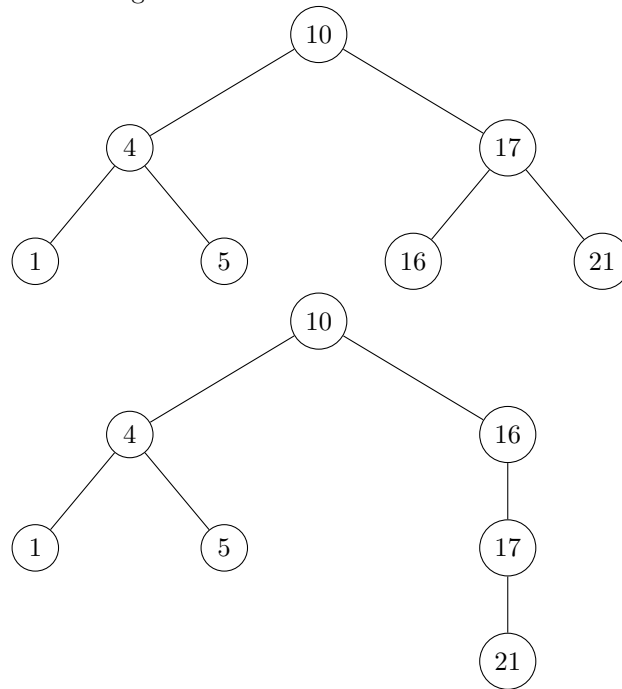
Exercise 12.1-2

The binary-search-tree property guarantees that all nodes in the left subtree are smaller, and all nodes in the right subtree are larger. The min-heap property only guarantees the general child-larger-than-parent relation, but doesn't distinguish between left and right children. For this reason, the min-heap property can't be used to print out the keys in sorted order in linear time because we have no way of knowing which subtree contains the next smallest element.

Exercise 12.1-3

Our solution to exercise 10.4-5 solves this problem.

Exercise 12.1-4

We call each algorithm on $T.root$. See algorithms PREORDER-TREE-WALK and POSTORDER-TREE-WALK.

Exercise 12.1-5

Suppose to a contradiction that we could build a BST in worst case time $o(n \lg(n))$. Then, to sort, we would just construct the BST and then read off the

---

Algorithm 1 PREORDER-TREE-WALK(x)

---

if $x \neq NIL$ **then**
    print x
    PREORDER-TREE-WALK(x.left)
    PREORDER-TREE-WALK(x.right)
end if
return

Algorithm 2 POSTORDER-TREE-WALK(x)

---

if $x \neq NIL$ **then**
    POSTORDER-TREE-WALK(x.left)
    POSTORDER-TREE-WALK(x.right)
    print x
end if
return

---

elements in an inorder traversal. This second step can be done in time $\Theta(n)$ by Theorem 12.1. Also, an inorder traversal must be in sorted order because the elements in the left subtree are all those that are smaller than the current element, and they all get printed out before the current element, and the elements of the right subtree are all those elements that are larger and they get printed out after the current element. This would allow us to sort in time $o(n \lg(n))$ a contradiction

Exercise 12.2-1

option c could not be the sequence of nodes explored because we take the left child from the 911 node, and yet somehow manage to get to the 912 node which cannot belong the left subtree of 911 because it is greater. Option e is also impossible because we take the right subtree on the 347 node and yet later come across the 299 node.

Exercise 12.2-2

See algorithms TREE-MINIMUM and TREE-MAXIMUM.

Algorithm 3 TREE-MINIMUM(x)

---

if $x.left \neq NIL$ **then**
    return $TREE - MINIMUM(x.left)$
else
    return x
end if

Algorithm 4 TREE-MAXIMUM(x)

---
if $x.right \neq NIL$ **then**
    return $TREE - MAXIMUM(x.right)$
else
    return x
end if

Exercise 12.2-3

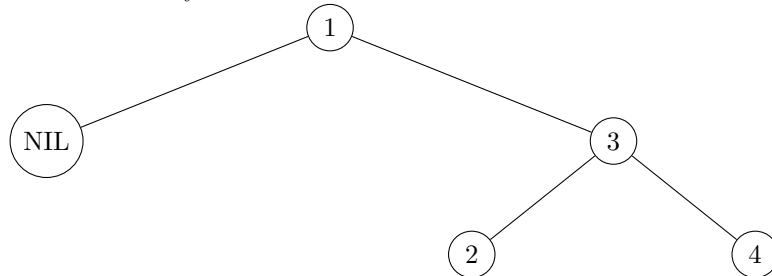---

Algorithm 5 TREE-PREDECESSOR(x)

---
if $x.left \neq NIL$ **then**
    return TREE-MAXIMUM(x.left)
end if
$y = x.p$
while $y \neq NIL$ and $x == y.left$ **do**
    $x = y$
    $y = y.p$
end while
return y

Exercise 12.2-4

Suppose we search for 4 in this tree. Then $A = \{2\}$, $B = \{1, 3, 4\}$ and $C = \emptyset$, and Professor Bunyan's claim fails since $1 < 2$.

Exercise 12.2-5

Suppose the node x has two children. Then it's successor is the minimum element of the BST rooted at $x.right$. If it had a left child then it wouldn't be the minimum element. So, it must not have a left child. Similarly, the predecessor must be the maximum element of the left subtree, so cannot have a right child.

Exercise 12.2-6

First we establish that y must be an ancestor of x. If y weren't an ancestor of x, then let z denote the first common ancestor of x and y. By the binary-search-tree property, $x < z < y$, so y cannot be the successor of x.

Next observe that $y.left$ must be an ancestor of x because if it weren't, then $y.right$ would be an ancestor of x, implying that $x > y$. Finally, suppose that y is not the lowest ancestor of x whose left child is also an ancestor of x. Let z denote this lowest ancestor. Then z must be in the left subtree of y, which implies $z < y$, contradicting the fact that y is the successor if x.

Exercise 12.2-7

To show this bound on the runtime, we will show that using this procedure, we traverse each edge twice. This will suffice because the number of edges in a tree is one less than the number of vertices.

Consider a vertex of a BST, say x. Then, we have that the edge between $x.p$ and x gets used when successor is called on $x.p$ and gets used again when it is called on the largest element in the subtree rooted at x. Since these are the only two times that that edge can be used, apart from the initial finding of tree minimum. We have that the runtime is $O(n)$. We trivially get the runtime is $\Omega(n)$ because that is the size of the output.

Exercise 12.2-8

Let x be the node on which we have called TREE-SUCCESSOR and y be the kth successor of x. Let z be the lowest common ancestor of x and y. Successive calls will never traverse a single edge more than twice since TREE-SUCCESSOR acts like a tree traversal, so we will never examine a single vertex more than three times. Moreover, any vertex whose key value isn't between x and y will be examined at most once, and it will occur on a simple path from x to z or y to z. Since the lengths of these paths are bounded by h, the running time can be bounded by $3k + 2h = O(k + h)$.

Exercise 12.2-9

If $x = y.left$ then calling successor on x will result in no iterations of the while loop, and so will return y. Similarly, if $x = y.right$, the while loop for calling predecessor(see exercise 3) will be run no times, and so y will be returned. Then, it is just a matter of recognizing what the problem asks to show is exactly that y is either predecessor(x) or successor(x).

Exercise 12.3-1

The initial call to TREE-INSERT-REC should be NIL,T.root,z

Exercise 12.3-2

Algorithm 6 TREE-INSERT-REC(y,x,z)
___
  if $x \neq NIL$ **then**
    if $z.key < x.key$ **then**
      TREE-INSERT-REC(x,x.left,z)
    else
      TREE-INSERT-REC(x,x,right,z)
    end if
  end if
  z.p = y
  if $y == NIL$ **then**
    T.root = z
  else if $z.key < y.key$ **then**
    y.left = z
  else
    y.right = z
  end if
___

The nodes examined in the while loop of TREE-INSERT are the same as those examined in TREE-SEARCH. In lines 9 through 13 of TREE-INSERT, only one additional node is examined.

Exercise 12.3-3

The worst case is that the tree formed has height n because we were inserting them in already sorted order. This will result in a runtime of $\Theta(n^2)$. In the best case, the tree formed is approximately balanced. This will mean that the height doesn't exceed $O(\lg(n))$. Note that it can't have a smaller height, because a complete binary tree of height h only has $\Theta(2^h)$ elements. This will result in a rutime of $O(n \lg(n))$. We showed $\Omega(n \lg(n))$ in exercise 12.1-5.
Exercise 12.3-4

Deletion is not commutative. In the following tree, deleting 1 then 2 yields a different from the one obtained by deleting 2 then 1.
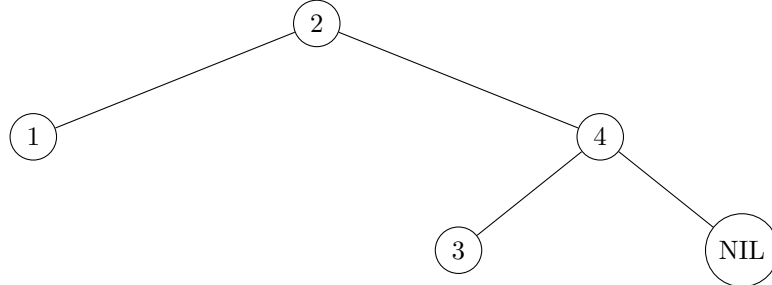
Exercise 12.3-5

Our insertion procedure follows closely our solution to 12.3-1, the difference being that once it finds the position to insert the given node, it updates the succ fields appropriately instead of the p field of z.

Algorithm 7 TREE-INSERT'(y,x,z)

---
  if $x \neq NIL$ **then**
    if $z.key < x.key$ **then**
      TREE-INSERT'(x,x.left,z)
    else
      TREE-INSERT'(x,x,right,z)
    end if
  end if
  if $y == NIL$ **then**
    T.root = y
  else if $z.key < y.key$ **then**
    y.left = z
    x.succ = y
  else
    y.right = z
    z.succ = y.succ
    y.succ = z
  end if

Our Search procedure is unchanged from the version given in the previous section

We will assume for the deletion procedure that all the keys are distinct, as that has been a frequent assumption throughout this chapter. This will however depend on it. Our deletion procedure first calls search until we are one step away from the node we are looking for, that is, it calls TREE-PRED(T.root,z.key)

Algorithm 8 TREE-PRED(x,k)

---
  if $k < x.key$ **then**
    $y = x.left$
  else
    $y = x.right$
  end if
  if $y == NIL$ **then**
    throw error
  else if $y.key = k$ **then**
    return x
  else
    return TREE-PRED(y,k)
  end if

It can use this TREE-PRED procedure to compute $u.p$ and $v.p$ in the

TRANSPLANT procedure. Since TREE-DELETE only calls TRANSPLANT a constant number of times, increasing the runtime of TRANSPLANT to $O(h)$ in this way causes the runtime of the new TREE-DELETE procedure to be $O(h)$.

Exercise 12.3-6

Update line 5 so that y is set equal to TREE-MAXIMUM(z.left). To implement the fair strategy, we could randomly decide each time TREE-DELETE is called whether or not to use the predecessor or successor.

Exercise 12.4-1

Consider all the possible positions of the largest element of the subset of $n + 3$ of size 4. Suppose it were in position $i + 4$ for some $i \leq n - 1$. Then, we have that there are $i + 3$ positions from which we can select the remaining three elements of the subset. Since every subset with different largest element is different, we get the total by just adding them all up (inclusion exclusion principle).

Exercise 12.4-2

To keep the average depth low but maximize height, the desired tree will be a complete binary search tree, but with a chain of length $c(n)$ hanging down from one of the leaf nodes. Let $k = \log(n - c(n))$ be the height of the complete binary search tree. Then the average height is approximately given by

$$\frac{1}{n} \left[\sum_{i=1}^{n-c(n)} \lg(i) + (k + 1) + (k + 2) + \ldots + (k + c(n)) \right] \approx \lg(n - c(n)) + \frac{c(n)^2}{2n}.$$

The upper bound is given by the largest $c(n)$ such that $\lg(n - c(n)) + \frac{c(n)^2}{2n} = \Theta(\lg n)$ and $c(n) = \omega(\lg n)$. One function which works is $\sqrt{n}$.

Exercise 12.4-3

Suppose we have the elements $\{1, 2, 3\}$. Then, if we construct a tree by a random ordering, then, we get trees which appear with probabilities some multiple of $\frac{1}{6}$. However, if we consider all the valid binary search trees on the key set of $\{1, 2, 3\}$. Then, we will have only five different possibilities. So, each will occur with probability $\frac{1}{5}$, which is a different probability distribution.

Exercise 12.4-4

The second derivative is $2^x \ln^2(2)$ which is always positive, so the function is convex.

Exercise 12.4-5

Suppose that when quicksort always selects it's elements to be in the middle $n^{1-k/2}$ of the elements each time. Then, the size of the problem shrinks by a power of at least $(1-k/2)$ each time. So, the greatest depth of recursion d will be so that $n^{(1-k/2)^d} \leq 2$, solving for d, we get $(1-k/2)^d \leq \log_n(2) = \lg(2)/\lg(n)$, so, $d \leq \log_{1-k/2}(\lg(2)) - \log_{1-k/2}(\lg(n)) = \log_{1-k/2}(\lg(2)) - \lg(\lg(n))/\lg(1-k/2)$.

Let A(n) denote the probability that when quicksorting a list of length n, some pivot is selected to not be in the middle $n^{1-k/2}$ of the numbers. This doesn't happen with probability $\frac{1}{n^{k/2}}$. Then, we have that the two subproblems are of size n_1, n_2 with $n_1 + n_2 = n - 1$ and $\max\{n_1, n_2\} \leq n^{1-k/2}$. So, $A(n) \leq \frac{1}{n^{k/2}} + T(n_1) + T(n_2)$ So, since we bounded the depth by $O(1/\lg(n))$ let $\{a_{i,j}\}_i$ be all the subproblem sizes left at depth j. So, $A(n) \leq \frac{1}{n^{k/2}} \sum_j \sum_i \frac{1}{a}$

Problem 12-1

a. Each insertion will add the element to the right of the rightmost leaf because the inequality on line 11 will always evaluate to false. This will result in the runtime being $\sum_{i=1}^n i \in \Theta(n^2)$

b. This strategy will result in each of the two children subtrees having a difference in size at most one. This means that the height will be $\Theta(\lg(n))$. So, the total runtime will be $\sum_{i=1}^n \lg(n) \in \Theta(n \lg(n))$

c. This will only take linear time since the tree itself will be height 0, and a single insertion into a list can be done in constant time.

d. The worst case performance is that every random choice is to the right (or all to the left) this will result in the same behavior as in the first part of this problem, $\Theta(n^2)$

   To compute the expected runtime informally, just notice that when randomly choosing, we will pick left roughly half the time, so, the tree will be roughly balanced, so, we have that the depth is roughly $\lg(n)$, so the expected runtime will be $n \lg(n)$.

Problem 12-2

The word at the root of the tree is necessarily before any word in its left or right subtree because it is both shorter, and the prefix of, every word in each of these trees. Moreover, every word in the left subtree comes before every word in the right subtree, so we need only perform a preorder traversal. This can be done recursively, as shown in exercise 12.1-4.

Problem 12-3

a. Since we are averaging over all nodes x the value of $d(x, T)$, it is $\frac{1}{n} \sum_{x \in T} d(x, T)$, but by definition, this is $\frac{1}{n} P(T)$.

b. Every non-root node has a contribution of one coming from the first edge from the root on its way to that node, every other edge in this path is counted by looking at the edges within the two subtrees rooted at the child of the original root. Since there are $n - 1$ non-root nodes, we have

$$P(T) = \sum_{x \in T} d(x, T) = \sum_{x \in T_L} d(x, T) + \sum_{x \in T_R} d(x, T) =$$

$$\sum_{x \in T_L} (d(x, T_L) + 1) + \sum_{x \in T_R} (d(x, T_R) + 1) = \sum_{x \in T_L} d(x, T_L) + \sum_{x \in T_R} d(x, T_R) + n - 1 =$$

$$P(T_L) + P(T_R) + n - 1$$

c. When we are randomly building our tree on n keys, we have n possibilities for the first element that we add to the tree, the key that will belong to the eventual root. Suppose that it had order statistic $i + 1$ for some i in $\{0, \ldots n - 1\}$. Then, we have that all the smaller elements will be to the left and all the larger elements will be in the subree to the right. However, they will all be in random order relative to each other, so, we will have $P(i) = E[P(T_L)]$ and $P(n - i - 1) = E[P(T_R)]$. So, we have have the desired inequality by averaging over the order statistic of the first term put into the BST.

d.

$$\frac{1}{n} \sum_{i=0}^{n-1} P(i) + P(n-i-1) + n - 1 = \frac{1}{n} \left(\sum_{i=0}^{n-1} P(i) + \sum_{i=0}^{n-1} P(n-i-1) + \sum_{i=0}^{n-1} (n-1) \right)$$

Then, we do the substitution $j = n - i - 1$ and do the simple thing of summing a constant for the third sum to get

$$= \frac{1}{n} \left(\sum_{i=0}^{n-1} P(i) + \sum_{j=0}^{n-1} P(j) + n(n-1) \right) = \frac{2}{n} \sum_{i=0}^{n-1} P(i) + n - 1$$

e. Our recurrence from the previous part is exactly the same as eq (7.6) which we showed in problem 7-3.e to have solution $\Theta(n \lg(n))$

f. Let the first pivot selected be the first element added to the binary tree. Since every element is compared to the root, and every element is compared to the first pivot, we have what we want. Then, let the next pivot for the left (resp. right) subarrays be the first element that is less than (resp. greater than) the root. Then, we have that the two subtrees form the same partition of the remaining elements as the two subarrays left form. We can than continue to recurse in this way. Since if holds at the first element, and the problems have the same recursive structure, we have that it holds at every element.

Problem 12-4

a. There is a single binary tree on one vertex consisting of just a root, so $b_0 = 1$. To count the number of binary trees with n nodes, we first choose a root from among the n vertices. If the root node we have chosen is the ith smallest element, the left subtree will have $i - 1$ vertices, and the right subtree will have $n - i$ vertices. The number of such left and right subtrees are counted by b_{i-1} and b_{n-i} respectively. Summing over all possibly choices of root vertex gives:

$$b_n = \sum_{k=1}^{n} b_{k-1} b_{n-k} = \sum_{k=0}^{n-1} b_k b_{n-k-1}.$$

b.

$$B(x) = \sum_{n=0}^{\infty} b_n x^n$$

$$= 1 + \sum_{n=1}^{\infty} b_n x^n$$

$$= 1 + \sum_{n=1}^{\infty} \sum_{k=0}^{n-1} b_k b_{n-k-1} x^n$$

$$= 1 + x \sum_{n=1}^{\infty} \sum_{k=0}^{n-1} b_k x^k b_{n-k-1} x^{n-k-1}$$

$$= 1 + x \sum_{n=0}^{\infty} \sum_{k=0}^{n} b_k x^k b_{n-k} x^{n-k}$$

$$= 1 + x B(x)^2.$$

Applying the quadratic formula and noting that the minus sign is to be taken so that $B(0) = 0$ proves the result.

c. Using the Taylor expansion of $\sqrt{1-4x}$ we have:

$$
\begin{aligned}
B(x) &= \frac{1}{2x}\left(1 - \sum_{n=0}^{\infty} \frac{1}{1-2n}\binom{2n}{n}x^n\right) \\
&= \frac{-1}{2x}\sum_{n=1}^{\infty} \frac{1}{1-2n}\binom{2n}{n}x^n \\
&= \frac{1}{2}\sum_{n=1}^{\infty} \frac{1}{2n-1}\binom{2n}{n}x^{n-1} \\
&= \frac{1}{2}\sum_{n=0}^{\infty} \frac{1}{2n+1}\binom{2n+2}{n+1}x^n.
\end{aligned}
$$

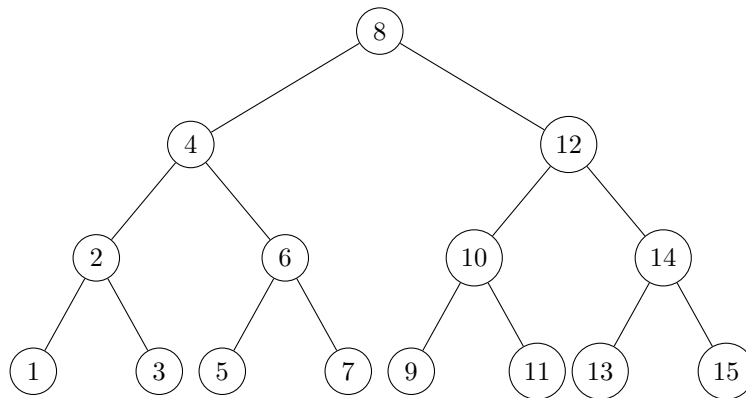Extracting the coefficient from x^n and simplifying yields the result.

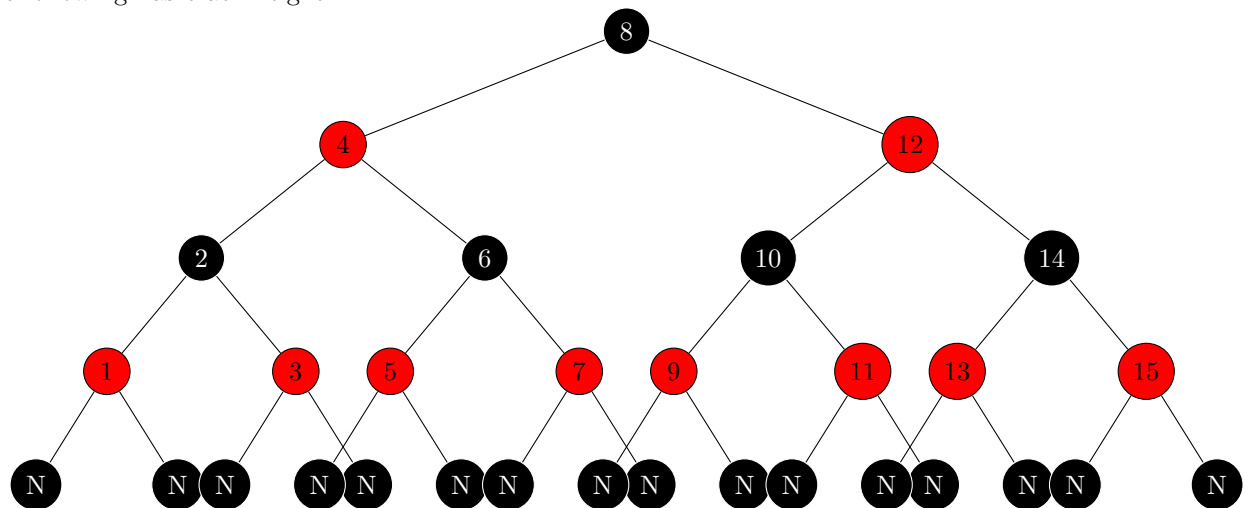d. The asymptotic follows from applying Sirling's formula to b_n.
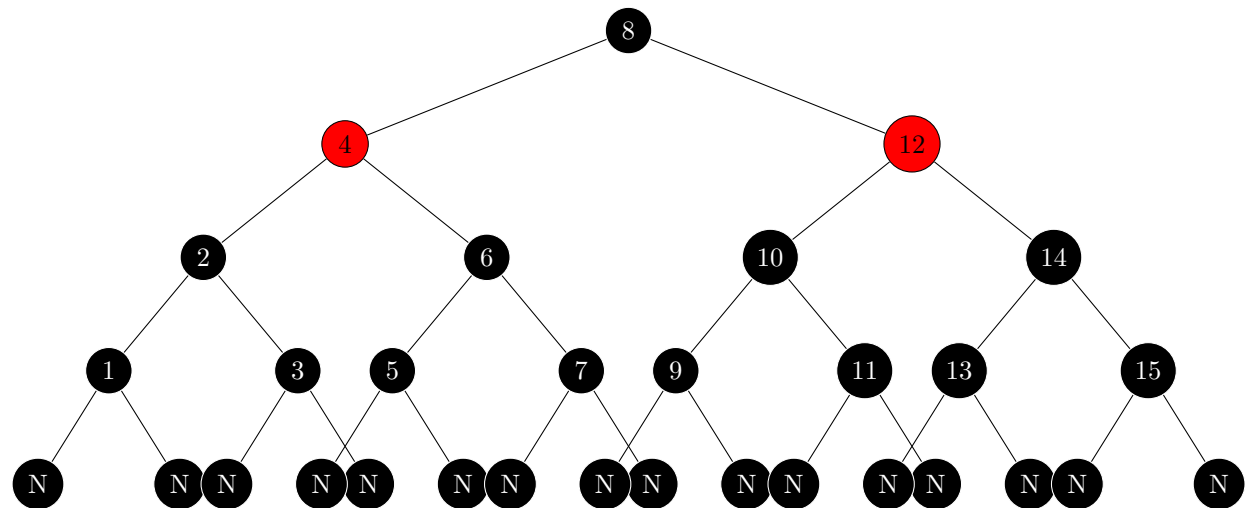
Chapter 13

Michelle Bodnar, Andrew Lohr

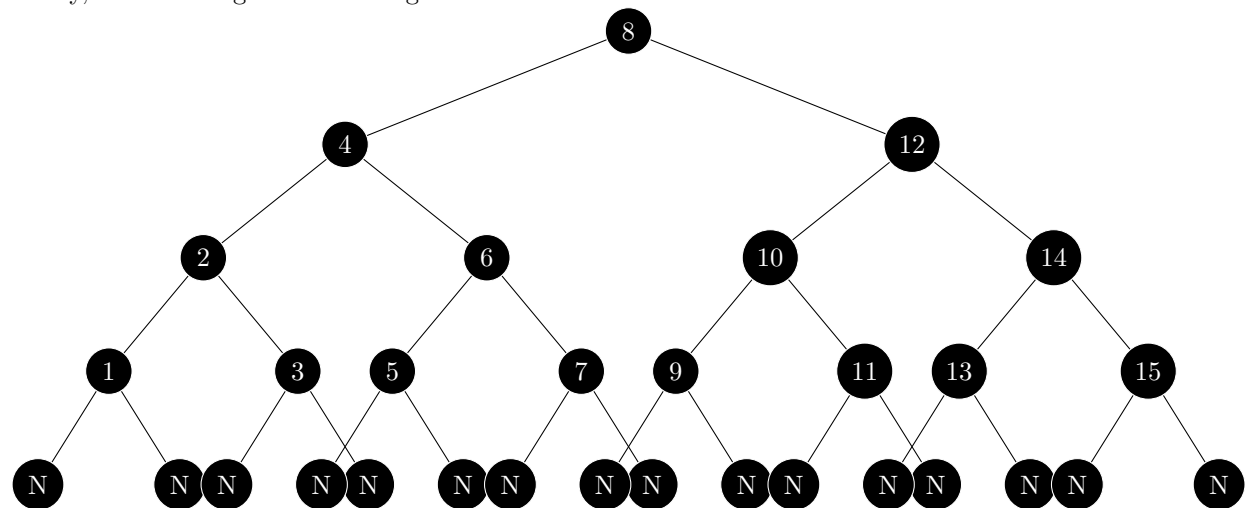April 12, 2016

Exercise 13.1-1

We shorten NIL to N so that it can be more easily displayed in the document. The following has black height 2.


The following has black height 3

Lastly, the following has black height 4.

Exercise 13.1-2

If the inserted node is red then it won't be a red-black tree because 35 will be the parent of 36, which is also colored red. If the inserted node is black it will also fail to be a red-black tree because there will be two paths from node 38 to T.nil which contain different numbers of black nodes, violating property 5. In the picture of the tree below, the NIL nodes have been omitted for space reasons.
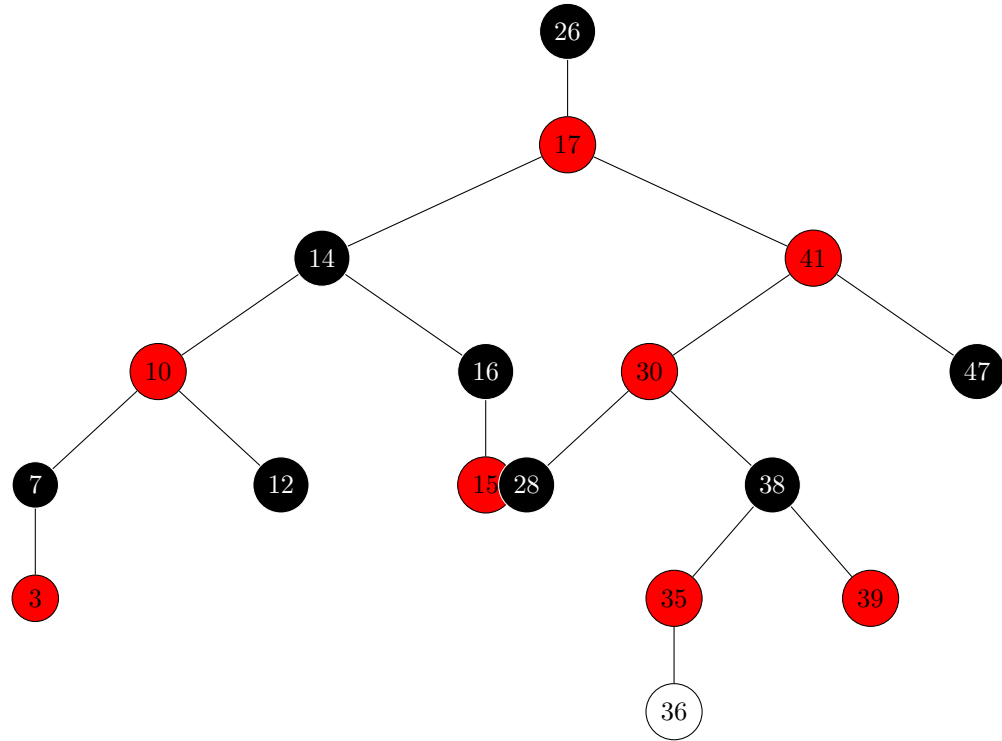
Exercise 13.1-3

It will. There was no red node introduced, so 4 will still be satisfied. Since the root is in every path from the root to the leaves, but no others. 5 will be satisfied because the only paths we will be changing the number of black nodes in are those coming from the root. All of these will increase by 1, and so will all be equal. 3 is trivially preserved, as no new leaves are introduced. 1 is also trivially preserved as only one node is changed and it is not changed to some mysterious third color.

Exercise 13.1-4

The possible degrees are 0 through 5, based on whether or not the black node was a root and whether it had one or two red children, each with either one or two black children. The depths could shrink by at most a factor of 1/2.

Exercise 13.1-5

Suppose we have the longest simple path $(a_1, a_2, \ldots a_s)$ and the shortest simple path $(b_1, b_2, \ldots, b_t)$. Then, by property 5 we know they have equal numbers of black nodes. By property 4, we know that neither contains a repeated red node. This tells us that at most $\lfloor \frac{s-1}{2} \rfloor$ of the nodes in the longest path are red.

This means that at least $\lceil \frac{s+1}{2} \rceil$ are black, so, $t \geq \lceil \frac{s+1}{2} \rceil$. So, if, by way of contradiction, we had that $s > t*2$, then $t \geq \lceil \frac{s+1}{2} \rceil \geq \lceil \frac{2t+2}{2} \rceil = t+1$ a contradiction.

Exercise 13.1-6

In a path from root to leaf we can have at most one red node between any two black nodes, so maximal height of such a tree is $2k+1$, where each path from root to leaf is alternating red and black nodes. To maximize internal nodes, we make the tree complete, giving a total of $2^{2k+1} - 1$ internal nodes. The smallest possible number of internal nodes comes from a complete binary tree, where every node is black. This has $2^{k+1} - 1$ internal nodes.

Exercise 13.1-7

Since each red node needs to have two black children, our only hope at getting a large number of internal red nodes relative to our number of black internal nodes is to make it so that the parent of every leaf is a red node. So, we would have a ratio of $\frac{2}{3}$ if we have the tree with a black root which has red children, and all of it's grandchildren be leaves. We can't do better than this because as we make the tree bigger, the ratio approaches $\frac{1}{2}$.

The smallest ratio is achieved by having a complete tree that is balanced and black as a raven's feather. For example, see the last tree presented in the solution to 13.1-1.

Exercise 13.2-1

See the algorithm for RIGHT-ROTATE.

Algorithm 1 RIGHT-ROTATE(T,x)

---
y = x.left
x.left = y.right
if $y.right \neq T.nil$ **then**
    t.right.p = x
end if
y.p = x.p
if x.p == T.nil **then**
    T.root = y
else if x == x.p.left **then**
    x.p.left = y
else
    x.p.right = y
end if
y.right =x
x.p =y

Exercise 13.2-2

We proceed by induction. In a tree with only one node, the root has neither a left nor a right child, so no rotations are valid. Suppose that a tree on $n \geq 0$ nodes has exactly $n-1$ rotations. Let T be a binary search tree on $n+1$ nodes. If $T.root$ has no right child then the root can only be involved in a right rotation, and the left child of T has n vertices, so it has exactly $n-1$ rotations, yielding a total of n for the whole tree. The argument is identical if $T.root$ has no left child. Finally, suppose $T.root$ has two children, and let k denote the number of nodes in the left subtree. Then the root can be either left or right rotated, contributing 2 to the count. By the induction hypothesis, $T.left$ has exactly $k-1$ rotations and $T.right$ has exactly $n-k-1-1$ rotations, so there are a total of $2+k-1+n-k-1-1 = n$ possible rotations, completing the proof.

Exercise 13.2-3

the depth of c decreases by one, the depth of b stays the same, and the depth of a increases by 1.

Exercise 13.2-4

Consider transforming an arbitrary n-node BT into a right-going chain as follows: Let the root and all successive right children of the root be the elements of the chain initial chain. For any node x which is a left child of a node on the chain, a single right rotation on the parent of x will add that node to the chain and not remove any elements from the chain. Thus, we can convert any BST to a right chain with at most $n-1$ right rotations. Let $r_1, r_2, \ldots, r_k$ be the sequence of rotations required to convert some BST T_1 into a right-going chain, and let $s_1, s_2, \ldots, s_m$ be the sequence of rotations required to convert some other BST T_2 to a right-going chain. Then $k < n$ and $m < n$, and we can convert T_1 to T_2 be performing the sequence $r_1, r_2, \ldots, r_k, s'_m s'_{m-1}, \ldots, s'_1$ where s'_i is the opposite rotation of s_i. Since $k+m < 2n$, the number of rotations required is $O(n)$.
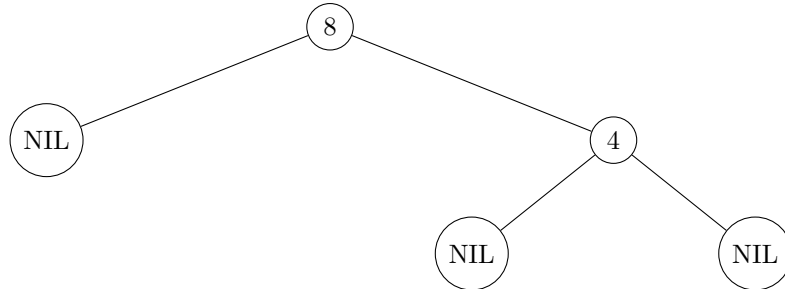
Exercise 13.2-5

Consider the BST for T_2 to be

5
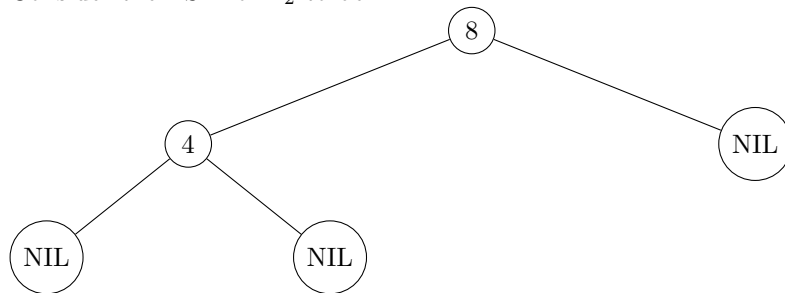
And let T_1 be

Then, there are no nodes for which its valid to call right rotate in T_1. Even though it is possible to right convert T_2 into T_1, the reverse is not possible.

For any BST T, define the quantity $f(T)$ to be the sum over all the nodes of the number of left pointers that are used in a simple path from the root to that node. Note that the contribution from each node is $O(n)$. Since there are only n nodes, we have that $f(T)$ is $O(n^2)$. Also, when we call RIGHT-ROTATE(T,x), then the contribution from x decreases by one, and the contribution from all other elements remain the same. Since $f(T)$ is a quantity that decreases by exactly one with every call of RIGHT-ROTATE, and begins $O(n^2)$, and never goes negative, we know that there can only be at most $O(n^2)$ calls of RIGHT-ROTATE on a BST.

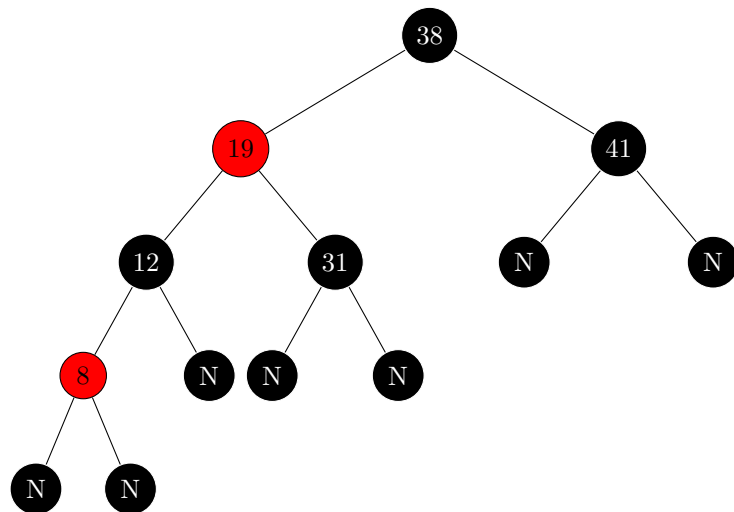**Exercise 13.3-1**

If we chose to set the color of z to black then we would be violating property 5 of being a red-black tree. Because any path from the root to a leaf under z would have one more black node than the paths to the other leaves
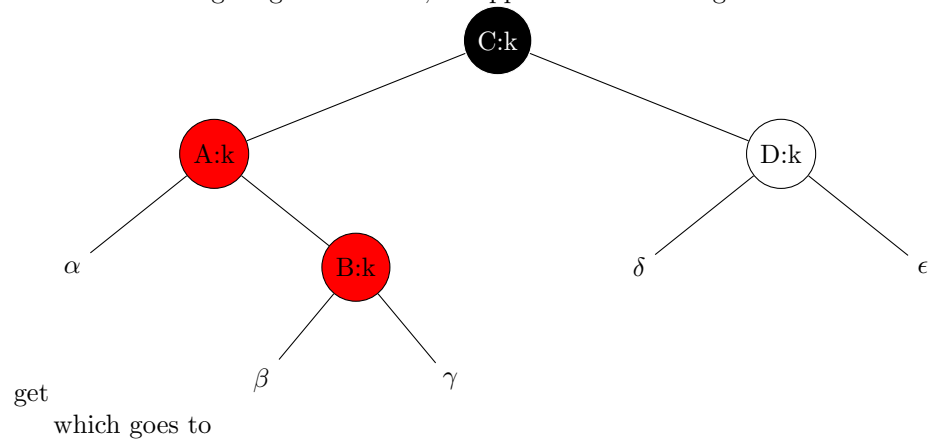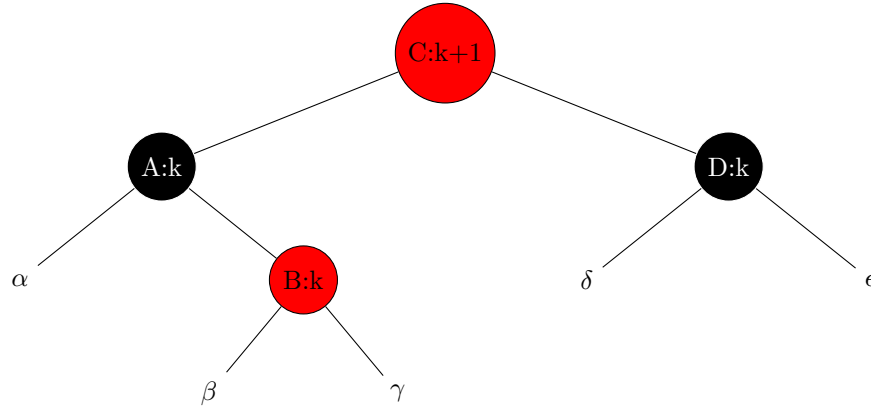
Exercise 13.3-2
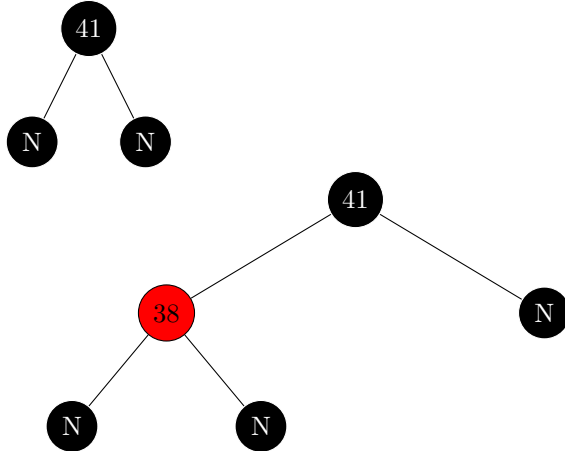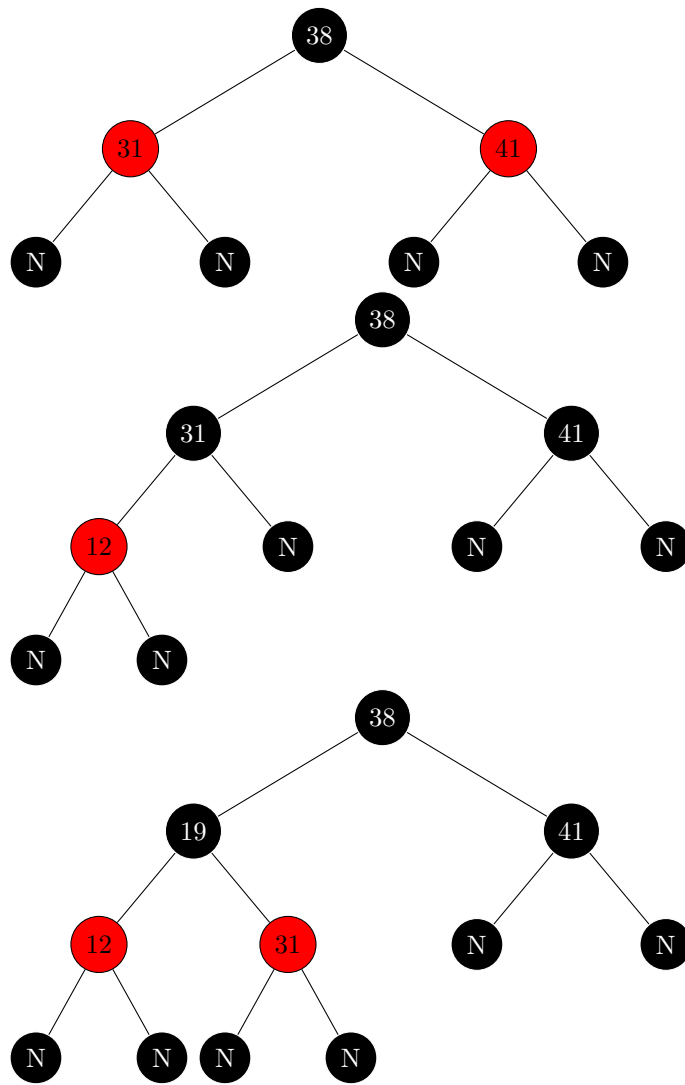
Exercise 13.3-3

For the z being a right child case, we append the black height of each node to

get

which goes to

note that while the black depths of the nodes may of changed, they are still well defined, and so they still satisfy condition 5 of being a red-black tree. Similar trees for when z is a left child.

Exercise 13.3-4

First observe that RB-INSERT-FIXUP only modifies the child of a node if it is already red, so we will never modify a child which is set to $T.nil$. We just need to check that the parent of the root is never set to red. Since the root and the parent of the root are automatically black, if z is at depth less than 2, the while loop will be broken. We only modify colors of nodes at most two levels above z, so the only case we need to worry about is if z is at depth 2. In this case we risk modifying the root to be red, but this is handled in line 16. When z is updated, it will either the root or the child of the root. Either way, the root and the parent of the root are still black, so the while condition is violated, making it impossibly to modify $T.nil$ to be red.

Exercise 13.3-5

Suppose we just added the last element. Then, prior to calling RB-INSERT-FIXUP, we have that it is red. In all of the fixup cases for an execution of the while loop, we have that the resulting tree fragment contains a red non-root node. This node will not be later made black on line 16 because it isn't the root.

Exercise 13.3-6

We need to remove line 8 from RB-INSERT and modify RB-INSERT-FIXUP. At any point in RB-INSERT-FIXUP we need only keep track of at most 2 ancestors: $z.p$ and $z.p.p$. We can find and store each of these nodes in $\log n$ time and use them for the duration of the call to RB-INSERT-FIXUP. This won't change the running time of RB-INSERT.
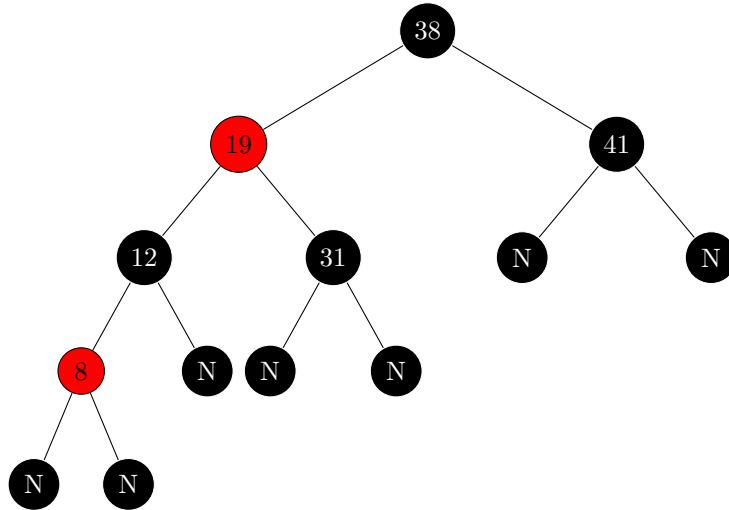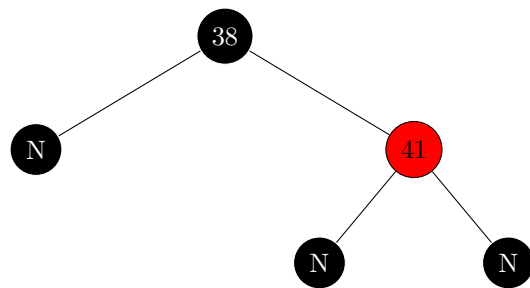
Exercise 13.4-1

There are two ways we may of left the while loop of RB-DELETE-FIXUP. The first is that we had $x = T.root$. In this case, we set $x.color = BLACK$ on line 23. So, we must have that the root is black. The other case is that we ended the while loop because we had $x.color == RED$, but had that $x \neq T.root$. This rules out case 4, because that has us setting $x = T.root$. In case 3, we don't set x to be red, or change x at all, so it couldn't of been the last case run. In case 2, we set nothing new to be RED, so this couldn't lead to exiting the while loop for this reason. In case 1, we make the sibling black and rotate it into the position of the parent. So, it wouldn't be possible to make the root red in this step because the only node we set to be red, we then placed a black node above.

Exercise 13.4-2

Suppose that both x and $x.p$ are red in RB-DELETE. This can only happen in the else-case of line 9. Since we are deleting from a red-black tree, the other child of $y.p$ which becomes $x's$ sibling in the call to RB-TRANSPLANT on line 14 must be black, so x is the only child of $x.p$ which is red. The while-loop condition of RB-DELETE-FIXUP(T,x) is immediately violated so we simply set $x.color = black$, restoring property 4.

Exercise 13.4-3
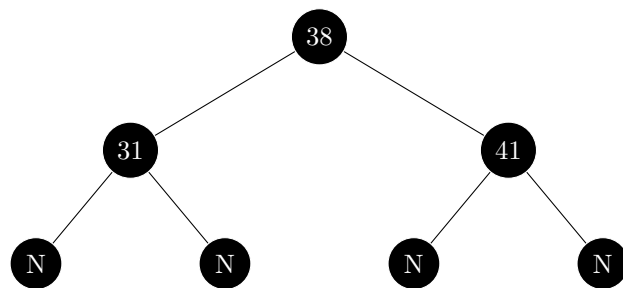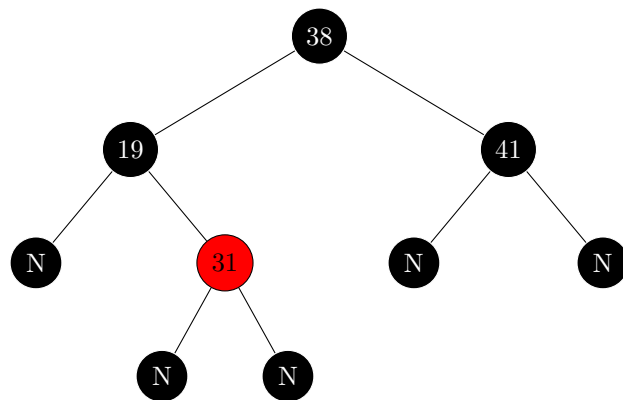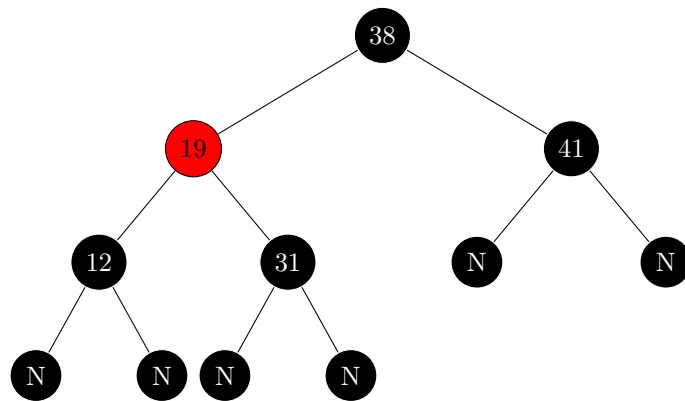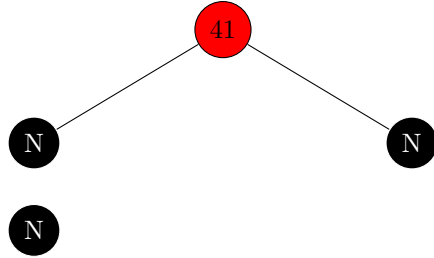
Exercise 13.4-4

Since it is possible that w is T.nil, any line of RB-DELETE-FIXUP(T,x) which examines or modifies w must be included. However, as described on page 317, x will never be T.nil, so we need not include those lines.

Exercise 13.4-5

Our count will include the root (if it is black).

Case 1: The count to each subtree is 2 both before and after

Case 2: The count to the subtrees α and β is 1+count(c) in both cases, and the count for the rest of the subtrees goes from 2+count(c) to 1+count(c). This decrease in the count for the other subtreese is handled by then having x represent an additional black.

Case 3: The count to ϵ and ζ is 2+count(c) both before and after, for all the other subtrees, it is 1+count(c) both before and after

Case 4: For α and β, the count goes from 1+count(c) to 2+count(c). For γ and δ, it is 1+count(c)+count(c') both before and after. For ϵ and ζ, it is 1+ count(c) both before and after. This increase in the count for α and β is because x before indicated an extra black.

Exercise 13.4-6

At the start of case 1 we have set w to be the sibling of x. We check on line 4 that $w.color == red$, which means that the parent of x and w cannot be red. Otherwise property 4 is violated. Thus, their concerns are unfounded.

Exercise 13.4-7

Suppose that we insert the elements $3, 2, 1$ in that order, then, the resulting tree will look like
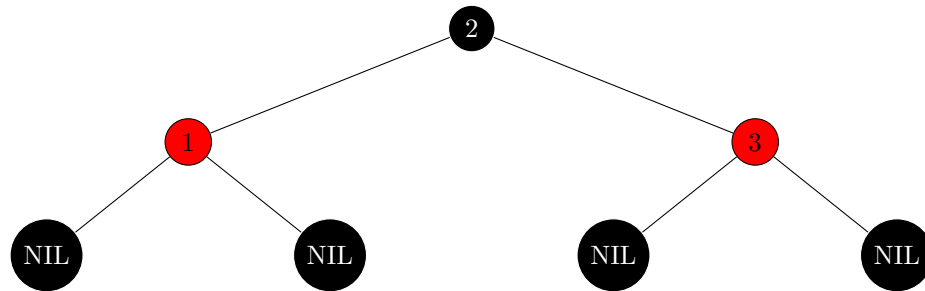
Then, after deleting 1, which was the last element added, the resulting tree is
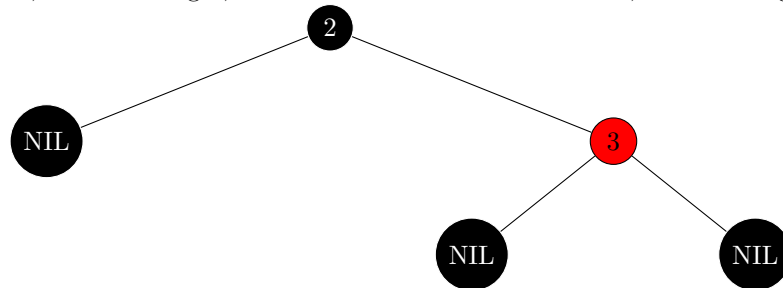
however, the tree we had before we inserted 1 in the first place was
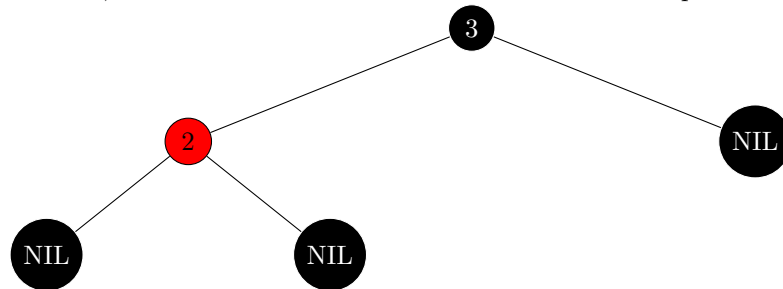
These two red black trees are clearly different

Problem 13-1

a. We need to make a new version of every node that is an ancestor of the node that is inserted or deleted.

b. See the algorithm, PERSISTENT-TREE-INSERT

c. Since the while loop will only run at most h times, since the distance from x to the root is increasing by 1 each time and bounded by the height. Also, since each iteration only takes a constant amount of time and uses a constant amount of additional space, we have that both the time and space complexity are $O(h)$.

d. When we insert an element, we need to make a new version of the root. So, any nodes that point to the root must have a new copy made so that they

13

Algorithm 2 PERSISTENT-TREE-INSERT(T,k)

$x = T.root$
if x==NIL **then**
    T.root = new node(key =k)
end if
while $x \neq NIL$ **do**
    y=x
    if k¡x.key **then**
        x= x.left
        y.left = copyof(x)
    else
        x= x.right
        y.right = copyof(x)
    end if
end while
z = new node(key = k, p = y)
if k ¡ y.key **then**
    y.left = z
else
    y.right = z
end if

---

point to the new root. So, all nodes of depth 1 must be copied. Similarly, all nodes that point to those must have new copies so that have the correct version. So, all nodes of depth 2 must be copied. Similarly, all nodes must be copied. So, we have that we need at least $\Omega(n)$ time and additional space.

e. Since the rebalancing operations can only change ancestors, and children of ancestors we only have to allocate at most $2h$ new nodes for each insertion, since the rest of the tree will be unchanged. This is of course assuming that we don't keep track of the parent pointers. This can be achieved by following the suggestions in 13.3-6 applied to both insert and delete. That is, we perform a search for the element where we store the $O(h)$ elements that are either ancestors or children of ancestors. Since these are the only nodes under consideration when doing the insertion and deletion procedure, then we can know their parents even though we aren't keeping track of the parent pointers for each node. Since the height stays $O(\lg(n))$, then, we have that everything can be done in $O(\lg(n))$.

Problem 13-2

a. When we call insert or delete we modify the black-height of the tree by at most 1 and we can modify it according to which case we're in, so no additional storage is required. When descending through T, we can determine the black-height of each node we visit in $O(1)$ time per node visited. Start by

determining the black height of the root in $O(\log n)$ time. As we move down the tree, we need only decrement the height by 1 for each black node we see to determine the new height which can be done in $O(1)$. Since there are $O(\log n)$ nodes on the path from root to leaf, the time per node is constant.

b. Find the black-height of T_1 in $O(\log n)$ time. Then find the black-height of T_2 in $O(\log n)$ time. Finally, set $z = T_1.root$. While the black height of z is strictly greater than $T_2.bh$, update z to be $z.right$ if such a child exists, otherwise update z to be $z.left$. Once the height of z is equal to $T_2.bh$, set y equal to z. The runtime of the algorithm is $O(\log n)$ since the height of T_1, and hence the number of iterations of the while-loop, is at most $O(\log n)$.

c. Let T denote the desired tree. Set $T.root = x$, $x.left = y$, $y.p = x$, $x.right = T_2.root$ and $T_2.root.p = x$. Every element of T_y is in T_1 which contains only elements smaller than x and every element of T_2 is larger than x. Since T_y and T_2 each have the binary search tree property, T does as well.

d. Color x red. Find y in T_1, as in part b, and form $T = T_y \cup \{x\} \cup T_2$ as in part c in constant time. Call T' = RB-TRANSPLANT(T,y,x). We have potentially violated the red black property if y's parent was red. To remedy this, call RB-INSERT-FIXUP(T', x).

e. In the symmetric situation, simply reverse the roles of T_1 and T_2 in parts b through d.

f. If $T_1.bh \geq T_2.bh$, run the steps outlined in part d. Otherwise, reverse the roles of parts T_1 and T_2 in d and then proceed as before. Either way, the algorithm takes $O(\log n)$ time because RB-INSERT-FIXUP is $O(\log n)$.

Problem 13-3

a. Let $T(h)$ denote the minimum size of an AVL tree of height h. Since it is height h, it must have the max of it's children's heights is equal to $h-1$. Since we are trying to get as few notes total as possible, suppose that the other child has as small of a height as is allowed. Because of the restriction of AVL trees, we have that the smaller child must be at least one less than the larger one, so, we have that $T(h) \geq T(h-1) + T(h-2) + 1$ where the +1 is coming from counting the root node. We can get inequality in the opposite direction by simply taking a tree that achieves the minimum number of number of nodes on height $h-1$ and on $h-2$ and join them together under another node. So, we have that $T(h) = T(h-1) + T(h-2) + 1$. Also, $T(0) = 0$,

15

$T(1) = 1$. This is both the same recurrence and initial conditions as the Fibonacci numbers. So, recalling equation (3.25), we have that

$$T(h) = \left\lfloor \frac{\phi^h}{\sqrt{5}} + \frac{1}{2} \right\rfloor \le n$$

Rearranging for h, we have

$$\frac{\phi^h}{\sqrt{5}} - \frac{1}{2} \le n$$

$$\phi^h \le \sqrt{5} \left(n + \frac{1}{2} \right)$$

$$h \le \frac{\lg(\sqrt{5}) + \lg(n + \frac{1}{2})}{\lg(\phi)} \in O(\lg(n))$$

b. Let UNBAL(x) denote x.left.h - x.right.h. Then, the algorithm BALANCE does what is desired. Note that because we are only rotating a single element at a time, the value of UNBAL(x) can only change by at most 2 in each step. Also, it must eventually start to change as the tree that was shorter becomes saturated with elements. We also fix any breaking of the AVL property that rotating may of caused by our recursive calls to the children.

Algorithm 3 BALANCE(x)

---

while $|UNBAL(x)| > 1$ **do**
    if $UNBAL(x) > 0$ **then**
        RIGHT-ROTATE(T,x)
    else
        LEFT-ROTATE(T,x)
    end if
    BALANCE(x.left)
    BALANCE(x.right)
end while

---

c. For the given algorithm AVL-INSERT(x,z), it correctly maintains the fact that it is a BST by the way we search for the correct spot to insert z. Also, we can see that it maintains the property of being AVL, because after inserting the element, it checks all of the parents for the AVL property, since those are the only places it could of broken. It then fixes it and also updates the height attribute for any of the nodes for which it may of changed.

d. Since both for loops only run for $O(h) = O(\lg(n))$ iterations, we have that that is the runtime. Also, only a single rotation will occur in the second while loop because when we do it, we will be decreasing the height of the subtree rooted there, which means that it's back down to what it was before, so all of it's ancestors will have unchanged heights, so, no further balancing will be required.

Algorithm 4 AVL-INSERT(x,z)

w = x
while $w \neq NIL$ **do**
    y = w
    if $z.key > y.key$ **then**
        w= w.right
    else
        w = w.left
    end if
end while

if $z.key > y.key$ **then**
    y.right = z
    if y.left = NIL **then**
        y.h = 1
    end if
else
    y.left = z
    if y.right = NIL **then**
        y.h = 1
    end if
end if
while $y \neq x$ **do**
    $y.h = 1 + \max\{y.left.h, y.right.h\}$
    if $y.left.h > y.right.h + 1$ **then**
        RIGHT-ROTATE(T,y)
    end if
    if $y.right.h > y.left.h + 1$ **then**
        LEFT-ROTATE(T,y)
        y= y.p
    end if
end while

Problem 13-4

a. The root r is uniquely determined because it must contain the smallest priority. Then we partition the set of nodes into those which have key values less than r and those which have values greater than r. We must make a treap out of each of these and make them the left and right children of r. By induction on the number of nodes, we see that the treap is uniquely determined.

b. Since choosing random priorities corresponds to inserting in a random order, the expected height of a treap is the same as the expected height of a randomly built binary search tree, $\Theta(\log n)$.

c. First insert a node as usual using the binary-search-tree insertion procedure. Then perform left and right rotations until the parent of the inserted node no longer has larger priority.

d. The expected runtime of TREAP-INSERT is $\Theta(\log n)$ since the expected height of a treap is $\Theta(\log n)$.

e. To insert x, we initially run the BST insert procedure, so x is a leaf node. Every time we perform a left rotation, we increase the length of the right spine of the left subtree by 1. Every time we perform a right rotation, we increase the length of the left spine of the right subtree by 1. Since we only perform left and right rotations, the claim follows.

f. If $X_{ik} = 1$ then the properties must hold by the binary-search-tree property and the definition of treap. On the other hand, suppose $y.key < z.key < x.key$ implies $y.priority < z.priority$. If y wasn't a child of x then taking z to be the lowest common ancestor of x and y would violate this. Since $y.priority > x.priority$, y must be a child of x. Since $y.key < x.key$, y is in the left subtree of x. If y is not in the right spine of the left subtree of x then there must exist some z such that $y.priority > z.priority > x.priority$ and $y.key < z.key < x.key$, a contradiction.

g. We need to compute the probability that the conditions of part f are satisfied. For all $z \in [i + 1, k - 1]$ we must have $x.priority < y.priority < z.priority$. There are $(k-i-1)!$ ways to permute the priorities corresponding to these z, out of $(k-i+1)!$ ways to permute the priorities corresponding to all elements in $[i, k]$. Cancellation gives $P\{X_{ik}\} = \frac{1}{(k-i+1)(k-i)}$.

18

h. We use part g then simplify the telescoping series:

$$E[C] = \sum_{j=1}^{k-1} E[X_{jk}]$$

$$= \sum_{j=1}^{k-1} \frac{1}{(k-j+1)(k-j)}$$

$$= \sum_{j=1}^{k-1} \frac{1}{j(j+1)}$$

$$= \sum_{j=1}^{k-1} \frac{1}{j} - \frac{1}{j+1}$$

$$= 1 - \frac{1}{k}.$$

i. A node y is in the left spine of the right subtree of x if and only if it would be in the right spine of the left subtree of x in the treap where every node with key k is replaced by a node with key $n - k$. Replacing k by $n - k$ in the expectation computation of part h gives the result.

j. By part e, the number of rotations is $C + D$. By linearity of expectation, $E[C + D] = 2 - \frac{1}{k} - \frac{1}{n-k+1} \leq 2$ for any choice of k.

Chapter 14

Michelle Bodnar, Andrew Lohr

April 12, 2016

Exercise 14.1-1

The call sequence is:
$OS - SELECT(T.root, 10)$
$OS - SELECT(T.root.left, 10)$
$OS - SELECT(T.root.left.right, 2)$
$OS - SELECT(T.root.left.right.left, 2)$
$OS - SELECT(T.root.left.right.left.right, 1)$
Then, we have that the node (with key 20) that is returned is $T.root.left.right.left.right$

Exercise 14.1-2

OS-RANK(T,x) operates as follows. r is set to 0 and y is set to x. On the first iteration of the while loop, y is set to the node with key 38. On the second iteration, r is increased to 2 and y is set to the node with key 30. On the third iteration, y is set to the node with key 41. On the fourth iteration r is increased to 15 and y is set to the node with key 26, the root. This breaks the while loop, and rank 15 is returned.

Exercise 14.1-3

See the algorithm OS-SELECT'
Exercise 14.1-4

See the algorithm RECURSIVE-OS-KEY-RANK.

Exercise 14.1-5

The desired result is OS-SELECT(T,OS-RANK(T,x)+i). This has runtime O(h), which by the properties of red black trees, is $O(\lg(n))$.

Exercise 14.1-6

First perform the usual BST insertion procedure on z, the node to be inserted. Then add 1 to the rank of every node on the path from the root to z

Algorithm 1 O

S-SELECT'(x,i)

  r = x.left.size

  while $i \neq r$ **do**

    if $i > r$ **then**

      x = x.right

      i = i-r

    else

      x=x.left

    end if

    r = x.left.size

  end while

  return x

Algorithm 2 RECURSIVE-OS-KEY-RANK(T,k)

1: **if** $T.root.key == k$ **then**

2:    Return $T.root.left.size + 1$

3: **else if** $T.root.key > k$ **then**

4:    Return $RECURSIVE - OS - KEY - RANK(T.root.left, k)$

5: **else**

6:    Return $RECURSIVE - OS - KEY - RANK(T.root.right, k) + T.root.left.size + 1$

7: **end if**

such that z is in the left subtree of that node. Since the added node is a leaf, it will have no subtrees so its rank will always be 1. When a left rotation is performed on x, its rank within its subtree will remain the same. The rank of $x.right$ will be increased by the rank of x, plus one. If we perform a right rotation on a node y, its rank will decrement by $y.left.rank + 1$. The rank of $y.left$ will remain unchanged. For deletion of z, decrement the rank of every node on the path from z to the root such that z is in the left subtree of that node. For any rotations, use the same rules as before.

Exercise 14.1-7

See the algorithm INV-COUNT(L). It does assume that all the elements of the list are distinct. To adapt it to the not neccesarily distinct case, each time that the we do the search, we should be selecting the element with that key that comes first in an inorder traversal.

Algorithm 3 INV-COUNT(L)

Construct an order statistic tree T for all the elements in L
$t = -|L|$
for i from 0 to $|L| - 1$ **do**
    $t = t + OS - RANK(T, SEARCH(T, L[i]))$
    remove the node corresponding to $L[i]$ from T
end for
return t

Exercise 14.1-8

Choose a point on the circle and assign it key value 1. The rank of any point will be its position when read in the sequence starting with point 1, and reading clockwise around the circle. Next, we label each point with a key. Reading clockwise around the circle, if a point's chordal companion has a key, assign it the same key. Otherwise, assign it the next lowest unused integer key value. Then use the algorithm given in the solution to problem 2-4 d to count the number of inversions (based on key value), with the caveat that an inversion from key i to key j (with $i > j$)doesn't count if the rank of the companion of i is smaller than the rank of j.

Exercise 14.2-1

Along all the nodes in an inorder traversal, add a prev and succ pointer. If we keep the head to be the first node, and, make it circularly linked, then this clearly allows for the four operations to work in constant time. We still need to be sure to maintain this linked list structure throughout all the tree modifications. Suppose we insert a node into the BST to be a left child, then we can

insert it into this doubly linked list immediately before it's parent, which can be done in constant time. Similarly, if it is a right child, then we would insert it immediately after its parent. Deletion of the element is just the usual deletion in a linked list.

Exercise 14.2-2

Since the black height of a node depends only on the black height and color of its children, Theorem 14.1 implies that we can maintain the attribute without affecting the asymptotic performance of the other red-black tree operations. The same is not true for maintaining the depths of nodes. If we delete the root of a tree we could potentially have to update the depths of $O(n)$ nodes, making the DELETE operation asymptotically slower than before.

Exercise 14.2-3

After performing the rotate operation, starting at the deeper of the two nodes that were moved by the rotate, say x, set $x.f = x.left.f \otimes x.a \otimes x.right.f$. Then, do the same thing for the higher up node in the rotation. For size, instead set $x.size = x.left.size + x.right.size + 1$ and then do the same for the higher up node after the rotation.

Exercise 14.2-4

The following algorithm runs in $\Theta(m + \lg n)$ time. There could be as many as $O(\lg n)$ recursive calls to locate the smallest and largest elements necessary to print, and every other constant time call will print one of the m keys between a and b. We can't do better than this because there are m keys to output, and to find a key requires $\lg n$ time to search.

Algorithm 4 RB-ENUMERATE(x,a,b)

  if $a \leq x.key \leq b$ **then**
     print x
  end if
  if $a \leq x.key$ and $x.left \neq NIL$ **then**
    $RB - ENUMERATE(x.left, a, b)$
  end if
  if $x.key \leq b$ and $x.right \neq NIL$ **then**
    $RB - ENUMERATE(x.right, a, b)$
  end if
  Return

Exercise 14.3-1

after rearranging the nodes, starting with the lower of the two nodes moved,

set it's max attribute to be the maximum of its right endpoint and the the two max attributes of its children. Do the same for the higher up of the two moved nodes.

Exercise 14.3-2

Change the weak inequality on line 3 to be strict.

Exercise 14.3-3

Consider the usual interval search given, but, instead of breaking out of the loop asa soon as we have an overlap, we just keep track of the most recently seen overlap, and keep going in the loop until we hit $T.nil$. We then return the most recently seen overlap. We have that this is the overlapping interval with minimum left end point because the search always goes to the left it contains an overlapping interval, and the left children are the ones with smaller left endpoint.

Exercise 14.3-4

Algorithm 5 INTERVAL(T,i)
---
if $T.root$ overlaps i **then**
    Print $T.root$
end if
if $T.root.left \neq T.nil$ and $T.root.left.max \geq i.low$ **then**
    INTERVAL(T.root.left,i)
end if
if $T.root.right \neq T.nil$ and $T.root.right.max \geq i.low$ and $T.root.right.key \leq i.high$ **then**
    INTERVAL(T.root.right,i)
end if

The algorithm examines each node at most twice and performs constant time checks so the runtime cannot exceed $O(n)$. If a recursive call is made on a branch of the tree, then that branch must contain an overlapping interval, so the runtime also cannot exceed $O(k \lg n)$ since the height is at most n and there are k intervals in the output list.

Exercise 14.3-5

We could modify the interval tree insertion procedure to, once you find a place to put the given interval, then we perform an insertion procedure based on the right hand endpoints. Then, to perform INTERVAL-SEARCH-EXACTLY(T,i) first perform a search for the left hand endpoint, then, perform a search for the right hand endpoint based on the BST rooted at this node for

the right hand endpoint, stopping the search if we ever come across a element with a different left hand endpoint.

Exercise 14.3-6

Store the elements in a red-black tree, where the key value is the value of each number itself. The auxiliary attribute stored at a node x will be the min gap between elements in the subtree rooted at x, the maximum value contained in subtree rooted at x, and the minimum value contained in the subtree rooted at x. The min gap at a leaf will be ∞. Since we can determine the attributes of a node x using only the information about the key at x, and the attributes in $x.left$ and $x.right$, Theorem 14.1 implies that we can maintain the values in all nodes of the tree during insertion and deletion without asymptotically affecting their $O(\lg n)$ performance. For MIN-GAP, just check the min gap at the root, in constant time.

Exercise 14.3-7

Let L be the set of left coordinates of rectangles. Let R be the set of right coordinates of rectangles. Sort both of these sets in $O(n \lg(n))$ time. Then, we will have a pointer to L and a pointer to R. If the pointer to L is smaller, call interval search on T for the up-down interval corresponding to this left hand side. If it contains something that intersects the up-down bounds of this rectangle, there is an intersection, so stop. Otherwise add this interval to T and increment the pointer to L. If R is the smaller one, remove the up-down interval that that right hand side corresponds to and increment the pointer to R. Since all the interval tree operations used run in time $O(\lg(n))$ and we only call them at most $3n$ times, we have that the runtime is $O(n \lg(n))$.

Problem 14-1

a. Suppose we have a point of maximum overlap p. Then, as long as we imagine moving the point p but don't pass any of the endpoints of any of the intervals, then we won't be changing the number of intervals containing p. So, we just move it to the right until we hit the endpoint of some interval, then, we have a point of maximum overlap that is the endpoint of an interval.

b. We will present a simple solution to this problem that runs in time $O(n \lg(n))$ which doesn't augment a red-black tree even though that is what is suggested by the hint. Consider a list of elements so that each element has a integer $x.pos$ and a field that says whether it is a left endpoint or a right endpoint $x.dir = L$ or R. Then, sort this list on the pos attribute of each element. Then, run through the list with a running total of how many intervals you are currently in, subtracting one for each right endpoint and adding one for each left endpoint. Also keep track of the running max of these values, and

the endpoint that has that value. Then, this point that attains the running max is what should be returned.

Problem 14-2

a. Create a circular doubly linked list. Continue to advance m places in the list (not counting the sentinel), followed by printing and deleting the current node, until the list is empty. Since m is a constant and we advance at most mn places, the runtime is $O(n)$.

b. Begin by creating an order statistic tree for the given elements, where rank starts at 0 and ends at $n-1$, the rank of a node is its position in the original order minus 1, and we store each element's value in an attribute $x.value$. Print the element with rank $m(n) - 1$, then delete it from the tree. Then proceed as follows: If you've just printed the kth value which had rank r, delete that node from the tree, and the $(k+1)^{st}$ value to be printed will have rank $r - 1 + m(n) \mod (n-k)$. Since deletion and lookup take $O(\lg n)$ and there are n nodes, the runtime is $O(n \lg n)$.

Chapter 15

Michelle Bodnar, Andrew Lohr

April 12, 2016

Exercise 15.1-1

Proceed by induction. The base case of $T(0) = 2^0 = 1$. Then, we apply the inductive hypothesis and recall equation (A.5) to get that

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) = 1 + \sum_{j=0}^{n-1} 2^j = 1 + \frac{2^n - 1}{2 - 1} = 1 + 2^n - 1 = 2^n$$

Exercise 15.1-2

Let $p_1 = 0$, $p_2 = 4$, $p_3 = 7$ and $n = 4$. The greedy strategy would first cut off a piece of length 3 since it has highest density. The remaining rod has length 1, so the total price would be 7. On the other hand, two rods of length 2 yield a price of 8.

Exercise 15.1-3

Now, instead of equation (15.1), we have that

$$r_n = \max\{p_n, r_1 + r_{n-1} - c, r_2 + r_{n-2} - c, \ldots, r_{n-1} + r_1 - c\}$$

And so, to change the top down solution to this problem, we would change MEMOIZED-CUT-ROD-AUX(p,n,r) as follows. The upper bound for i on line 6 should be $n - 1$ instead of n. Also, after the for loop, but before line 8, set $q = \max\{q - c, p[i]\}$.

Exercise 15.1-4

Create a new array called s. Initialize it to all zeros in MEMOIZED-CUT-ROD(p,n) and pass it as an additional argument to MEMOIZED-CUT-ROD-AUX(p,n,r,s). Replace line 7 in MEMOIZED-CUT-ROD-AUX by the following: $t = p[i] + MEMOIZED - CUT - ROD - AUX(p, n - i, r, s)$. Following this, if $t > q$, set $q = t$ and $s[n] = i$. Upon termination, $s[i]$ will contain the size of the first cut for a rod of size i.

Exercise 15.1-5

The subproblem graph for $n = 4$ looks like
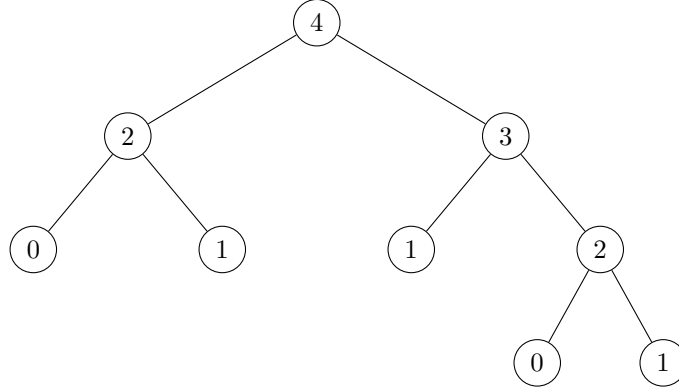
The number of vertices in the tree to compute the nth Fibonacci will follow the recurrence
$$V(n) = 1 + V(n-2) + V(n-1)$$

And has initial condition $V(1) = V(0) = 1$. This has solution $V(n) = 2 * Fib(n) - 1$ which we will check by direct substitution. For the base cases, this is simple to check. Now, by induction, we have

$$V(n) = 1 + 2 * Fib(n-2) - 1 + 2 * Fib(n-1) - 1 = 2 * Fib(n) - 1$$

The number of edegs will satisfy the recurrence

$$E(n) = 2 + E(n-1) + E(n-2)$$

and having base cases $E(1) = E(0) = 0$. So, we show by induction that we have $E(n) = 2 * Fib(n) - 2$. For the base cases it clearly holds, and by induction, we have

$$E(n) = 2 + 2 * Fib(n-1) - 2 + 2 * Fib(n-2) - 2 = 2 * Fib(n) - 2$$

We will present a $O(n)$ bottom up solution that only keeps track of the the two largest subproblems so far, since a subproblem can only depend on the solution to subproblems at most two less for Fibonacci.

Exercise 15.2-1

An optimal parenthesization of that sequence would be $(A_1 A_2)((A_3 A_4)(A_5 A_6))$ which will require $5 * 50 * 6 + 3 * 12 * 5 + 5 * 10 * 3 + 3 * 5 * 6 + 5 * 3 * 6 = 1500 + 180 + 150 + 90 + 90 = 2010$.

Exercise 15.2-2

Algorithm 1 DYN-FIB(n)

---

  prev = 1
  prevprev = 1
  if $n \leq 1$ **then**
     return 1
  end if
  for i=2 upto n **do**
     tmp = prev + prevprev
     prevprev = prev
     prev = tmp
  end for
  return prev

Algorithm 2 MATRIX-CHAIN-MULTIPLY(A,s,i,j)

---

  if $i == j$ **then**
     Return A_i
  end if
  Return   MATRIX-CHAIN-MULTIPLY(A,s,i,s[i,j]) · MATRIX-CHAIN-MULTIPLY(A,s,s[i,j]+1,j)

The following algorithm actually performs the optimal multiplication, and is recursive in nature:

Exercise 15.1-3

By indution we will show that $P(n)$ from eq (15.6) is $\geq 2^n - 1 \in \Omega(2^n)$. The base case of n=1 is trivial. Then, for $n \geq 2$, by induction and eq (15.6), we have

$$P(n) = \sum_{k=1}^{n-1} P(k)P(n-k) \geq \sum_{k=1}^{n-1} 2^k 2^{n-k} = (n-1)(2^n - 1) \geq 2^n - 1$$

So, the conclusion holds.

Exercise 15.2-4

The subproblem graph for matrix chain multiplication has a vertex for each pair (i, j) such that $1 \leq i \leq j \leq n$, corresponding to the subproblem of finding the optimal way to multiply $A_i A_{i+1} \cdots A_j$. There are $n(n-1)/2 + n$ vertices. Vertex (i, j) is connected by an edge directed to vertex (k, l) if $k = i$ and $k \leq l < j$ or $l = j$ and $i < k \leq j$. A vertex (i, j) has outdegree $2(j - i)$. There are $n - k$ vertices such that $j - i = k$, so the total number of edges is

$$\sum_{k=0}^{n-1} 2k(n-k).$$

3

Exercise 15.1-5

We count the number of times that we reference a different entry in m than the one we are computing, that is, 2 times the number of times that line 10 runs.

$$\sum_{l=2}^{n} \sum_{i=l}^{n-l+1} \sum_{k=i}^{i+l-2} 2 = \sum_{l=2}^{n} \sum_{i=1}^{n-l+1} (l-1)2 = \sum_{l=2}^{n} 2(l-1)(n-l+1)$$

$$= \sum_{l=1}^{n-1} 2l(n-l)$$

$$= 2n \sum_{l=1}^{n-1} l - 2 \sum_{l=1}^{n-1} l^2$$

$$= n^2(n-1) - \frac{(n-1)(n)(2n-1)}{3}$$

$$= n^3 - n^2 - \frac{2n^3 - 3n^2 + n}{3}$$

$$= \frac{n^3 - n}{3}$$

Exercise 15.2-6

We proceed by induction on the number of matrices. A single matrix has no pairs of parentheses. Assume that a full parenthesization of an n-element expression has exactly $n-1$ pairs of parentheses. Given a full parenthesization of an $n+1$-element expression, there must exist some k such that we first multiply $B = A_1 \cdots A_k$ in some way, then multiply $C = A_{k+1} \cdots A_{n+1}$ in some way, then multiply B and C. By our induction hypothesis, we have $k-1$ pairs of parentheses for the full parenthesization of B and $n+1-k-1$ pairs of parentheses for the full parenthesization of C. Adding these together, plus the pair of outer parentheses for the entire expression, yields $k-1+n+1-k-1+1 = (n+1)-1$ parentheses, as desired.

Exercise 15.3-1

The runtime of of enumerating is just $n * P(n)$, while is we were running RECURSIVE-MATRIX-CHAIN, it would also have to run on all of the internal nodes of the subproblem tree. Also, the enumeration approach wouldn't have as much overhead.

Exercise 15.3-2

Let $[i..j]$ denote the call to Merge Sort to sort the elements in positions i through j of the original array. The recursion tree will have $[1..n]$ as its root, and at any node $[i..j]$ will have $[i..(j-i)/2]$ and $[(j-i)/2+1..j]$ as its left

and right children, respectively. If $j - i = 1$, there will be no children. The memoization approach fails to speed up Merge Sort because the subproblems aren't overlapping. Sorting one list of size n isn't the same as sorting another list of size n, so there is no savings in storing solutions to subproblems since each solution is used at most once.

Exercise 15.3-3

This modification of the matrix-chain-multiplication problem does still exhibit the optimal substructure property. Suppose we split a maximal multiplication of $A_1, \ldots, A_n$ between A_k and A_{k+1} then, we must have a maximal cost multiplication on either side, otherwise we could substitute in for that side a more expensive multiplication of $A_1, \ldots, A_n$.

Exercise 15.3-4

Suppose that we are given matrices A_1, A_2, A_3, and A_4 with dimensions such that $p_0, p_1, p_2, p_3, p_4 = 1000, 100, 20, 10, 1000$. Then $p_0 p_k p_4$ is minimized when $k = 3$, so we need to solve the subproblem of multiplying $A_1 A_2 A_3$, and also A_4 which is solved automatically. By her algorithm, this is solved by splitting at $k = 2$. Thus, the full parenthesization is $(((A_1 A_2) A_3) A_4)$. This requires $1000 \cdot 100 \cdot 20 + 1000 \cdot 20 \cdot 10 + 1000 \cdot 10 \cdot 1000 = 12,200,000$ scalar multiplications. On the other hand, suppose we had fully parenthesized the matrices to multiply as $((A_1(A_2 A_3))A_4)$. Then we would only require $100 \cdot 20 \cdot 10 + 1000 \cdot 100 \cdot 10 + 1000 \cdot 10 \cdot 1000 = 11,020,000$ scalar multiplications, which is fewer than Professor Capulet's method. Therefore her greedy approach yields a suboptimal solution.

Exercise 15.3-5

The optimal substructure property doesn't hold because the number of pieces of length i used on one side of the cut affects the number allowed on the other. That is, there is information about the particular solution on one side of the cut that changes what is allowed on the other.

To make this more concrete, suppose the rod was length 4, the values were $l_1 = 2, l_2 = l_3 = l_4 = 1$, and each piece has the same worth regardless of length. Then, if we make our first cut in the middle, we have that the optimal solution for the two rods left over is to cut it in the middle, which isn't allowed because it increases the total number of rods of length 1 to be too large.

Exercise 15.3-6

First we assume that the commission is always zero. Let k denote a currency which appears in an optimal sequence s of trades to go from currency 1 to currency n. p_k denote the first part of this sequence which changes currencies from 1 to k and q_k denote the rest of the sequence. Then p_k and q_k are both

optimal sequences for changing from 1 to k and k to n respectively. To see this, suppose that p_k wasn't optimal but that p'_k was. Then by changing currencies according to the sequence $p'_k q_k$ we would have a sequence of changes which is better than s, a contradiction since s was optimal. The same argument applies to q_k.

Now suppose that the commissions can take on arbitrary values. Suppose we have currencies 1 through 6, and $r_{12} = r_{23} = r_{34} = r_{45} = 2$, $r_{13} = r_{35} = 6$, and all other exchanges are such that $r_{ij} = 100$. Let $c_1 = 0$, $c_2 = 1$, and $c_k = 10$ for $k \geq 3$. The optimal solution in this setup is to change 1 to 3, then 3 to 5, for a total cost of 13. An optimal solution for changing 1 to 3 involves changing 1 to 2 then 2 to 3, for a cost of 5, and an optimal solution for changing 3 to 5 is to change 3 to 4 then 4 to 5, for a total cost of 5. However, combining these optimal solutions to subproblems means making more exchanges overall, and the total cost of combining them is 18, which is not optimal.

Exercise 15.4-1

An LCS is $\langle 1, 0, 1, 0, 1, 0 \rangle$. A concise way of seeing this is by noticing that the first list contains a "00" while the second contains none, Also, the second list contains two copies of "11" while the first contains none. Inorder to reconcile this, any LCS will have to skip at least three elements. Since we managed to do this, we know that our common subsequence was maximal.

Exercise 15.4-2

The algorithm PRINT-LCS(c,X,Y) prints the LCS of X and Y from the completed table by computing only the necessary entries of B on the fly. It runs in $O(m + n)$ time because each iteration of the while loop decrements either i or j or both by 1, and halts when either reaches 0. The final for loop iterates at most $\min(m, n)$ times.

Exercise 15.4-3

Exercise 15.4-4

Since we only use the previous row of the c table to compute the current row, we compute as normal, but when we go to compute row k, we free row $k-2$ since we will never need it again to compute the length. To use even less space, observe that to compute $c[i, j]$, all we need are the entries $c[i-1, j], c[i-1, j-1]$, and $c[i, j-1]$. Thus, we can free up entry-by-entry those from the previous row which we will never need again, reducing the space requirement to $\min(m, n)$. Computing the next entry from the three that it depends on takes $O(1)$ time and space.

Exercise 15.4-5

Given a list of numbers L, make a copy of L called L' and then sort L'.

Algorithm 3 PRINT-LCS(c,X,Y)

$n = c[X.length, Y.length]$
Initialize an array s of length n
$i = X.length$ and $j = Y.length$
while $i > 0$ and $j > 0$ **do**
    if $x_i == y_j$ **then**
        $s[n] = x_i$
        $n = n - 1$
        $i = i - 1$
        $j = j - 1$
    else if $c[i - 1, j] \geq c[i, j - 1]$ **then**
        $i = i - 1$
    else
        $j = j - 1$
    end if
end while
for $k = 1$ to $s.length$ **do**
    Print $s[k]$
end for

Algorithm 4 MEMO-LCS-LENGTH-AUX(X,Y,c,b)

---
m $= |X|$
n $= |Y|$
if $c[m, n]! = 0$ or $m == 0$ or $n == 0$ **then**
    return
end if
if $x_m == y_n$ **then**
    $b[m, n] = \nwarrow$
    c[m,n] =MEMO-LCS-LENGTH-AUX(X[1,..., m-1],Y[1,...,n-1],c,b) +1
else if $MEMO - LCS - LENGTH - AUX(X[1, \ldots, m - 1], Y, c, b) \geq MEMO - LCS - LENGTH - AUX(X, Y[1, \ldots, n - 1], c, b)$ **then**
    $b[m, n] = \uparrow$
    c[m,n] =MEMO-LCS-LENGTH-AUX(X[1,..., m-1],Y,c,b)
else
    $b[m, n] = \leftarrow$
    c[m,n] =MEMO-LCS-LENGTH-AUX(X,Y[1,...,n-1],c,b)
end if

Algorithm 5 MEMO-LCS-LENGTH(X,Y)

---
let c be a (passed by reference) $|X|$ by $|Y|$ array initiallized to 0
let b be a (passed by reference) $|X|$ by $|Y|$ array
MEMO-LCS-LENGTH-AUX(X,Y,c,b)
return c and b

Then, just run the LCS algorithm on these two lists. The longest common subsequence must be monotone increasing because it is a subsequence of L' which is sorted. It is also the longest monotone increasing subsequence because being a subsequence of L' only adds the restriction that the subsequence must be monotone increasing. Since $|L| = |L'| = n$, and sorting L can be done in $o(n^2)$ time, the final running time will be $O(|L||L'|) = O(n^2)$.

Exercise 15.4-6

The algorithm LONG-MONOTONIC(S) returns the longest monotonically increasing subsequence of S, where S has length n. The algorithm works as follows: a new array B will be created such that $B[i]$ contains the last value of a longest monotonically increasing subsequence of length i. A new array C will be such that $C[i]$ contains the monotonically increasing subsequence of length i with smallest last element seen so far. To analyze the runtime, observe that the entries of B are in sorted order, so we can execute line 9 in $O(\log(n))$ time. Since every other line in the for-loop takes constant time, the total run-time is $O(n \log n)$.

Algorithm 6 LONG-MONOTONIC(S)

1: Initialize an array B of integers length of n, where every value is set equal to ∞.
2: Initialize an array C of empty lists length n.
3: $L = 1$
4: **for** $i = 1$ to n **do**
5:     **if** $A[i] < B[1]$ **then**
6:         $B[1] = A[i]$
7:         $C[1].head.key = A[i]$
8:     **else**
9:         Let j be the largest index of B such that $B[j] < A[i]$
10:         $B[j+1] = A[i]$
11:         $C[j+1] = C[j]$
12:         $C[j+1].insert(A[i])$
13:         **if** $j+1 > L$ **then**
14:             $L = L + 1$
15:         **end if**
16:     **end if**
17: **end for**
18: Print $C[L]$

Exercise 15.5-1

Run the given algorithm with the initial argument of $i = 1$ and $j = m[1].length$.
Exercise 15.5-2

Algorithm 7 CONSTRUCT-OPTIMAL-BST(root,i,j)

---
if $i > j$ **then**
    return nil
end if
if $i == j$ **then**
    return a node with key k_i and whose children are nil
end if
let n be a node with key $k_{root[i,j]}$
n.left = CONSTRUCT-OPTIMAL-BST(root,i,root[i,j]-1)
n.right = CONSTRUCT-OPTIMAL-BST(root,root[i,j]+1,j)
return n

After painstakingly working through the algorithm and building up the tables, we find that the cost of the optimal binary search tree is 3.12. The tree takes the following structure:
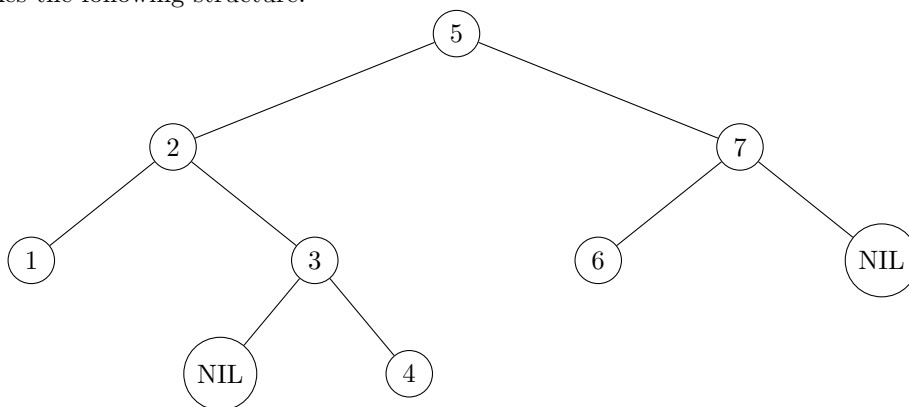
Exercise 15.5-3

Each of the $\Theta(n^2)$ values of $w[i, j]$ would require computing those two sums, both of which can be of size $O(n)$, so, the asymptotic runtime would increase to $O(n^3)$.

Exercise 15.5-4

Change the for loop of line 10 in OPTIMAL-BST to "for $r = r[i, j - 1]$ to $r[i+1, j]$". Knuth's result implies that it is sufficient to only check these values because optimal root found in this range is in fact the optimal root of some binary search tree. The time spent within the for loop of line 6 is now $\Theta(n)$. This is because the bounds on r in the new for loop of line 10 are nonoverlapping. To see this, suppose we have fixed l and i. On one iteration of the for loop of line 6, the upper bound on r is $r[i + 1, j] = r[i + 1, i + l - 1]$. When we increment i by 1 we increase j by 1. However, the lower bound on r for the next iteration subtracts this, so the lower bound on the next iteration is

$r[i+1, j+1-1] = r[i+1, j]$. Thus, the total time spent in the for loop of line 6 is $\Theta(n)$. Since we iterate the outer for loop of line 5 n times, the total runtime is $\Theta(n^2)$.

Problem 15-1

Since any longest simple path must start by going through some edge out of s, and thereafter cannot pass through s because it must be simple, that is,

$$LONGEST(G, s, t) = 1 + \max_{s \sim s'}\{LONGEST(G|_{V\setminus\{s\}}, s', t)\}$$

with the base case that if $s = t$ then we have a length of 0.

A naive bound would be to say that since the graph we are considering is a subset of the vertices, and the other two arguments to the substructure are distinguished vertices, then, the runtime will be $O(|V|^2 2^{|V|})$. We can see that we can actually will have to consider this many possible subproblems by taking $|G|$ to be the complete graph on $|V|$ vertices.

Problem 15-2

Let $A[1..n]$ denote the array which contains the given word. First note that for a palindrome to be a subsequence we must be able to divide the input word at some position i, and then solve the longest common subsequence problem on $A[1..i]$ and $A[i+1..n]$, possibly adding in an extra letter to account for palindromes with a central letter. Since there are n places at which we could split the input word and the LCS problem takes time $O(n^2)$, we can solve the palindrome problem in time $O(n^3)$.

Problem 15-3

First sort all the points based on their x coordinate. To index our subproblem, we will give the rightmost point for both the path going to the left and the path going to the right. Then, we have that the desired result will be the subproblem indexed by v,v where v is the rightmost point. Suppose by symmetry that we are further along on the left-going path, that the leftmost path is going to the ith one and the right going path is going until the jth one. Then, if we have that $i > j + 1$, then we have that the cost must be the distance from the $i - 1$st point to the ith plus the solution to the subproblem obtained where we replace i with $i - 1$. There can be at most $O(n^2)$ of these subproblem, but solving them only requires considering a constant number of cases. The other possibility for a subproblem is that $j \leq i \leq j + 1$. In this case, we consider for every k from 1 to j the subproblem where we replace i with k plus the cost from kth point to the ith point and take the minimum over all of them. This case requires considering $O(n)$ things, but there are only $O(n)$ such cases. So, the final runtime is $O(n^2)$.

Problem 15-4

First observe that the problem exhibits optimal substructure in the following way: Suppose we know that an optimal solution has k words on the first line. Then we must solve the subproblem of printing neatly words $l_{k+1}, \ldots, l_n$. We build a table of optimal solutions solutions to solve the problem using dynamic programming. If $n - 1 + \sum_{k=1}^{n} l_k < M$ then put all words on a single line for an optimal solution. In the following algorithm Printing-Neatly(n), $C[k]$ contains the cost of printing neatly words l_k through l_n. We can determine the cost of an optimal solution upon termination by examining $C[1]$. The entry $P[k]$ contains the position of the last word which should appear on the first line of the optimal solution of words $l_1, l_2, \ldots, l_n$. Thus, to obtain the optimal way to place the words, we make $L_{P[1]}$ the last word on the first line, $l_{P[P[1]]}$ the last word on the second line, and so on.

Algorithm 8 Printing-Neatly(n)

1: Let $P[1..n]$ and $C[1..n]$ be a new tables.
2: **for** $k = n$ downto 1 **do**
3:      **if** $\sum_{i=k}^{n} l_i + n - k < M$ **then**
4:          $C[k] = 0$
5:      **end if**
6:      $q = \infty$
7:      **for** $j = 1$ downto $n - k$ **do**
8:          **if** $\sum_{m=1}^{j} l_{k+j} + j - 1 < M$ and $(M - \sum_{m=1}^{j} l_{k+j} + j - 1) + C[k+j+1] < q$ **then**
9:              $q = (M - \sum_{m=1}^{j} l_{k+j} + j - 1) + C[k + j + 1]$
10:              $P[k] = k + j$
11:          **end if**
12:      **end for**
13:      $C[k] = q$
14: **end for**

Problem 15-5

a. We will index our subproblems by two integers, $1 \leq i \leq m$ and $1 \leq j \leq n$. We will let i indicate the rightmost element of x we have not processed and j indicate the rightmost element of y we have not yet found matches for. For a solution, we call $EDIT(x, y, i, j)$

b. We will set $cost(delete) = cost(insert) = 2, cost(copy) = -1, cost(replace) = 1$, and $cost(twiddle) = cost(kill) = \infty$. Then a minimum cost translation of the first string into the second corresponds to an alignment. where we view a copy or a replace as incrementing a pointer for both strings. A insert as putting a space at the current position of the pointer in the first string. A

Algorithm 9 EDIT(x,y,i,j)

---

let $m = x.length$ and $n = y.length$
if $i = m$ **then**
    return (n-j)cost(insert)
end if
if $j = n$ **then**
    return $\min\{(m - i)cost(delete), cost(kill)\}$
end if
$o_1, \ldots, o_5$ initialized to ∞
if x[i] = y[j] **then**
    $o_1 = cost(copy) + EDIT(x, y, i + 1, j + 1)$
end if
$o_2 = cost(replace) + EDIT(x, y, i + 1, j + 1)$
$o_3 = cost(delete) + EDIT(x, y, i + 1, j)$
$o_4 = cost(insert) + EDIT(x, y, i, j + 1)$
if $i < m - 1$ and $j < n - 1$ **then**
    if $x[i] = y[j + 1]$ and $x[i + 1] = y[j]$ **then**
        $o_5 = cost(twiddle) + EDIT(x, y, i + 2, j + 2)$
    end if
end if
return $\min_{i\in[5]}\{o_i\}$

---

delete operation means putting a space in the current position in the second string. Since twiddles and kills have infinite costs, we will have neither of them in a minimal cost solution. The final value for the alignment will be the negative of the minimum cost sequence of edits.

Problem 15-6

The problem exhibits optimal substructure in the following way: If the root r is included in an optimal solution, then we must solve the optimal subproblems rooted at the grandchildren of r. If r is not included, then we must solve the optimal subproblems on trees rooted at the children of r. The dynamic programming algorithm to solve this problem works as follows: We make a table C indexed by vertices which tells us the optimal conviviality ranking of a guest list obtained from the subtree with root at that vertex. We also make a table G such that $G[i]$ tells us the guest list we would use when vertex i is at the root. Let T be the tree of guests. To solve the problem, we need to examine the guest list stored at $G[T.root]$. First solve the problem at each leaf L. If the conviviality ranking at L is positive, $G[L] = \{L\}$ and $C[L] = L.conviv$. Otherwise $G[L] = \emptyset$ and $C[L] = 0$. Iteratively solve the subproblems located at parents of nodes at which the subproblem has been solved. In general for a

node x,

$$C[x] = \min \left(\sum_{y \text{ is a child of } x} C[y], \sum_{y \text{ is a grandchild of } x} C[y] \right).$$

The runtime of the algorithm is $O(n^2)$ where n is the number of vertices, because we solve n subproblems, each in constant time, but the tree traversals required to find the appropriate next node to solve could take linear time.

Problem 15-7

a. Our substructure will consist of trying to find suffixes of s of length one less starting at all the edges leaving ν_0 with label σ_0. if any of them have a solution, then, there is a solution. If none do, then there is none. See the algorithm VITERBI for details.

Algorithm 10 $VITERBI(G, s, \nu_0)$

---
  if s.length $= 0$ **then**
      return ν_0
  end if
  for edges $(\nu_0, \nu_1) \in V$ for some ν_1 **do**
      if $\sigma(\nu_0, \nu_1) = \sigma_1$ **then**
         $res = VITERBI(G, (\sigma_2, \ldots, \sigma_k), \nu_1)$
         if res != NO-SUCH-PATH **then**
            return ν_0, res
         end if
      end if
  end for
  return NO-SUCH-PATH

Since the subproblems are indexed by a suffix of s (of which there are only k) and a vertex in the graph, there are at most $O(k|V|)$ different possible arguments. Since each run may require testing a edge going to every other vertex, and each iteration of the for loop takes at most a constant amount of time other than the call to PROB=VITERBI, the final runtime is $O(k|V|^2)$

b. For this modification, we will need to try all the possible edges leaving from ν_0 instead of stopping as soon as we find one that works. The substructure is very similar. We'll make it so that instead of just returning the sequence, we'll have the algorithm also return the probability of that maximum probability sequence, calling the fields seq and prob respectively. See the algorithm PROB-VITERBI

Since the runtime is indexed by the same things, we have that we will call it with at most $O(k|V|)$ different possible arguments. Since each run may

Algorithm 11 $PROB - VITERBI(G, s, \nu_0)$

---

if s.length $= 0$ **then**
    return ν_0
end if
let $sols.seq = NO - SUCH - PATH$, and $sols.prob = 0$
for edges $(\nu_0, \nu_1) \in V$ for some ν_1 **do**
    if $\sigma(\nu_0, \nu_1) = \sigma_1$ **then**
        $res = PROB - VITERBI(G, (\sigma_2, \ldots, \sigma_k), \nu_1)$
        if $p(\nu_0, \nu_1) \cdot res.prob >= sols.prob$ **then**
            $sols.prob = p(\nu_0, \nu_1) \cdot res.prob$ and $sols.seq = \nu_0, res.seq$
        end if
    end if
end for
return sols

---

require testing a edge going to every other vertex, and each iteration of the for loop takes at most a constant amount of time other than the call to PROB=VITERBI, the final runtime is $O(k|V|^2)$

Problem 15-8

a. If $n > 1$ then for every choice of pixel at a given row, we have at least 2 choices of pixel in the next row to add to the seam (3 if we're not in column 1 or n). Thus the total number of possibilities is bounded below by 2^m.

b. We create a table $D[1..m, 1..n]$ such that $D[i,j]$ stores the disruption of an optimal seam ending at position $[i, j]$, which started in row 1. We also create a table $S[i,j]$ which stores the list of ordered pairs indicating which pixels were used to create the optimal seam ending at position (i, j). To find the solution to the problem, we look for the minimum k entry in row m of table D, and use the list of pixels stored at $S[m, k]$ to determine the optimal seam. To simplify the algorithm Seam(A), let $MIN(a, b, c)$ be the function which returns -1 if a is the minimum, 0 if b is the minimum, and 1 if c is the minimum value from among a, b, and c. The time complexity of the algorithm is $O(mn)$.

Problem 15-9

The subproblems will be indexed by contiguous subarrays of the arrays of cuts needed to be made. We try making each possible cut, and take the one with cheapest cost. Since there are m to try, and there are at most m^2 possible things to index the subproblems with, we have that the m dependence is that the solution is $O(m^3)$. Also, since each of the additions is of a number that

Algorithm 12 Seam(A)

Initialize tables $D[1..m, 1..n]$ of zeros and $S[1..m, 1..n]$ of empty lists
for $i = 1$ to n **do**
    $S[1, i] = (1, i)$
    $D[1, i] = d_{1i}$
end for
for $i = 2$ to m **do**
    for $j = 1$ to n **do**
        if $j == 1$ **then** //Handles the left-edge case
            if $D[i - 1, j] < D[i - 1, j + 1]$ **then**
                $D[i, j] = D[i - 1, j] + d_{ij}$
                $S[i, j] = S[i - 1, j].insert(i, j)$
            else
                $D[i, j] = D[i - 1, j + 1] + d_{ij}$
                $S[i, j] = S[i - 1, j + 1].insert(i, j)$
            end if
        else if $j == n$ **then** //Handles the right-edge case
            if $D[i - 1, j - 1] < D[i - 1, j]$ **then**
                $D[i, j] = D[i - 1, j - 1] + d_{ij}$
                $S[i, j] = S[i - 1, j - 1].insert(i, j)$
            else
                $D[i, j] = D[i - 1, j] + d_{ij}$
                $S[i, j] = S[i - 1, j].insert(i, j)$
            end if
        end if
        $x = MIN(D[i - 1, j - 1], D[i - 1, j], D[i - 1, j + 1])$
        $D[i, j] = D[i - 1, j + x]$
        $S[i, j] = S[i - 1, j + x].insert(i, j)$
    end for
end for
$q = 1$
for $j = 1$ to n **do**
    if $D[m, j] < D[m, q]$ **then** $q = j$
    end if
end for
Print the list stored at $S[m, q]$.

is $O(n)$, each of the iterations of the for loop may take time $O(\lg(n) + \lg(m))$, so, the final runtime is $O(m^3 \lg(n))$. The given algorithm will return (cost,seq) where cost is the cost of the cheapest sequence, and seq is the sequence of cuts to make

---
Algorithm 13 CUT-STRING(L,i,j,l,r)
---
if $l = r$ **then**
    return $(0,[])$
end if
$mincost = \infty$
for k from i to j **do**
    if $l + r + CUT - STRING(L, i, k, l, L[k]).cost + CUT - STRING(L, k, j, L[k], j).cost < mincost$ **then**
        $mincost = l + r + CUT - STRING(L, i, k, l, L[k]).cost + CUT - STRING(L, k, j, L[k], j).cost$
        $minseq = L[k]$ concatenated with the sequence returned from $CUT - STRING(L, i, k, l, L[k])$ and from $CUT - STRING(L, i, k, l, L[k])$
    end if
end for
return (mincost,minseq)

Problem 15-10

a. Without loss of generality, suppose that there exists an optimal solution S which involves investing d_1 dollars into investment k and d_2 dollars into investement m in year 1. Further, suppose in this optimal solution, you don't move your money for the first j years. If $r_{k1} + r_{k2} + \ldots + r_{kj} > r_{m1} + r_{m2} + \ldots + r_{mj}$ then we can perform the usual cut-and-paste maneuver and instead invest $d_1 + d_2$ dollars into investment k for j years. Keeping all other investments the same, this results in a strategy which is at least as profitable as S, but has reduced the number of different investments in a given span of years by 1. Continuing in this way, we can reduce the optimal strategy to consist of only a single investment each year.

b. If a particular investment strategy is the year-one-plan for a optimal investment strategy, then we must solve two kinds of optimal suproblem: either we maintain the strategy for an additional year, not incurring the money-moving fee, or we move the money, which amounts to solving the problem where we ignore all information from year 1. Thus, the problem exhibits optimal substructure.

c. The algorithm works as follows: We build tables I and R of size 10 such that $I[i]$ tells which investment should be made (with all money) in year i, and

$R[i]$ gives the total return on the investment strategy in years i through 10.

Algorithm 14 Invest(d,n)

Initialize tables I and R of size 11, all filled with zeros
for $k = 10$ downto 1 **do**
    $q = 1$
    for $i = 1$ to n **do**
        if $r_{ik} > r_{qk}$ **then** // i now holds the investment which looks best for
a given year
            $q = i$
        end if
    end for
    if $R[k+1] + dr_{I[k+1]k} - f_1 > R[k+1] + dr_{qk} - f_2$ **then** //If revenue is
greater when money is not moved
        $R[k] = R[k+1] + dr_{I[k+1]k} - f_1$
        $I[k] = I[k+1]$
    else
        $R[k] = R[k+1] + dr_{qk} - f_2$
        $I[k] = q$
    end if
end for
Return I as an optimal strategy with return $R[1]$.

---

d. The previous investment strategy was independent of the amount of money
   you started with. When there is a cap on the amount you can invest, the
   amount you have to invest in the next year becomes relevant. If we know
   the year-one-strategy of an optimal investment, and we know that we need
   to move money after the first year, we're left with the problem of investing a
   different initial amount of money, so we'd have to solve a subproblem for every
   possible initial amount of money. Since there is no bound on the returns,
   there's also no bound on the number of subproblems we need to solve.

Problem 15-11

    Our subproblems will be indexed by and integer $i \in [n]$ and another integer
$j \in [D]$. i will indicate how many months have passed, that is, we will restrict
ourselves to only caring about $(d_i, \ldots, d_n)$. j will indicate how many machines
we have in stock initially. Then, the recurrence we will use will try producing
all possible numbers of machines from 1 to $[D]$. Since the index space has size
$O(nD)$ and we are only running through and taking the minimum cost from D
many options when computing a particular subproblem, the total runtime will
be $O(nD^2)$.

Problem 15-12

We will make an $N+1$ by $X+1$ by $P+1$ table. The runtime of the algorithm is $O(NXP)$.

Algorithm 15 Baseball(N,X,P)
___
Initialize an $N + 1$ by $X + 1$ table B
Initialize an array P of length N
for $i = 0$ to N **do**
    $B[i, 0] = 0$
end for
for $j = 1$ to X **do**
    $B[0, j] = 0$
end for
for $i = 1$ to N **do**
    for $j = 1$ to X **do**
        if $j < i.cost$ **then**
            $B[i, j] = B[i - 1, j]$
        end if
        $q = B[i - 1, j]$
        $p = 0$
        for $k = 1$ to p **do**
            if $B[i - 1, j - i.cost] + i.value > q$ **then**
                $q = B[i - 1, j - i.cost] + i.value$
                $p = k$
            end if
        end for
        $B[i, j] = q$
        $P[i] = p$
    end for
end for
Print: The total VORP is $B[N, X]$ and the players are:
$i = N$
$j = X$
$C = 0$
for $k = 1$ to N **do** //Prints the players from the table
    if $B[i, j] \neq B[i - 1, j]$ **then**
        Print $P[i]$
        $j = j - i.cost$
        $C = C + i.cost$
    end if
    $i = i - 1$
end for
Print: The total cost is C
___

Chapter 16

Michelle Bodnar, Andrew Lohr

May 5, 2017

Exercise 16.1-1

The given algorithm would just stupidly compute the minimum of the $O(n)$ numbers or return zero depending on the size of S_{ij}. There are a possible number of subproblems that is $O(n^2)$ since we are selecting i and j so that $1 \leq i \leq j \leq n$. So, the runtime would be $O(n^3)$.

Exercise 16.1-2

This becomes exactly the same as the original problem if we imagine time running in reverse, so it produces an optimal solution for essentially the same reasons. It is greedy because we make the best looking choice at each step.

Exercise 16.1-3

As a counterexample to the optimality of greedily selecting the shortest, suppose our activity times are $\{(1,9),(8,11),(10,20)\}$ then, picking the shortest first, we have to eliminate the other two, where if we picked the other two instead, we would have two tasks not one.

As a counterexample to the optimality of greedily selecting the task that conflicts with the fewest remaining activities, suppose the activity times are $\{(-1,1),(2,5),(0,3),(0,3),(0,3),(4,7),(6,9),(8,11),(8,11),(8,11),(10,12)\}$. Then, by this greedy strategy, we would first pick $(4,7)$ since it only has a two conflicts. However, doing so would mean that we would not be able to pick the only optimal solution of $(-1,1),(2,5),(6,9),(10,12)$.

As a counterexample to the optimality of greedily selecting the earliest start times, suppose our activity times are $\{(1,10),(2,3),(4,5)\}$. If we pick the earliest start time, we will only have a single activity, $(1,10)$, whereas the optimal solution would be to pick the two other activities.

Exercise 16.1-4

Maintain a set of free (but already used) lecture halls F and currently busy lecture halls B. Sort the classes by start time. For each new start time which you encounter, remove a lecture hall from F, schedule the class in that room,

and add the lecture hall to B. If F is empty, add a new, unused lecture hall to F. When a class finishes, remove its lecture hall from B and add it to F. Why this is optimal: Suppose we have just started using the mth lecture hall for the first time. This only happens when ever classroom ever used before is in B. But this means that there are m classes occurring simultaneously, so it is necessary to have m distinct lecture halls in use.

Exercise 16.1-5

Run a dynamic programming solution based off of the equation (16.2) where the second case has "1" replaced with "v_k". Since the subproblems are still indexed by a pair of activities, and each calculation requires taking the minimum over some set of size $\le |S_{ij}| \in O(n)$. The total runtime is bounded by $O(n^3)$.

Exercise 16.2-1

A optimal solution to the fractional knapsack is one that has the highest total value density. Since we are always adding as much of the highest value density we can, we are going to end up with the highest total value density. Suppose that we had some other solution that used some amount of the lower value density object, we could substitute in some of the higher value density object meaning our original solution could not have been optimal.

Exercise 16.2-2

Suppose we know that a particular item of weight w is in the solution. Then we must solve the subproblem on $n - 1$ items with maximum weight $W - w$. Thus, to take a bottom-up approach we must solve the 0-1 knapsack problem for all items and possible weights smaller than W. We'll build an $n + 1$ by $W + 1$ table of values where the rows are indexed by item and the columns are indexed by total weight. (The first row and column of the table will be a dummy row). For row i column j, we decide whether or not it would be advantageous to include item i in the knapsack by comparing the total value of of a knapsack including items 1 through $i - 1$ with max weight j, and the total value of including items 1 through $i - 1$ with max weight $j - i.weight$ and also item i. To solve the problem, we simply examine the n, W entry of the table to determine the maximum value we can achieve. To read off the items we include, start with entry n, W. In general, proceed as follows: if entry i, j equals entry $i-1, j$, don't include item i, and examine entry $i-1, j$ next. If entry i, j doesn't equal entry $i - 1, j$, include item i and examine entry $i - 1, j - i.weight$ next. See algorithm below for construction of table:

Exercise 16.2-3

At each step just pick the lightest (and most valuable) item that you can pick. To see this solution is optimal, suppose that there were some item j that we included but some smaller, more valuable item i that we didn't. Then, we could replace the item j in our knapsack with the item i. it will definitely fit

Algorithm 1 0-1 Knapsack(n,W)

1: Initialize an $n + 1$ by $W + 1$ table K
2: **for** $j = 1$ to W **do**
3:     $K[0, j] = 0$
4: **end for**
5: **for** $i = 1$ to n **do**
6:     $K[i, 0] = 0$
7: **end for**
8: **for** $i = 1$ to n **do**
9:     **for** $j = 1$ to W **do**
10:         **if** $j < i.weight$ **then**
11:            $K[i, j] = K[i - 1, j]$
12:         **end if**
13:         $K[i, j] = \max(K[i - 1, j], K[i - 1, j - i.weight] + i.value)$
14:     **end for**
15: **end for**

---

because i is lighter, and it will also increase the total value because i is more valuable.

Exercise 16.2-4

The greedy solution solves this problem optimally, where we maximize distance we can cover from a particular point such that there still exists a place to get water before we run out. The first stop is at the furthest point from the starting position which is less than or equal to m miles away. The problem exhibits optimal substructure, since once we have chosen a first stopping point p, we solve the subproblem assuming we are starting at p. Combining these two plans yields an optimal solution for the usual cut-and-paste reasons. Now we must show that this greedy approach in fact yields a first stopping point which is contained in some optimal solution. Let O be any optimal solution which has the professor stop at positions $o_1, o_2, \ldots, o_k$. Let g_1 denote the furthest stopping point we can reach from the starting point. Then we may replace o_1 by g_2 to create a modified solution G, since $o_2 - o_1 < o_2 - g_1$. In other words, we can actually make it to the positions in G without running out of water. Since G has the same number of stops, we conclude that g_1 is contained in some optimal solution. Therefore the greedy strategy works.

Exercise 16.2-5

Consider the leftmost interval. It will do no good if it extends any further left than the leftmost point, however, we know that it must contain the leftmost point. So, we know that it's left hand side is exactly the leftmost point. So, we just remove any point that is within a unit distance of the left most point since

they are contained in this single interval. Then, we just repeat until all points are covered. Since at each step there is a clearly optimal choice for where to put the leftmost interval, this final solution is optimal.

Exercise 16.2-6

First compute the value of each item, defined to be it's worth divided by its weight. We use a recursive approach as follows: Find the item of median value, which can be done in linear time as shown in chapter 9. Then sum the weights of all items whose value exceeds the median and call it M. If M exceeds W then we know that the solution to the fractional knapsack problem lies in taking items from among this collection. In other words, we're now solving the fractional knapsack problem on input of size $n/2$. On the other hand, if the weight doesn't exceed W, then we must solve the fractional knapsack problem on the input of $n/2$ low-value items, with maximum weight $W - M$. Let $T(n)$ denote the runtime of the algorithm. Since we can solve the problem when there is only one item in constant time, the recursion for the runtime is $T(n) = T(n/2) + cn$ and $T(1) = d$, which gives runtime of $O(n)$.

Exercise 16.2-7

Since an identical permutation of both sets doesn't affect this product, suppose that A is sorted in ascending order. Then, we will prove that the product is maximized when B is also sorted in ascending order. To see this, suppose not, that is, there is some $i < j$ so that $a_i < a_j$ and $b_i > b_j$. Then, consider only the contribution to the product from the indices i and j. That is, $a_i^{b_i} a_j^{b_j}$, then, if we were to swap the order of b_1 and b_j, we would have that contribution be $a_i^{b_j} a_j^{b_i}$. we can see that this is larger than the previous expression because it differs by a factor of $(\frac{a_j}{a_i})^{b_i - b_j}$ which is bigger than one. So, we couldn't of maximized the product with this ordering on B.

Exercise 16.3-1

If we have that $x.freq = b.freq$, then we know that b is tied for lowest frequency. In particular, it means that there are at least two things with lowest frequency, so $y.freq = x.freq$. Also, since $x.freq \leq a.freq \leq b.freq = x.freq$, we must have $a.freq = x.freq$.

Exercise 16.3-2

Let T be a binary tree corresponding to an optimal prefix code and suppose that T is not full. Let node n have a single child x. Let T' be the tree obtained by removing n and replacing it by x. Let m be a leaf node which is a descendant of x. Then we have

$$cost(T') \leq \sum_{c \in C \setminus \{m\}} c.freq \cdot d_T(c) + m.freq(d_T(m)-1) < \sum_{c \in C} c.freq \cdot d_T(c) = cost(T)$$

which contradicts the fact that T was optimal. Therefore every binary tree corresponding to an optimal prefix code is full.

Exercise 16.3-3

An optimal Huffman code would be

$$0000000 \rightarrow a$$
$$0000001 \rightarrow b$$
$$000001 \rightarrow c$$
$$00001 \rightarrow d$$
$$0001 \rightarrow e$$
$$001 \rightarrow f$$
$$01 \rightarrow g$$
$$1 \rightarrow h$$

This generalizes to having the first n Fibonacci numbers as the frequencies in that the $k < n$th most frequent letter has codeword $0^{k-1}1$, and the nth most frequent letter having codeword 0^{n-1}. To see this holds, we will prove the recurrence
$$\sum_{i=0}^{n-1} F(i) = F(n+1) - 1$$

This will show that we should join together the letter with frequency $F(n)$ with the result of joining together the letters with smaller frequencies. We will prove it by induction. For $n = 1$ is is trivial to check. Now, suppose that we have $n - 1 \geq 1$, then,

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

See also Lemma 19.2.

To use this fact, to show the desired Huffman code is optimal, we claim that as we are greedily combining nodes, that at each stage we can maintain one node which contains all of the least frequent letters. Initially, this consists of just the least frequent letter. Then, at stage k, inductively, we assume that it contains the k least frequent letters. This means that it has weight $\sum_{i=0}^{k-1} F(i) = F(k-1) - 1$. All of the other nodes at this stage have weights $\{F(k), F(k+1), ...F(n-1)\}$. So, clearly the two lowest weight nodes are this

node containing the k least frequent letters and the node containing the k least frequent letter. Therefore, the greedy algorithm tells us that we should combine those nodes leaving us with a node containing the $k+1$ least frequent letters for stage $k + 1$, completing the induction. Of course, the code that we have given is not uniquely optimal, because left and right children (represented by 0 and 1 respectively) are an artificial choice, we could assume that at each stage we are adding in the singleton as the 1 child of the new parent, getting us our code.

Exercise 16.3-4

Let x be a leaf node. Then $x.freq$ is added to the cost of each internal node which is an ancestor of x exactly once, so its total contribution to the new way of computing cost is $x.freq \cdot d_T(x)$, which is the same as its old contribution. Therefore the two ways of computing cost are equivalent.

Exercise 16.3-5
We construct this codeword with monotonically increasing lengths by always resolving ties in terms of which two nodes to join together by joining together those with the two latest occurring earliest elements. We will show that the ending codeword has that the least frequent words are all having longer code-words. Suppose to a contradiction that there were two words, w_1 and w_2 so that w_1 appears more frequently, but has a longer codeword. This means that it was involved in more merge operation than w_2 was. However, since we are always merging together the two sets of words with the lowest combined frequency, this would contradict the fact that w_1 has a higher frequency than w_2.

Exercise 16.3-6

First observe that any full binary tree has exactly $2n - 1$ nodes. We can encode the structure of our full binary tree by performing a preorder traversal of T. For each node that we record in the traversal, write a 0 if it is an internal node and a 1 if it is a leaf node. Since we know the tree to be full, this uniquely determines its structure. Next, note that we can encode any character of C in $\lceil \lg n \rceil$ bits. Since there are n characters, we can encode them in order of appearance in our preorder traversal using $n\lceil \lg n \rceil$ bits.

Exercise 16.3-7

Instead of grouping together the two with lowest frequency into pairs that have the smallest total frequency, we will group together the three with lowest frequency in order to have a final result that is a ternary tree. The analysis of optimality is almost identical to the binary case. We are placing the symbols of lowest frequency lower down in the final tree and so they will have longer codewords than the more frequently occurring symbols

Exercise 16.3-8

6

For any 2 characters, the sum of their frequencies exceeds the frequency of any other character, so initially Huffman coding makes 128 small trees with 2 leaves each. At the next stage, no internal node has a label which is more than twice that of any other, so we are in the same setup as before. Continuing in this fashion, Huffman coding builds a complete binary tree of height $\lg(256) = 8$, which is no more efficient than ordinary 8-bit length codes.

Exercise 16.3-9

If every possible character is equally likely, then, when constructing the Huffman code, we will end up with a complete binary tree of depth 7. This means that every character, regardless of what it is will be represented using 8 bits. This is exactly as many bits as was originally used to represent those characters, so the total length of the file will not decrease at all.

Exercise 16.4-1

The first condition that S is a finite set is a given. To prove the second condition we assume that $k \geq 0$, this gets us that $\mathcal{I}_k$ is nonempty. Also, to prove the hereditary property, suppose $A \in \mathcal{I}_k$ this means that $|A| \leq k$. Then, if $B \subseteq A$, this means that $|B| \leq |A| \leq k$, so $B \in \mathcal{I}_k$. Lastly, we prove the exchange property by letting $A, B \in \mathcal{I}_k$ be such that $|A| < |B|$. Then, we can pick any element $x \in B \setminus A$, then, $|A \cup \{x\}| = |A| + 1 \leq |B| \leq k$, so, we can extend A to $A \cup \{x\} \in \mathcal{I}_k$.

Exercise 16.4-2

Let $c_1, \ldots, c_m$ be the columns of T. Suppose $C = \{c_{i_1}, \ldots, c_{i_k}\}$ is dependent. Then there exist scalars $d_1, \ldots, d_k$ not all zero such that $\sum_{j=1}^{k} d_j c_{i_j} = 0$. By adding columns to C and assigning them to have coefficient 0 in the sum, we see that any superset of C is also dependent. By contrapositive, any subset of an independent set must be independent. Now suppose that A and B are two independent sets of columns with $|A| > |B|$. If we couldn't add any column of A to be whilst preserving independence then it must be the case that every element of A is a linear combination of elements of B. But this implies that B spans a $|A|$-dimensional space, which is impossible. Therefore our independence system must satisfy the exchange property, so it is in fact a matroid.

Exercise 16.4-3

Condition one of being a matroid is still satisfied because the base set hasn't changed. Next we show that $\mathcal{I}'$ is nonempty. Let A be any maximal element of $\mathcal{I}$ then, we have that $S - A \in \mathcal{I}'$ because $S - (S - A) = A \subseteq A$ which is maximal in $\mathcal{I}$. Next we show the hereditary property, suppose that $B \subseteq A \in \mathcal{I}'$, then,

there exists some $A' \in \mathcal{I}$ so that $S - A \subseteq A'$, however, $S - B \supseteq S - A \subseteq A'$ so $B \in \mathcal{I}'$.

Lastly we prove the exchange property. That is, if we have $B, A \in \mathcal{I}'$ and $|B| < |A|$ we can find an element x in $A - B$ to add to B so that it stays independent. We will split into two cases.

Our first case is that $|A| = |B| + 1$. We clearly need to select x to be the single element in $A - B$. Since $S - B$ contains a maximal independent set

Our second case is if the first case does not hold. Let C be a maximal independent set of $\mathcal{I}$ contained in $S - A$. Pick an aribitrary set of size $|C| - 1$ from some maximal independent set contained in $S - B$, call it D. Since D is a subset of a maximal independent set, it is also independent, and so, by the exchange property, there is some $y \in C - D$ so that $D \cup \{y\}$ is a maximal independent set in $\mathcal{I}$. Then, we select x to be any element other than y in $A - B$. Then, $S - (B \cup \{x\})$ will still contain $D \cup \{y\}$. This means that $B \cup \{x\}$ is independent in $(I)'$

Exercise 16.4-4

Suppose $X \subset Y$ and $Y \in \mathcal{I}$. Then $(X \cap S_i) \subset (Y \cap S_i)$ for all i, so $|X \cap S_i| \leq |Y \cap S_i| \leq 1$ for all $1 \leq i \leq k$. Therefore $\mathcal{M}$ is closed under inclusion.

Now Let $A, B \in \mathcal{I}$ with $|A| = |B| + 1$. Then there must exist some j such that $|A \cap S_j| = 1$ but $B \cap S_j = 0$. Let $a = A \cap S_j$. Then $a \notin B$ and $|(B \cup \{a\}) \cap S_j| = 1$. Since $|(B \cup \{a\}) \cap S_i| = |B \cap S_i|$ for all $i \neq j$, we must have $B \cup \{a\} \in \mathcal{I}$. Therefore $\mathcal{M}$ is a matroid.

Exercise 16.4-5

Suppose that W is the largest weight that any one element takes. Then, define the new weight function $w_2(x) = 1 + W - w(x)$. This then assigns a strictly positive weight, and we will show that any independent set that that has maximum weight with respect to w_2 will have minimum weight with respect to w. Recall Theorem 16.6 since we will be using it, suppose that for our matriod, all maximal independent sets have size S. Then, suppose M_1 and M_2 are maximal independent sets so that M_1 is maximal with respect to w_2 and M_2 is minimal with respect to w. Then, we need to show that $w(M_1) = w(M_2)$. Suppose not to achieve a contradiction, then, by minimality of M_2, $w(M_1) > w(M_2)$. Rewriting both sides in terms of w_2, we have $w_2(M_2) - (1 + W)S > w_2(M_1) - (1 + W)S$, so, $w_2(M_2) > w_2(M_1)$. This however contradicts maximality of M_1 with respect to w_2. So, we must have that $w(M_1) = w(M_2)$. So, a maximal independent set that has the largest weight with respect to w_2 also has the smallest weight with respect to w.

Exercise 16.5-1

With the requested substitution, the instance of the problem becomes

a_i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	10	20	30	40	50	60	70

We begin by just greedily constructing the matroid, adding the most costly to leave incomplete tasks first. So, we add tasks 7,6,5,4,3. Then, in order to schedule tasks 1 or 2 we need to leave incomplete more important tasks. So, our final schedule is $\langle 5, 3, 4, 6, 7, 1, 2 \rangle$ to have a total penalty of only $w_1 + w_2 = 30$.

Exercise 16.5-2

Create an array B of length n containing zeros in each entry. For each element $a \in A$, add 1 to $B[a.deadline]$. If $B[a.deadline] > a.deadline$, return that the set is not independent. Otherwise, continue. If successfully examine every element of A, return that the set is independent.

Problem 16-1

a. Always give the highest denomination coin that you can without going over. Then, repeat this process until the amount of remaining change drops to 0.

b. Given an optimal solution $(x_0, x_1, \ldots, x_k)$ where x_i indicates the number of coins of denomination c^i. We will first show that we must have $x_i < c$ for every $i < k$. Suppose that we had some $x_i \geq c$, then, we could decrease x_i by c and increase x_{i+1} by 1. This collection of coins has the same value and has $c - 1$ fewer coins, so the original solution must of been non-optimal. This configuration of coins is exactly the same as you would get if you kept greedily picking the largest coin possible. This is because to get a total value of V, you would pick $x_k = \lfloor Vc^{-k} \rfloor$ and for $i < k$, $x_i \lfloor (V mod c^{i+1}) c^{-i} \rfloor$. This is the only solution that satisfies the property that there aren't more than c of any but the largest denomination because the coin amounts are a base c representation of $V mod c^k$.

c. Let the coin denominations be $\{1, 3, 4\}$, and the value to make change for be 6. The greedy solution would result in the collection of coins $\{1, 1, 4\}$ but the optimal solution would be $\{3, 3\}$.

d. See algorithm MAKE-CHANGE(S,v) which does a dynamic programming solution. Since the first forloop runs n times, and the inner for loop runs k times, and the later while loop runs at most n times, the total running time is $O(nk)$.

Problem 16-2

a. Order the tasks by processing time from smallest to largest and run them in that order. To see that this greedy solution is optimal, first observe that the problem exhibits optimal substructure: if we run the first task in an optimal

---

Algorithm 2 MAKE-CHANGE(S,v)

Let *numcoins* and *coin* be empty arrays of length v, and any any attempt to
access them at indices in the range $-\max(S), -1$ should return ∞
for i from 1 to v **do**
    bestcoin =nil
    $bestnum = \infty$
    for c in S **do**
        if $numcoins[i - c] + 1 < bestnum$ **then**
            bestnum = numcoins[i-c]
            bestcoin = c
        end if
    end for
    numcoins[i] = bestnum
    coin[i] = bestcoin
end for
let change be an empty set
iter = v
while $iter > 0$ **do**
    add coin[iter] to change
    $iter = iter - coin[iter]$
end while
return change

---

solution, then we obtain an optimal solution by running the remaining tasks
in a way which minimizes the average completion time. Let O be an optimal
solution. Let a be the task which has the smallest processing time and let b
be the first task run in O. Let G be the solution obtained by switching the
order in which we run a and b in O. This amounts reducing the completion
times of a and the completion times of all tasks in G between a and b by the
difference in processing times of a and b. Since all other completion times
remain the same, the average completion time of G is less than or equal to
the average completion time of O, proving that the greedy solution gives an
optimal solution. This has runtime $O(n \lg n)$ because we must first sort the
elements.

b. Without loss of generality we my assume that every task is a unit time task.
Apply the same strategy as in part (a), except this time if a task which we
would like to add next to the schedule isn't allowed to run yet, we must skip
over it. Since there could be many tasks of short processing time which have
late release time, the runtime becomes $O(n^2)$ since we might have to spend
$O(n)$ time deciding which task to add next at each step.

Problem 16-3

10

a. First, suppose that a set of columns is not linearly independent over $\mathbb{F}_2$ then, there is some subset of those columns, say S so that a linear combination of S is 0. However, over $\mathbb{F}_2$, since the only two elements are 1 and 0, a linear combination is a sum over some subset. Suppose that this subset is S', note that it has to be nonempty because of linear dependence. Now, consider the set of edges that these columns correspond to. Since the columns had their total incidence with each vertex 0 in $\mathbb{F}_2$, it is even. So, if we consider the subgraph on these edges, then every vertex has a even degree. Also, since our S' was nonempty, some component has an edge. Restrict our attention to any such component. Since this component is connected and has all even vertex degrees, it contains an Euler Circuit, which is a cycle.

Now, suppose that our graph had some subset of edges which was a cycle. Then, the degree of any vertex with respect to this set of edges is even, so, when we add the corresponding columns, we will get a zero column in $\mathbb{F}_2$.

Since sets of linear independent columns form a matroid, by problem 16.4-2, the acyclic sets of edges form a matroid as well.

b. One simple approach is to take the highest weight edge that doesn't complete a cycle. Another way to phrase this is by running Kruskal's algorithm (see Chapter 23) on the graph with negated edge weights.

c. Consider the digraph on [3] with the edges $(1, 2), (2, 1), (2, 3), (3, 2), (3, 1)$ where (u, v) indicates there is an edge from u to v. Then, consider the two acyclic subsets of edges $B = (3, 1), (3, 2), (2, 1)$ and $A = (1, 2), (2, 3)$. Then, adding any edge in $B - A$ to A will create a cycle. So, the exchange property is violated.

d. Suppose that the graph contained a directed cycle consisting of edges corresponding to columns S. Then, since each vertex that is involved in this cycle has exactly as many edges going out of it as going into it, the rows corresponding to each vertex will add up to zero, since the outgoing edges count negative and the incoming vertices count positive. This means that the sum of the columns in S is zero, so, the columns were not linearly independent.

e. There is not a perfect correspondence because we didn't show that not containing a directed cycle means that the columns are linearly independent, so there is not perfect correspondence between these sets of independent columns (which we know to be a matroid) and the acyclic sets of edges (which we know not to be a matroid).

Problem 16-4

a. Let O be an optimal solution. If a_j is scheduled before its deadline, we can always swap it with whichever activity is scheduled at its deadline without

changing the penalty. If it is scheduled after its deadline but $a_j.deadline \leq j$ then there must exist a task from among the first j with penalty less than that of a_j. We can then swap a_j with this task to reduce the overall penalty incurred. Since O is optimal, this can't happen. Finally, if a_j is scheduled after its deadline and $a_j.deadline > j$ we can swap a_j with any other late task without increasing the penalty incurred. Since the problem exhibits the greedy choice property as well, this greedy strategy always yields on optimal solution.

b. Assume that MAKE-SET(x) returns a pointer to the element x which is now it its own set. Our disjoint sets will be collections of elements which have been scheduled at contiguous times. We'll use this structure to quickly find the next available time to schedule a task. Store attributes $x.low$ and $x.high$ at the representative x of each disjoint set. This will give the earliest and latest time of a scheduled task in the block. Assume that UNION(x, y) maintains this attribute. This can be done in constant time, so it won't affect the asymptotics. Note that the attribute is well-defined under the union operation because we only union two blocks if they are contiguous. Without loss of generality we may assume that task a_1 has the greatest penalty, task a_2 has the second greatest penalty, and so on, and they are given to us in the form of an array A where $A[i] = a_i$. We will maintain an array D such that $D[i]$ contains a pointer to the task with deadline i. We may assume that the size of D is at most n, since a task with deadline later than n can't possibly be scheduled on time. There are at most $3n$ total MAKE-SET, UNION, and FIND-SET operations, each of which occur at most n times, so by Theorem 21.14 the runtime is $O(n\alpha(n))$.

Problem 16-5

a. Suppose there are m distinct elements that could be requested. There may be some room for improvement in terms of keeping track of the furthest in future element at each position. If you maintain a (double circular) linked list with a node for each possible cache element and an array so that in index i there is a pointer corresponding to the node in the linked list corresponding to the possible cache request i. Then, starting with the elements in an arbitrary order, process the sequence $\langle r_1, \ldots, r_n \rangle$ from right to left. Upon processing a request move the node corresponding to that request to the beginning of the linked list and make a note in some other array of length n of the element at the end of the linked list. This element is tied for furthest-in-future. Then, just scan left to right through the sequence, each time just checking some set for which elements are currently in the cache. It can be done in constant time to check if an element is in the cache or not by a direct address table. If an element need be evicted, evict the furthest-in-future one noted earlier. This algorithm will take time $O(n + m)$ and use additional space $O(m + n)$.

Algorithm 3 SCHEDULING-VARIATIONS(A)

1: Initialize an array D of size n.
2: **for** $i = 1$ to n **do**
3:     $a_i.time = a_i.deadline$
4:     **if** $D[a_i.deadline] \neq NIL$ **then**
5:         $y = \text{FIND-SET}(D[a_i.deadline])$
6:         $a_i.time = y.low - 1$
7:     **end if**
8:     $x = \text{MAKE-SET}(a_i)$
9:     $D[a_i.time] = x$
10:     $x.low = x.high = a_i.time$
11:     **if** $D[a_i.time - 1] \neq NIL$ **then**
12:         $\text{UNION}(D[a_i.time - 1], D[a_i.time])$
13:     **end if**
14:     **if** $D[a_i.time + 1] \neq NIL$ **then**
15:         $\text{UNION}(D[a_i.time], D[a_i.time + 1])$
16:     **end if**
17: **end for**

If we were in the stupid case that $m > n$, we could restrict our attention to the possible cache requests that actually happen, so we have a solution that is $O(n)$ both in time and in additional space required.

b. Index the subproblems $c[i, S]$ by a number $i \in [n]$ and a subset $S \in \binom{[m]}{k}$. Which indicates the lowest number of misses that can be achieved with an initial cache of S starting after index i. Then,

$$c[i, S] = \min_{x \in \{S\}} \left(c[i + 1, \{r_i\} \cup (S - \{x\})] + (1 - \chi_{\{r_i\}}(x)) \right)$$

which means that x is the element that is removed from the cache unless it is the current element being accessed, in which case there is no cost of eviction.

c. At each time we need to add something new, we can pick which entry to evict from the cache. We need to show the there is an exchange property. That is, if we are at round i and need to evict someone, suppose we evict x. Then, if we were to instead evict the furthest in future element y, we would have no more evictions than before. To see this, since we evicted x, we will have to evict someone else once we get to x, whereas, if we had used the other strategy, we wouldn't of had to evict anyone until we got to y. This is a point later in time than when we had to evict someone to put x back into the cache, so we could, at reloading y, just evict the person we would of evicted when we evicted someone to reload x. This causes the same number of misses unless there was an access to that element that wold of been evicted at reloading x some point in between when x any y were needed, in which case furthest in future would be better.

13

Chapter 17

Michelle Bodnar, Andrew Lohr

October 25, 2017

Exercise 17.1-1

It woudn't because we could make an arbitrary sequence of $MULTIPUSH(k), MULTIPOP(k)$. The cost of each will be $\Theta(k)$, so the average runtime of each will be $\Theta(k)$ not $O(1)$.

Exercise 17.1-2

Suppose the input is a 1 followed by $k - 1$ zeros. If we call DECREMENT we must change k entries. If we then call INCREMENT on this it reverses these k changes. Thus, by calling them alternately n times, the total time is $\Theta(nk)$.

Exercise 17.1-3

Note that this setup is similar to the dynamic tables discussed in section 17.4. Let n be arbitrary, and have the cost of operation i be $c(i)$. Then,

$$\sum_{i=1}^{n} c(i) = \sum_{i=1}^{\lceil \lg(n) \rceil} 2^i + \sum_{i \leq n \text{ not a power of } 2} 1 \leq \sum_{i=1}^{\lceil \lg(n) \rceil} 2^i + n = 2^{1+\lceil \lg(n) \rceil} - 1 + n \leq 4n - 1 + n \leq 5n \in O(n)$$

So, since to find the average, we divide by n, the average runtime of each command is $O(1)$.

Exercise 17.2-1

To every stack operation, we charge twice. First we charge the actual cost of the stack operation. Second we charge the cost of copying an element later on. Since we have the size of the stack never exceed k, and there are always k operations between backups, we always overpay by at least enough. So, the amortized cost of the operation is constant. So, the cost of the n operation is $O(n)$.

Exercise 17.2-2

Assign the cost 3 to each operation. The first operation costs 1, so we have a credit of 21. Now suppose that we have nonnegative credit after having performed the 2^ith operation. Each of the $2^i - 1$ operations following has cost 1. Since we pay 3 for each, we build up a credit of 2 from each of them, giving us $2(2^i - 1) = 2^{i+1} - 2$ credit. Then for the 2^{i+1}th operation, the 3 credits we pay gives us a total of $2^{i+1} + 1$ to use towards the actual cost of 2^{i+1}, leaving us with 1 credit. Thus, for any operation we have nonnegative credit. Since the amortized cost of each operation is $O(1)$, an upper bound on the total actual cost of n operations is $O(n)$.

Exercise 17.2-3

For each time we set a bit to 1, we both pay a dollar for eventually setting it back to zero (in the usual manner as the counter is incremented). But we also pay a third dollar in the event that even after the position has been set back to zero, we check about zeroing it out during a reset operation. We also increment the position of the highest order bit (as needed). Then, while doing the reset operation, we will only need consider those positions less significant than the highest order bit. Because of this, we have at least paid one extra dollar before, because we had set the bit at that position to one at least once for the highest order bit to be where it is. Since we have only put down a constant amortized cost at each setting of a bit to 1, the amortized cost is constant because each increment operation involves setting only a single bit to 1. Also, the amortized cost of a reset is zero because it involves setting no bits to one. It's true cost has already been paid for.

Exercise 17.3-1

Define $\Phi'(D) = \Phi(D) - \Phi(D_0)$. Then, we have that $\Phi(D) \geq \Phi(D_0)$ implies $\Phi'(D) = \Phi(D) - \Phi(D_0) \geq \Phi(D_0) - \Phi(D_0) = 0$. and $\Phi'(D_0) = \phi(D_0) - \Phi(D_0) = 0$. Lastly, the amortized cost using Φ' is $c_i + \Phi'(D_i) - \Phi'(D_{i-1}) = c_i + (\Phi(D_i) - \Phi(D_0)) - (\Phi(D_i) - \Phi(D_{i-1})) = c_i + \Phi(D_i) - \Phi(D_{i-1})$ which is the amortized cost using Φ.

Exercise 17.3-2

Let $\Phi(D_i) = k + 3$ if $i = 2^k$. Otherwise, let k be the largest integer such that $2^k \leq i$. Then define $\Phi(D_i) = \Phi(D_{2^k}) + 2(i - 2^k)$. Also, define $\Phi(D_0) = 0$. Then $\Phi(D_i) \geq 0$ for all i. The potential difference $\Phi(D_i) - \Phi(D_{i-1})$ is 2 if i is not a power of 2, and is $-2^k + 3$ if $i = 2^k$. Thus, the total amortized cost of n operations is $\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} 3 = 3n = O(n)$.

Exercise 17.3-3

Make the potential function be equal to $n \lg(n)$ where n is the size of the min-heap. Then, there is still a cost of $O(\lg(n))$ to insert, since only an amount

of ammortization that is about $\lg(n)$ was spent to increase the size of the heap by 1. However, since extract min decreases the size of the heap by 1, the actual cost of the operation is offset by a change in potential of the same order, so only a constant amount of work is needed.

Exercise 17.3-4

Since $D_n = s_n$, $D_0 = s_0$, and the amortized cost of n stack operations starting from an empty stack is is $O(n)$, equation 17.3 implies that the amortized cost is $O(n) + s_n - s_0$.

Exercise 17.3-5

Suppose that we have that $n \geq cb$. Since the counter begins with b 1's, we'll make all of our amortized cost $2 + \frac{1}{c}$. Then the additional cost of $\frac{1}{c}$ over the course of n operations amounts to paying an extra $\frac{n}{c} \geq b$ which was how much we were behind by when we started. Since the amortized cost of each operation is $2 + \frac{1}{c}$ it is in $O(1)$ so the total cost is in $O(n)$.

Exercise 17.3-6

We'll use the accounting method for the analysis. Assign cost 3 to the ENQUEUE operation and 0 to the DEQUEUE operation. Recall the implementation of 10.1-6 where we enqueue by pushing on to the top of stack 1, and dequeue by popping from stack 2. If stack 2 is empty, then we must pop every element from stack 1 and push it onto stack 2 before popping the top element from stack 2. For each item that we enqueue we accumulate 2 credits. Before we can dequeue an element, it must be moved to stack 2. Note: this might happen prior to the time at which we wish to dequeue it, but it will happen only once overall. One of the 2 credits will be used for this move. Once an item is on stack 2 its pop only costs 1 credit, which is exactly the remaining credit associated to the element. Since each operation's cost is $O(1)$, the amortized cost per operation is $O(1)$.

Exercise 17.3-7

We'll store all our elements in an array, and if ever it is too large, we will copy all the elements out into an array of twice the length. To delete the larger half, we first find the element m with order statistic $\lceil |S|/2 \rceil$ by the algorithm presented in section 9.3. Then, scan through the array and copy out the elements that are smaller or equal to m into an array of half the size. Since the delete half operation takes time $O(|S|)$ and reduces the number of elements by $\lfloor |S|/2 \rfloor \in \Omega(|S|)$, we can make these operations take amortized constant time by selecting our potential function to be linear in $|S|$. Since the insert operation only increases $|S|$ by one, we have that there is only a constant amount of work going towards satisfying the potential, so the total amortized cost of an insertion

3

is still constant. To output all the elements just iterate through the array and output each.

Exercise 17.4-1

By theorems 11.6-11.8, the expected cost of performing insertions and searches in an open address hash table approaches infinity as the load factor approaches one, for any load factor fixed away from 1, the expected time is bounded by a constant though. The expected value of the actual cost my not be O(1) for every insertion because the actual cost may include copying out the current values from the current table into a larger table because it became too full. This would take time that is linear in the number of elements stored.

Exercise 17.4-2

First suppose that $\alpha_i \geq 1/2$. Then we have

$$
\begin{aligned}
\hat{c}_i &= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\
&= 1 + 2 \cdot num_i - size_i - 2 \cdot num_i - 2 + size_i \\
&= -2.
\end{aligned}
$$

On the other hand, if $\alpha_i < 1/2$ then we have

$$
\begin{aligned}
\hat{c}_i &= 1 + (size_i/2 - num_i) - (2 \cdot num_{i-1} - size_{i-1}) \\
&= 1 + size_i/2 - num_i - 2 \cdot num_i - 2 + size_i \\
&= -1 + \frac{3}{2}(size_i - 2 \cdot num_i) \\
&\leq -1 + \frac{3}{2}(size_i - (size_i - 1)) \\
&\leq 1.
\end{aligned}
$$

Either way, the amortized cost is bounded above by a constant.

Exercise 17.4-3

If a resizing is not triggered, we have

$$
\begin{aligned}
\hat{c}_i &= C_i + \Phi_i - \Phi_{i-1} \\
&= 1 + |2 \cdot num_i - size_i| - |2 \cdot num_{i-1} - size_{i-1}| \\
&= 1 + |2 \cdot num_i - size_i| - |2 \cdot num_i + 2 - size_i| \\
&\leq 1 + |2 \cdot num_i - size_i| - |2 \cdot num_i - size_i| + 2 \\
&= 3
\end{aligned}
$$

However, if a resizing is triggered, suppose that $\alpha_{i-1} < \frac{1}{2}$. Then the actual cost is $num_i + 1$ since we do a deletion and move all the rest of the items. Also,

since we resize when the load factor drops below $\frac{1}{3}$, we have that $size_{i-1}/3 = num_{i-1} = num_i + 1$.

$$\hat{c}_i = c_i + \phi_i - \Phi_{i-1}$$
$$= num_i + 1 + |2 \cdot num_i - size_i| - |2 \cdot num_{i-1} - size_{i-1}|$$
$$\leq num_i + 1 + \left(\frac{2}{3}size_{i-1} - 2 \cdot num_i\right) - (size_{i-1} - 2 \cdot num_i - 2)$$
$$= num_i + 1 + (2 \cdot num_i + 2 - 2 \cdot num_i) - (3 \cdot num_i + 3 - 2 \cdot num_i)$$
$$= 2$$

The last case, that we had the load factor was greater than or equal to $\frac{1}{2}$, will not trigger a resizing because we only resize when the load drops below $\frac{1}{3}$.

Problem 17-1

a. Initialize a second array of length n to all trues, then, going through the indices of the original array in any order, if the corresponding entry in the second array is true, then swap the element at the current index with the element at the bit-reversed position, and set the entry in the second array corresponding to the bit-reversed index equal to false. Since we are running $rev_k < n$ times, the total runtime is $O(nk)$.

b. Doing a bit reversed increment is the same thing as adding a one to the leftmost position where all carries are going to the left instead of the right. See the algorithm BIT-REVERSED-INCREMENT(a)

Algorithm 1 BIT-REVERSED-INCREMENT(a)

---
let m be a 1 followed by k-1 zeroes
while m bitwise-AND a is not zero **do**
    a = a bitwise-XOR m
    shift m right by 1
end whilereturn m bitwise-OR a

By a similar analysis to the binary counter (just look at the problem in a mirror), this BIT-REVERSED-INCREMENT will take constant amortized time. So, to perform the bit-reversed permutation, have a normal binary counter and a bit reversed counter, then, swap the values of the two counters and increment. Do not swap however if those pairs of elements have already been swapped, which can be kept track of in a auxiliary array.

c. The BIT-REVERSED-INCREMENT procedure given in the previous part only uses single shifts to the right, not arbitrary shifts.

Problem 17-2

a. We linearly go through the lists and binary search each one since we don't know the relationship between one list an another. In the worst case, every list is actually used. Since list i has length 2^i and it's sorted, we can search it in $O(i)$ time. Since i varies from 0 to $O(\lg n)$, the runtime of SEARCH is $O((\lg n)^2)$.

b. To insert, we put the new element into A_0 and update the lists accordingly. In the worst case, we must combine lists $A_0, A_1, \ldots, A_{m-1}$ into list A_m. Since merging two sorted lists can be done linearly in the total length of the lists, the time this takes is $O(2^m)$. In the worst case, this takes time $O(n)$ since m could equal k.

We'll use the accounting method to analyse the amortized cost. Assign a cost of $\lg(n)$ to each insertion. Thus, each item carries $\lg(n)$ credit to pay for its later merges as additional items are inserted. Since an individual item can only be merged into a larger list and there are only $\lg(n)$ lists, the credit pays for all future costs the item might incur. Thus, the amortized cost is $O(\lg(n))$.

c. Find the smallest m such that $n_m \neq 0$ in the binary representation of n. If the item to be deleted is not in list A_m, remove it from its list and swap in an item from A_m, arbitrarily. This can be done in $O(\lg n)$ time since we may need to search list A_k to find the element to be deleted. Now simply break list A_m into lists $A_0, A_1, \ldots, A_{m-1}$ by index. Since the lists are already sorted, the runtime comes entirely from making the splits, which takes $O(m)$ time. In the worst case, this is $O(\lg n)$.

Problem 17-3

a. Since we have $O(x.size)$ auxiliary space, we will take the tree rooted at x and write down an inorder traversal of the tree into the extra space. This will only take linear time to do because it will visit each node thrice, once when passing to its left child, once when the nodes value is output and passing to the right child, and once when passing to the parent. Then, once the inorder traversal is written down, we can convert it back to a binary tree by selecting the median of the list to be the root, and recursing on the two halves of the list that remain on both sides. Since we can index into the middle element of a list in constant time, we will have the recurrence $T(n) = 2T(n/2) + 1$, which has solution that is linear. Since both trees come from the same underlying inorder traversal, the result is a BST since the original was. Also, since the root at each point was selected so that half the elements are larger and half the elements are smaller, it is a $1/2$-balanced tree.

b. We will show by induction that any tree with $\leq \alpha^{-d} + d$ elements has a depth of at most d. This is clearly true for $d = 0$ because any tree with

6

a single node has depth 0, and since $\alpha^0 = 1$, we have that our restriction on the number of elements requires there to only be one. Now, suppose that in some inductive step we had a contradiction, that is, some tree of depth d that is α balanced but has more than α^{-d} elements. We know that both of the subtrees are alpha balanced, and by being alpha balanced at the root, we have $root.left.size \leq \alpha \cdot root.size$ which implies $root.right.size > root.size - \alpha \cdot root.size - 1$. So, $root.right.size > (1 - \alpha)root.size - 1 > (1 - \alpha)\alpha^{-d} + d - 1 = (\alpha^{-1} - 1)\alpha^{-d+1} + d - 1 \geq \alpha^{-d+1} + d - 1$ which is a contradiction to the fact that it held for all smaller values of d because any child of a tree of depth d has depth $d - 1$.

c. The potential function is a sum of $\Delta(x)$ each of which is the absolute value of a quantity, so, since it is a sum of nonnegative values, it is nonnegative regardless of the input BST.

   If we suppose that our tree is $1/2-$ balanced, then, for every node x, we'll have that $\Delta(x) \leq 1$, so, the sum we compute to find the potential will be over no nonzero terms.

d. Suppose that we have a tree that has become no longer α balanced because it's left subtree has become too large. This means that $x.left.size > \alpha x.size = (\alpha - \frac{1}{2})x.size + \frac{1}{2}\alpha.size$. This means that we had at least $c(\alpha - \frac{1}{2})x.size$ units of potential. So, we need to select $c \geq \frac{1}{\alpha - \frac{1}{2}}$.

e. Suppose that our tree is α balanced. Then, we know that performing a search takes time $O(\lg(n))$. So, we perform that search and insert the element that we need to insert or delete the element we found. Then, we may have made the tree become unbalanced. However, we know that since we only changed one position, we have only changed the Δ value for all of the parents of the node that we either inserted or deleted. Therefore, we can rebuild the balanced properties starting at the lowest such unbalanced node and working up. Since each one only takes amortized constant time, and there are $O(\lg(n))$ many trees made unbalanced, tot total time to rebalanced every subtree is $O(\lg(n))$ amortized time.

Problem 17-4

a. If we insert a node into a complete binary search tree whose lowest level is all red, then there will be $\Omega(\lg n)$ instances of case 1 required to switch the colors all the way up the tree. If we delete a node from an all-black, complete binary tree then this also requires $\Omega(\lg n)$ time because there will be instances of case 2 at each iteration of the while-loop.

b. For RB-INSERT, cases 2 and 3 are terminating. For RB-DELETE, cases 1 and 3 are terminating.

7

c. After applying case 1, z's parent and uncle have been changed to black and z's grandparent is changed to red. Thus, there is a ned loss of one red node, so $\Phi(T') = \Phi(T) - 1$.

d. For case 1, there is a single decrease in the number of red nodes, and thus a decrease in the potential function. However, a single call to RB-INSERT-FIXUP could result in $\Omega(\lg n)$ instances of case 1. For cases 2 and 3, the colors stay the same and each performs a rotation.

e. Since each instance of case 1 requires a specific node to be red, it can't decrease the number of red nodes by more than $\Phi(T)$. Therefore the potential function is always non-negative. Any insert can increase the number of red nodes by at most 1, and one unit of potential can pay for any structural modifications of any of the 3 cases. Note that in the worst case, the call to RB-INSERT has to perform k case-1 operations, where k is equal to $\Phi(T_i) - \Phi(T_{i-1})$. Thus, the total amortized cost is bounded above by $2(\Phi(T_n) - \Phi(T_0)) \leq n$, so the amortized cost of each insert is $O(1)$.

f. In case 1 of RB-INSERT, we reduce the number of black nodes with two red children by 1 and we at most increase the number of black nodes with no red children by 1, leaving a net loss of at most 1 to the potential function. In our new potential function, $\Phi(T_n) - \Phi(T_0) \leq n$. Since one unit of potential pays for each operation and the terminating cases cause constant structural changes, the total amortized cost is $O(n)$ making the amortized cost of each RB-INSERT-FIXUP $O(1)$.

g. In case 2 of RB-DELETE, we reduce the number of black nodes with two red children by 1, thereby reducing the potential function by 2. Since the change in potential is at least negative 1, it pays for the structural modifications. Since the other cases cause constant structural changes, the total amortized cost is $O(n)$ making the amortized cost of each RB-DELETE-FIXUP $O(1)$.

h. As described above, whether we insert or delete in any of the cases, the potential function always pays for the changes made if they're nonterminating. If they're terminating then they already take constant time, so the amortized cost of any operation in a sequence of m inserts and deletes is $O(1)$, making the toal amortized cost $O(m)$.

Problem 17-5

a. Since the heuristic is picked in advance, given any sequence of requests given so far, we can simulate what ordering the heuristic will call for, then, we will

pick our next request to be whatever element will of been in the last position of the list. Continuing until all the requests have been made, we have that the cost of this sequence of accesses is $= mn$.

b. The cost of finding an element is $= rank_L(x)$ and since it needs to be swapped with all the elements before it, of which there are $rank_L(x) - 1$, the total cost is $2 \cdot rank_L(x) - 1$.

c. Regardless of the heuristic used, we first need to locate the element, which is left where ever it was after the previous step, so, needs $rank_{L_{i-1}}(x)$. After that, by definition, there are t_i transpositions made, so, $c_i^* = rank_{L_{i-1}}(x) + t_i^*$.

d. If we perform a transposition of elements y and z, where y is towards the left. Then there are two cases. The first is that the final ordering of the list in L_i^* is with y in front of z, in which case we have just increased the number of inversions by 1, so the potential increases by 2. The second is that in L_I^* z occurs before y, in which case, we have just reduced the number of inversions by one, reducing the potential by 2. In both cases, whether or not there is an inversion between y or z and any other element has not changed, since the transposition only changed the relative ordering of those two elements.

e. By definition, A and B are the only two of the four categories to place elements that precede x in L_{i-1}, since there are $|A| + |B|$ elements preceding it, it's rank in L_{i-1} is $|A| + |B| + 1$. Similarly, the two categories in which an element can be if it precedes x in L_{i-1}^* are A and C, so, in L_{i-1}^*, x has rank $|A| + |C| + 1$.

f. We have from part d that the potential increases by 2 if we transpose two elements that are being swapped so that their relative order in the final ordering is being screwed up, and decreases by two if they are begin placed into their correct order in L_i^*. In particular, they increase it by at most 2. since we are keeping track of the number of inversions that may not be the direct effect of the transpositions that heuristic H made, we see which ones the Move to front heuristic may of added. In particular, since the move to front heuristic only changed the relative order of x with respect to the other elements, moving it in front of the elements that preceded it in L_{i-1}, we only care about sets A and B. For an element in A, moving it to be behind A created an inversion, since that element preceded x in L_i^*. However, if the element were in B, we are removing an inversion by placing x in front of it.

g. First, we apply parts b and f to the expression for $\hat{c}_i$ to get $\hat{c}_i \leq 2 \cdot rank_L(x) - 1 + 2(|A| - |B| + t_i^*)$. Then, applying part e, we get this is $= 2(|A| + |B| + 1) - 1 + 2(|A| - |B| + t_i^*) = 4|A| - 1 + 2t_i^* \leq 4(|A| + |C| + 1) + 4t_i^* = 4(rank_{L_{i-1}^*}(x) + t_i^*)$. Finally, by part c, this bound is equal to $4c_i^*$.

h. We showed that the amortized cost of each operation under the move to front heuristic was at most four times the cost of the operation using any other heuristic. Since the amortized cost added up over all these operation is at

9

most the total (real) cost, so we have that the total cost with movetofront is
at most four times the total cost with an arbitrary other heuristic.

Chapter 18

Michelle Bodnar, Andrew Lohr

March 2, 2017

Exercise 18.1-1

If we allow $t = 0$, then, since we only know that internal nodes have to have at least $t - 1$ keys, it may be the case that some internal nodes represent no keys, a bad situation indeed.

Exercise 18.1-2

The possible values of t are 2 and 3. Every non-root node has at least 1 (resp. 2) keys and at most 3 (resp. 5) keys. The value of t cannot exceed 3 since some nodes have only 2 keys.

Exercise 18.1-3
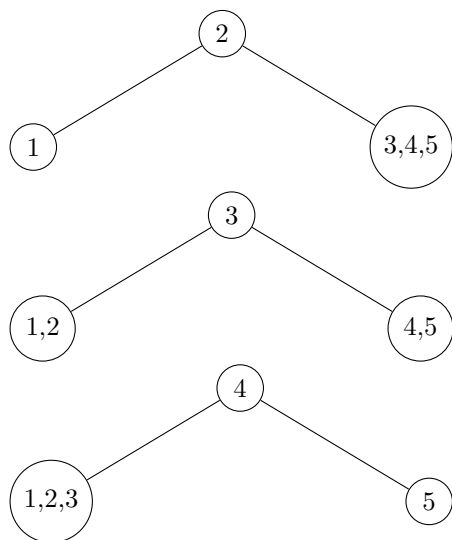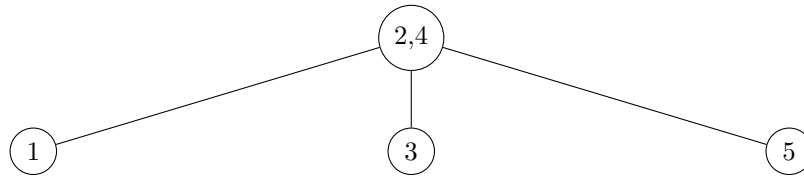
Exercise 18.1-4

The maximum number of nodes is achieved when every node has $2t$ children. In this case, there are $1 + 2t + (2t)^2 + \ldots + (2t)^h = \frac{1-(2t)^{h+1}}{1-2t}$ nodes. Since every node has at most $2t - 1$ keys, there are at most $(2t)^{h+1} - 1$ keys.

Exercise 18.1-5

We would get a t=2 B-tree. It would have one, two, or three keys depending on if it has zero, one, or two red children respectively. Suppose that the left child is red, then it's keys becomes the first one, and that red node's children become the first and second children of the new node. Similarly, if it is the right child that is red, that key becomes the last key listed with the new node, and the red nodes children become the second to last and last children of the new node.

Exercise 18.2-1

FQT
— C
— K
  — H
  — LM
— RS
— VW

FQT
— C
— K
  — H
  — LMN
— RS
— VW

FQT
— C
— KM
  — H
  — L
  — NP
— RS
— VW

FQT
— AC
— KM
  — H
  — L
  — NP
— RS
— VW
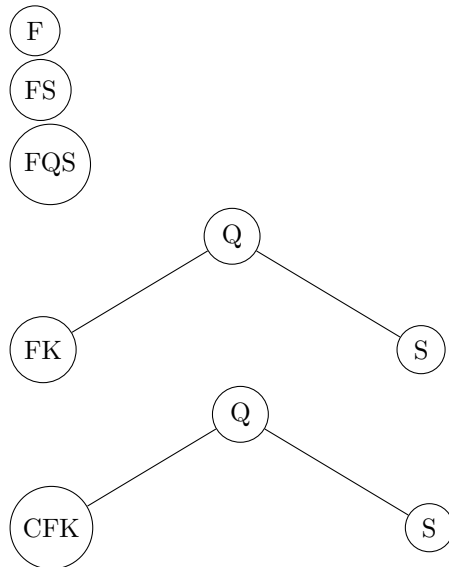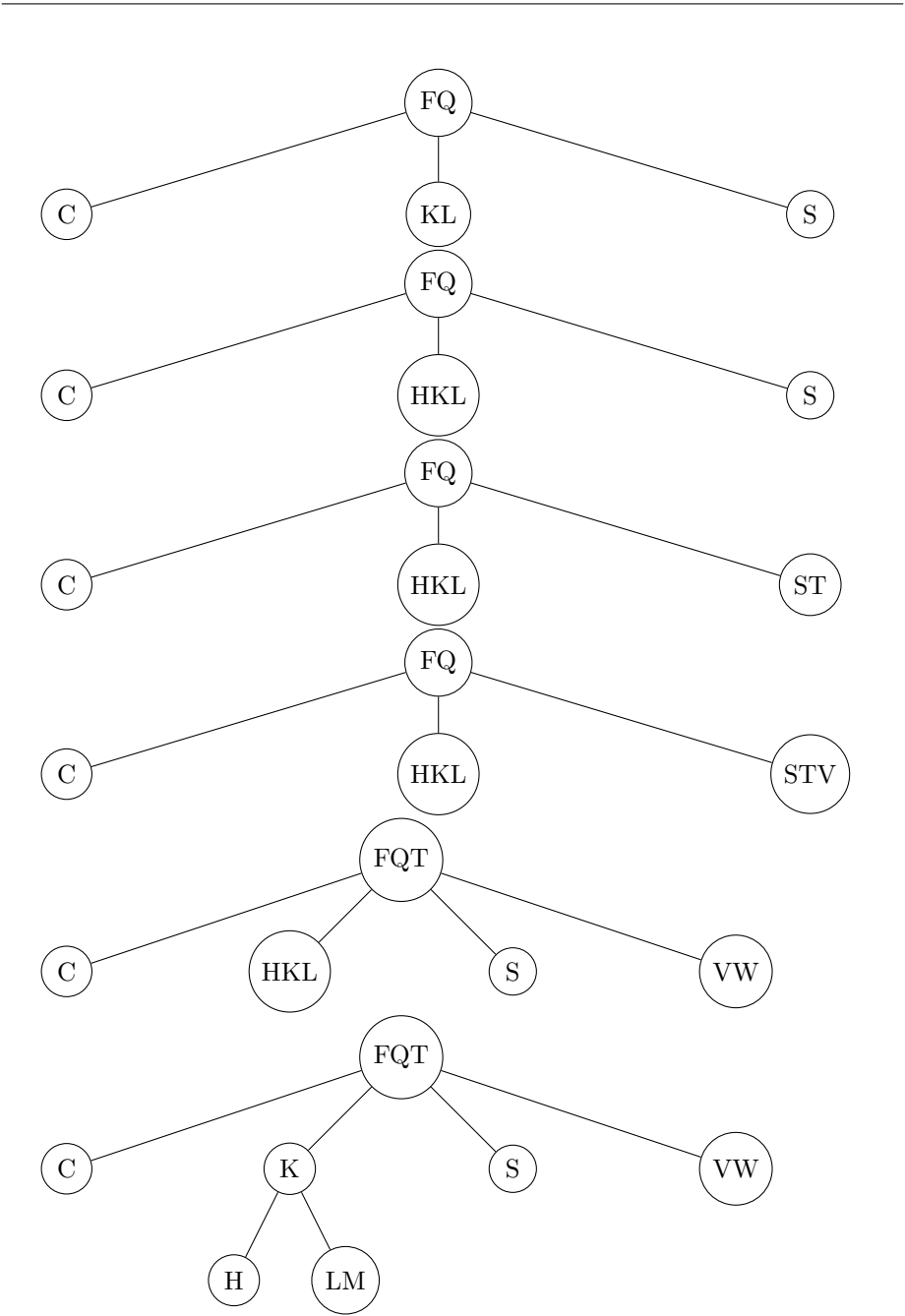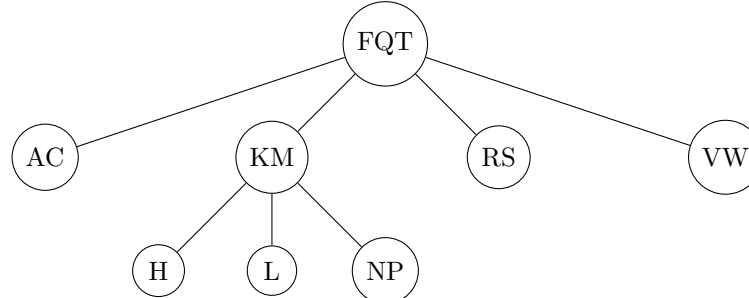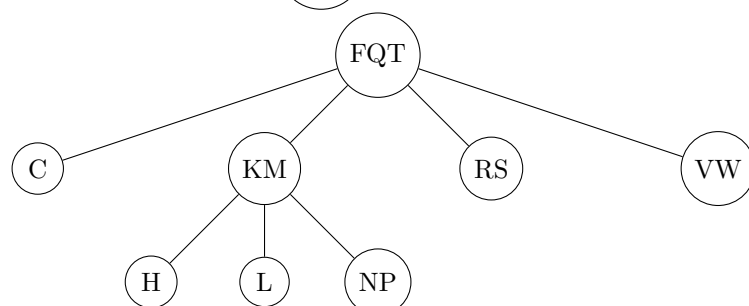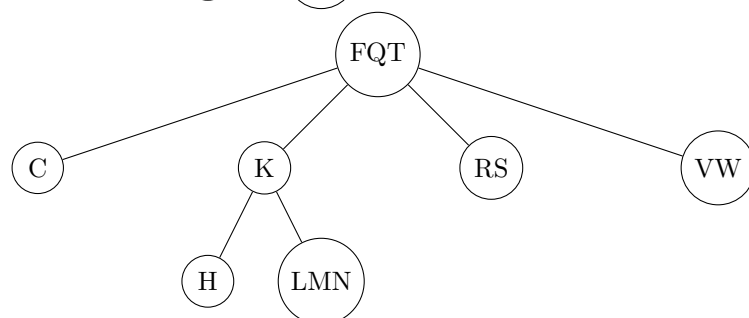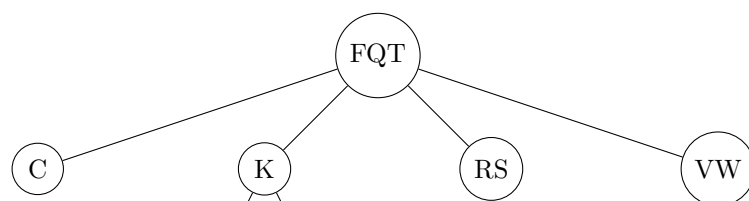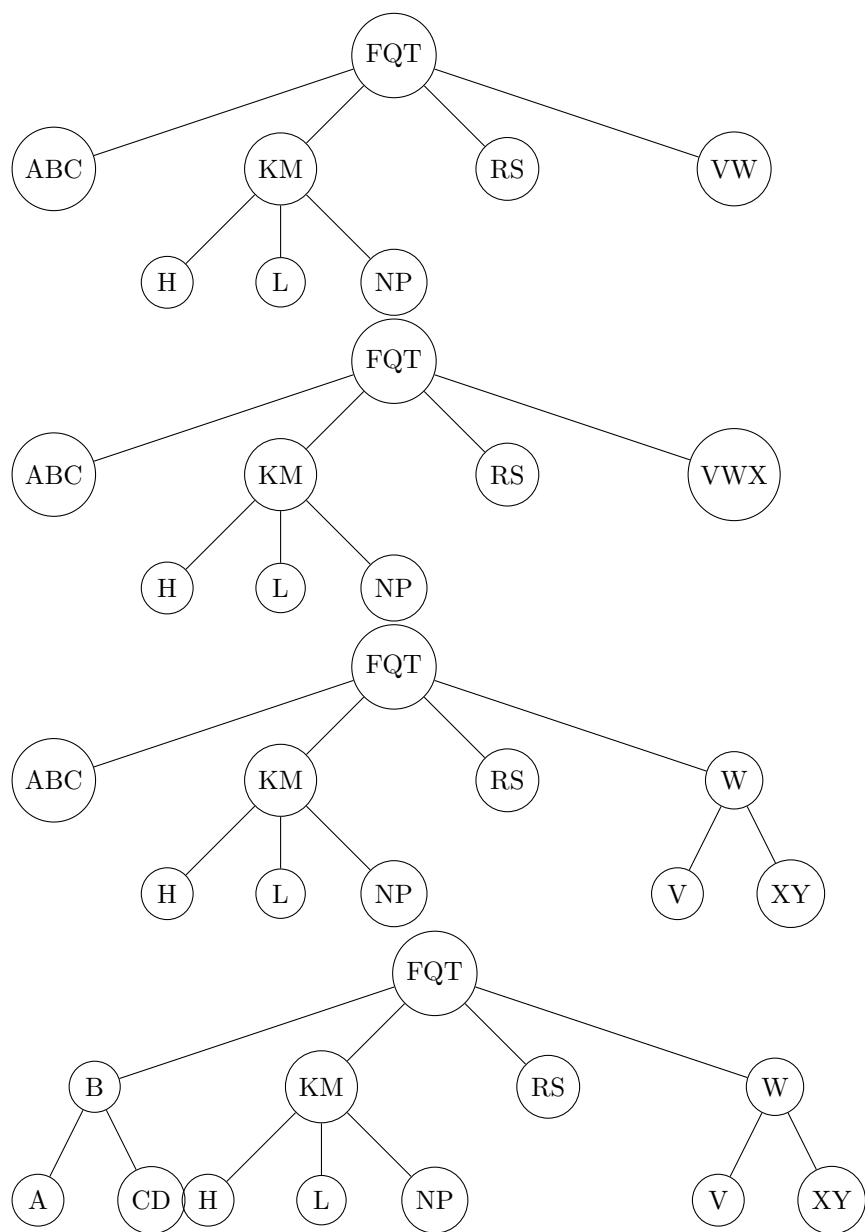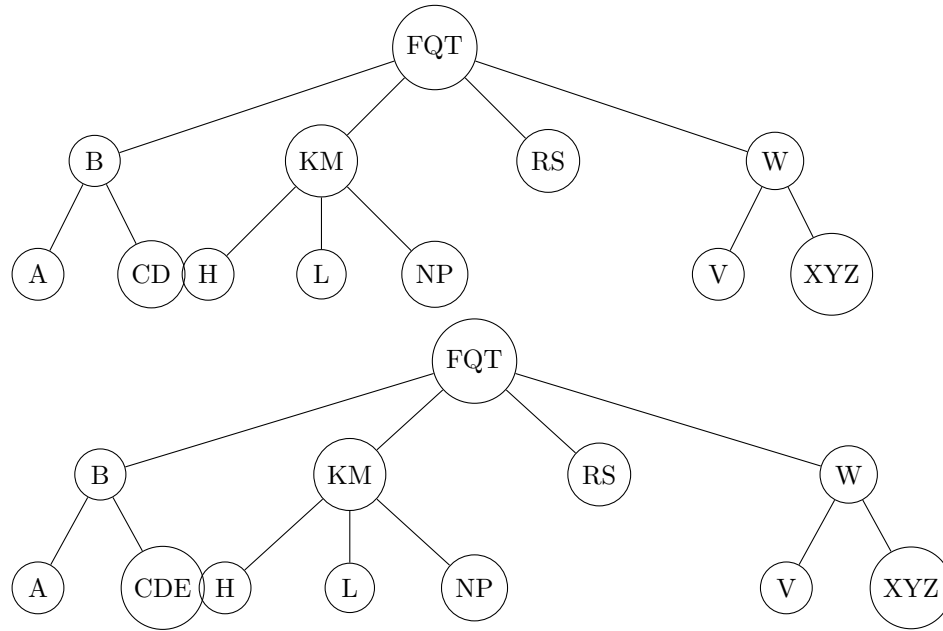
Exercise 18.2-2

Lines 1, 2, and 13 of B-TREE-SPLIT-CHILD guarantee that there are no redundant DISK-WRITE operations performed in this part of the algorithm, since each of these lines necessarily makes a change to nodes z, y, and x respectively. B-TREE-INSERT makes no calls to DISK-READ or DISK-WRITE. In B-TREE-INSERT-NONFULL, we only reach line 8 after executing line 7, which modifies x, so line 8 isn't redundant. The only call to DISK-READ occurs at line 12. Since calls to B-TREE-INSERT-NONFULL are made recursively on successive children, line 12 will never be redundant. Thus, no redundant read or write operations are ever performed.

Exercise 18.2-3

To find the minimum key, just always select the first child until you are on a leaf, then return the first key. To find the predecessor of a given key, fist find it. if it's on a leaf then just return the preceding key. If it's not a leaf, then return the largest element(in an analogous way to finding minimum) of the child that immediately precedes the key just found.

Exercise 18.2-4

The final tree can have as many as $n-1$ nodes. Unless $n=1$ there cannot ever be n nodes since we only ever insert a key into a non-empty node, so there will always be at least one node with 2 keys. Next observe that we will never have more than one key in a node which is not a right spine of our B-tree. This

is because every key we insert is larger than all keys stored in the tree, so it will be inserted into the right spine of the tree. Nodes not in the right spine are a result of splits, and since $t = 2$, every split results in child nodes with one key each. The fewest possible number of nodes occurs when every node in the right spine has 3 keys. In this case, $n = 2h + 2^{h+1} - 1$ where h is the height of the B-tree, and the number of nodes is $2^{h+1} - 1$. Asymptotically these are the same, so the number of nodes is $\Theta(n)$.

Exercise 18.2-5

You would modify the insertion procedure by, in B-TREE-Insert, check if the node is a leaf, and if it is, only split it if there twice as many keys stored as expected. Also, if an element needs to be inserted into a full leaf, we would split the leaf into two separate leaves, each of which doesn't have too many keys stored in it.

Exercise 18.2-6

If we use binary search rather than linear search, the CPU time becomes $O(\log_2(t) \log_t(n)) = O(\log_2(n))$ by the change of base formula.

Exercise 18.2-7

By Theorem 18.1, we have that the height of a B-tree on n elements is bounded by $\log_t \frac{n+1}{2}$. The number of page reads needed during a search is at worst the height. Since the cost per page access is now also a function of t, the time required for the search is $c(t) = (a + bt) \log_t \frac{n+1}{2}$. To minimize this expression, we'll take a derivative with respect to t. $c'(t) = b \log_t \frac{n+1}{2} - (a + bt) \frac{\ln\left(\frac{n+1}{2}\right)}{t \ln(t)^2}$. Then, setting this equal to zero, we have that

$$b \log_t \frac{n+1}{2} = (a + bt) \frac{\ln\left(\frac{n+1}{2}\right)}{t \ln(t)^2}$$

$$b \ln \frac{n+1}{2} = (a + bt) \frac{\ln\left(\frac{n+1}{2}\right)}{t \ln(t)}$$

$$t \ln(t) = (\frac{a}{b} + t)$$

$$t(\ln(t) - 1) = \frac{a}{b}$$

For our particular values of $a = 5$, and $b = 10$, we can solve this equation numerically to get an approximate maxima of 3.18, so selecting t=3 will minimize the worst case cost of a search in the tree.

Exercise 18.3-1
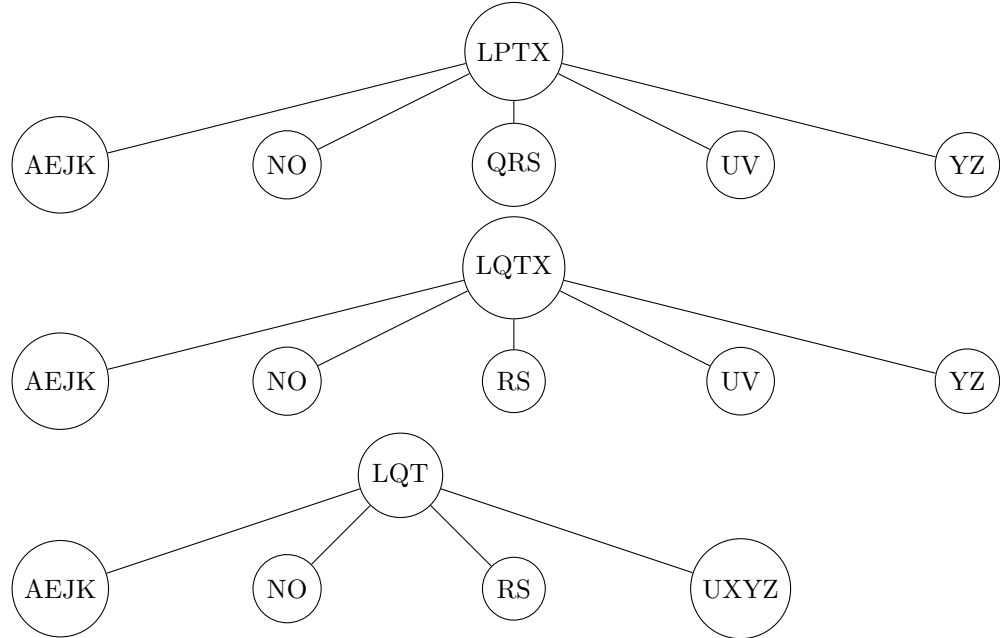
Exercise 18.3-2

   The algorithm B-TREE-DELETE(x, k) is a recursive procedure which deletes key k from the B-tree rooted at node x. The functions $\text{PRED}(k, x)$ and $\text{SUCC}(k, x)$ return the predecessor and successor of k in the B-tree rooted at x respectively. The cases where k is the last key in a node have been omitted because the pseudocode is already unwieldy. For these, we simply use the left sibling as opposed to the right sibling, making the appropriate modifications to the indexing in the for-loops.

Problem 18-1

a. We will have to make a disk access for each stack operation. Since each of these disk operations takes time $\Theta(m)$, the CPU time is $\Theta(mn)$.

b. Since only every mth push starts a new page, the number of disk operations is approximately n/m, and the CPU runtime is $\Theta(n)$, since both the contribution from the cost of the disk access and the actual running of the push operations are both $\Theta(n)$.

c. If we make a sequence of pushes until it just spills over onto the second page, then alternate popping and pulling many times, the asymptotic number of disk accesses and CPU time is of the same order as in part a. This is because

---

Algorithm 1 B-TREE-DELETE(x,k)

1: **if** $x.leaf$ **then**
2:    **for** $i = 1$ to $x.n$ **do**
3:       **if** $x.key_i == k$ **then**
4:          Delete key k from x
5:          $x.n = x.n - 1$
6:          DISK-WRITE(x)
7:          Return
8:       **end if**
9:    **end for**
10: **end if**
11: $i = 1$
12: **while** $x.key_i < k$ **do**
13:    $i = i + 1$
14: **end while**
15: **if** $x.key_i == k$ **then** // If k is in node x at position i
16:    DISK-READ($x.c_i$)
17:    **if** $x.c_i.n \geq t$ **then**
18:       $k' = PRED(k, x.c_i)$
19:       $x.key_i = k'$
20:       DISK-WRITE(x)
21:       B-TREE-DELETE($x.c_i, k'$)
22:       Return
23:       DISK-READ($x.c_{i+1}$) $x.c_{i+1}.n \geq t$
24:       $k' = SUCC(k, x.c_i)$
25:       $x.key_i = k'$
26:       DISK-WRITE(x)
27:       B-TREE-DELETE($x.c_{i+1}, k'$)
28:       Return
29:       $y = x.c_i$
30:       $z = x.c_{i+1}$
31:       $m = y.n$
32:       $p = z.n$
33:       $y.key_{m+1} = k$
34:       **for** $j = 1$ to p **do**
35:          $y.key_{m+1+j} = z.key_j$
36:       **end for**
37:       $y.n = m + p + 1$
38:       **for** $j = i + 1$ to $x.n - 1$ **do**
39:          $x.c_j = x.c_{j+1}$
40:       **end for**
41:       $x.n = x.n - 1$
42:       FREE(z)
43:       DISK-WRITE(x)
44:       DISK-WRITE(y)
45:       DISK-WRITE(z)
46:       B-TREE-DELETE(y, k)
47:       Return
48:    **end if**
49: **end if**

9

```
50:  DISK-READ(x.c_i)
51:  if x.c_i.n ≥ t then
52:      B-TREE-DELETE(x.c_i, k)
53:      Return
54:      DISK-READ(x.c_{i+1}) x.c_{i+1}.n ≥ t
55:      x.c_i.key_t = x.key_i
56:      x.c_i.n = x.c_i.n + 1
57:      x.key_i = x.c_{i+1}.key_1
58:      x.c_i.c_{t+1} = x.c_{i+1}.c_1
59:      x.c_i.n = t
60:      x.c_{i+1}.n = x.c_{i+1}.n - 1
61:      for j = 1 to x.c_{i+1}.n do
62:          x.c_{i+1}.key_j = x.c_{i+1}.key_{j+1}
63:      end for
64:      DISK-WRITE(x)
65:      DISK-WRITE(x.c_i)
66:      DISK-WRITE(x.c_{i+1})
67:      B-TREE-DELETE(x.c_i, k)
68:      y = x.c_i
69:      z = x.c_{i+1}
70:      m = y.n
71:      p = z.n
72:      y.key_{m+1} = x.key_i
73:      for j = 1 to p do
74:          y.key_{m+1+j} = z.key_j
75:      end for
76:      y.n = m + p + 1
77:      for j = i + 1 to x.n - 1 do
78:          x.c_j = x.c_{j+1}
79:      end for
80:      x.n = x.n - 1
81:      FREE(z)
82:      DISK-WRITE(x)
83:      DISK-WRITE(y)
84:      DISK-WRITE(z)
85:      B-TREE-DELETE(y, k)
86:      if x.n == 0 then //This occurs when the root contains no keys
87:          Free(x)
88:      end if
89:      Return
90:  end if
```

when we are doing that alternating of pops and pushes, each one triggers a disk access.

d. We define the potential of the stack to be the absolute value of the difference between the current size of the stack and the most recently passed multiple of m. This potential function means that the initial stack which has size 0, is also a multiple of m, so the potential is zero. Also, as we do a stack operation we either increase or decrease the potential by one. For us to have to load a new page from disk and write an old one to disk, we would need to be at least m positions away from the most recently visited multiple of m, because we would have had to just cross a page boundary. This cost of loading and storing a page takes (real) cpu time of $\Theta(m)$. However, we just had a drop in the potential function of order $\Theta(m)$. So, the amortized cost of this operation is $O(1)$.

Problem 18-2

a. For insertion it will suffice to explain how to update height when we split a node. Suppose node x is split into nodes y and z, and the median of x is merged into node w. The height of w remains unchanged unless x was the root (in which case $w.height = x.height + 1$). The height of y or z will often change. We set $y.height = \max_i y.c_i.height + 1$ and $z.height = \max_i z.c_i.height + 1$. Each update takes $O(t)$. Since a call to B-TREE-INSERT makes at most h splits where h is the height of the tree, the total time it takes to update heights is $O(th)$, preserving the asymptotic running time of insert. For deletion the situation is even simple. The only time the height changes is when the root has a single node and it is merged with its subtree nodes, leaving an empty root node to be deleted. In this case, we update the height of the new node to be the (old) height of the root minus 1.

b. Without loss of generality, assume $h' \geq h''$. We essentially wish to merge T'' into T' at a node of height h'' using node x. To do this, find the node at depth $h' - h''$ on the right spine of T'. Add x as a key to this node, and T'' as the additional child. If it should happen that the node was already full, perform a split operation.

c. Let x_i be the node encountered after i steps on path p. Let l_i be the index of the largest key stored in x_i which is less than or equal to k. We take $k_i' = x_i.key_{l_i}$ and T_{i-1}' to be the tree whose root node consists of the keys in x_i which are less than $x_i.key_{l_i}$, and all of their children. In general, $T_{i-1}'.height \geq T_i'.height$. For S'', we take a similar approach. They keys will be those in nodes passed on p which are immediately greater than k, and the trees will be rooted at a node consisting of the larger keys, with the

11

associated subtrees. When we reach the node which contains k, we don't assign a key, but we do assign a tree.

d. Let T_1 and T_2 be empty trees. Consider the path p from the root of T to k. Suppose we have reached node x_i. We join tree T'_{i-1} to T_1, then insert k'_i into T_1. We join T''_{i-1} to T_2 and insert k''_i into T_2. Once we have encountered the node which contains k at $x_m.key_k$, join $x_m.c_k$ with T_1 and $x_m.c_{k+1}$ with T_2. We will perform at most 2 join operations and 1 insert operation for each level of the tree. Using the runtime determined in part (b), and the fact that when we join a tree T' to T_1 (or T'' to T_2 respectively) the height difference is $T'.height - T_1.height$. Since the heights are nondecreasing of successive tree that are joined, we get a telescoping sum of heights. The first tree has height h, where h is the height of T, and the last tree has height 0. Thus, the runtime is $O(2(h+h)) = O(\lg n)$.

Chapter 19

Michelle Bodnar, Andrew Lohr

April 12, 2016

Exercise 19.2-1

First, we take the subtrees rooted at 24, 17, and 23 and add them to the root list. Then, we set H.min to 18. Then, we run consolidate. First this has its degree 2 set to the subtree rooted at 18. Then the degree 1 is the subtree rooted at 38. Then, we get a repeated subtree of degree 2 when we consider the one rooted at 24. So, we make it a subheap by placing the 24 node under 18. Then, we consider the heap rooted at 17. This is a repeat for heaps of degree 1, so we place the heap rooted at 38 below 17. Lastly we consider the heap rooted at 23, and then we have that all the different heaps have distinct degrees and are done, setting H.min to the smallest, that is, the one rooted at 17.

The three heaps that we end up with in our root list are:
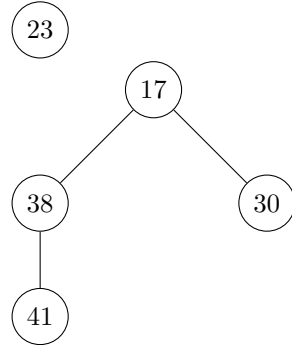
and
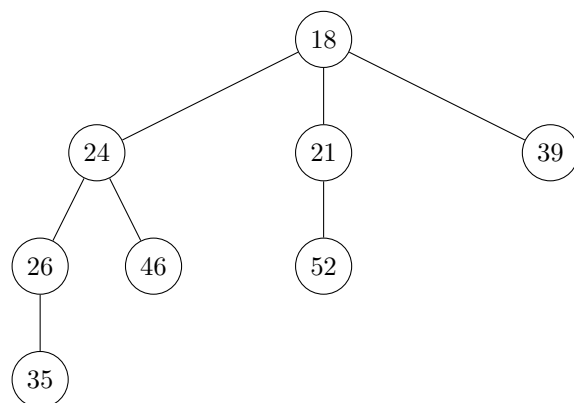
Exercise 19.3-1

A root in the heap became marked because it at some point had a child whose key was decreased. It doesn't add the potential for having to do any more actual work for it to be marked. This is because the only time that markedness is checked is in line 3 of cascading cut. This however is only ever run on nodes whose parent is non NIL. Since every root has NIL as it parent, line 3 of cascading cut will never be run on this marked root. It will still cause the potential function to be larger than needed, but that extra computation that was paid in to get the potential function higher will never be used up later.

Exercise 19.3-2

Recall that the actual cost of FIB-HEAP-DECREASE-KEY is $O(c)$, where c is the number of calls made to CASCADING-CUT. If c_i is the number of calls made on the ith key decrease, then the total time of n calls to FIB-HEAP-DECREASE-KEY is $\sum_{i=1}^{n} O(c_i)$. Next observe that every call to CASCADING-CUT moves a node to the root, and every call to a root node takes $O(1)$. Since no roots ever become children during the course of these calls, we must have that $\sum_{i=1}^{n} c_i = O(n)$. Therefore the aggregate cost is $O(n)$, so the average, or amortized, cost is $O(1)$.

Exercise 19.4-1

Add three nodes then delete one. This gets us a chain of length 1. Then, add three nodes, all with smaller values than the first three, and delete one of them. Then, delete the leaf that is only at depth 1. This gets us a chain of length 2. Then, make a chain of length two using this process except with all smaller keys. Then, upon a consolidate being forced, we will have that the remaining heap will have one path of length 3 and one of length 2, with a root that is unmarked. So, just run decrease key on all of the children along the shorter path, starting with those of shorter depth. Then, extract min the appropriate number of times. Then what is left over will be just a path of length 3. We can

continue this process ad infinitum. It will result in a chain of arbitrarily long length where all but the leaf is marked. It will take time exponential in n, but that's none of our concern.

More formally, we will make the following procedure $linear(n, c)$ that makes heap that is a linear chain of n nodes and has all of its keys between c and $c+2^n$. Also, as a precondition of running $linear(n, c)$, we have all the keys currently in the heap are less than c. As a base case, define $linear(1, c)$ to be the command insert(c). Define $linear(n + 1, c)$ as follows, where the return value list of nodes that lie on the chain but aren't the root

$S_1 = linear(n, c)$
$S_2 = linear(n, c + 2^n)$
$x.key = -\infty$
$insert(x)$
$extractmin()$
  for each entry in S_1, delete that key
  The heap now has the desired structure, return S_2

Exercise 19.4-2

Following the proof of lemma 19.1, if x is any node if a Fibonacci heap, $x.degree = m$, and x has children $y_1, y_2, \ldots, y_m$, then $y_1.degree \geq 0$ and $y_i.degree \geq i - k$. Thus, if s_m denotes the fewest nodes possible in a node of degree m, then we have $s_0 = 1, s_1 = 2, \ldots, s_{k-1} = k$ and in general, $s_m = k + \sum_{i=0}^{m-k} s_i$. Thus, the difference between s_m and s_{m-1} is s_{m-k}. Let $\{f_m\}$ be the sequence such that $f_m = m + 1$ for $0 \leq m < k$ and $f_m = f_{m-1} + f_{m-k}$ for $m \geq k$. If $F(x)$ is the generating function for f_m then we have $F(x) = \frac{1-x^k}{(1-x)(1-x-x^k)}$. Let α be a root of $x^k = x^{k-1} + 1$. We'll show by induction that $f_{m+k} \geq \alpha^m$. For the base cases:

$$f_k = k + 1 \geq 1 = \alpha^0$$
$$f_{k+1} = k + 3 \geq \alpha^1$$
$$\vdots$$
$$f_{k+k} = k + \frac{(k+1)(k+2)}{2} = k + k + 1 + \frac{k(k+1)}{2} \geq 2k + 1 + \alpha^{k-1} \geq \alpha^k.$$

In general, we have

$$f_{m+k} = f_{m+k-1} + f_m \geq \alpha^{m-1} + \alpha^{m-k} = \alpha^{m-k}(\alpha^{k-1} + 1) = \alpha^m.$$

Next we show that $f_{m+k} = k + \sum_{i=0}^{m} f_i$. The base case is clear, since $f_k = f_0 + k = k + 1$. For the induction step, we have

$$f_{m+k} = f_{m-1-k} + f_m = k + \sum_{i=0}^{m-1} f_i + f_m = k + \sum_{i=0}^{m} f_i.$$

3

Observe that $s_i \geq f_{i+k}$ for $0 \leq i < k$. Again, by induction, for $m \geq k$ we have

$$s_m = k + \sum_{i=0}^{m-k} s_i \geq k + \sum_{i=0}^{m-k} f_{i+k} \geq k + \sum_{i=0}^{m} f_i = f_{m+k}$$

so in general, $s_m \geq f_{m+k}$. Putting it all together, we have:

$$size(x) \geq s_m$$
$$\geq k + \sum_{i=k}^{m} s_{i-k}$$
$$\geq k + \sum_{i=k}^{m} f_i$$
$$\geq f_{m+k}$$
$$\geq \alpha^m.$$

Taking logs on both sides, we have

$$\log_\alpha n \geq m.$$

In other words, provided that α is a constant, we have a logarithmic bound on the maximum degree.

Problem 19-1

a. It can take actual time proportional to the number of children that x had because for each child, when placing it in the root list, their parent pointer needs to be updated to be NIL instead of x.

b. Line 7 takes actual time bounded by x.degree since updating each of the children of x only takes constant time. So, if c is the number of cascading cuts that are done, the actual cost is $O(c + x.degree)$.

c. From the cascading cut, we marked at most one more node, so, $m(H') \leq 1 + m(H)$ regardless of the number of calls to cascading cut, because only the highest thing in the chain of calls actually goes from unmarked to marked. Also, the number of children increases by the number of children that x had, that is $t(H') = x.degree + t(H)$. Putting these together, we get that

$$\Phi(H') \leq t(H) + x.degree + 2(1 + m(H))$$

d. The asymptotic time is $\Theta(x.degree) = \Theta(\lg(n))$ which is the same asyptotic time that was required for the original deletion method.

Problem 19-2

a. We proceed by induction to prove all four claims simultaneously. When $k = 0$, B_0 has $2^0 = 1$ node. The height of B_0 is 0. The only possible depth is 0, at which there are $\binom{0}{0} = 1$ node. Finally, the root has degree 0 and it has no children. Now suppose the claims hold for k. B_{k+1} is formed by connecting two copies of B_k, so it has $2^k + 2^k = 2^{k+1}$ nodes. The height of the tree is the height of B_k plus 1, since we have added an extra edge connecting the root of B_k to the new root of the tree, so the height is $k + 1$. At depth i we get a contribution of $\binom{k}{i-1}$ from the first tree, and a contribution of $\binom{k}{i}$ from the second. Summing these and applying a common binomial identity gives $\binom{k+1}{i}$. Finally, the degree of the root is the sum of 1, and the degree of the root of B_k, which is $1 + k$. If we number the children left to right by $k, k-1, \ldots, 0$, then the first child corresponds to the root of B_k by definition. The remaining children correspond to the proper roots of subtrees by the induction hypothesis.

b. Let $n.b$ denote the binary expansion of n. The fact that we can have at most one of each binomial tree corresponds to the fact that we can have at most 1 as any digit of $n.b$. Since each binomial tree has a size which is a power of 2, the binomial trees required to represent n nodes are uniquely determined. We include B_k if and only if the kth position of $n.b$ is 1. Since the binary representation of n has at most $\lfloor \lg n \rfloor + 1$ digits, this also bounds the number of trees which can be used to represent n nodes.

c. Given a node x, let $x.key$, $x.p$, $x.c$, and $x.s$ represent the attributes key, parent, left-most child, and sibling to the right, respectively. The pointer attributes have value NIL when no such node exists. The root list will be stored in a singly linked list. MAKE-HEAP() is implemented by initializing an empty list for the root list and returning a pointer to the head of the list, which contains NIL. This takes constant time. To insert: Let x be a node with key k , to be inserted. Scan the root list to find the first m such that B_m is not one of the trees in the binomial heap. If there is no B_0, simply create a single root node x. Otherwise, union $x, B_0, B_1, \ldots, B_{m-1}$ into a B_m tree. Remove all root nodes of the unioned trees from the root list, and update it with the new root. Since each join operation is logarithmic in the height of the tree, the total time is $O(\lg n)$. MINIMUM just scans the root list and returns the minimum in $O(\lg n)$, since the root list has size at most $O(\lg n)$. EXTRACT-MIN finds and deletes the minimum, then splits the tree B_m which contained the minimum into its component binomial trees $B_0, B_1, \ldots, B_{m-1}$ in $O(\lg n)$ time. Finally, it unions each of these with any existing trees of the same size in $O(\lg n)$ time. To implement UNION, suppose we have two binomial heaps consisting of trees $B_{i_1}, B_{i_2}, \ldots, B_{i_k}$ and $B_{j_1}, B_{j_2}, \ldots, B_{j_m}$ respectively. Simply union corresponding trees of the same size between the two heaps, then do another check and join any newly created trees which have caused additional duplicates. Note: we will perform at

most one union on any fixed size of binomial tree so the total running time is still logarithmic in n, where we assume that n is sum of the sizes of the trees which we are unioning. To implement DECREASE-KEY, simply swap the node whose key was decreased up the tree until it satisfies the min-heap property. To implement DELETE, note that every binomial tree consists of two copies of a smaller binomial tree, so we can write the procedure recursively. If the tree is a single node, simply delete it. If we wish to delete from B_k, first split the tree into its constituent copies of B_{k-1}, and recursively call delete on the copy of B_{k-1} which contains x. If this results in two binomial trees of the same size, simply union them.

d. The Fibonacci heap will look like a binomial heap, except that multiple copies of a given binomial tree will be allowed. Since the only trees which will appear are binomial trees and B_k has 2^k nodes, we must have $2^k \leq n$, which implies $k \leq \lfloor \lg n \rfloor$. Since the largest root of any binomial tree occurs at the root, and on B_k it is degree k, this also bounds the largest degree of a node.

e. INSERT and UNION will no longer have amortized $O(1)$ running time because CONSOLIDATE has runtime $O(\lg n)$. Even if no nodes are consolidated, the runtime is dominated by the check that all degrees are distinct. Since calling UNION on a heap and a single node is the same as insertion, it must also have runtime $O(\lg n)$. The other operations remain unchanged.

Problem 19-3

a. If $k < x.key$ just run the decrease key procedure. If $k > x.key$, delete the current value x and insert x again with a new key. Both of these cases only need $O(\lg(n))$ amortized time to run.

b. Suppose that we also had an additional cost to the potential function that was proportional to the size of the structure. This would only increase when we do an insertion, and then only by a constant amount, so there aren't any worries concerning this increased potential function raising the amortized cost of any operations. Once we've made this modification, to the potential function, we also modify the heap itself by having a doubly linked list along all of the leaf nodes in the heap. To prune we then pick any leaf node, remove it from it's parent's child list, and remove it from the list of leaves. We repeat this $\min(r, H.n)$ times. This causes the potential to drop by an amount proportional to r which is on the order of the actual cost of what just happened since the deletions from the linked list take only constant amounts of time each. So, the amortized time is constant.

Problem 19-4

a. Traverse a path from root to leaf as follows: At a given node, examine the attribute $x.small$ in each child-node of the current node. Proceed to the child node which minimizes this attribute. If the children of the current node are leaves, then simply return a pointer to the child node with smallest key. Since the height of the tree is $O(\lg n)$ and the number of children of any node is at most 4, this has runtime $O(\lg n)$.

b. Decrease the key of x, then traverse the simple path from x to the root by following the parent pointers. At each node y encountered, check the attribute $y.small$. If $k < y.small$, set $y.small = k$. Otherwise do nothing and continue on the path.

c. Insert works the same as in a B-tree, except that at each node it is assumed that the node to be inserted is "smaller" than every key stored at that node, so the runtime is inherited. If the root is split, we update the height of the tree. When we reach the final node before the leaves, simply insert the new node as the leftmost child of that node.

d. As with B-TREE-DELETE, we'll want to ensure that the tree satisfies the properties of being a 2-3-4 tree after deletion, so we'll need to check that we're never deleting a leaf which only has a single sibling. This is handled in much the same way as in chapter 18. We can imagine that dummy keys are stored in all the internal nodes, and carry out the deletion process in exactly the same way as done in exercise 18.3-2, with the added requirement that we update the height stored in the root if we merge the root with its child nodes.

e. EXTRACT-MIN simply locates the minimum as done in part a, then deletes it as in part d.

f. This can be done by implementing the join operation, as in Problem 18-2 (b).

Chapter 20

Michelle Bodnar, Andrew Lohr

April 12, 2016

Exercise 20.1-1

To modify these structure to allow for multiple elements, instead of just storing a bit in each of the entries, we can store the head of a linked list representing how many elements of that value that are contained in the structure, with a NIL value to represent having no elements of that value.

Exercise 20.1-2

All operations will remain the same, except instead of the leaves of the tree being an array of integers, they will be an array of nodes, each of which stores $x.key$ in addition to whatever additional satellite data you wish.

Exercise 20.1-3

To find the successor of a given key k from a binary tree, call the procedure $SUCC(x, T.root)$. Note that this will return NIL if there is no entry in the tree with a larger key.

Exercise 20.1-4

The new tree would have height k. INSERT would take $O(k)$, MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR, and DELETE would take $O(ku^{1/k})$.

Exercise 20.2-1

See the two algorithms, PROTO-vEB-MAXIMUM and PROTO-vEB-PREDECESSOR.

Exercise 20.2-2

When we delete a key, we need to check membership of all keys of that cluster to know how to update the summary structure. There are $\sqrt{u}$ of these, and each membership takes $O(\lg \lg u)$ time to check. With the recursive calls, recurrence for running time is

Algorithm 1 SUCC(k,x)

---

  if $k < x.key$ **then**
    if $x.left == NIL$ **then**
      return x
    else
      if $SUCC(k, x.left) == NIL$ **then**
        return x
      else
        return SUCC(k,x.left)
      end if
    end if
  else
    if $x.right == NIL$ **then**
      return NIL
    else
      return SUCC(k,x.right)
    end if
  end if

Algorithm 2 PROTO-vEB-MAXIMUM(V)

---

  if $V.u == 2$ **then**
    if V.A[1] == 1 **then**
      return 1
    else if V.A[0] == 1 **then**
      return 0
    else
      return NIL
    end if
  else
    max-cluster = PROTO-vEB-MAXIMUM($V.summary$)
    if max-cluster ==NIL **then**
      return NIL
    else
      $offset = PROTO - vEB - MINIMUM(V.cluster[max - cluster])$
      return $index(max - cluster, offset)$
    end if
  end if

Algorithm 3 PROTO-vEB-PREDECESSOR(V,x)

---
if V.u==2 **then**
    if x==1 and V.A[0] ==1 **then**
        return 0
    else
        return NIL
    end if
else
    $offset = PROTO-vEB-PREDECESSOR(V.cluster[high(x)], low(x))$
    if $offset \neq NIL$ **then**
        return index(high(x),offest)
    else
        pred-cluster = PROTO-vEB-PREDECESSOR(V.summary,high(x))
        if pred-cluster ==NIL **then**
            return NIL
        else
            return index(succ-cluster, PROTO-vEB-MAXIMUM(V.cluster[pred-cluster]))
        end if
    end if
end if

$$T(u) = T(\sqrt{u}) + O(\sqrt{u} \lg \lg u).$$

We make the substitution $m = \lg u$ and $S(m) = T(2^m)$. Then we apply the Master Theorem, using case 3, to solve the recurrence. Substituting back, we find that the runtime is $T(u) = O(\sqrt{u} \lg \lg u)$.

Exercise 20.2-3

We would keep the same as before, but insert immediately after the else, a check of whether $n = 1$. If it doesn't continue as usual, but if it does, then we can just immediately set the summary bit to zero, null out the pointer in the table, and be done immediately. This has the upside that it can sometimes save up to $\lg \lg u$. The procedure has the big downside that the number of elements that are in the set could be as high as $\lg(\lg(u))$, in which case $\lg(u)$ many bits are needed to store n.

Exercise 20.2-4

The array A found in a proto van Emde Boas structure of size 2 should now support integers, instead of just bits. All other pats of the structure will remain the same. The integer will store the number of duplicates at that position. The modifications to insert, delete, minimum, successor, etc... will be minor. Only the base cases will need to be updated.

Algorithm 4 PROTO-vEB-DELETE(V, x)
---
  if $V.u == 2$ **then**
      $V.A[x] = 0$
  else
      PROTO-vEB-DELETE($V.cluster[high(x)], low(x)$)
      $inCluster = False$
      for $i = high(x) \cdot \sqrt{u}$ to $(high(x) + 1) \cdot \sqrt{u} - 1$ **do**
          if PROTO-vEB-MEMBER($V.cluster[high(x)], i$) **then**
              $inCluster = True$
              Break
          end if
      end for
      if $inCluster == False$ **then**
          PROTO-vEB-DELETE($V.summary, high(x)$)
      end if
  end if

Exercise 20.2-5

The only modification necessary would be for the u=2 trees. They would need to also include a length two array that had pointers to the corresponding satellite data which would be populated in case the corresponding entry in A were 1.

Exercise 20.2-6

This algorithm recursively allocates proper space and appropriately initializes attributes for a proto van Emde Boas structure of size u.

Algorithm 5 Make-Proto-vEB(u)

  $V = allocate\text{-}node()$
  $V.u = u$
  if $u == 2$ **then**
      Allocate-Array A of size 2
      $V.A[1] = V.A[0] = 0$
  else
      $V.summary = $ Make-Proto-vEB($\sqrt{u}$)
      for $i = 0$ to $\sqrt{u} - 1$ **do**
          $V.cluster[i] = $ Make-Proto-vEB($\sqrt{u}$)
      end for
  end if

Exercise 20.2-7

For line 9 to be executed, we would need that in the summary data, we also had a NIL returned. This could of either happened through line 9, or 6. Eventually though, it would need to happen in line 6, so, there must be some number of summarizations that happened of V that caused us to get an empty u=2 vEB. However, a summarization has an entry of one if any of the corresponding entries in the data structure are one. This means that there are no entries in V, and so, we have that V is empty.

Exercise 20.2-8

For MEMBER the runtime recurrence is $T(u) = T(u^{1/4}) + O(1)$, whose solution is again $O(\lg \lg(u))$. For MINIMUM, $T(u) = 2T(u^{1/4}) + O(1)$. Making a substitution and applying case 1 of the master theorem, this is $O(\sqrt{\lg u})$. For SUCCESSOR, $T(u) = 2T(u^{1/4}) + O(\lg u)$. By case 3 of the master theorem, this is $O(\lg u)$. For INSERT, $T(u) = 2T(u^{1/4}) + O(1)$. This is the same as MINIMUM, which is $O(\sqrt{\lg u})$. To analyze DELETE, we need to update the recurrence to reflect the fact that DELETE depends on MEMBER. The new recurrence is $T(u) = T(u^{1/4}) + O(u^{1/4} \lg \lg u)$. By case 3 of the master theorem, this is $O(u^{1/4} \lg \lg u)$.

Exercise 20.3-1

To support duplicate keys, for each u=2 vEB tree, instead of storing just a bit in each of the entries of its array, it should store an integer representing how many elements of that value the vEB contains.

Exercise 20.3-2

For any key which is a minimum on some vEB, we'll need to store its satellite data with the min value since the key doesn't appear in the subtree. The rest of the satellite data will be stored alongside the keys of the vEB trees of size 2. Explicitly, for each non-summary vEB tree, store a pointer in addition to min. If min is NIL, the pointer should also point to NIL. Otherwise, the pointer should point to the satellite data associated with that minimum. In a size 2 vEB tree, we'll have two additional pointers, which will each point to the minimum's and maximum's satellite data, or NIL if these don't exist. In the case where min=max, the pointers will point to the same data.

Exercise 20.3-3

We define the procedure for any u that is a power of 2. If $u = 2$, then, just slap that fact together with an array of length 2 that contains 0 in both entries.

If $u = 2^k > 2$, then, we create an empty vEB tree called Summary with $u = 2^{\lceil k/2 \rceil}$. We also make an array called cluster of length $2^{\lceil k/2 \rceil}$ with each

entry initialized to an empty vEB tree with $u = 2^{\lfloor k/2 \rfloor}$. Lastly, we create a min and max element, both initialized to NIL.

Exercise 20.3-4

Suppose that x is already in V and we call INSERT. Then we can't satisfy lines 1, 3, 6, or 10, so we will enter the else case on line 9 every time, causing an infinite loop. Now suppose we call DELETE when x isn't in V. If there is only a single element in V, lines 1 through 3 will delete it, regardless of what element it is. To enter the elseif of line 4, x can't be equal to 0 or 1 and the vEB tree must be of size 2. In this case, we delete the max element, regardless of what it is. Since the recursive call always puts us in this case, we always delete an element we shouldn't. To avoid these issue, keep and updated auxiliary array A with u elements. Set $A[i] = 0$ if i is not in the tree, and 1 if it is. Since we can perform constant time updates to this array, it won't affect the runtime of any of our operations. When inserting x, check first to be sure $A[x] == 0$. If it's not, simply return. If it is, set $A[x] = 1$ and proceed with insert as usual. When deleting x, check if $A[x] == 1$. If it isn't, simply return. If it is, set $A[x] = 0$ and proceed with delete as usual.

Exercise 20.3-5

Similar to the analysis of (20.4), we will analyze:

$$T(u) \leq T(u^{1-1/k}) + T(u^{1/k}) + O(1)$$

This is a good choice for analysis because for many operations we first check the summary vEB tree, which will have size $u^{1/k}$ (the second term). And then possible have to check a vEB tree somewhere in cluster, which will have size $u^{1-1/k}$(the first term). We let $T(2^m) = S(m)$, so the equation becomes

$$S(m) \leq S(m(1-1/k)) + S(m/k) + O(1).$$

If $k > 2$ the first term dominates, so by master theorem, we'll have that $S(m)$ is $O(\lg(m))$, this means that T will be $O(\lg(\lg(u)))$ just as in the original case where we took squareroots.

Exercise 20.3-6

Set $n = u/\lg\lg u$. Then performing n operations takes $c(u + n\lg\lg u)$ time for some constant c. Using the aggregate amortized analysis, we divide by n to see that the amortized cost of each operations is $c(\lg\lg u + \lg\lg u) = O(\lg\lg u)$ per operation. Thus we need $n \geq u/\lg\lg u$.

Problem 20-1

6

a. Lets look at what has to be stored for a vEB tree. Each vEB tree contains one vEB tree of size $\sqrt[+]{u}$ and $\sqrt[+]{u}$ vEB trees of size $\sqrt[-]{u}$. It also is storing three numbers each of order $O(u)$, so they need $\Theta(\lg(u))$ space each. Lastly, it needs to store $\sqrt[+]{u}$ many pointers to the cluster vEB trees. We'll combine these last two contributions which are $\Theta(\lg(u))$ and $\Theta(\sqrt[+]{u})$ respectively into a single term that is $\Theta \sqrt[+]{u}$. This gets us the recurrence

$$P(u) = P(\sqrt[+]{u}) + \sqrt[+]{u}P(\sqrt[-]{u}) + \Theta(\sqrt{u})$$

Then, we have that $u = 2^{2m}$ (which follows from the assumption that $\sqrt{u}$ was an integer), this equation becomes

$$P(u) = (1 + 2^m)P(2^m) + \Theta(\sqrt{u}) = (1 + \sqrt{u})P(\sqrt{u}) + \Theta(\sqrt{u})$$

as desired.

b. We recall from our solution to problem 3-6.e (it seems like so long ago now) that given a number n, a bound on the number of times that we need to take the squareroot of a number before it falls below 2 is $\lg(\lg(n))$. So, if we just unroll out recurrence, we get that

$$P(u) \leq \left(\prod_{i=1}^{\lg(\lg(u))} (u^{1/2^i} + 1) \right) P(2) + \sum_{i=1}^{\lg(\lg(u))} \Theta(u^{1/2^i})(u^{1/2i} + 1)$$

The first product has a highest power of u corresponding to always multiplying the first terms of each binomial. The power in this term is equal to $\sum_{i=1}^{\lg(\lg(u))} \frac{1}{2^i}$ which is a partial sum of a geometric series whose sum is 1. This means that the first term is $o(u)$. The order of the ith term in the summation appearing in the formula is $u^{2/2^i}$. In particular, for $i = 1$ is it $O(u)$, and for any $i > 1$, we have that $2/2^i < 1$, so those terms will be $o(u)$. Putting it all together, the largest term appearing is $O(u)$, and so, $P(u)$ is $O(u)$.

c. For this problem we just use the version written for normal vEB trees, with minor modifications. That is, since there are entries in cluster that may not exist, and summary may of not yet been initialized, just before we try to access either, we check to see if it's initialized. If it isn't, we do so then.

d. As in the previous problem, we just wait until just before either of the two things that may of not been allocated try to get used then allocate them if need be.

e. Since the initialization performed only take constant time, those modifications don't ruin the the desired runtime bound for the original algorithms already had. So, our responses to parts c and d are $O(\lg(\lg(n)))$.

f. As mentioned in the errata, this part should instead be changed to $O(n \lg(n))$ space. When we are adding an element, we may have to add an entry to a dynamic hash table, which means that a constant amount of extra space

Algorithm 6 RS-vEB-TREE-INSERT(V,x)

> **if** $V.min == NIL$ **then**
>> vEB-EMPTY-TREE-INSERT(V,x)
>
> **else**
>> **if** $x < V.min$ **then**
>>> swap V.min with x
>>
>> **end if**
>> **if** $V.u > 2$ **then**
>>> **if** $V.summary == NIL$ **then**
>>>> $V.summary = CREATE - NEW - RD - vEB - TREE(\sqrt[+]{V.u})$
>>>
>>> **end if**
>>> **if** $lookup(V.cluster, low(x)) == NIL$ **then**
>>>> insert into V.summary with key high(x) what is returned by
>>
>> $CREATE - NEW - RD - vEB - TREE(\sqrt{V.u})$
>>> **end if**
>>> **if** $vEB - TREE - MINIMUM(lookup(V.cluster, high(x))) == NIL$
>
> **then**
>>>> vEB-TREE-INSERT(V.summary,high(x))
>>>> vEB-EMPTY-TREE-INSERT(lookup(V.cluster,high(x)),low(x))
>>>
>>> **else**
>>>> vEB-TREE-INSERT(lookup(V.cluster,high(x)),low(x))
>>>
>>> **end if**
>>
>> **end if**
>> **if** $x > V.max$ **then**
>>> $V.max = x$
>>
>> **end if**
>
> **end if**

Algorithm 7 RS-vEB-TREE-SUCCESSOR(V,x)

---

if $V.u == 2$ **then**
    if $x == 0$ and $V.max == 1$ **then**
        return 1
    else
        return NIL
    end if
else if $V.min \neq NIL$ and $x < V.min$ **then**
    return V.min
else
    if $lookup(V.cluster, low(x)) == NIL$ **then**
        insert into V.summary with key high(x) what is returned by
$CREATE - NEW - RD - vEB - TREE(\sqrt{V.u})$
    end if
    max-low = vEB-TREE-MAXIMUM(lookup(V.cluster,high(x)))
    if $max - low \neq NIL$ and $low(x) < max - low$ **then**
        return$index(high(x), vEB \qquad - \qquad TREE \qquad -$
$SUCCESSOR(lookup(V.summary, high(x)), low(x)))$
    else
        if $V.summary == NIL$ **then**
            $V.summary = CREATE - NEW - RD - vEB - TREE(\sqrt[+]{V.u})$
        end if
        $succ - cluster = vEB - TREE - SUCCESSOR(V.summary, high(x))$
        if succ-cluster==NIL **then**
            return NIL
        else
            return$index(succ \qquad - \qquad cluster, vEB \qquad - \qquad TREE \qquad -$
$MINIMUM(lookup(V.summary, succ - cluster)))$
        end if
    end if
end if

---

would be needed. If we are adding an element to that table, we also have to add an element to the RS-vEB tree in the summary, but the entry that we add in the cluster will be a constant size RS-vEB tree. We can charge the cost of that addition to the summary table to the making the minimum element entry that we added in the cluster table. Since we are always making at least one element be added as a new min entry somewhere, this amortization will mean that it is only a constant amount of time in order to store the new entry.

g. It only takes a constant amount of time to create an empty RS-vEB tree. This is immediate since the only dependence on u in CREATE-NEW-RS-vEB-TREE(u) is on line 2 when V.u is initialized, but this only takes a constant amount of time. Since nothing else in the procedure depends on u, it must take a constant amount of time.

Problem 20-2

a) By 11.5, the perfect hash table uses $O(m)$ space to store m elements. In a universe of size u, each element contributes $\lg u$ entries to the hash table, so the requirement is $O(n \lg u)$. Since the linked list requires $O(n)$, the total space requirement is $O(n \lg u)$.

b) MINIMUM and MAXIMUM are easy. We just examine the first and last elements of the associated doubly linked list. MEMBER can actually be performed in $O(1)$, since we are simply checking membership in a perfect hash table. PREDECESSOR and SUCCESSOR are a bit more complicated. Assume that we have a binary tree in which we store all the elements and their prefixes. When we query the hash table for an element, we get a pointer to that element's location in the binary search tree, if the element is in the tree, and NULL otherwise. Moreover, assume that every leaf node comes with a pointer to its position in the doubly linked list. Let x be the number whose successor we seek. Begin by performing a binary search of the prefixes in the hash table to find the longest hashed prefix y which matches a prefix of x. This takes $O(\lg \lg u)$ since we can check if any prefix is in the hash table in $O(1)$. Observe that y can have at most one child in the BST, because if it had both children then one of these would share a longer prefix with x. If the left child is missing, have the left child pointer point to the largest labeled leaf node in the BST which is less than y. If the right child is missing, use its pointer to point to the successor of y. If If y is a leaf node then $y = x$, so we simply follow the pointer to x in the doubly linked list, in $O(1)$, and its successor is the next element on the list. If y is not a leaf node, we follow its predecessor or successor node, depending on which we need. This gives us $O(1)$ access to the proper element, so the total runtime is $O(\lg \lg u)$. INSERT and DELETE must take $O(\lg u)$ since we need to insert one entry into the hash table for each of their bits and update the pointers.

10

c) The hash table has $\lg u$ entries for each of the $n/\lg u$ groups, so it stores a total of n entries, making it size $O(n)$. There are $n/\lg u$ binary trees of size $\lg u$, so they take $O(n)$ space. Finally, the linked list takes $O(n/\lg u)$ space. Thus, the total space requirement is $O(n)$.

d) For MINIMUM: Starting with the linked list, locate the minimum representative. This is $O(1)$ since we can just look at the start of the doubly linked list. Then use the hash table to find its corresponding binary tree in $O(1)$. Since this binary tree contains $lg(u)$ elements and is balanced, its height is $\lg \lg u$, so we can find its minimum in $O(\lg \lg u)$. The procedure is similar for MAXIMUM.

e) We start by finding the smallest representative greater than or equal to x. To do this, store the representatives in the structure described above, with runtimes given in parts a and b, and call SUCCESSOR(x) to find the proper binary search tree to look in. Since $n \leq u$ we can do this in $O(\lg \lg u)$. Next we search the binary search tree for x. Since its height is $\lg u$, the total runtime is $O(\lg \lg u)$.

f) Again, if we can find the largest representative greater than or equal to x, we can determine which binary tree contains the predecessor or successor of x. To do this, just call PREDECESSOR or SUCCESSOR on x to locate the appropriate tree in $O(\lg \lg u)$. Since the tree has height $\lg u$, we can find the predecessor or successor in $(O \lg \lg u)$.

g) Insertion and deletion into a binary tree of height $\lg u$ is $\Omega(\lg \lg u)$. In addition to this, we may have to update the representatives of the groups which can only increase the running time. representatives.

h) We can relax the requirements and only impose the condition that each group has at least $\frac{1}{2}\lg u$ elements and at most $2\lg u$ elements. If a red-black tree is too big, we split it in half at the median. If a red-black tree is too small, we merge it with a neighboring tree. If this causes the merged tree to become too large, we split it at the median. If a tree splits, we create a new representative. If two trees merge, we delete the lost representative. Any split or merge takes $O(\lg u)$ since we have to insert or delete an element in the data structure storing our representatives, which by part b takes $O(\lg u)$. However, we only split a tree after at least $\lg u$ insertions, since the size of one of the red-black trees needs to increase from $\lg u$ to $2\lg u$ and we only merge two trees after at least $(1/2)\lg u$ deletions, because the size of the merging tree needs to have decreased from $\lg u$ to $(1/2)\lg u$. Thus, the amortized cost of the merges, splits, and updates to representatives is $O(1)$ per insertion or deletion, so the amortized cost is $O(\lg \lg u)$ as desired.

Chapter 21

Michelle Bodnar, Andrew Lohr

April 12, 2016

Exercise 21.1-1

EdgeProcessed											
initial	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$	$\{k\}$
(d,i)	$\{a\}$	$\{b\}$	$\{c\}$	$\{d,i\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$		$\{j\}$	$\{k\}$
(f,k)	$\{a\}$	$\{b\}$	$\{c\}$	$\{d,i\}$	$\{e\}$	$\{f,k\}$	$\{g\}$	$\{h\}$		$\{j\}$	
(g,i)	$\{a\}$	$\{b\}$	$\{c\}$	$\{d,i,g\}$	$\{e\}$	$\{f,k\}$		$\{h\}$		$\{j\}$	
(b,g)	$\{a\}$	$\{b,d,i,g\}$	$\{c\}$		$\{e\}$	$\{f,k\}$		$\{h\}$		$\{j\}$	
(a,h)	$\{a,h\}$	$\{b,d,i,g\}$	$\{c\}$		$\{e\}$	$\{f,k\}$				$\{j\}$	
(i,j)	$\{a,h\}$	$\{b,d,i,g,j\}$	$\{c\}$		$\{e\}$	$\{f,k\}$					
(d,k)	$\{a,h\}$	$\{b,d,i,g,j,f,k\}$	$\{c\}$		$\{e\}$						
(b,j)	$\{a,h\}$	$\{b,d,i,g,j,f,k\}$	$\{c\}$		$\{e\}$						
(d,f)	$\{a,h\}$	$\{b,d,i,g,j,f,k\}$	$\{c\}$		$\{e\}$						
(g,j)	$\{a,h\}$	$\{b,d,i,g,j,f,k\}$	$\{c\}$		$\{e\}$						
(a,e)	$\{a,h,e\}$	$\{b,d,i,g,j,f,k\}$	$\{c\}$								

So, the connected components that we are left with are $\{a, h, e\}$, $\{b, d, i, g, j, f, k\}$, and $\{c\}$.

Exercise 21.1-2

First suppose that two vertices are in the same connected component. Then there exists a path of edges connecting them. If two vertices are connected by a single edge, then they are put into the same set when that edge is processed. At some point during the algorithm every edge of the path will be processed, so all vertices on the path will be in the same set, including the endpoints. Now suppose two vertices u and v wind up in the same set. Since every vertex starts off in its own set, some sequence of edges in G must have resulted in eventually combining the sets containing u and v. From among these, there must be a path of edges from u to v, implying that u and v are in the same connected component.

Exercise 21.1-3

Find set is called twice on line 4, this is run once per edge in the graph, so, we have that find set is run $2|E|$ times. Since we start with $|V|$ sets, at the end

1

only have k, and each call to UNION reduces the number of sets by one, we have that we have to of made $|V| - k$ calls to UNION.

Exercise 21.2-1

The three algorithms follow the english description and are provided here. There are alternate versions using the weighted union heuristic, suffixed with WU.

Algorithm 1 MAKE-SET(x)

Let o be an object with three fields, next, value, and set
Let L be a linked list object with head = tail = o
o.next = NIL
o.set = L
o.value = x
return L

Algorithm 2 FIND-SET(x)
---
return o.set.head.value

Algorithm 3 UNION(x,y)

L1= x.set
L2 = y.set
L1.tail.next = L2.head
z = L2.head
while $z.next \neq NIL$ **do**
    z.set = L1
end while
L1.tail = L2.tail
return L1

Exercise 21.2-2

Originally we have 16 sets, each containing x_i. In the following, we'll replace x_i by i. After the for loop in line 3 we have:

$$\{1, 2\}, \{3, 4\}, \{5, 6\}, \{7, 8\}, \{9, 10\}, \{11, 12\}, \{13, 14\}, \{15, 16\}.$$

After the for loop on line 5 we have

$$\{1, 2, 3, 4\}, \{5, 6, 7, 8\}, \{9, 10, 11, 12\}, \{13, 14, 15, 16\}.$$

Line 7 results in:

Algorithm 4 MAKE-SET-WU(x)

  L = MAKE-SET(x)
  L.size = 1
  return L

Algorithm 5 UNION-WU(x,y)

  L1= x.set
  L2 = y.set
  if $L1.size \geq L2.size$ **then**
     L = UNION(x,y)
  else
     L = UNION(y,x)
  end if
  L.size = L1.size + L2.size
  return L

$$\{1, 2, 3, 4, 5, 6, 7, 8\}, \{9, 10, 11, 12\}, \{13, 14, 15, 16\}.$$

Line 8 results in:

$$\{1, 2, 3, 4, 5, 6, 7, 8\}, \{9, 10, 11, 12, 13, 14, 15, 16\}.$$

Line 9 results in:

$$\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16\}.$$

FIND-SET(x_2) and FIND-SET(x_9) each return pointers to x_1.

Exercise 21.2-3

During the proof of theorem 21.1, we concluded that the time for the n UNION operations to run was at most $O(n \lg(n))$. This means that each of them took an amortized time of at most $O(\lg(n))$. Also, since there is only a constant actual amount of work in performing MAKE-SET and FIND-SET operations, and none of that ease is used to offset costs of UNION operations, they both have $O(1)$ runtime.

Exercise 21.2-4

We call MAKE-SET n times, which contributes $\Theta(n)$. In each union, the smaller set is of size 1, so each of these takes $\Theta(1)$ time. Since we union $n-1$ times, the runtime is $\Theta(n)$.

Exercise 21.2-5

For each member of the set, we will make its first field which used to point back to the set object point instead to the last element of the linked list. Then, given any set, we can find its last element by going ot the head and following the pointer that that object maintains to the last element of the linked list. This only requires following exactly two pointers, so it takes a constant amount of time. Some care must be taken when unioning these modified sets. Since the set representative is the last element in the set, when we combine two linked lists, we place the smaller of the two sets before the larger, since we need to update their set representative pointers, unlike the original situation, where we update the representative of the objects that are placed on to the end of the linked list.

Exercise 21.2-6

Instead of appending the second list to the end of the first, we can imagine splicing it into the first list, in between the head and the elements. Store a pointer to the first element in S_1. Then for each element x in S_2, set $x.head = S_1.head$. When the last element of S_2 is reached, set its next pointer to the first element of S_1. If we always let S_2 play the role of the smaller set, this works well with the weighted-union heuristic and don't affect the asymptotic running time of UNION.

Exercise 21.3-1
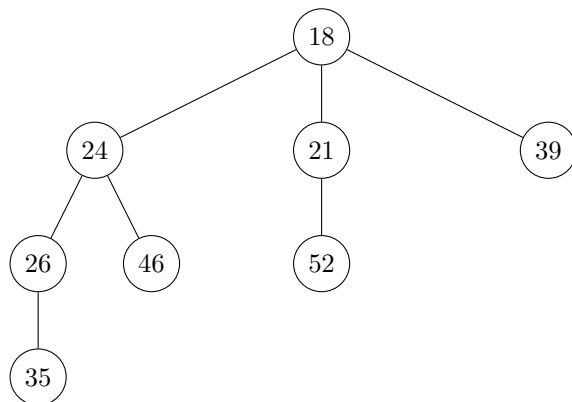
Exercise 21.3-2

To implement FIND-SET nonrecursively, let x be the element we call the function on. Create a linked list A which contains a pointer to x. Each time we most one element up the tree, insert a pointer to that element into A. Once the root r has been found, use the linked list to find each node on the path from the root to x and update its parent to r.

Exercise 21.3-3

Suppose that $n' = 2^k$ is the smallest power of two less than n. To see that this sequences of operations does take the required amount of time, we'll first note that after each iteration of the for loop indexed by j, we have that the elements $x_1, \ldots, x_{n'}$ are in trees of depth i. So, after we finish the outer for loop, we have that $x_1 \ldots x_{n'}$ all lie in the same set, but are represented by a tree of depth $k \in \Omega(\lg(n))$. Then, since we repeatedly call FIND-SET on an item that is $\lg(n)$ away from its set representative, we have that each one takes time $\lg(n)$. So, the last for loop alltogther takes time $m \lg(n)$.

Algorithm 6 Sequence of operations for Exercise 21.3-3

---

   for i=1..n **do**
      MAKE-SETx_i
   end for
   for $i = 1..k$ **do**
      for $j = 1..n' - 2^{i=1} by 2^i$ **do**
         UNION($x_i, x_{i+2^{j-1}}$)
      end for
   end for
   for $i = 1..m$ **do**
      FIND-SET(x_1)
   end for

Exercise 21.3-4

In addition to each tree, we'll store a linked list (whose set object contains a single tail pointer) with which keeps track of all the names of elements in the tree. The only additional information we'll store in each node is a pointer $x.l$ to that element's position in the list. When we call MAKE-SET(x), we'll also create a new linked list, insert the label of x into the list, and set $x.l$ to a pointer to that label. This is all done in $O(1)$. FIND-SET will remain unchanged. UNION(x, y) will work as usual, with the additional requirement that we union the linked lists of x and y. Since we don't need to update pointers to the head, we can link up the lists in constant time, thus preserving the runtime of UNION. Finally, PRINT-SET(x) works as follows: first, set $s = $ FIND-SET(x). Then print the elements in the linked list, starting with the element pointed to by x. (This will be the first element in the list). Since the list contains the same number of elements as the set and printing takes $O(1)$, this operation takes linear time in the number of set members.

Exercise 21.3-5

Clearly each MAKE-SET and LINK operation only takes time $O(1)$, so, supposing that n is the number of FIND-SET operations occuring after the making and linking, we need to show that all the FIND-SET operations only

5

take time $O(n)$. To do this, we will ammortize some of the cost of the FIND-SET operations into the cost of the MAKE-SET operations. Imagine paying some constant amount extra for each MAKE-SET operation. Then, when doing a FIND-SET(x) operation, we have three possibilities. First, we could have that x is the representative of its own set. In this case, it clearly only takes constant time to run. Second, we could have that the path from x to its set's representative is already compressed, so it only takes a single step to find the set representative. In this case also, the time required is constant. Lastly, we could have that x is not the representative and it's path has not been compressed. Then, suppose that there are k nodes between x and its representative. The time of this find-set operation is $O(k)$, but it also ends up compressing the paths of k nodes, so we use that extra amount that we paid during the MAKE-SET operations for these k nodes whose paths were compressed. Any subsequent call to find set for these nodes will take only a constant amount of time, so we would never try to use the work that amortization amount twice for a given node.

Exercise 21.4-1

The initial value of x.rank is 0, as it is initialized in line 2 of the MAKE-SET(x) procedure. When we run LINK(x,y), whichever one has the larger rank is placed as the parent of the other, and if there is a tie, the parent's rank is incremented. This means that after any LINK(y,x), the two nodes being linked satisfy this strict inequality of ranks. Also, if we have that $x \neq x.p$, then, we have that x is not its own set representative, so, any linking together of sets that would occur would not involve x, but that's the only way for ranks to increase, so, we have that x.rank must remain constant after that point.

Exercise 21.4-2

We'll prove the claim by strong induction on the number of nodes. If $n = 1$, then that node has rank equal to $0 = \lfloor \lg 1 \rfloor$. Now suppose that the claim holds for $1, 2, \ldots, n$ nodes. Given $n + 1$ nodes, suppose we perform a UNION operation on two disjoint sets with a and b nodes respectively, where $a, b \leq n$. Then the root of the first set has rank at most $\lfloor \lg a \rfloor$ and the root of the second set has rank at most $\lfloor \lg b \rfloor$. If the ranks are unequal, then the UNION operation preserves rank and we are done, so suppose the ranks are equal. Then the rank of the union increases by 1, and the resulting set has rank $\lfloor \lg a \rfloor + 1 \leq \lfloor \lg(n+1)/2 \rfloor + 1 = \lfloor \lg(n+1) \rfloor$.

Exercise 21.4-3

Since their value is at most $\lfloor \lg(n) \rfloor$, we can represent them using $\Theta(\lg(\lg(n)))$ bits, and may need to use that many bits to represent a number that can take that many values.

Exercise 21.4-4

MAKE-SET takes constant time and both FIND-SET and UNION are bounded by the largest rank among all the sets. Exercise 21.4-2 bounds this from about by $\lfloor \lg n \rfloor$, so the actual cost of each operation is $O(\lg n)$. Therefore the actual cost of m operations is $O(m \lg n)$.

Exercise 21.4-5

He isn't correct, suppose that we had that $rank(x.p) > A_2(rank(x))$ but that $rank(x.p.p) = 1 + rank(x.p)$, then we would have that $level(x.p) = 0$, but $level(x) \geq 2$. So, we don't have that $level(x) \leq level(x.p)$ even though we have that the ranks are monotonically increasing as we go up in the tree. Put another way, even though the ranks are monotonically increasing, the rate at which they are increasing (roughly captured by the level vales) doesn't have to, itself be increasing.

Exercise 21.4-6

First observe that by a change of variables, $\alpha'(2^{n-1}) = \alpha(n)$. Earlier in the section we saw that $\alpha(n) \leq 3$ for $0 \leq n \leq 2047$. This means that $\alpha'(n) \leq 2$ for $0 \leq n \leq 2^{2046}$, which is larger than the estimated number of atoms in the observable universe. To prove the improved bound of $O(m\alpha'(n))$ on the operations, the general structure will be essentially the same as that given in the section. First, modify bound 21.2 by observing that $A_{\alpha'(n)}(x.rank) \geq A_{\alpha'(n)}(1) \geq \lg(n+1) > x.p.rank$ which implies $level(x) \leq \alpha'(n)$. Next, redefine the potential replacing $\alpha(n)$ by $\alpha'(n)$. Lemma 21.8 now goes through just as before. All subsequent lemmas rely on these previous observations, and their proofs go through exactly as in the section, yielding the bound.

Problem 21-1

a.

index	value
1	4
2	3
3	2
4	6
5	8
6	1

b. As we run the for loop, we are picking off the smallest of the possible elements to be removed, knowing for sure that it will be removed by the next unused EXTRACT-MIN operation. Then, since that EXTRACT-MIN operation is used up, we can pretend that it no longer exists, and combine the set of things that were inserted by that segment with those inserted by the next, since we know that the EXTRACT-MIN operation that had separated the

two is now used up. Since we proceed to figure out what the various extract operations do one at a time, by the time we are done, we have figured them all out.

c. We let each of the sets be represented by a disjoint set structure. To union them (as on line 6) just call UNION. Checking that they exist is just a matter of keeping track of a linked list of which ones exist(needed for line 5), initially containing all of them, but then, when deleting the set on line 6, we delete it from the linked list that we were maintaining. The only other interaction with the sets that we have to worry about is on line 2, which just amounts to a call of FIND-SET(j). Since line 2 takes amortized time $\alpha(n)$ and we call it exactly n times, then, since the rest of the for loop only takes constant time, the total runtime is $O(n\alpha(n))$.

Problem 21-2

a. MAKE-TREE and GRAFT are both constant time operations. FIND-DEPTH is linear in the depth of the node. In a sequence of m operations the maximal depth which can be achieved is $m/2$, so FIND-DEPTH takes at most $O(m)$. Thus, m operations take at most $O(m^2)$. This is achieved as follows: Create $m/3$ new trees. Graft them together into a chain using $m/3$ calls to GRAFT. Now call FIND-DEPTH on the deepest node $m/3$ times. Each call takes time at least $m/3$, so the total runtime is $\Omega((m/3)^2) = \Omega(m^2)$. Thus the worst-case runtime of the m operations is $\Theta(m^2)$.

b. Since the new set will contain only a single node, its depth must be zero and its parent is itself. In this case, the set and its corresponding tree are indistinguishable.

Algorithm 7 MAKE-TREE(v)

$v = $ Allocate-Node()
$v.d = 0$
$v.p = v$
Return v

---

c. In addition to returning the set object, modify FIND-SET to also return the depth of the parent node. Update the pseudodistance of the current node v to be $v.d$ plus the returned pseudodistance. Since this is done recursively, the running time is unchanged. It is still linear in the length of the find path. To implement FIND-DEPTH, simply recurse up the tree containing v, keeping a running total of pseudodistances.

d. To implement GRAFT we need to find v's actual depth and add it to the pseudodistance of the root of the tree S_i which contains r.

Algorithm 8 FIND-SET(v)

---
if $v \neq v.p$ **then**
    $(v.p, d) = FIND - SET(v.p)$
    $v.d = v.d + d$
    Return $(v.p, v.d)$
else
    Return $(v, 0)$
end if

Algorithm 9 GRAFT(r,v)

$(x, d1) = $ FIND-SET(r)
$(y, d2) = $ FIND-SET(v)
if $x.rank > y.rank$ **then**
    $y.p = x$
    $x.d = x.d + d2 + y.d$
else
    $x.p = y$
    $x.d = x.d + d2$
    if $x.rank == y.rank$ **then**
        $y.rank = y.rank + 1$
    end if
end if

---

    e. The three implemented operations have the same asymptotic running time as MAKE, FIND, and UNION for disjoint sets, so the worst-case runtime of m such operations, n of which are MAKE-TREE operations, is $O(m\alpha(n))$.

Problem 21-3

a. Suppose that we let $\leq_{LCA}$ to be an ordering on the vertices so that $u \leq_{LCA} v$ if we run line 7 of $LCA(u)$ before line 7 of $LCA(v)$. Then, when we are running line 7 of $LCA(u)$, we immediately go on to the for loop on line 8. So, while we are doing this for loop, we still haven't called line 7 of LCA(v). This means that v.color is white, and so, the pair {u,v} is not considered during the run of LCA(u). However, during the for loop of LCA(v), since line 7 of LCA(u) has already run, u.color = black. This means that we will consider the pair {u,v} during the running of LCA(v).

It is not obvious what the ordering $\leq_{LCA}$ is, as it will be implementation dependent. It depends on the order in which child vertices are iterated in the for loop on line 3. That is, it doesn't just depend on the graph structure.

b. We suppose that it is true prior to a given call of LCA, and show that this property is preserved throughout a run of the procedure, increasing the number of disjoint sets by one by the end of the procedure. So, supposing

9

that u has depth d and there are d items in the disjoint set data structure before it runs, it increases to d+1 disjoint sets on line 1. So, by the time we get to line 4, and call LCA of a child of u, there are d+1 disjoint sets, this is exactly the depth of the child. After line 4, there are now $d + 2$ disjoint sets, so, line 5 brings it back down to $d + 1$ disjoint sets for the subsequent times through the loop. After the loop, there are no more changes to the number of disjoint sets, so, the algorithm terminates with d+1 disjoint sets, as desired. Since this holds for any arbitrary run of LCA, it holds for all runs of LCA.

c. Suppose that the pair u and v have the least common ancestor w. Then, when running $LCA(w)$, u will be in the subtree rooted at one of w's children, and v will be in another. WLOG, suppose that the subtree containing u runs first. So, when we are done with running that subtree, all of their ancestor values will point to w and their colors will be black, and their ancestor values will not change until $LCA(w)$ returns. However, we run LCA(v) before $LCA(w)$ returns, so in the for loop on line 8 of LCA(v), we will be considering the pair $\{u, v\}$, since $u.color == BLACK$. Since u.ancestor is still w, that is what will be output, which is the correct answer for their LCA.

d. The time complexity of lines 1 and 2 are just constant. Then, for each child, we have a call to the same procedure, a UNION operation which only takes constant time, and a FIND-SET operation which can take at most amortized inverse Ackerman's time. Since we check each and every thing that is adjacent to u for being black, we are only checking each pair in P at most twice in lines 8-10, among all the runs of LCA. This means that the total runtime is $O(|T|\alpha(|T|) + |P|)$.

Chapter 22

Michelle Bodnar, Andrew Lohr

September 17, 2017

Exercise 22.1-1

Since it seems as though the list for the neighbors of each vertex v is just an undecorated list, to find the length of each would take time $O(out-degree(v))$. So, the total cost will be $\sum_{v \in V} O(outdegree(v)) = O(|E| + |V|)$. Note that the $|V|$ showing up in the asymptotics is necessary, because it still takes a constant amount of time to know that a list is empty. This time could be reduced to $O(|V|)$ if for each list in the adjacency list representation, we just also stored its length.

To compute the in degree of each vertex, we will have to scan through all of the adjacency lists and keep counters for how many times each vertex has appeared. As in the previous case, the time to scan through all of the adjacency lists takes time $O(|E| + |V|)$.

Exercise 22.1-2

The adjacency list representation:

$$1 : 2, 3$$
$$2 : 1, 4, 5$$
$$3 : 1, 6, 7$$
$$4 : 2$$
$$5 : 2$$
$$6 : 3$$
$$7 : 3.$$

The adjacency matrix representation:

$$
\begin{bmatrix}
0 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0
\end{bmatrix}.
$$

Exercise 22.1-3

For the adjacency matrix representation, to compute the graph transpose, we just take the matrix transpose. This means looking along every entry above the diagonal, and swapping it with the entry that occurs below the diagonal. This takes time $O(|V|^2)$.

For the adjacency list representation, we will maintain an initially empty adjacency list representation of the transpose. Then, we scan through every list in the original graph. If we are in the list corresponding to vertex v and see u as an entry in the list, then we add an entry of v to the list in the transpose graph corresponding to vertex u. Since this only requires a scan through all of the lists, it only takes time $O(|E| + |V|)$

Exercise 22.1-4

Create an array A of size $|V|$. For a list in the adjacency list corresponding to vertex v, examine items on the list one by one. If any item is equal to v, remove it. If vertex u appears on the list, examine $A[u]$. If it's not equal to v, set it equal to v. If it's equal to v, remove u from the list. Since we have constant time lookup in the array, the total runtime is $O(V + E)$.

Exercise 22.1-5

From the adjacency matrix representation, if we take the square of the matrix, we are left an edge between all pairs of vertices that are separated by a path of exactly 2, so, to get the desired notion of the square of a graph, we also just have to add in the vertices that are separated by only a single edge in G, that is the entry u, v in the final resulting matrix should be one iff either $G^2[u, v]$ or $G[u, v]$ are one. Taking the square of a matrix can be done with a matrix multiplication, which at the time of writing, can be most efficiently done by the Coppersmith-Windograd algorithm which takes time $O(|V|^{2.3728639})$. Since the other operation for computing the final result only takes time $O(|V|^2)$, the total runtime is $O(|V|^{2.3728639})$.

If we are given an adjacency list representation, we can find the desired resulting graph by first computing the transpose graph G^T from exercise 22.1-3 in $O(|V| + |E|)$ time. Then, our initally empty adjacency list representation of

G^2 will be added to as follows. As we scan though the list of each vertex, say v, and see a entry going to u, then we add u to the list corresponding to v, but also add u to the list of everything on v's list in G^T. This means that we may take as much as $O(|E||V| + |V|)$ time since, we have to spend potentially $|V|$ time as we process each edge.

Exercise 22.1-6

Start by examining position $(1,1)$ in the adjacency matrix. When examining position (i, j), if a 1 is encountered, examine position $(i+1, j)$. If a 0 is encountered, examine position $(i, j + 1)$. Once either i or j is equal to $|V|$, terminate. I claim that if the graph contains a universal sink, then it must be at vertex i. To see this, suppose that vertex k is a universal sink. Since k is a universal sink, row k in the adjacency matrix is all 0's, and column k is all 1's except for position (k, k) which is a 0. Thus, once row k is hit, the algorithm will continue to increment j until $j = |V|$. To be sure that row k is eventually hit, note that once column k is reached, the algorithm will continue to increment i until it reaches k. This algorithm runs in $O(V)$ and checking whether or not i in fact corresponds to a sink is done in $O(V)$. Therefore the entire process takes $O(V)$.

Exercise 22.1-7

We have two cases, one for the diagonal entries and one for the non-diagonal entries.

The entry of $[i, i]$ for some i represents the sum of the in and ourt degrees of the vertex that i corresponds to. To see this, we recall that an entry in a matrix product is the dot product of row i in B and column i in B^T. But, column i in B^T is the same as row i in B. So, we have that the entry is just row i of B dotted with itself, that is

$$\sum_{j=1}^{|E|} b_{ij}^2$$

However, since b_{ij} only takes values in $\{-1, 0, 1\}$, we have that b_{ij}^2 only takes values in $\{0, 1\}$, taking zero iff $b_{i,j}$ is zero. So, the entry is the sum of all nonzero entries in row i of B, Since each edge leaving i is -1 and each edge going to i is 1, we are counting all the edges that either leave or enter i, as we wanted to show.

Now, suppose that our entry is indexed by $[i, j]$ where $i \neq j$. This is the dot product of row i in B with column j in B^T, which is row j in B. So, the entry is equal to

$$\sum_{k=1}^{|E|} b_{i,k} \cdot b_{j,k}$$

Each term in this sum is -1 if k goes between i and j, or 0 if it doesn't. Since we can't have that two different vertices ar both on the same side of an edge,

no terms may ever be 1. So, the entry is just -1 if there is an edge between i and j, and zero otherwise.

Exercise 22.1-8

The expected lookup time is $O(1)$, but in the worst case it could take $O(V)$. If we first sorted vertices in each adjacency list then we could perform a binary search so that the worst case lookup time is $O(\lg V)$, but this has the disadvantage of having a much worse expected lookup time.

Exercise 22.2-1

$vertex$	d	π
1	∞	NIL
2	3	4
3	0	NIL
4	2	5
5	1	3
6	1	3

Exercise 22.2-2

These are the results when we examine adjacent vertices in lexicographic order:

Vertex	d	π
r	4	s
s	3	w
t	1	u
u	0	NIL
v	5	r
w	2	t
x	1	u
y	1	u

Exercise 22.2-3

As mentioned in the errata, the question should state that we are to show that a single bit suffices by removing line 18. To see why it is valid to remove line 18, consider the possible transitions between colors that can occur. In particular, it is impossible for a white vertex to go straight to black. This is because inorder for a vertex to be colored black, it must of been assigned to u on line 11. This means that we have to of enqueued the vertex in the queue at some point. This can only occur on line 17, however, if we are running line 17 on a vertex, we have to of run line 14 on it, giving it the color GRAY. Then, notice that the only testing of colors that is done anywhere is on line 13, in which we test whiteness. Since line 13 doesn't care if a vertex is GRAY or BLACK, and

we only ever assign black to a gray vertex, we don't affect the running of the algorithm at all by removing line 18. Since, once we remove line 18, we ever assign BLACK to a vertex, we can represent the color by a single bit saying whether the vertex is WHITE or GRAY.

Exercise 22.2-4

If we use an adjacency matrix, for each vertex u we dequeue we'll have to examine all vertices v to decide whether or not v is adjacent to u. This makes the for-loop of line 12 $O(V)$. In a connected graph we enqueue every vertex of the graph, so the worst case runtime becomes $O(V^2)$.

Exercise 22.2-5

First, we will show that the vale d assigned to a vertex is independent of the order that entries appear in adjacency lists. To do this, we rely on theorem 22.5 which proves correctness of BFS. In particular, that we have $\nu.d = \delta(s, \nu)$ at the end of the procedure. Since $\delta(s, \nu)$ is a property of the underlying graph, no matter which representation of the graph in terms of adjacency lists that we choose, this value will not change. Since the d values are equal to this thing that doesn't change when we mess with the adjacency lists, it too doesn't change when we mess with the adjacency lists.

Now, to show that π does depend on the ordering of the adjacency lists, we will be using Figure 22.3 as a guide. First, we note that in the given worked out procedure, we have that in the adjacency list for w, t precedes x. Also, in the worked out procedure, we have that $u.\pi = t$. Now, suppose instead that we had x preceding t in the adjacency list of w. Then, it would get added to the queue before t, which means that it would u as it's child before we have a chance to process the children of t. This will mean that $u.\pi = x$ in this different ordering of the adjacency list for w.

Exercise 22.2-6

Let G be the graph shown in the first picture, $G' = (V, E_\pi)$ be the graph shown in the second picture, and 1 be the source vertex. Let's see why E_π can never be produced by running BFS on G. Suppose that 2 precedes 5 in the adjacency list of 1. We'll dequeue 2 before 5, so $3.\pi$ and $4.\pi$ must both equal 2. However, this is not the case. Thus, 5 must have preceded 2 in the adjacency list. However, this implies that $3.\pi$ and $4.\pi$ both equal 5, which again isn't true. Nonetheless, it is easily seen that the unique simple path in G' from 1 to any vertex is a shortest path in G.

5

4   1

2

3

5

4   1

2

3
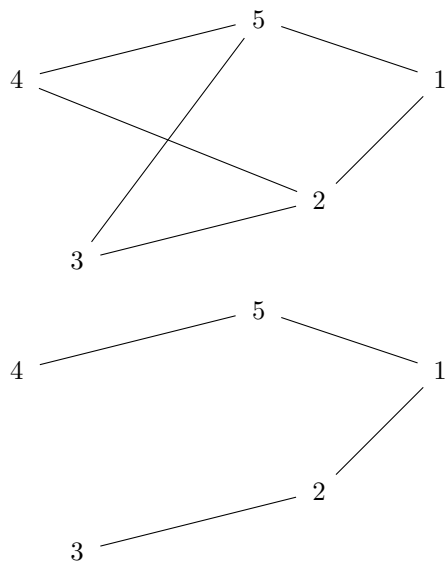
Exercise 22.2-7

This problem is basically just a obfuscated version of two coloring. We will try to color the vertices of this graph of rivalries by two colors, "babyface" and "heel". To have that no two babyfaces and no two heels have a rivalry is the same as saying that the coloring is proper. To two color, we perform a breadth first search of each connected component to get the d values for each vertex. Then, we give all the odd ones one color say "heel", and all the even d values a different color. We know that no other coloring will succeed where this one fails since if we gave any other coloring, we would have that a vertex v has the same color as $v.\pi$ since v and $v.\pi$ must have different parities for their d values. Since we know that there is no better coloring, we just need to check each edge to see if this coloring is valid. If each edge works, it is possible to find a designation, if a single edge fails, then it is not possible. Since the BFS took time $O(n + r)$ and the checking took time $O(r)$, the total runtime is $O(n + r)$.

Exercise 22.2-8

Suppose that a and b are the endpoints of the path in the tree which achieve the diameter, and without loss of generality assume that a and b are the unique pair which do so. Let s be any vertex in T. I claim that the result of a single BFS will return either a or b (or both) as the vertex whose distance from s is greatest. To see this, suppose to the contrary that some other vertex x is shown to be furthest from s. (Note that x cannot be on the path from a to b, otherwise we could extend). Then we have $d(s, a) < d(s, x)$ and $d(s, b) < d(s, x)$. Let c denote the vertex on the path from a to b which minimizes $d(s, c)$. Since the graph is in fact a tree, we must have $d(s, a) = d(s, c) + d(c, a)$ and $d(s, b) = d(s, c) + d(c, b)$.

(If there were another path, we could form a cycle). Using the triangle inequality and inequalities and equalities mentioned above we must have

$$d(a,b) + 2d(s,c) = d(s,c) + d(c,b) + d(s,c) + d(c,a) < d(s,x) + d(s,c) + d(c,b).$$

I claim that $d(x,b) = d(s,c) + d(s,b)$. It not, then by the triangle inequality we must have a strict less-than. In other words, there is some path from x to b which does not go through c. This gives the contradiction, because it implies there is a cycle formed by concatenating these paths. Then we have

$$d(a,b) < d(a,b) + 2d(s,c) < d(x,b).$$

Since it is assumed that $d(a,b)$ is maximal among all pairs, we have a contradiction. Therefore, since trees have $|V| - 1$ edges, we can run BFS a single time in $O(V)$ to obtain one of the vertices which is the endpoint of the longest simple path contained in the graph. Running BFS again will show us where the other one is, so we can solve the diameter problem for trees in $O(V)$.

Exercise 22.2-9

First, the algorithm computes a minimum spanning tree of the graph. Note that this can be done using the procedures of Chapter 23. It can also be done by performing a breadth first search, and restricting to the edges between v and $v.\pi$ for every v. To aide in not double counting edges, fix any ordering $\leq$ on the vertices before hand. Then, we will construct the sequence of steps by calling $MAKE - PATH(s)$ where s was the root used for the BFS.

Algorithm 1 MAKE-PATH(u)

for v adjacent to u in the original graph, but not in the tree such that $u \leq v$ **do**
    go to v and back to u
end for
for v adjacent to u in the tree, but not equal to $u.\pi$ **do**
    go to v
    perform the path proscribed by MAKE-PATH(v)
end for
go to $u.\pi$

Exercise 22.3-1
For directed graphs:

$from\backslash to$	$BLACK$	$GRAY$	$WHITE$
$BLACK$	$Allkinds$	$Back, Cross$	$Back, Cross$
$GRAY$	$Tree, Forward, Cross$	$Tree, Forward, Back$	$Back, Cross$
$WHITE$	$Cross, Tree, Forward$	$Cross, Back$	$allkinds$

For undirected graphs, note that the lower diagonal is defined by the upper diagonal:

$from\backslash to$	$BLACK$	$GRAY$	$WHITE$
$BLACK$	$Allkinds$	$Allkinds$	$Allkinds$
$GRAY$	$-$	$Tree, Forward, Back$	$Allkinds$
$WHITE$	$-$	$-$	$Allkinds$

Exercise 22.3-2

The following table gives the discovery time and finish time for each vetex in the graph.

Vertex	Discovered	Finished
q	1	16
r	17	20
s	2	7
t	8	15
u	18	19
v	3	6
w	4	5
x	9	12
y	13	14
z	10	11

The tree edges are: $(q,s), (s,v), (v,w), (q,t), (t,x), (x,z), (t,y), (r,u)$. The back edges are: $(w,s), (y,q), (z,x)$. The forward edge is: (q,w). The cross edges are: $(u,y), (r,y)$.

Exercise 22.3-3

As pointed out in figure 22.5, the parentheses structure of the DFS of figure 22.4 is $((((())())))(()())$

Exercise 22.3-4

Treat white vertices as 0 and non-white vertices as 1. Since we never check whether or not a vertex is black, deleting line 8 doesn't matter. We need only know whether a vertex is white to get the same results. Once a vertex has been colored grey, the non-white condition is satisfied and there is no need to mark it as black.

Exercise 22.3-5

a. Since we have that $u.d < v.d$, we know that we have first explored u before v. This rules out back edges and rules out the possibility that v is on a tree that has been explored before exploring u's tree. Also, since we return from v before returning from u, we know that it can't be on a tree that was explored after exploring u. So, This rules out it being a cross edge. Leaving us with the only possibilities of being a tree edge or forward edge.

8

To show the other direction, suppose that (u, v) is a tree or forward edge. In that case, since v occurs further down the tree from u, we know that we have to explored u before v, this means that $u.d < v.d$. Also, since we have to of finished v before coming back up the tree, we have that $v.f < u.f$. The last inequality to show is that $v.d < v.f$ which is trivial.

b. By similar reasoning to part a, we have that we must have v being an ancestor of u on the DFS tree. This means that the only type of edge that could go from u to v is a back edge.

To show the other direction, suppose that (u, v) is a back edge. This means that we have that v is above u on the DFS tree. This is the same as the second direction of part a where the roles of u and v are reversed. This means that the inequalities follow for the same reasons.

c. Since we have that $v.f < u.d$, we know that either v is a descendant of u or it comes on some branch that is explored before u. Similarly, since $v.d < u.d$, we either have that u is a descendant of v or it comes on some branch that gets explored before u. Putting these together, we see that it isn't possible for both to be descendants of each other. So, we must have that v comes on a branch before u, So, we have that u is a cross edge.

To See the other direction, suppose that (u, v) is a cross edge. This means that we have explored v at some point before exploring u, otherwise, we would have taken the edge from u to v when exploring u, which would make the edge either a forward edge or a tree edge. Since we explored v first, and the edge is not a back edge, we must of finished exploring v before starting u, so we have the desired inequalities.

Exercise 22.3-6

By Theorem 22.10, every edge of an undirected graph is either a tree edge or a back edge. First suppose that v is first discovered by exploring edge (u, v). Then by definition, (u, v) is a tree edge. Moreover, (u, v) must have been discovered before (v, u) because once (v, u) is explored, v is necessarily discovered. Now suppose that v isn't first discovered by (u, v). Then it must be discovered by (r, v) for some $r \neq u$. If u hasn't yet been discovered then if (u, v) is explored first, it must be a back edge since v is an ancestor of u. If u has been discovered then u is an ancestor of v, so (v, u) is a back edge.

Exercise 22.3-7

See the algorithm DFS-STACK(G). Note that by a similar justification to 22.2-3, we may remove line 8 from the original DFS-VISIT algorithm without changing the final result of the program, that is just working with the colors white and gray.

Exercise 22.3-8

Algorithm 2 DFS-STACK(G)

---

for every $u \in G.V$ **do**
    u.color = WHITE
    $u.\pi$ = NIL
end for
time = 0
S is an empty stack
while there is a white vertex u in G **do**
    S.push(u)
    while S is nonempty **do**
        $v = S.pop$
        time++
        v.d = time
        for all neighbors w of v **do**
            if w.color == WHITE **then**
                w.color = GRAY
                $w.\pi$ = v
                $S.push(w)$
            end if
        end for
        time++
        v.f = time
    end while
end while

Consider a graph with 3 vertices u, v, and w, and with edges $(w, u), (u, w)$, and (w, v). Suppose that DFS first explores w, and that w's adjacency list has u before v. We next discover u. The only adjacent vertex is w, but w is already grey, so u finishes. Since v is not yet a descendant of u and u is finished, v can never be a descendant of u.

Exercise 22.3-9

Consider the Directed graph on the vertices $\{1, 2, 3\}$, and having the edges $(1, 2), (1, 3), (2, 1)$ then there is a path from 2 to 3, however, if we start a DFS at 1 and process 2 before 3, we will have $2.f = 3 < 2 = 2.d$ which provides a counterexample to the given conjecture.

Exercise 22.3-10

We need only update DFS-VISIT. If G is undirected we don't need to make any modifications. We simply note that lines 11 through 16 will never be executed.

Algorithm 3 DFS-VISIT-PRINT(G,u)

1: $time = time + 1$
2: $u.d = time$
3: $u.color = GRAY$
4: **for** each $v \in G.Adj[u]$ **do**
5:      **if** $v.color == white$ **then**
6:          Print "(u, v) is a Tree edge"
7:          $v.\pi = u$
8:          DFS-VISIT-PRINT(G, v)
9:      **else if** $v.color == grey$ **then**
10:          Print "(u, v) is a Back edge"
11:      **else**
12:          **if** $v.d > u.d$ **then**
13:              Print "(u, v) is a Forward edge"
14:          **else**
15:              Print "(u, v) is a Cross edge"
16:          **end if**
17:      **end if**
18: **end for**

Exercise 22.3-11

Suppose that we have a directed graph on the vertices $\{1, 2, 3\}$ and having edges $(1, 2), (2, 3)$ then, 2 has both incoming and outgoing edges. However, if we pick our first root to be 3, that will be in it's own DFS tree. Then, we pick our

second root to be 2, since the only thing it points to has already been marked BLACK, we wont be exploring it. Then, picking the last root to be 1, we don't screw up the fact that 2 is along in a DFS tree despite the fact that it has both an incoming and outgoing edge in G.

Exercise 22.3-12

The modifications work as follows: Each time the if-condition of line 8 is satisfied in DFS-CC, we have a new root of a tree in the forest, so we update its cc label to be a new value of k. In the recursive calls to DFS-VISIT-CC, we always update a descendant's connected component to agree with its ancestor's.

Algorithm 4 DFS-CC(G)

1: **for** each vertex $u \in G.V$ **do**
2:      $u.color = white$
3:      $u.\pi = NIL$
4: **end for**
5: $time = 0$
6: $k = 1$
7: **for** each vertex $u \in G.V$ **do**
8:      **if** $u.color == white$ **then**
9:          $u.cc = k$
10:          $k = k + 1$
11:          DFS-VISIT-CC(G,u)
12:      **end if**
13: **end for**

Algorithm 5 DFS-VISIT-CC(G,u)

1: $time = time + 1$
2: $u.d = time$
3: $u.color = GRAY$
4: **for** each $v \in G.Adj[u]$ **do**
5:      $v.cc = u.cc$
6:      **if** $v.color == white$ **then**
7:          $v.\pi = u$
8:          DFS-VISIT-CC(G, v)
9:      **end if**
10: **end for**
11: $u.color = black$
12: $time = time + 1$
13: $u.f = time$

Exercise 22.3-13

This can be done in time $O(|V||E|)$. To do this, first perform a topological sort of the vertices. Then, we will contain for each vertex a list of it's ancestors with in degree 0. We compute these lists for each vertex in the order starting from the earlier ones topologically. Then, if we ever have a vertex that has the same degree 0 vertex appearing in the lists of two of its immediate parents, we know that the graph is not singly connected. however, if at each step we have that at each step all of the parents have disjoint sets of degree 0 vertices as ancestors, the graph is singly connected. Since, for each vertex, the amount of time required is bounded by the number of vertices times the in degree of the particular vertex, the total runtime is bounded by $O(|V||E|)$.

Exercise 22.4-1

Our start and finish times from performing the DFS are

$label$	d	f
m	1	20
q	2	5
t	3	4
r	6	19
u	7	8
y	9	18
v	10	17
w	11	14
z	12	13
x	15	16
n	21	26
o	22	25
s	23	24
p	27	28

And so, by reading off the entries in decreasing order of finish time, we have the sequence $p, n, o, s, m, r, y, v, x, w, z, u, q, t$.

Exercise 22.4-2

The algorithm works as follows. The attribute $u.paths$ of node u tells the number of simple paths from u to v, where we assume that v is fixed throughout the entire process. To count the number of paths, we can sum the number of paths which leave from each of u's neighbors. Since we have no cycles, we will never risk adding a partially completed number of paths. Moreover, we can never consider the same edge twice among the recursive calls. Therefore, the total number of executions of the for-loop over all recursive calls is $O(V + E)$. Calling SIMPLE-PATHS(s, t) yields the desired result.

Algorithm 6 SIMPLE-PATHS(u,v)

1: **if** $u == v$ **then**
2:     Return 1
3: **else if** $u.paths \neq NIL$ **then**
4:     Return $u.paths$
5: **else**
6:     **for** each $w \in Adj[u]$ **do**
7:        $u.paths = u.paths+$ SIMPLE-PATHS(w,v)
8:     **end for**
9:     Return $u.paths$
10: **end if**

Exercise 22.4-3

We can't just use a depth first search, since that takes time that could be worst case linear in $|E|$. However we will take great inspiration from DFS, and just terminate early if we end up seeing an edge that goes back to a visited vertex. Then, we should only have to spend a constant amount of time processing each vertex. Suppose we have an acyclic graph, then this algorithm is the usual DFS, however, since it is a forest, we have $|E| \leq |V| - 1$ with equality in the case that it is connected. So, in this case, the runtime of $O(|E| + |V|)$ $O(|V|)$. Now, suppose that the procedure stopped early, this is because it found some edge coming from the currently considered vertex that goes to a vertex that has already been considered. Since all of the edges considered up to this point didn't do that, we know that they formed a forest. So, the number of edges considered is at most the number of vertices considered, which is $O(|V|)$. So, the total runtime is $O(|V|)$.

Exercise 22.4-4

This is not true. Consider the graph G consisting of vertices a, b, c, and d. Let the edges be $(a, b), (b, c), (a, d), (d, c)$, and (c, a). Suppose that we start the DFS of TOPOLOGICAL-SORT at vertex c. Assuming that b appears before d in the adjacency list of a, the order, from latest to earliest, of finish times is c, a, d, b. The "bad" edges in this case are (b, c) and (d, c). However, if we had instead ordered them by a, b, d, c then the only bad edges would be (c, a). Thus TOPOLOGICAL-SORT doesn't always minimizes the number of "bad" edges.

Exercise 22.4-5

Consider having a list for each potential in degree that may occur. We will also make a pointer from each vertex to the list that contains it. The initial construction of this can be done in time $O(|V| + |E|)$ because it only requires computing the in degree of each vertex, which can be done in time

$O(|V| + |E|)$ (see problem 22.1-3). Once we have constructed this sequence of lists, we repeatedly extract any element from the list corresponding to having in degree zero. We spit this out as the next element in the topological sort. Then, for each of the children c of this extracted vertex, we remove it from the list that contains it and insert it into the list of in degree one less. Since a deletion and an insertion in a doubly linked list can be done in constant time, and we only have to do this for each child of each vertex, it only has to be done $|E|$ many times. Since at each step, we are outputting some element of in degree zero with respect to all the vertices that hadn't yet been output, we have successfully output a topological sort, and the total runtime is just $O(|E| + |V|)$. We also know that we can always have that there is some element to extract from the list of in degree 0, because otherwise we would have a cycle somewhere in the graph. To see this, just pick any vertex and traverse edges backwards. You can keep doing this indefinitely because no vertex has in degree zero. However, there are only finitely many vertices, so at some point you would need to find a repeat, which would mean that you have a cycle.

If the graph was not acyclic to begin with, then we will have the problem of having an empty list of vertices of in degree zero at some point. That is, if the vertices left lie on a cycle, then none of them will have in degree zero.

Exercise 22.5-1

It can either stay the same or decrease. To see that it is possible to stay the same, just suppose you add some edge to a cycle. To see that it is possible to decrease, suppose that your original graph is on three vertices, and is just a path passing through all of them, and the edge added completes this path to a cycle. To see that it cannot increase, notice that adding an edge cannot remove any path that existed before. So, if u and v are in the same connected component in the original graph, then there are a path from one to the other, in both directions. Adding an edge wont disturb these two paths, so we know that u and v will still be in the same SCC in the graph after adding the edge. Since no components can be split apart, this means that the number of them cannot increase since they form a partition of the set of vertices.

Exercise 22.5-2

The finishing times of each vertex were computed in exercise 22.3-2. The forest consists of 5 trees, each of which is a chain. We'll list the vertices of each tree in order from root to leaf: r, u, $q - y - t$, $x - z$, and $s - w - v$.

Exercise 22.5-3

Professor Bacon's suggestion doesn't work out. As an example, suppose that our graph is on the three vertices $\{1, 2, 3\}$ and consists of the edges $(2, 1), (2, 3), (3, 2)$. Then, we should end up with $\{2, 3\}$ and $\{1\}$ as our SCC's. However, a possible DFS starting at 2 could explore 3 before 1, this would mean that the finish

time of 3 is lower than of 1 and 2. This means that when we first perform the DFS starting at 3. However, a DFS starting at 3 will be able to reach all other vertices. This means that the algorithm would return that the entire graph is a single SCC, even though this is clearly not the case since there is neither a path from 1 to 2 of from 1 to 3.

Exercise 22.5-4

First observe that C is a strongly connected component of G if and only if it is a strongly connected component of G^T. Thus the vertex sets of G^{SCC} and $(G^T)^{SCC}$ are the same, which implies the vertex sets of $((G^T)^{SCC})^T$ and G^{SCC} are the same. It suffices to show that their edge sets are the same. Suppose (v_i, v_j) is an edge in $((G^T)^{SCC})^T$. Then (v_j, v_i) is an edge in $(G^T)^{SCC}$. Thus there exist $x \in C_j$ and $y \in C_i$ such that (x, y) is an edge of G^T, which implies (y, x) is an edge of G. Since components are preserved, this means that (v_i, v_j) is an edge in G^{SCC}. For the opposite implication we simply note that for any graph G we have $(G^T)^T = G$.

Exercise 22.5-5

Given the procedure given in the section, we can compute the set of vertices in each of the strongly connected components. For each vertex, we will give it an entry SCC, so that $v.SCC$ denotes the strongly connected component (vertex in the component graph) that v belongs to. Then, for each edge (u, v) in the original graph, we add an edge from $u.SCC$ to $v.SCC$ if one does not already exist. This whole process only takes a time of $O(|V| + |E|)$. This is because the procedure from this section only takes that much time. Then, from that point, we just need a constant amount of work checking the existence of an edge in the component graph, and adding one if need be.

Exercise 22.5-6

By Exercise 22.5-5 we can compute the component graph in $O(V + E)$ time, and we may as well label each node with its component as we go (see exercise 22.3-12 for the specifics), as well as creating a list for each component which contains the vertices in that component by forming an array A such that $A[i]$ contains a list of the vertices in the ith connected component. Then run DFS again, and for each edge encountered, check whether or not it connects two different components. If it doesn't, delete it. If it does, determine whether it is the first edge connecting them. If not, delete it. This can be done in constant time per edge since we can store the component edge information in a k by k matrix, where k is the number of connected components. The runtime of this is thus $O(V + E)$. Now the only edges we have are a minimal number which connect distinct connected components. The last step is place edges within the connected components in a minimal way. The fewest edges which can be used to create a connected component with n vertices is n, and this is done with a

cycle. For each connected component, let $v_1, v_2, \ldots, v_k$ be the vertices in that component. We find these by using the array A created earlier. Add in the edges $(v_1, v_2), (v_2, v_3), \ldots, (v_k, v_1)$. This is linear in the number of vertices, so the total runtime is $O(V + E)$.

Exercise 22.5-7

First compute the component graph as in 22.5-5. Then, in order to have that every vertex either has a path to or from every other vertex, we need that this component graph also has this property. Since this is acyclic, we can perform a topological sort on it. For this to be the case, we want that there is a single path through this dag that hits every single vertex. This can only happen in the DAG if each vertex has an edge going to the vertex that appears next in the topological ordering. See the algorithm IS-SEMI-CONNECTED(G).

Algorithm 7 IS-SEMI-CONNECTED(G)

Compute the component graph of G, call it G'
Perform a topological sort on G' to get the ordering of its vertices $v_1, v_2, \ldots, v_k$.
for i=1..k-1 **do**
    if there is no edge from v_i to v_{i+1} **then**
        return FALSE
    end if
end for
return TRUE

Problem 22-1

a) 1. If we found a back edge, this means that there are two vertices, one a descendant of the other, but there is already a path from the ancestor to the child that doesn't involve moving up the tree. This is a contradiction since the only children in the bfs tree are those that are a single edge away, which means there cannot be any other paths to that child because that would make it more than a single edge away. To see that there are no forward edges, We do a similar procedure. A forward edge would mean that from a given vertex we notice it has a child that has already been processed, but this cannot happen because all children are only one edge away, and for it to of already been processed, it would need to have gone through some other vertex first.

   2. An edge is placed on the list to be processed if it goes to a vertex that has not yet been considered. This means that the path from that vertex to the root must be at least the distance from the current vertex plus 1. It is also at most that since we can just take the path that consists of going to the current vertex and taking its path to the root.

17

3. We know that a cross edge cannot be going to a depth more than one less, otherwise it would be used as a tree edge when we were processing that earlier element. It also cannot be going to a vertex of depth more than one more, because we wouldn't of already processed a vertex that was that much further away from the root. Since the depths of the vertices in the cross edge cannot be more than one apart, the conclusion follows by possibly interchanging the roles of u and v, which we can do because the edges are unordered.

b) 1. To have a forward edge, we would need to have already processed a vertex using more than one edge, even though there is a path to it using a single edge. Since breadth first search always considers shorter paths first, this is not possible.

2. Suppose that (u, v) is a tree edge. Then, this means that there is a path from the root to v of length $u.d + 1$ by just appending (u, v) on to the path from the root to u. To see that there is no shorter path, we just note that we would of processed v sooner, and so wouldn't currently have a tree edge if there were.

3. To see this, all we need to do is note that there is some path from the root to v of length $u.d+1$ obtained by appending (u, v) to $v.d$. Since there is a path of that length, it serves as an upper bound on the minimum length of all such paths from the root to v.

4. It is trivial that $0 \leq v.d$, since it is impossible to have a path from the root to v of negative length. The more interesting inequality is $v.d \leq u.d$. We know that there is some path from v to u, consisting of tree edges, this is the defining property of (u, v) being a back edge. This means that is $v, v_1, v_2, \ldots, v_k, u$ is this path (it is unique because the tree edges form a tree). Then, we have that $u.d = v_k.d+1 = v_{k-1}.d+2 = \cdots = v_1.d+k = v.d + k + 1$. So, we have that $u.d > v.d$.

   In fact, we just showed that we have the stronger conclusion, that $0 \leq v.d < u.d$.

Problem 22-2

a. First suppose the root r of G_π is an articulation point. Then the removal of r from G would cause the graph to disconnect, so r has at least 2 children in G. If r has only one child v in G_π then it must be the case that there is a path from v to each of r's other children. Since removing r disconnects the graph, there must exist vertices u and w such that the only paths from u to w contain r. To reach r from u, the path must first reach one of r's children. This child is connect to v via a path which doesn't contain r. To reach w, the path must also leave r through one of its children, which is also reachable by v. This implies that there is a path from u to w which doesn't contain r, a contradiction.

Now suppose r has at least two children u and v in G_π. Then there is no path from u to v in G which doesn't go through r, since otherwise u would be an ancestor of v. Thus, removing r disconnects the component containing u and the component containing v, so r is an articulation point.

b. Suppose that v is a nonroot vertex of G_π and that v has a child s such that neither s nor any of s's descendants have back edges to a proper ancestor of v. Let r be an ancestor of v, and remove v from G. Since we are in the undirected case, the only edges in the graph are tree edges or back edges, which means that every edge incident with s takes us to a descendant of s, and no descendants have back edges, so at no point can we move up the tree by taking edges. Therefore r is unreachable from s, so the graph is disconnected and v is an articulation point.

Now suppose that for every child of v there exists a descendant of that child which has a back edge to a proper ancestor of v. Remove v from G. Every subtree of v is a connected component. Within a given subtree, find the vertex which has a back edge to a proper ancestor of v. Since the set T of vertices which aren't descendants of v form a connected component, we have that every subtree of v is connected to T. Thus, the graph remains connected after the deletion of v so v is not an articulation point.

c. Since v is discovered before all of its descendants, the only back edges which could affect $v.low$ are ones which go from a descendant of v to a proper ancestor of v. If we know $u.low$ for every child u of v, then we can compute $v.low$ easily since all the information is coded in its descendants. Thus, we can write the algorithm recursively: If v is a leaf in G_π then $v.low$ is the minimum of $v.d$ and $w.d$ where (v, w) is a back edge. If v is not a leaf, v is the minimum of $v.d$, $w.d$ where w is a back edge, and $u.low$, where u is a child of v. Computing $v.low$ for a vertex is linear in its degree. The sum of the vertices' degrees gives twice the number of edges, so the total runtime is $O(E)$.

d. First apply the algorithm of part (c) in $O(E)$ to compute $v.low$ for all $v \in V$. If $v.low = v.d$ if and only if no descendant of v has a back edge to a proper ancestor of v, if and only if v is not an articulation point. Thus, we need only check $v.low$ versus $v.d$ to decide in constant time whether or not v is an articulation point, so the runtime is $O(E)$.

e. An edge (u, v) lies on a simple cycle if and only if there exists at least one path from u to v which doesn't contain the edge (u, v), if and only if removing (u, v) doesn't disconnect the graph, if and only if (u, v) is not a bridge.

f. A edge (u, v) lies on a simple cycle in an undirected graph if and only if either both of its endpoints are articulation points, or one of its endpoints is an articulation point and the other is a vertex of degree 1. Since we can compute all articulation points in $O(E)$ and we can decide whether or not a vertex has degree 1 in constant time, we can run the algorithm in part d and then decide whether each edge is a bridge in constant time, so we can find all bridges in $O(E)$ time.

g. It is clear that every nonbridge edge is in some biconnected component, so we need to show that if C_1 and C_2 are distinct biconnected components, then they contain no common edges. Suppose to the contrary that (u, v) is in both C_1 and C_2. Let (a, b) be any edge in C_1 and (c, d) be any edge in C_2. Then (a, b) lies on a simple cycle with (u, v), consisting of the path $a, b, p_1, \ldots, p_k, u, v, p_{k+1}, \ldots, p_n, a$. Similarly, (c, d) lies on a simple cycle with (u, v) consisting of the path $c, d, q_1, \ldots, q_m, u, v, q_{m+1}, \ldots, q_l, c$. This means $a, b, p_1, \ldots, p_k, u, q_m, \ldots, q_1, d, c, q_l, \ldots, q_{m+1}, v, p_{k+1}, \ldots, p_n, a$ is a simple cycle containing (a, b) and (c, d), a contradiction. Thus, the biconnected components form a partition.

h. Locate all bridge edges in $O(E)$ time using the algorithm described in part f. Remove each bridge from E. The biconnected components are now simply the edges in the connected components. Assuming this has been done, run the following algorithm, which clearly runs in $O(E)$ where E is the number of edges *originally* in G.

Algorithm 8 BCC(G)

1: **for** each vertex $u \in G.V$ **do**
2:     $u.color = white$
3: **end for**
4: $k = 1$
5: **for** each vertex $u \in G.V$ **do**
6:     **if** $u.color == white$ **then**
7:         $k = k + 1$
8:         VISIT-BCC(G,u, k)
9:     **end if**
10: **end for**

Problem 22-3

a. First, we'll show that it is necessary to have in degree equal out degree for each vertex. Suppose that there was some vertex v for which the two were not equal, suppose wlog that in-degree - out-degree = a ¿ 0. Note that we

20

Algorithm 9 VISIT-BCC(G,u,k)

1: $u.color = GRAY$
2: **for** each $v \in G.Adj[u]$ **do**
3:     $(u, v).bcc = k$
4:     **if** $v.color == white$ **then**
5:         VISIT-BCC(G, v, k)
6:     **end if**
7: **end for**

---

may assume that in degree is greater because otherwise we would just look at the transpose graph in which we traverse the cycle backwards. If v is the start of the cycle as it is listed, just shift the starting and ending vertex to any other one on the cycle. Then, in whatever cycle we take going though v, we must pass through v some number of times, in particular, after we pass through it a times, the number of unused edges coming out of v is zero, however, there are still unused edges goin in that we need to use. This means that there is no hope of using those while still being a tour, becase we would never be able to escape v and get back to the vertex where the tour started.

Now, we show that it is sufficient to have the in degree and out degree equal for every vertex. To do this, we will generalize the problem slightly so that it is more amenable to an inductive approach. That is, we will show that for every graph G that has two vertices v and u so that all the vertices have the same in and out degree except that the indegree is one greater for u and the out degree is one greater for v, then there is an Euler path from v to u. This clearly lines up with the original statement if we pick $u = v$ to be any vertex in the graph. We now perform induction on the number of edges. If there is only a single edge, then taking just that edge is an Euler tour. Then, suppose that we start at v and take any edge coming out of it. Consider the graph that is obtained from removing that edge, it inductively contains an Euler tour that we can just post-pend to the edge that we took to get out of v.

b. To actually get the Euler circuit, we can just arbitrarily walk any way that we want so long as we don't repeat an edge, we will necessarily end up with a valid Euler tour. This is implemented in the following algorithm, EULER-TOUR(G) which takes time $O(|E|)$. It has this runtime because the for loop will get run for every edge, and takes a constant amount of time. Also, the process of initializing each edge's color will take time proportional to the number of edges.

Problem 22-4

Begin by locating the element v of minimal label. We would like to make $u.min = v.label$ for all u such that $u \rightsquigarrow v$. Equivalently, this is the set of ver-

Algorithm 10 EULER-TOUR(G)

---
color all edges white
let (v, u) be any edge
let L be a list containing just v.
while there is some white edge (v, w) coming out of v **do**
    color (v,w) black
    v = w
    append v to L
end while

---

tices u which are reachable from v in G^T. We can implement the algorithm as follows, assuming that $u.min$ is initially set equal to NIL for all vertices $u \in V$, and simply call the algorithm on G^T.

Algorithm 11 REACHABILITY(G)

1: Use counting sort to sort the vertices by label from smallest to largest
2: **for** each vertex $u \in V$ **do**
3:     **if** $u.min == NIL$ **then**
4:         REACHABILITY-VISIT$(u, u.label)$
5:     **end if**
6: **end for**

Algorithm 12 REACHABILITY-VISIT(u, k)

1: $u.min = k$
2: **for** $v \in G.Adj[u]$ **do**
3:     **if** $v.min == NIL$ **then**
4:         REACHABILITY-VISIT(v, k)
5:     **end if**
6: **end for**

Chapter 23

Michelle Bodnar, Andrew Lohr

April 12, 2016

Exercise 23.1-1

Suppose that A is an empty set of edges. Then, make any cut that has (u, v) crossing it. Then, since that edge is of minimal weight, we have that (u, v) is a light edge of that cut, and so it is safe to add. Since we add it, then, once we finish constructing the tree, we have that (u, v) is contained in a minimum spanning tree.

Exercise 23.1-2

Let G be the graph with 4 vertices: u, v, w, z. Let the edges of the graph be $(u, v), (u, w), (w, z)$ with weights 3, 1, and 2 respectively. Suppose A is the set $\{(u, w)\}$. Let $S = A$. Then S clearly respects A. Since G is a tree, its minimum spanning tree is itself, so A is trivially a subset of a minimum spanning tree. Moreover, every edge is safe. In particular, (u, v) is safe but not a light edge for the cut. Therefore Professor Sabatier's conjecture is false.

Exercise 23.1-3

Let T_0 and T_1 be the two trees that are obtained by removing edge (u, v) from a MST. Suppose that V_0 and V_1 are the vertices of T_0 and T_1 respectively. Consider the cut which separates V_0 from V_1. Suppose to a contradiction that there is some edge that has weight less than that of (u, v) in this cut. Then, we could construct a minimum spanning tree of the whole graph by adding that edge to $T_1 \cup T_0$. This would result in a minimum spanning tree that has weight less than the original minimum spanning tree that contained (u, v).

Exercise 23.1-4

Let G be a graph on 3 vertices, each connected to the other 2 by an edge, and such that each edge has weight 1. Since every edge has the same weight, every edge is a light edge for a cut which it spans. However, if we take all edges we get a cycle.

Exercise 23.1-5

Let A be any cut that causes some vertices in the cycle on once side of the cut, and some vertices in the cycle on the other. For any of these cuts, we know that the edge e is not a light edge for this cut. Since all the other cuts wont have the edge e crossing it, we won't have that the edge is light for any of those cuts either. This means that we have that e is not safe.

Exercise 23.1-6

Suppose that for every cut of the graph there is a unique light edge crossing the cut, but that the graph has 2 spanning trees T and T'. Since T and T' are distinct, there must exist edges (u, v) and (x, y) such that (u, v) is in T but not T' and (x, y) is in T' but not T. Let $S = \{u, x\}$. There is a unique light edge which spans this cut. Without loss of generality, suppose that it is not (u, v). Then we can replace (u, v) by this edge in T to obtain a spanning tree of strictly smaller weight, a contradiction. Thus the spanning tree is unique.

For a counter example to the converse, let $G = (V, E)$ where $V = \{x, y, z\}$ and $E = \{(x, y), (y, z), (x, z)\}$ with weights 1, 2, and 1 respectively. The unique minimum spanning tree consists of the two edges of weight 1, however the cut where $S = \{x\}$ doesn't have a unique light edge which crosses it, since both of them have weight 1.

Exercise 23.1-7

First, we show that the subset of edges of minimum total weight that connects all the vertices is a tree. To see this, suppose not, that it had a cycle. This would mean that removing any of the edges in this cycle would mean that the remaining edges would still connect all the vertices, but would have a total weight that's less by the weight of the edge that was removed. This would contradict the minimality of the total weight of the subset of vertices. Since the subset of edges forms a tree, and has minimal total weight, it must also be a minimum spanning tree.

To see that this conclusion is not true if we allow negative edge weights, we provide a construction. Consider the graph K_3 with all edge weights equal to -1. The only minimum weight set of edges that connects the graph has total weight -3, and consists of all the edges. This is clearly not a MST because it is not a tree, which can be easily seen because it has one more edge than a tree on three vertices should have. Any MST of this weighted graph must have weight that is at least -2.

Exercise 23.1-8

Suppose that L' is another sorted list of edge weights of a minimum spanning tree. If $L' \neq L$, there must be a first edge (u, v) in T or T' which is of smaller weight than the corresponding edge (x, y) in the other set. Without

loss of generality, assume (u, v) is in T. Let C be the graph obtained by adding (u, v) to L'. Then we must have introduced a cycle. If there exists an edge on that cycle which is of larger weight than (u, v), we can remove it to obtain a tree C' of weight strictly smaller than the weight of T', contradicting the fact that T' is a minimum spanning tree. Thus, every edge on the cycle must be of lesser or equal weight than (u, v). Suppose that every edge is of strictly smaller weight. Remove (u, v) from T to disconnect it into two components. There must exist some edge besides (u, v) on the cycle which would connect these, and since it has smaller weight we can use that edge instead to create a spanning tree with less weight than T, a contradiction. Thus, some edge on the cycle has the same weight as (u, v). Replace that edge by (u, v). The corresponding lists L and L' remain unchanged since we have swapped out an edge of equal weight, but the number of edges which T and T' have in common has increased by 1. If we continue in this way, eventually they must have every edge in common, contradicting the fact that their edge weights differ somewhere. Therefore all minimum spanning trees have the same sorted list of edge weights.

Exercise 23.1-9

Suppose that there was some cheaper spanning tree than T'. That is, we have that there is some T'' so that $w(T'') < w(T')$. Then, let S be the edges in T but not in T'. We can then construct a minimum spanning tree of G by considering $S \cup T''$. This is a spanning tree since $S \cup T'$ is, and T'' makes all the vertices in V' connected just like T' does. However, we have that $w(S \cup T'') = w(S) + w(T'') < w(S) + w(T') = w(S \cup T') = w(T)$. This means that we just found a spanning tree that has a lower total weight than a minimum spanning tree. This is a contradiction, and so our assumption that there was a spanning tree of V' cheaper than T' must be false.

Exercise 23.1-10

Suppose that T is no longer a minimum spanning tree for G with edge weights given by w'. Let T' be a minimum spanning tree for this graph. Then we have we have $w'(T') < w(T) - k$. Since the edge (x, y) may or may not be in T' we have $w(T') \leq w'(T') + k < w(T)$, contradicting the fact that T was minimal under the weight function w.

Exercise 23.1-11

If we were to add in this newly decreased edge to the given tree, we would be creating a cycle. Then, if we were to remove any one of the edges along this cycle, we would still have a spanning tree. This means that we look at all the weights along this cycle formed by adding in the decreased edge, and remove the edge in the cycle of maximum weight. This does exactly what we want since we could only possibly want to add in the single decreased edge, and then, from there we change the graph back to a tree in the way that makes its total weight

minimized.

Exercise 23.2-1

Suppose that we wanted to pick T as our minimum spanning tree. Then, to obtain this tree with Kruskal's algorithm, we will order the edges first by their weight, but then will resolve ties in edge weights by picking an edge first if it is contained in the minimum spanning tree, and treating all the edges that aren't in T as being slightly larger, even though they have the same actual weight. With this ordering, we will still be finding a tree of the same weight as all the minimum spanning trees $w(T)$. However, since we prioritize the edges in T, we have that we will pick them over any other edges that may be in other minimum spanning trees.

Exercise 23.2-2

At each step of the algorithm we will add an edge from a vertex in the tree created so far to a vertex not in the tree, such that this edge has minimum weight. Thus, it will be useful to know, for each vertex not in the tree, the edge from that vertex to some vertex in the tree of minimal weight. We will store this information in an array A, where $A[u] = (v, w)$ if w is the weight of (u, v) and is minimal among the weights of edges from u to some vertex v in the tree built so far. We'll use $A[u].1$ to access v and $A[u].2$ to access w.

Algorithm 1 PRIM-ADJ(G,w,r)

Initialize A so that every entry is (NIL, ∞)
$T = \{r\}$
for $i = 1$ to V **do**
    if $Adj[r, i] \neq 0$ **then**
        $A[i] = (r, w(r, i))$
    end if
end for
for each $u \in V - T$ **do**
    $k = \min_i A[i].2$
    $T = T \cup \{k\}$
    $k.\pi = A[k].1$
    for $i = 1$ to V **do**
        if $Adj[k, i] \neq 0$ and $Adj[k, i] < A[i].2$ **then**
            $A[i] = (k, Adj[k, i])$
        end if
    end for
end for

Exercise 23.2-3

Prim's algorithm implemented with a Binary heap has runtime $O((V + E) \lg(V))$, which in the sparse case, is just $O(V \lg(V))$. The implementation with Fibonacci heaps is $O(E + V \lg(V)) = O(V + V \lg(V)) = O(V \lg(V))$. So, in the sparse case, the two algorithms have the same asymptotic runtimes.

In the dense case, we have that the binary heap implementation has runtime $O((V + E) \lg(V)) = O((V + V^2) \lg(V)) = O(V^2 \lg(V))$. The Fibonacci heap implementation however has a runtime of $O(E + V \lg(V)) = O(V^2 + V \lg(V)) = O(V^2)$. So, in the dense case, we have that the Fibonacci heap implementation is asymptotically faster.

The Fibonacci heap implementation will be asymptotically faster so long as $E = \omega(V)$. Suppose that we have some function that grows more quickly than linear, say f, and $E = f(V)$. The binary heap implementation will have runtime $O((V + E) \lg(V)) = O((V + f(V)) \lg(V)) = O(f(V) \lg(V))$. However, we have that the runtime of the Fibonacci heap implementation will have runtime $O(E + V \lg(V)) = O(f(V) + V \lg(V))$. This runtime is either $O(f(V))$ or $O(V \lg(V))$ depending on if $f(V)$ grows more or less quickly than $V \lg(V)$ respectively. In either case, we have that the runtime is faster than $O(f(V) \lg(V))$.

Exercise 23.2-4

If the edge weights are integers in the range from 1 to $|V|$, we can make Kruskal's algorithm run in $O(E\alpha(V))$ time by using counting sort to sort the edges by weight in linear time. I would take the same approach if the edge weights were integers bounded by a constant, since the runtime is dominated by the task of deciding whether an edge joins disjoint forests, which is independent of edge weights.

Exercise 23.2-5

If there the edge weights are all in the range $1, \ldots, |V|$, then, we can imagine adding the edges to an array of lists, where the edges of weight i go into the list in index i in the array. Then, to decrease an element, we just remove it from the list currently containing it(constant time) and add it to the list corresponding to its new value(also constant time). To extract the minimum wight edge, we maintain a linked list among all the indices that contain non-empty lists, which can also be maintained with only a constant amount of extra work. Since all of these operations can be done in constant time, we have a total runtime $O(E+V)$.

If the edge weights all lie in some bounded universe, suppose in the range 1 to W. Then, we can just vEB tree structure given in chapter 20 to have the two required operations performed in time $O(\lg(\lg(W)))$, which means that the total runtime could be made $O((V + E) \lg(\lg(W)))$.

Exercise 23.2-6

For input drawn from a uniform distribution I would use bucket sort with

5

Kruskal's algorithm, for expected linear time sorting of edges by weight. This would achieve expected runtime $O(E\alpha(V))$.

Exercise 23.2-7
We first add all the edges to the new vertex. Then, we preform a DFS rooted at that vertex. As we go down, we keep track of the largest weight edge seen so far since each vertex above us in the DFS. We know from exercise 23.3-6 that in a directed graph, we don't need to consider cross or forward edges. Every cycle that we detect will then be formed by a back edge. So, we just remove the edge of greatest weight seen since we were at the vertex that the back edge is going to. Then, we'll keep going until we've removed one less than the degree of the vertex we added many edges. This will end up being linear time since we can reuse part of the DFS that we had already computed before detecting each cycle.

Exercise 23.2-8

Professor Borden is mistaken. Consider the graph with 4 vertices: a, b, c, and d. Let the edges be $(a, b), (b, c), (c, d), (d, a)$ with weights 1, 5, 1, and 5 respectively. Let $V_1 = \{a, d\}$ and $V_2 = \{b, c\}$. Then there is only one edge incident on each of these, so the trees we must take on V_1 and V_2 consist of precisely the edges (a, d) and (b, c), for a total weight of 10. With the addition of the weight 1 edge that connects them, we get weight 11. However, an MST would use the two weight 1 edges and only one of the weight 5 edges, for a total weight of 7.

Problem 23-1

a. To see that the second best minimum spanning tree need not be unique, we consider the following example graph on four vertices. Suppose the vertices are $\{a, b, c, d\}$, and the edge weights are as follows:

	a	b	c	d
a	$-$	1	4	3
b	1	$-$	5	2
c	4	5	$-$	6
d	3	2	6	$-$

Then, the minimum spanning tree has weight 7, but there are two spanning trees of the second best weight, 8.

b. We are trying to show that there is a single edge swap that can demote our minimum spanning tree to a second best minimum spanning tree.

In obtaining the second best minimum spanning tree, there must be some cut of a single vertex away from the rest for which the edge that is added is not light, otherwise, we would find the minimum spanning tree, not the second best minimum spanning tree. Call the edge that is selected for that cut for

the second best minimum spanning tree (x, y). Now, consider the same cut, except look at the edge that was selected when obtaining T, call it (u, v). Then, we have that if consider $T - \{(u, v)\} \cup \{(x, y)\}$, it will be a second best minimum spanning tree. This is because if the second best minimum spanning tree also selected a non-light edge for another cut, it would end up more expensive than all the minimum spanning trees. This means that we need for every cut other than the one that the selected edge was light. This means that the choices all align with what the minimum spanning tree was.

c. We give here a dynamic programming solution. Suppose that we want to find it for (u, v). First, we will identify the vertex x that occurs immediately after u on the simple path from u to v. We will then make $max[u, v]$ equal to the max of $w((u, x))$ and $max[w, v]$.Lastly, we just consider the case that u and v are adjacent, in which case the maximum weight edge is just the single edge between the two. If we can find x in constant time, then we will have the whole dynamic program running in time $O(V^2)$, since that's the size of the table that's being built up.

To find x in constant time, we preprocess the tree. We first pick an arbitrary root. Then, we do the preprocessing for Tarjan's off-line least common ancestors algorithm(See problem 21-3). This takes time just a little more than linear, $O(|V|\alpha(|V|))$. Once we've computed all the least common ancestors, we can just look up that result at some point later in constant time. Then, to find the w that we should pick, we first see if $u = LCA(u, v)$ if it does not, then we just pick the parent of u in the tree. If it does, then we flip the question on its head and try to compute $max[v, u]$, we are guaranteed to not have this situation of $v = LCA(v, u)$ because we know that u is an ancestor of v.

d. We provide here an algorithm that takes time $O(V^2)$ and leave open if there exists a linear time solution, that is a $O(E + V)$ time solution. First, we find a minimum spanning tree in time $O(E + V \lg(V))$, which is in $O(V^2)$. Then, using the algorithm from part c, we find the double array max. Then, we take a running minimum over all pairs of vertices u, v, of the value of $w(u, v) - max[u, v]$. If there is no edge between u and v, we think of the weight being infinite. Then, for the pair that resulted in the minimum value of this difference, we add in that edge and remove from the minimum spanning tree, an edge that is in the path from u to v that has weight max[u,v].

Problem 23-2

a. We'll show that the edges added at each step are safe. Consider an unmarked vertex u. Set $S = \{u\}$ and let A be the set of edges in the tree so far. Then the cut respects A, and the next edge we add is a light edge, so it is safe for A. Thus, every edge in T before we run Prim's algorithm is safe for T. Any edge that Prim's would normally add at this point would have to connect two

of the trees already created, and it would be chosen as minimal. Moreover, we choose exactly one between any two trees. Thus, the fact that we only have the smallest edges available to us is not a problem. The resulting tree must be minimal.

b. We argue by induction on the number of vertices in G. We'll assume that $|V| > 1$, since otherwise MST-REDUCE will encounter an error on line 6 because there is no way to choose v. Let $|V| = 2$. Since G is connected, there must be an edge between u and v, and it is trivially of minimum weight. They are joined, and $|G'.V| = 1 = |V|/2$. Suppose the claim holds for $|V| = n$. Let G be a connected graph on $n + 1$ vertices. Then $G'.V \leq n/2$ prior to the final vertex v being examined in the for-loop of line 4. If v is marked then we're done, and if v isn't marked then we'll connect it to some other vertex, which must be marked since v is the last to be processed. Either way, v can't contribute an additional vertex to $G'.V$, so $|G'.V| \leq n/2 \leq (n+1)/2$.

c. Rather than using the disjoint set structures of chapter 21, we can simply use an array to keep track of which component a vertex is in. Let A be an array of length $|V|$ such that $A[u] = v$ if $v = FIND - SET(u)$. Then FIND-SET(u) can now be replaced with $A[u]$ and UNION(u, v) can be replaced by $A[v] = A[u]$. Since these operations run in constant time, the runtime is $O(E)$.

d. The number of edges in the output is monotonically decreasing, so each call is $O(E)$. Thus, k calls take $O(kE)$ time.

e. The runtime of Prim's algorithm is $O(E + V \lg V)$. Each time we run MST-REDUCE, we cut the number of vertices at least in half. Thus, after k calls, the number of vertices is at most $|V|/2^k$. We need to minimize $E + V/2^k \lg(V/2^k) + kE = E + \frac{V \lg(V)}{2^k} - \frac{Vk}{2^k} + kE$ with respect to k. If we choose $k = \lg \lg V$ then we achieve the overall running time of $O(E \lg \lg V)$ as desired. To see that this value of k minimizes, note that the $\frac{Vk}{2^k}$ term is always less than the kE term since $E \geq V$. As k decreases, the contribution of kE decreases, and the contribution of $\frac{V \lg V}{2^k}$ increases. Thus, we need to find the value of k which makes them approximately equal in the worst case, when $E = V$. To do this, we set $\frac{\lg V}{2^k} = k$. Solving this exactly would involve the Lambert W function, but the nicest elementary function which gets close is $k = \lg \lg V$.

f. We simply set up the inequality $E \lg \lg V < E + V \lg V$ to find that we need $E < \frac{V \lg V}{\lg \lg V - 1} = O\left(\frac{V \lg V}{\lg \lg V}\right)$.

Problem 23-3

a. To see that every minimum spanning tree is also a bottleneck spanning tree. Suppose that T is a minimum spanning tree. Suppose there is some edge in it (u, v) that has a weight that's greater than the weight of the bottleneck

spanning tree. Then, let V_1 be the subset of vertices of V that are reachable from u in T, without going though v. Define V_2 symmetrically. Then, consider the cut that separates V_1 from V_2. The only edge that we could add across this cut is the one of minimum weight, so we know that there are no edge across this cut of weight less than $w(u, v)$. However, we have that there is a bottleneck spanning tree with less than that weight. This is a contradiction because a bottleneck spanning tree, since it is a spanning tree, must have an edge across this cut.

b. To do this, we first process the entire graph, and remove any edges that have weight greater than b. If the remaining graph is selected, we can just arbitrarily select any tree in it, and it will be a bottleneck spanning tree of weight at most b. Testing connectivity of a graph can be done in linear time by running a breadth first search and then making sure that no vertices remain white at the end.

c. Write down all of the edge weights of vertices. Use the algorithm from section 9.3 to find the median of this list of numbers in time $O(E)$. Then, run the procedure from part b with this median value as the one that you are testing for there to be a bottleneck spanning tree with weight at most. Then there are two cases:

First, we could have that there is a bottleneck spanning tree with weight at most this median. Then just throw the edges with weight more than the median, and repeat the procedure on this new graph with half the edges.

Second, we could have that there is no bottleneck spanning tree with at most that weight. Then, we should run the procedure from problem 23-2 to contract all of the edges that have weight at most this median weight. This takes time $O(E \lg(\lg(V)))$ and then we are left solving the problem on a graph that now has half the vertices.

Problem 23-4

a. This does return an MST. To see this, we'll show that we never remove an edge which must be part of a minimum spanning tree. If we remove e, then e cannot be a bridge, which means that e lies on a simple cycle of the graph. Since we remove edges in nonincreasing order, the weight of every edge on the cycle must be less than or equal to that of e. By exercise 23.1-5, there is a minimum spanning tree on G with edge e removed.

To implement this, we begin by sorting the edges in $O(E \lg E)$ time. For each edge we need to check whether or not $T - \{e\}$ is connected, so we'll need to run a DFS. Each one takes $O(V + E)$, so doing this for all edges takes $O(E(V + E))$. This dominates the running time, so the total time is $O(E^2)$.

9

b. This doesn't return an MST. To see this, let G be the graph on 3 vertices a, b, and c. Let the eges be (a, b), (b, c), and (c, a) with weights 3, 2, and 1 respectively. If the algorithm examines the edges in their order listed, it will take the two heaviest edges instead of the two lightest.

An efficient implementation will use disjoint sets to keep track of connected components, as in MST-REDUCE in problem 23-2. Trying to union within the same component will create a cycle. Since we make $|V|$ calls to MAKE-SET and at most $3|E|$ calls to FIND-SET and UNION, the runtime is $O(E\alpha(V))$.

c. This does return an MST. To see this, we simply quote the result from exercise 23.1-5. The only edges we remove are the edges of maximum weight on some cycle, and there always exists a minimum spanning tree which doesn't include these edges. Moreover, if we remove an edge from every cycle then the resulting graph cannot have any cycles, so it must be a tree.

To implement this, we use the approach taken in part (b), except now we also need to find the maximum weight edge on a cycle. For each edge which introduces a cycle we can perform a DFS to find the cycle and max weight edge. Since the tree at that time has at most one cycle, it has at most $|V|$ edges, so we can run DFS in $O(V)$. The runtime is thus $O(EV)$.

Chapter 24

Michelle Bodnar, Andrew Lohr

April 24, 2017

Exercise 24.1-1

If we change our source to z and use the same ordering of edges to decide what to relax, the d values after successive iterations of relaxation are:

s	t	x	y	z
∞	∞	∞	∞	0
2	∞	7	∞	0
2	5	7	9	0
2	5	6	9	0
2	4	6	9	0

The π values are:

s	t	x	y	z
NIL	NIL	NIL	NIL	NIL
z	NIL	z	NIL	NIL
z	x	z	s	NIL
z	x	y	s	NIL
z	x	y	s	NIL

Now, if we change the weight of edge (z, x) to 4 and rerun with s as the source, we have that the d values after successive iterations of relaxation are:

s	t	x	y	z
0	∞	∞	∞	∞
0	6	∞	7	∞
0	6	4	7	2
0	2	4	7	2
0	2	4	7	-2

The π values are:

s	t	x	y	z
NIL	NIL	NIL	NIL	NIL
NIL	s	NIL	s	NIL
NIL	s	y	s	t
NIL	x	y	s	t
NIL	x	y	s	t

Note that these values are exactly the same as in the worked example. The difference that changing this edge will cause is that there is now a negative weight cycle, which will be detected when it considers the edge (z, x) in the for loop on line 5. Since $x.d = 4 > -2 + 4 = z.d + w(z, x)$, it will return false on line 7.

Exercise 24.1-2

Suppose there is a path from s to v. Then there must be a shortest such path of length $\delta(s, v)$. It must have finite length since it contains at most $|V| - 1$ edges and each edge has finite length. By Lemma 24.2, $v.d = \delta(s, v) < \infty$ upon termination. On the other hand, suppose $v.d < \infty$ when BELLMAN-FORD terminates. Recall that $v.d$ is monotonically decreasing throughout the algorithm, and RELAX will update $v.d$ only if $u.d + w(u, v) < v.d$ for some u adjacent to v. Moreover, we update $v.\pi = u$ at this point, so v has an ancestor in the predecessor subgraph. Since this is a tree rooted at s, there must be a path from s to v in this tree. Every edge in the tree is also an edge in G, so there is also a path in G from s to v.

Exercise 24.1-3

Before each iteration of the for loop on line 2, we make a backup copy of the current d values for all the vertices. Then, after each iteration, we check to see if any of the d values changed. If none did, then we immediately terminate the for loop. This clearly works because if one iteration didn't change the values of d, nothing will of changed on later iterations, and so they would all proceed to not change any of the d values.

Exercise 24.1-4

If there is a negative weight cycle on some path from s to v such that u immediately preceeds v on the path, then $v.d$ will strictly decrease every time RELAX(u, v, w) is called. If there is no negative weight cycle, then $v.d$ can never decrease after lines 1 through 4 are executed. Thus, we just update all vertices v which satisfy the if-condition of line 6. In particular, replace line 7 with $v.d = -\infty$.

Exercise 24.1-5

Initially, we will make each vertex have a D value of 0, which corresponds to taking a path of length zero starting at that vertex. Then, we relax along each edge exactly $V - 1$ times. Then, we do one final round of relaxation, which if any thing changes, indicated the existence of a negative weight cycle. The code for this algorithm is identical to that for Bellman ford, except instead of initializing the values to be infinity except at the source which is zero, we initialize every d value to be infinity. We can even recover the path of minimum length for each

vertex by looking at their π values.

Note that this solution assumes that paths of length zero are acceptable. If they are not to your liking then just initialize each vertex to have a d value equal to the minimum weight edge that they have adjacent to them.

Exercise 24.1-6

Begin by calling a slightly modified version of DFS, where we maintain the attribute $v.d$ at each vertex which gives the weight of the unique simple path from s to v in the DFS tree. However, once $v.d$ is set for the first time we will never modify it. It is easy to update DFS to keep track of this without changing its runtime. At first sight of a back edge (u, v), if $v.d > u.d + w(u, v)$ then we must have a negative-weight cycle because $u.d + w(u, v) - v.d$ represents the weight of the cycle which the back edge completes in the DFS tree. To print out the vertices print v, u, $u.\pi$, $u.\pi.\pi$, and so on until v is reached. This has runtime $O(V + E)$.

Exercise 24.2-1

If we run the procedure on the DAG given in figure 24.5, but start at vertex r, we have that the d values after successive iterations of relaxation are:

r	s	t	x	y	z
0	∞	∞	∞	∞	∞
0	5	3	∞	∞	∞
0	5	3	11	∞	∞
0	5	3	10	7	5
0	5	3	10	7	5
0	5	3	10	7	5

The π values are:

r	s	t	x	y	z
NIL	NIL	NIL	NIL	NIL	NIL
NIL	r	r	NIL	NIL	NIL
NIL	r	r	s	NIL	NIL
NIL	r	r	t	t	t
NIL	r	r	t	t	t
NIL	r	r	t	t	t

Exercise 24.2-2

When we reach vertex v, the last vertex in the topological sort, it must have out-degree 0. Otherwise there would be an edge pointing from a later vertex to an earlier vertex in the ordering, a contradiction. Thus, the body of the for-loop of line 4 is never entered for this final vertex, so we may as well not consider it.

Exercise 24.2-3

Introduce two new dummy tasks, with cost zero. The first one having edges going to every task that has no in edges. The second having an edge going to it from every task that has no out edges. Now, construct a new directed graph in which each e gets mapped to a vertex v_e and there is an edge $(v_e, v_{e'})$ with cost w if and only if edge e goes to the same vertex that e' comes from, and that vertex has weight w. Then, every path through this dual graph corresponds to a path through the original graph. So, we just look for the most expensive path in this DAG which has weighted edges using the algorithm from this section.

Exercise 24.2-4

We will compute the total number of paths by counting the number of paths whose start point is at each vertex v, which will be stored in an attribute $v.paths$. Assume that initial we have $v.paths = 0$ for all $v \in V$. Since all vertices adjacent to u occur later in the topological sort and the final vertex has no neighbors, line 4 is well-defined. Topological sort takes $O(V + E)$ and the nested for-loops take $O(V + E)$ so the total runtime is $O(V + E)$.

Algorithm 1 PATHS(G)

1: topologically sort the vertices of G
2: **for** each vertex u, taken in reverse topologically sorted order **do**
3:     **for** each vertex $v \in G.Adj[u]$ **do**
4:         $u.paths = u.paths + 1 + v.paths$
5:     **end for**
6: **end for**

Exercise 24.3-1

We first have s as the source, in this case, the sequence of extractions from the priority queue are: s, t, y,x,z. The d values after each iteration are:

s	t	x	y	z
0	3	∞	5	∞
0	3	9	5	∞
0	3	9	5	11
0	3	9	5	11
0	3	9	5	11

The π values are:

s	t	x	y	z
NIL	s	NIL	NIL	NIL
NIL	s	t	s	NIL
NIL	s	t	s	y
NIL	s	t	s	y
NIL	s	t	s	y

Now, if we repeat the procedure, except having z as the source, we have that the d values are

s	t	x	y	z
3	∞	7	∞	0
3	6	7	8	0
3	6	7	8	0
3	6	7	8	0
3	6	7	8	0

The π values are:

s	t	x	y	z
z	NIL	z	NIL	NIL
z	s	z	s	NIL
z	s	z	s	NIL
z	s	z	s	NIL
z	s	z	s	NIL

Exercise 24.3-2

Consider any graph with a negative cycle. RELAX is called a finite number of times but the distance to any vertex on the cycle is $-\infty$, so DIJKSTRA's algorithm cannot possibly be correct here. The proof of theorem 24.6 doesn't go through because we can no longer guarantee that $\delta(s, y) \leq \delta(s, u)$.

Exercise 24.3-3

It does work correctly to modify the algorithm like that. Once we are at the point of considering the last vertex, we know that it's current d value is at at least as large as the largest of the other vertices. Since none of the edge weights are negative, its d value plus the weight of any edge coming out of it will be at least as large as the d values of all the other vertices. This means that the relaxations that occur will not change any of the d values of any vertices, and so not change their π values.

Exercise 24.3-4

Begin by verifying that proposed shortest paths tree is indeed a tree. Next check that $s.d = 0$. Then, for each vertex in $V \backslash \{s\}$, examine all edges coming

into V. Check that $v.\pi$ is the vertex which minimizes $u.d + w(u, v)$ for all vertices u for which there is an edge (u, v), and that $v.d = v.\pi.d + w(v.\pi, v)$. If this is ever false, return false. Otherwise, return true. This takes $O(V + E)$ time. Now we must check that this correctly checks whether or not the d and π attributes match those of some shortest-paths tree. Suppose that this is not true; ie, that the algorithm terminates without returning false on input which doesn't correspond to correctly computed minimum distances and a minimum spanning tree. Let v be a vertex which is incorrect which minimizes $\min(v.d, \delta(s, v))$. Break ties by choosing the vertex closest to s along the proposed shortest paths tree T. (If there are still ties, choose arbitrarily from among them).

First suppose $v.d \leq \delta(s, v)$. Then we must have $v.\pi.d < v.d$ since $v.\pi.d$ is closer to the root along T, so if equality held we would have selected $v.\pi$ instead of v. By construction, $v.\pi.d$ must be correct. Furthermore, if v had a better neighbor than $v.\pi$ according to the estimated distances, the algorithm would have identified it at terminated. Thus, if $v.\pi$ cannot be the parent of v on a shortest paths tree, there must exist a neighbor u of v such that $u.d$ was computed incorrectly, but such that $\delta(s, u) + w(u, v) < v.d$. However, this implies $\delta(s, u) < v.d$, which means we should have initially chosen u instead of v.

Now suppose $\delta(s, v) < v.d$. If v has no incident edges of weight 0 then each vertex $u \in N(v) = \{u | (u, v) \in E\}$ has been computed correctly. Since $v.d$ must equal $u.d + w(u, v)$ for some such vertex $u \in N(v)$, the algorithm would detect if v were incorrectly computed. Thus, v must have at least one incident edge of weight 0. Let $S = \{v' | (v', v) \in E, w(v', v) = 0\}$. Then no $v' \in S$ can be the parent of v in the proposed shortest paths tree, since it would have been chosen instead of v to begin with. Furthermore, since $v.d$ is incorrect and each $u \in N(v) - S$ has strictly lower estimated distances, the estimated distances of each $u \in N(v) - S$ must be correct. This implies that the only possible parents of v in a true shortest paths tree must lie in S. Let z be one such vertex. Then $\delta(s, z) = \delta(s, v)$, so we may apply the exact same argument as above to z. Continuing in this fashion, follow 0-weight edges back towards the root along some true shortest paths tree. Since $0 \leq \delta(s, v) < v.d$, this process will eventually terminate because the estimated distance is nonzero, so either we'll run out of 0-weight edges or we'll hit the root and the algorithm will correctly identify an overestimated distance.

Exercise 24.3-5

Consider the graph on 5 vertices $\{a, b, c, d, e\}$, and with edges $(a, b), (b, c), (c, d), (a, e), (e, c)$ all with weight 0. Then, we could pull vertices off of the queue in the order a, e, c, b, d. This would mean that we relax (c, d) before (b, c). However, a shortest pat to d is $(a, b), (b, c), (c, d)$. So, we would be relaxing an edge that appears later on this shortest path before an edge that appears earlier.

Exercise 24.3-6

6

We now view the weight of a path as the reliability of a path, and it is computed by taking the product of the reliabilities of the edges on the path. Our algorithm will be similar to that of DIJKSTRA, and have the same runtime, but we now wish to maximize weight, and RELAX will be done inline by checking products instead of sums, and switching the inequality since we want to maximize reliability. Finally, we track that path from y back to x and print the vertices as we go.

Algorithm 2 RELIABILITY(G, r, x, y)

 1: INITIALIZE-SINGLE-SOURCE(G, x)
 2: $S = \emptyset$
 3: $Q = G.V$
 4: **while** $Q \neq \emptyset$ **do**
 5:     $u = $ EXTRACT-MIN(Q)
 6:     $S = S \cup \{u\}$
 7:     **for** each vertex $v \in G.Adj[u]$ **do**
 8:         **if** $v.d < u.d \cdot r(u, v)$ **then**
 9:             $v.d = u.d \cdot r(u, v)$
10:             $v.\pi = u$
11:         **end if**
12:     **end for**
13: **end while**
14: **while** $y \neq x$ **do**
15:     Print y
16:     $y = y.\pi$
17: **end while**
18: Print x

Exercise 24.3-7

Each edge is replaced with a number of edges equal to its weight, and one less than that many vertices. That is, $|V'| = \sum_{(v,u) \in E} w(v, u) - 1$. Similarly, $|E'| = \sum_{(v,u) \in E} w(v, u)$. Since we can bound each of these weights by W, we can say that $|V'| \leq W|E| - |E|$ so there are at most $W|E| - |E| + |V|$ vertices in G'. A breadth first search considers vertices in an order so that u and v satisfy $u.d < v.d$ it considers u before v. Similarly, since each iteration of the while loop in Dijkstra's algorithm considers the vertex with lowest d value in the queue, we will also be considering vertices with smaller d values first. So, the two order of considering vertices coincide.

Exercise 24.3-8

We will modify Dijkstra's algorithm to run on this graph by changing the way the priority queue works, taking advantage of the fact that its members

will have keys in the range $[0, WV] \cup \{\infty\}$, since in the worst case we have to compute the distance from one end of a chain to the other, with $|V|$ vertices each connected by an edge of weight W. In particular, we will create an array A such that $A[i]$ holds a linked list of all vertices whose estimated distance from s is i. We'll also need the attribute $u.list$ for each vertex u, which points to u's spot in the list stored at $A[u.d]$. Since the minimum distance is always increasing, k will be at most VW, so the algorithm spends $O(VW)$ time in the while loop on line 9, over all iterations of the for-loop of line 8. We spend only $O(V + E)$ time executing the for-loop of line 14 because we look through each adjacency list only once. All other operations take $O(1)$ time, so the total runtime of the algorithm is $O(VW) + O(V + E) = O(VW + E)$.

Algorithm 3 MODIFIED-DIJKSTRA(G,w,s)

1: **for** each $v \in G.V$ **do** $v.d = VW + 1$ $v.\pi = NIL$
2: **end for**
3: $s.d = 0$
4: Initialize an array A of length $VW + 2$
5: $A[0].insert(s)$
6: Set $A[VW + 1]$ equal to a linked list containing every vertex except s
7: $k = 0$
8: **for** $i = 1$ to $|V|$ **do**
9:     **while** $A[k] = NIL$ **do**
10:         $k = k + 1$
11:     **end while**
12:     $u = A[k].head$
13:     $A[k].delete(u)$
14:     **for** each vertex $v \in G.Adj[u]$ **do**
15:         **if** $v.d > u.d + w(u, v)$ **then**
16:             $A[v.d].delete(v.list)$
17:             $v.d = u.d + w(u, v)$
18:             $v.\pi = u$
19:             $A[v.d].insert(v)$
20:             $v.list = A[v.d].head$
21:         **end if**
22:     **end for**
23: **end for**

Exercise 24.3-9
We can modify the construction given in the previous exercise to avoid having to do a linear search through the array of lists A. To do this, we can just keep a set of the indices of A that contain a non-empty list. Then, we can just maintain this as we move the vertices between lists, and replace the while loop in the previous algorithm with just getting the minimum element in the set.

One way of implementing this set structure is with a self-balancing binary

search tree. Since the set consists entirely of indices of A which has length W, the size of the tree is at most W. We can find the minimum element, delete an element and insert in element all in time $O(\lg(W))$ in a self balancing binary serach tree.

Another way of doing this, since we know that the set is of integers drawn from the set $\{1, \ldots, W\}$, is by using a vEB tree. This allows the insertion, deletion, and find minimum to run in time $O(\lg(\lg(W)))$.

Since for every edge, we potentially move a vertex from one list to another, we also need to possibly perform a deletion and insertion in the set. Also, at the beginning of the outermost for loop, we need to perform a get min operation, if removing this element would cause the list in that index of A to become empty, we have to delete it from our set of indices. So, we need to perform a set operation for each vertex, and also for each edge. There is only a constant amount of extra work that has to be done for each, so the total runtime is $O((V + E) \lg \lg(W))$ which is also $O((V + E) \lg(W))$.

Exercise 24.3-10

The proof of correctness, Theorem 24.6, goes through exactly as stated in the text. The key fact was that $\delta(s, y) \leq \delta(s, u)$. It is claimed that this holds because there are no negative edge weights, but in fact that is stronger than is needed. This always holds if y occurs on a shortest path from s to u and $y \neq s$ because all edges on the path from y to u have nonnegative weight. If any had negative weight, this would imply that we had "gone back" to an edge incident with s, which implies that a cycle is involved in the path, which would only be the case if it were a negative-weight cycle. However, these are still forbidden.

Exercise 24.4-1

Our vertices of the constraint graph will be $\{v_0, v_1, v_2, v_3, v_4, v_5, v_6\}$. The edges will be $(v_0, v_1), (v_0, v_2), (v_0, v_3), (v_0, v_4), (v_0, v_5), (v_0, v_6), (v_2, v_1), (v_4, v_1), (v_3, v_2), (v_5, v_2), (v_6, v_2), (v_6, v_3),$ with edge weights $0, 0, 0, 0, 0, 0, 1, -4, 2, 7, 5, 10, 2, -1, 3, -8$ respectively. Then, computing $(\delta(v_0, v_1), \delta(v_0, v_2), \delta(v_0, v_3), \delta(v_0, v_4), \delta(v_0, v_5), \delta(v_0, v_6))$, we get $(-5, -3, 0, -1, -6, -8)$ which is a feasible solution by Theorem 24.9.

Exercise 24.4-2

There is no feasible solution because the constraint graph contains a negative-weight cycle: $(v_1, v_4, v_2, v_3, v_5, v_1)$ has weight -1.

Exercise 24.4-3

No, it cannot be positive. This is because for every vertex $v \neq v_0$, there is an edge (v_0, v) with weight zero. So, there is some path from the new vertex to every other of weight zero. Since $\delta(v_0, v)$ is a minimum weight of all paths, it cannot be greater than the weight of this weight zero path that consists of a

9

single edge.

Exercise 24.4-4

To solve the single source shortest path problem we must have that for each edge (v_i, v_j), $\delta(s, v_j) \leq \delta(s, v_i) + w(v_i, v_j)$, and $\delta(s, s) = 0$. We we will use these as our inequalities.

Exercise 24.4-5

We can follow the advice of problem 14.4-7 and solve the system of constraints on a modified constraint graph in which there is no new vertex v_0. This is simply done by initializing all of the vertices to have a d value of 0 before running the iterated relaxations of Bellman Ford. Since we don't add a new vertex and the n edges going from it to to vertex corresponding to each variable, we are just running Bellman Ford on a graph with n vertices and m edges, and so it will have a runtime of $O(mn)$.

Exercise 24.4-6

To obtain the equality constraint $x_i = x_j + b_k$ we simply use the inequalities $x_i - x_j \leq b_k$ and $x_j - x_i \leq -b_k$, then solve the problem as usual.

Exercise 24.4-7

We could avoid adding in the additional vertex by instead initializing the d value for each vertex to be 0, and then running the bellman ford algorithm. These modified initial conditions are what would result from looking at the vertex v_0 and relaxing all of the edges coming off of it. After we would of processed the edges coming off of v_0, we can never consider it again because there are no edges going to it. So, we can just initialize the vertices to what they would be after relaxing the edges coming off of v_0.

Exercise 24.4-8

Bellman-Ford correctly solves the system of difference constraints so $Ax \leq b$ is always satisfied. We also have that $x_i = \delta(v_0, v_i) \leq w(v_0, v_i) = 0$ so $x_i \leq 0$ for all i. To show that $\sum x_i$ is maximized, we'll show that for any feasible solution $(y_1, y_2, \ldots, y_n)$ which satisfies the constraints we have $y_i \leq \delta(v_0, v_i) = x_i$. Let $v_0, v_{i_1}, \ldots, v_{i_k}$ be a shortest path from v_0 to v_i in the constraint graph. Then we must have the constraints $y_{i_2} - y_{i_1} \leq w(v_{i_1}, v_{i_2}), \ldots, y_{i_k} - y_{i_{k-1}} \leq w(v_{i_{k-1}}, v_{i_k})$. Summing these up we have

$$y_i \leq y_i - y_1 \leq \sum_{m=2}^{k} w(v_{i_m}, v_{i_{m-1}}) = \delta(v_0, v_i) = x_i.$$

Exercise 24.4-9

We can see that the Bellman-Ford algorithm run on the graph whose construction is described in this section causes the quantity $\max\{x_i\} - \min\{x_i\}$ to be minimized. We know that the largest value assigned to any of the vertices in the constraint graph is a 0. It is clear that it won't be greater than zero, since just the single edge path to each of the vertices has cost zero. We also know that we cannot have every vertex having a shortest path with negative weight. To see this, notice that this would mean that the pointer for each vertex has it's p value going to some other vertex that is not the source. This means that if we follow the procedure for reconstructing the shortest path for any of the vertices, we have that it can never get back to the source, a contradiction to the fact that it is a shortest path from the source to that vertex.

Next, we note that when we run Bellman-Ford, we are maximizing $\min\{x_i\}$. The shortest distance in the constraint graphs is the bare minimum of what is required in order to have all the constraints satisfied, if we were to increase any of the values we would be violating a constraint.

This could be in handy when scheduling construction jobs because the quantity $\max\{x_i\} - \min\{x_i\}$ is equal to the difference in time between the last task and the first task. Therefore, it means that minimizing it would mean that the total time that all the jobs takes is also minimized. And, most people want the entire process of construction to take as short of a time as possible.

Exercise 24.4-10

Consider introducing the dummy variable x. Let $y_i = x_i + x$. Then $(y_1, \ldots, y_n)$ is a solution to a system of difference constraints if and only if $(x_1, \ldots, x_n)$ is. Moreover, we have $x_i \le b_k$ if and only if $y_i - x \le b_k$ and $x_i \ge b_k$ if and only if $y_i - x \ge b_k$. Finally, $x_i - x_j \le b$ if and only if $y_i - y_j \le b$. Thus, we construct our constraint graph as follows: Let the vertices be $v_0, v_1, v_2, \ldots, v_n, v$. Draw the usual edges among the v_i's, and weight every edge from v_0 to another vertex by 0. Then for each constraint of the form $x_i \le b_k$, create edge (x, y_i) with weight b_k. For every constraint of the form $x_i \ge b_k$, create edge (y_i, x) with weight $-b_k$. Now use Bellman-Ford to solve the problem as usual. Take whatever weight is assigned to vertex x, and subtract it from the weights of every other vertex to obtain the desired solution.

Exercise 24.4-11

To do this, just take the floor of (largest integer that is less than or equal to) each of the b values and solve the resulting integer difference problem. These modified constraints will be admitting exactly the same set of assignments since we required that the solution have integer values assigned to the variables. This is because since the variables are integers, all of their differences will also be integers. For an integer to be less than or equal to a real number, it is necessary and sufficient for it to be less than or equal to the floor of that real number.

Exercise 24.4-12

To solve the problem of $Ax \leq b$ where the elements of b are real-valued we carry out the same procedure as before, running Bellman-Ford, but allowing our edge weights to be real-valued. To impose the integer condition on the x_i's, we modify the RELAX procedure. Suppose we call RELAX(v_i, v_j, w) where v_j is required to be integral valued. If $v_j.d > \lfloor v_i.d + w(v_i, v_j) \rfloor$, set $v_j.d = \lfloor v_i.d + w(v_i, v_j) \rfloor$. This guarantees that the condition that $v_j.d - v_i.d \leq w(v_i, v_j)$ as desired. It also ensures that v_j is integer valued. Since the triangle inequality still holds, $x = (v_1.d, v_2.d, \ldots, v_n.d)$ is a feasible solution for the system, provided that G contains no negative weight cycles.

Exercise 24.5-1

Since the induced shortest path trees on $\{s, t, y\}$ and on $\{t, x, y, z\}$ are independent and have to possible configurations each, there are four total arising from that. So, we have the two not shown in the figure are the one consisting of the edges $\{(s, t), (s, y), (y, x), (x, z)\}$ and the one consisting of the edges $\{(s, t), (t, y), (t, x), (y, z)\}$.

Exercise 24.5-2

Let G have 3 vertices s, x, and y. Let the edges be $(s, x), (s, y), (x, y)$ with weights 1, 1, and 0 respectively. There are 3 possible trees on these vertices rooted at s, and each is a shortest paths tree which gives $\delta(s, x) = \delta(s, y) = 1$.

Exercise 24.5-3

To modify Lemma 24.10 to allow for possible shortest path weights of ∞ and $-\infty$, we need to define our addition as $\infty + c = \infty$, and $-\infty + c = -\infty$. This will make the statement behave correctly, that is, we can take the shortest path from s to u and tack on the edge (u, v) to the end. That is, if there is a negative weight cycle on your way to u and there is an edge from u to v, there is a negative weight cycle on our way to v. Similarly, if we cannot reach v and there is an edge from u to v, we cannot reach u.

Exercise 24.5-4

Suppose u is the vertex which first caused $s.\pi$ to be set to a non-NIL value. Then we must have $0 = s.d > u.d + w(u, s)$. Let p be the path from s to u in the shortest paths tree so far, and C be the cycle obtained by following that path from s to u, then taking the edge (u, s). Then we have $w(C) = w(p) + w(u, s) = u.d + w(u, s) < 0$, so we have a negative-weight cycle.

Exercise 24.5-5

Suppose that we have a grap hon three vertices $\{s, u, v\}$ and containing edges $(s, u), (s, v), (u, v), (v, u)$ all with weight 0. Then, there is a shortest path from s to v of s, u, v and a shortest path from s to u of s, v, u. Based off of these, we could set $v.pi = u$ and $u.\pi = v$. This then means that there is a cycle consisting of u, v in G_π.

Exercise 24.5-6

We will prove this by induction on the number of relaxations performed. For the base-case, we have just called INITIALIZE-SINGLE-SOURCE(G, s). The only vertex in V_π is s, and there is trivially a path from s to itself. Now suppose that after any sequence of n relaxations, for every vertex $v \in V_\pi$ there exists a path from s to v in G_π. Consider the $(n + 1)^{st}$ relaxation. Suppose it is such that $v.d > u.d + w(u, v)$. When we relax v, we update $v.\pi = u.\pi$. By the induction hypothesis, there was a path from s to u in G_π. Now v is in V_π, and the path from s to u, followed by the edge $(u, v) = (v.\pi, v)$ is a path from s to v in G_π, so the claim holds.

Exercise 24.5-7

We know by 24.16 that a G_π forms a tree after a sequence of relaxation steps. Suppose that T is the tree formed after performing all the relaxation steps of the Bellman Ford algorithm. While finding this tree would take many more than $V - 1$ relaxations, we just want to say that there is some sequence of relaxations that gets us our answer quickly, not necessarily proscribe what those relaxations are. So, our sequence of relaxations will be all the edges of T in an order so that we never relax an edge that is below an unrelaxed edge in the tree(a topological ordering). This guarantees that G_π will be the same as was obtained through the slow, proven correct, Bellman-Ford algorithm. Since any tree on V vertices has $V - 1$ edges, we are only relaxing $V - 1$ edges.

Exercise 24.5-8

Since the negative-weight cycle is reachable from s, let v be the first vertex on the cycle reachable from s (in terms of number of edges required to reach v) and $s = v_0, v_1, \ldots, v_k = v$ be a simple path from s to v. Start by performing the relaxations to v. Since the path is simple, every vertex on this path is encountered for the first time, so its shortest path estimate will always decrease from infinity. Next, follow the path around from v back to v. Since v was the first vertex reached on the cycle, every other vertex will have shortest-path estimate set to ∞ until it is relaxed, so we will change these for every relaxation around the cycle. We now create the infinite sequence of relaxations by continuing to relax vertices around the cycle indefinitely. To see why this always causes the shortest-path estimate to change, suppose we have just reached vertex x_i, and

the shortest-path estimates have been changed for every prior relaxation. Let $x_1, x_2, \ldots, x_n$ be the vertices on the cycle. Then we have $x_{i-1}.d + w(x_{i-1}, x) = x_{i-2}.d + w(x_{i-2}, x_{i-1}) + w(x_{i-1}, w_i) = \ldots = x_i.d + \sum_{j=1}^{n} w(x_j) < w.d$ since the cycle has negative weight. Thus, we must update the shortest-path estimate of x_i.

Problem 24-1

a. Since in G_f edges only go from vertices with smaller index to vertices with greater index, there is no way that we could pick a vertex, and keep increasing it's index, and get back to having the index equal to what we started with. This means that G_f is acyclic. Similarly, there is no way to pick an index, keep decreasing it, and get back to the same vertex index. By these definitions, since G_f only has vertices going from lower indices to higher indices, $(v_1, \ldots, v_{|V|})$ is a topological ordering of the vertices. Similarly, for G_b, $(v_{|V|}, \ldots, v_1)$ is a topological ordering of the vertices.

b. Suppose that we are trying to find the shortest path from s to v. Then, list out the vertices of this shortest path $v_{k_1}, v_{k_2}, \ldots, v_{k_m}$. Then, we have that the number of times that the sequence $\{k_i\}_i$ goes from increasing to decreasing or from decreasing to increasing is the number of passes over the edges that are necessary to notice this path. This is because any increasing sequence of vertices will be captured in a pass through E_f and any decreasing sequence will be captured in a pass through E_b. Any sequence of integers of length $|V|$ can only change direction at most $\lfloor |V|/2 \rfloor$ times. However, we need to add one more in to account for the case that the source appears later in the ordering of the vertices than v_{k_2}, as it is in a sense initially expecting increasing vertex indices, as it runs through E_f before E_b.

c. It does not improve the asymptotic runtime of Bellman ford, it just drops the runtime from having a leading coefficient of 1 to a leading coefficient of $\frac{1}{2}$. Both in the original and in the modified version, the runtime is $O(EV)$.

Problem 24-2

1. Suppose that box $x = (x_1, \ldots, x_d)$ nests with box $y = (y_1, \ldots, y_d)$ and box y nests with box $z = (z_1, \ldots, z_d)$. Then there exist permutations π and σ such that $x_{\pi(1)} < y_1, \ldots, x_{\pi(d)} < y_d$ and $y_{\sigma(1)} < z_1, \ldots, y_{\sigma(d)} < z_d$. This implies $x_{\pi(\sigma(1))} < z_1, \ldots, x_{\pi(\sigma(d))} < z_d$, so x nests with z and the nesting relation is transitive.

2. Box x nests inside box y if and only if the increasing sequence of dimensions of x is component-wise strictly less than the increasing sequence of dimensions of y. Thus, it will suffice to sort both sequences of dimensions and compare them. Sorting both length d sequences is done in $O(d \lg d)$,

and comparing their elements is done in $O(d)$, so the total time is $O(d \lg d)$.

3. We will create a nesting-graph G with vertices $B_1, \ldots, B_n$ as follows. For each pair of boxes B_i, B_j, we decide if one nests inside the other. If B_i nests in B_j, draw an arrow from B_i to B_j. If B_j nests in B_i, draw an arrow from B_j to B_i. If neither nests, draw no arrow. To determine the arrows efficiently, after sorting each list of dimensions in $O(nd \lg d)$ we can sort all boxes' sorted dimensions lexicographically in $O(dn \lg n)$ using radix sort. By transitivity, it will suffice to test adjacent nesting relations. Thus, the total time to build this graph is $O(nd \max \lg d, \lg n)$. Next, we need to find the longest chain in the graph.

Problem 24-3

a. To do this we take the negative of the natural log (or any other base will also work) of all the values c_i that are on the edges between the currencies. Then, we detect the presence or absence of a negative weight cycle by applying Bellman Ford. To see that the existence of an arbitrage situation is equivalent to there being a negative weight cycle in the original graph, consider the following sequence of steps:

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdot \cdots \cdot R[i_k, i_1] > 1$$
$$\ln(R[i_1, i_2]) + \ln(R[i_2, i_3]) + \cdots + \ln(R[i_k, i_1]) > 0$$
$$-\ln(R[i_1, i_2]) - \ln(R[i_2, i_3]) - \cdots - \ln(R[i_k, i_1]) < 0$$

b. To do this, we first perform the same modification of all the edge weights as done in part a of this problem. Then, we wish to detect the negative weight cycle. To do this, we relax all the edges $|V| - 1$ many times, as in Bellman-Ford algorithm. Then, we record all of the d values of the vertices. Then, we relax all the edges $|V|$ more times. Then, we check to see which vertices had their d value decrease since we recorded them. All of these vertices must lie on some (possibly disjoint) set of negative weight cycles. Call S this set of vertices. To find one of these cycles in particular, we can pick any vertex in S and greedily keep picking any vertex that it has an edge to that is also in S. Then, we just keep an eye out for a repeat. This finds us our cycle. We know that we will never get to a dead end in this process because the set S consists of vertices that are in some union of cycles, and so every vertex has out degree at least 1.

Problem 24-4

15

a. We can do this in $O(E)$ by the algorithm described in exercise 24.3-8 since our "priority queue" takes on only integer values and is bounded in size by E.

b. We can do this in $O(E)$ by the algorithm described in exercise 24.3-8 since w takes values in $\{0, 1\}$ and $V = O(E)$.

c. If the ith digit, read from left to right, of $w(u, v)$ is 0, then $w_i(u, v) = 2w_{i-1}(u, v)$. If it is a 1, then $w_i(u, v) = 2w_{i-1}(u, v) + 1$. Now let $s = v_0, v_1, \ldots, v_n = v$ be a shortest path from s to v under w_i. Note that any shortest path under w_i is necessarily also a shortest path under w_{i-1}. Then we have

$$\delta_i(s, v) = \sum_{m=1}^{n} w_i(v_{m-1}, v_m)$$
$$\leq \sum_{m=1}^{n} [2w_{i-1}(u, v) + 1]$$
$$\leq 2 \sum_{m=1}^{n} w_{i-1}(u, v) + n$$
$$\leq 2\delta_{i-1}(s, v) + |V| - 1.$$

On the other hand, we also have

$$\delta_i(s, v) = \sum_{m=1}^{n} w_i(v_{m-1}, v_m)$$
$$\geq \sum_{m=1}^{n} 2w_{i-1}(v_{m-1}, v_m)$$
$$\geq 2\delta_{i-1}(s, v)$$

d. Note that every quantity in the definition of $\hat{w}_i$ is an integer, so $\hat{w}_i$ is clearly an integer. Since $w_i(u, v) \geq 2w_{i-1}(u, v)$, it will suffice to show that $w_{i-1}(u, v) + \delta_{i-1}(s, u) \geq \delta_{i-1}(s, v)$ to prove nonnegativity. This follows immediately from the triangle inequality.

e. First note that $s = v_0, v_1, \ldots, v_n = v$ is a shortest path from s to v with respect to $\hat{w}$ if and only if it is a shortest path with respect to w. Then we

16

have

$$\hat{\delta}_i(s, v) = \sum_{m=1}^{n} w_i(v_{m-1}, v_m) + 2\delta_{i-1}(s, v_{m-1}) - 2\delta_{i-1}(s, v_m)$$

$$= \sum_{m=1}^{n} w_i(v_{m-1}, v_m) - 2\delta_{i-1}(s, v_n)$$

$$= \delta_i(s, v) - 2\delta_{i-1}(s, v)$$

f. By part a we can compute $\hat{\delta}_i(s, v)$ for all $v \in V$ in $O(E)$ time. If we have already computed δ_{i-1} then we can compute δ_i in $O(E)$ time. Since we can compute δ_1 in $O(E)$ by part b, we can compute δ_i from scratch in $O(iE)$ time. Thus, we can compute $\delta = \delta_k$ in $O(Ek) = O(E \lg W)$ time.

Problem 24-5

a. If $\mu^* = 0$, then we have that the lowest that $\frac{1}{k} \sum_{i=1}^{k} w(e_i)$ can be is zero. This means that the lowest $\sum_{i=1}^{k} w(e_i)$ can be is 0. This means that no cycle can have negative weight. Also, we know that for any path from s to v, we can make it simple by removing any cycles that occur. This means that it had a weight equal to some path that has at most $n-1$ edges in it. Since we take the minimum over all possible number of edges, we have the minimum over all paths.

b. To show that

$$\max_{0 \le k \le n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} \ge 0$$

we need to show that

$$\max_{0 \le k \le n-1} \delta_n(s, v) - \delta_k(s, v) \ge 0$$

Since we have that $\mu^* = 0$, there aren't any negative weight cycles. This means that we can't have the minimum cost of a path decrease as we increase the possible length of the path past $n - 1$. This means that there will be a path that at least ties for cheapest when we restrict to the path being less than length n. Note that there may also be cheapest path of longer length since we necessarily do have zero cost cycles. However, this isn't guaranteed since the zero cost cycle may not lie along a cheapest path from s to v.

c. Since the total cost of the cycle is 0, and one part of it has cost x, in order to balance that out, the weight of the rest of the cycle has to be $-x$. So, suppose we have some shortest length path from s to u, then, we could traverse the path from u to v along the cycle to get a path from s to u that has length $\delta(s, u) + x$. This gets us that $\delta(s, v) \le \delta(s, u) + x$. To see the converse

17

inequality, suppose that we have some shortest length path from s to v. Then, we can traverse the cycle going from v to u. We already said that this part of the cycle had total cost $-x$. This gets us that $\delta(s, u) \leq \delta(s, v) - x$. Or, rearranging, we have $\delta(s, u) + x \leq \delta(s, v)$. Since we have inequalities both ways, we must have equality.

d. To see this, we find a vertex v and natural number $k \leq n-1$ so that $\delta_n(s, v) - \delta_k(s, v) = 0$. To do this, we will first take any shortest length, smallest number of edges path from s to any vertex on the cycle. Then, we will just keep on walking around the cycle until we've walked along n edges. Whatever vertex we end up on at that point will be our v. Since we did not change the d value of v after looking at length n paths, by part a, we know that there was some length of this path, say k, which had the same cost. That is, we have $\delta_n(s, v) = \delta_k(s, v)$.

e. This is an immediate result of the previous problem and part b. part b says that for all v the inequality holds, so, we have

$$\min_{v \in V} \max_{0 \leq k \leq n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} \geq 0$$

The previous part says that there is some v on each minimum weight cycle so that

$$\max_{0 \leq k \leq n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} = 0$$

which means that

$$\min_{v \in V} \max_{0 \leq k \leq n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} \leq 0$$

Putting the two inequalities together, we have the desired equality.

f. if we add t to the weight of each edge, the mean weight of any cycle becomes $\mu(c) = \frac{1}{k} \sum_{i=1}^{k} (w(e_i) + t) = \frac{1}{k} \left(\sum_i^k w(e_i) \right) + \frac{kt}{k} = \frac{1}{k} \left(\sum_i^k w(e_i) \right) + t$. This is the original, unmodified mean weight cycle, plus t. Since this is how the mean weight of every cycle is changed, the lowest mean weight cycle stays the lowest mean weight cycle. This means that μ^* will increase by t. Suppose that we first compute μ^*. Then, we subtract from every edge weight the value μ^*. This will make the new μ^* equal zero, which by part e means that $\min_{v \in V} \max_{0 \leq k \leq n-1} \frac{\delta_n(s,v) - \delta_k(s,v)}{n-k} = 0$. Since they are both equal to zero, they are both equal to each other.

g. By the previous part, it suffices to compute the expression on the previous line. We will start by creating a table that lists $\delta_k(s, v)$ for every $k \in \{1, \ldots, n\}$ and $v \in V$. This can be done in time $O(V(E + V))$ by creating a $|V|$ by $|V|$ table, where the kth row and vth column represent $\delta_k(s, v)$ when wanting to compute a particular entry, we need look at a number of entries in the previous row equal to the in degree of the vertex we

want to compute. So, summing over the computation required for each row, we need $O(E + V)$. Note that this total runtime can be bumped down to $O(VE)$ by not including in the table any isolated vertices, this will ensure that $E \in \Omega(V)$ So, $O(V(E + V))$ becomes $O(VE)$. Once we have this table of values computed, it is simple to just replace each row with the last row minus what it was, and divide each entry by $n - k$, then, find the min column in each row, and take the max of those numbers.

Problem 24-6

We'll use the Bellman-Ford algorithm, but with a careful choice of the order in which we relax the edges in order to perform a smaller number of RELAX operations. In any bitonic path there can be at most two distinct increasing sequences of edge weights, and similarly at most two distinct decreasing sequences of edge weights. Thus, by the path-relaxation property, if we relax the edges in order of increasing weight then decreasing weight three times (for a total of six times relaxing every edge) the we are guaranteed that $v.d$ will equal $\delta(s, v)$ for all $v \in V$. Sorting the edges takes $O(E \lg E)$. We relax every edge 6 times, taking $O(E)$. Thus the total runtime is $O(E \lg E) + O(E) = O(E \lg E)$, which is asymptotically faster than the usual $O(VE)$ runtime of Bellman-Ford.

Chapter 25

Michelle Bodnar, Andrew Lohr

April 20, 2016

Exercise 25.1-1

First, the slow way:

$$
L^{(1)} = \begin{pmatrix}
0 & \infty & \infty & \infty & -1 & \infty \\
1 & 0 & \infty & 2 & \infty & \infty \\
\infty & 2 & 0 & \infty & \infty & -8 \\
-4 & \infty & \infty & 0 & 3 & \infty \\
\infty & 7 & \infty & \infty & 0 & \infty \\
\infty & 5 & 10 & \infty & \infty & 0
\end{pmatrix}
$$

$$
L^{(2)} = \begin{pmatrix}
0 & 6 & \infty & \infty & -1 & \infty \\
-2 & 0 & \infty & 2 & 0 & \infty \\
3 & -3 & 0 & 4 & \infty & -8 \\
-4 & 10 & \infty & 0 & -5 & \infty \\
8 & 7 & \infty & 9 & 0 & \infty \\
6 & 5 & 10 & 7 & \infty & 0
\end{pmatrix}
$$

$$
L^{(3)} = \begin{pmatrix}
0 & 6 & \infty & 8 & -1 & \infty \\
-2 & 0 & \infty & 2 & -3 & \infty \\
-2 & -3 & 0 & -1 & 2 & -8 \\
-4 & 2 & \infty & 0 & -5 & \infty \\
5 & 7 & \infty & 9 & 0 & \infty \\
3 & 5 & 10 & 7 & 5 & 0
\end{pmatrix}
$$

$$
L^{(4)} = \begin{pmatrix}
0 & 6 & \infty & 8 & -1 & \infty \\
-2 & 0 & \infty & 2 & -3 & \infty \\
-5 & -3 & 0 & -1 & -3 & -8 \\
-4 & 2 & \infty & 0 & -5 & \infty \\
5 & 7 & \infty & 9 & 0 & \infty \\
3 & 5 & 10 & 7 & 2 & 0
\end{pmatrix}
$$

$$
L^{(5)} = \begin{pmatrix}
0 & 6 & \infty & 8 & -1 & \infty \\
-2 & 0 & \infty & 2 & -3 & \infty \\
-5 & -3 & 0 & -1 & -6 & -8 \\
-4 & 2 & \infty & 0 & -5 & \infty \\
5 & 7 & \infty & 9 & 0 & \infty \\
3 & 5 & 10 & 7 & 2 & 0
\end{pmatrix}
$$

Then, since we have reached $L^{(n-1)}$ we can stop since we know that since there are no negative weight cycles, taking higher powers will not cause the matrix to change at all. By the fast method, we get that

$$
L^{(1)} = \begin{pmatrix}
0 & \infty & \infty & \infty & -1 & \infty \\
1 & 0 & \infty & 2 & \infty & \infty \\
\infty & 2 & 0 & \infty & \infty & -8 \\
-4 & \infty & \infty & 0 & 3 & \infty \\
\infty & 7 & \infty & \infty & 0 & \infty \\
\infty & 5 & 10 & \infty & \infty & 0
\end{pmatrix}
$$

$$
L^{(2)} = \begin{pmatrix}
0 & 6 & \infty & \infty & -1 & \infty \\
-2 & 0 & \infty & 2 & 0 & \infty \\
3 & -3 & 0 & 4 & \infty & -8 \\
-4 & 10 & \infty & 0 & -5 & \infty \\
8 & 7 & \infty & 9 & 0 & \infty \\
6 & 5 & 10 & 7 & \infty & 0
\end{pmatrix}
$$

$$
L^{(4)} = \begin{pmatrix}
0 & 6 & \infty & 8 & -1 & \infty \\
-2 & 0 & \infty & 2 & -3 & \infty \\
-5 & -3 & 0 & -1 & -3 & -8 \\
-4 & 2 & \infty & 0 & -5 & \infty \\
5 & 7 & \infty & 9 & 0 & \infty \\
3 & 5 & 10 & 7 & 2 & 0
\end{pmatrix}
$$

$$
L^{(8)} = \begin{pmatrix}
0 & 6 & \infty & 8 & -1 & \infty \\
-2 & 0 & \infty & 2 & -3 & \infty \\
-5 & -3 & 0 & -1 & -6 & -8 \\
-4 & 2 & \infty & 0 & -5 & \infty \\
5 & 7 & \infty & 9 & 0 & \infty \\
3 & 5 & 10 & 7 & 2 & 0
\end{pmatrix}
$$

We stop since $8 \geq 5 = n - 1$.

Exercise 25.1-2

This is consistent with the fact that the shortest path from a vertex to itself is the empty path of weight 0. If there were another path of weight less than 0 then it must be a negative-weight cycle, since it starts and ends at v_i.

Exercise 25.1-3

This matrix corresponds to the identity matrix in normal matrix multiplication. This is because anytime that we take a min of any number with infinity, we get that same number back. Another way of seeing this is that we can interpret this strange version of matrix multipllication as allowing any of our shortest paths to be a path described by one matrix followed by a path described by the other. However, the given matrix corresponds to there being no paths between

any of the vertices. This means that we won't change any of the shortest paths that are described by the matrix we are multiplying it by since we are allowing nothing more for those paths.

Exercise 25.1-4

To verify associativity, we need to check that $(W^i W^j)W^p = W^i(W^j W^p)$ for all i, j, p, where we use the matrix multiplication defined by the EXTEND-SHORTEST-PATHS procedure. Consider entry (a, b) of the left hand side. This is:

$$\min_{1 \le k \le n} [W^i W^j]_{a,k} + W^p_{k,b} = \min_{1 \le k \le n} \min_{1 \le q \le n} W^i_{a,q} + W^j_{q,k} + W^p_{k,b}$$
$$= \min_{1 \le q \le n} W^i_{a,q} + \min_{1 \le k \le n} W^j_{q,k} + W^p_{k,b}$$
$$= \min_{1 \le q \le n} W^i_{a,q} + [W^j W^p]_{q,b}$$

which is precisely entry (a, b) of the right hand side.

Exercise 25.1-5

We can express finding the shortest path from a single vertex with the modified version of matrix multiplication described in the section. We initially let v_1 be a vector indexed by the vertices of the graph. It is infinity when the corresponding vertex has no edge going to it from the source vertex, s. It is the weight of the edge going to it from s if there is one. Lastly, it is zero in the entry corresponding to s itself. Essentially, we are only taking the row of the W matrix that corresponds to s. Then, we define $v_{i+1} = v_i W$. Then, we stop computing v_i once we compute v_{n-1}. This vector then contains the correct shortest distances from the source to each vertex. Since each time we multiply the vector by the matrix, we only have to consider the entries which are non-infinite in W. There are only $|E| + |V|$ of these non-finfinite entries. So, we have that the time required for each time we mutiply the vector by the matrix, we take time at most $O(E)$. So, the total runtime would be $O(EV)$ just as in Bellman-Ford. The similarities don't stop there however. This is because v_i represents the shortest distance to each vector from s using at most i edges, and each time we multiply by W corresponds to relaxing every edge.

Exercise 25.1-6

For each source vertex v_i we need to compute the shortest-paths tree for v_i. To do this, we need to compute the predecessor for each $j \ne i$. For fixed i and j, this is the value of k such that $L_{i,k} + w(k, j) = L_{i,j}$. Since there are n vertices whose trees need computing, n vertices for each such tree whose predecessors need computing, and it takes $O(n)$ to compute this for each one (checking each possible k), the total time is $O(n^3)$.

Exercise 25.1-7
To have the procedure compute the predecessor along the shortest path, see the modified procedures, EXTEND-SHORTEST-PATH-MOD and SLOW-ALL-PAIRS-SHORTEST-PATHS-MOD

Algorithm 1 EXTEND-SHORTEST-PATH-MOD(Π, L,W)

---

n= L.rows
Let $L' = (l'_{ij})$ be a new $n \times n$ matrix.
$\Pi' = (\pi'_{ij})$ is a new $n \times n$ matrix
for i=1 to n **do**
    for j = 1 to n **do**
        $l'_{ij} = \infty$
        $\pi_{ij} = NIL$
        for k=1 to n **do**
            if $l_{ik} + l_{jk} < l_{ij}$ **then**
                $l_{ij} = l_{ik} + l_{jk}$
                $\pi'_{ij} = \pi_{kj}$
            end if
        end for
    end for
end for
return Π', L'

Algorithm 2 SLOW-ALL-PAIRS-SHORTEST-PATHS-MOD(W)

---

n= W.rows
$L^{(1)} = W$
$\Pi^{(1)} = (\pi^{(1)}_{ij})$ where $\pi^{(1)}_{ij} = i$ if there is an edge from i to j, and NIL otherwise.
for m=2 to n-1 **do**
    $\Pi^{(m)}, L^{(m)} = $ EXTEND-SHORTEST-PATH-MOD($\Pi^{(m-1)}, L^{(m-1)}$,W)
end for
return $\Pi^{(n-1)}, L^{(n-1)}$

Exercise 25.1-8

We can overwrite matrices as we go. Let $A \star B$ denote multiplication defined by the EXTEND-SHORTEST-PATHS procedure. Then we modify FASTER-ALL-EXTEND-SHORTEST-PATHS(W). We initially create an n by n matrix L. Delete line 5 of the algorithm, and change line 6 to $L = W \star W$, followed by $W = L$.

Exercise 25.1-9

For the modification, keep computing for one step more than the original, that is, we compute all the way up to $L^{(2^{k+1})}$ where $2^k > n - 1$. Then, if

there aren't any negative weight cycles, then, we will have that the two matrices should be equal since having no negative weight cycles means that between any two vertices, there is a path that is tied for shortest and contains at most $n - 1$ edges. However, if there is a cycle of negative total weight, we know that it's length is at most n, so, since we are allowing paths to be larger by $2^k \geq n$ between these two matrices, we have that we would need to have all of the vertices on the cycle have their distance reduce by at least the negative weight of the cycle. Since we can detect exactly when there is a negative cycle, based on when these two matrices are different. This algorithm works. It also only takes time equal to a single matrix multiplication which is littlee oh of the unmodified algorithm.

Exercise 25.1-10

A negative-weight cycle appears when $W_{i,i}^m < 0$ for some m and i. Each time a new *power* of W is computed, we simply check whether or not this happens, at which point the cycle has length m. The runtime is $O(n^4)$.

Exercise 25.2-1

k	D^k					
0	0	∞	∞	∞	−1	∞
	1	0	∞	2	∞	∞
	∞	2	0	∞	∞	−8
	−4	∞	∞	0	3	∞
	∞	7	∞	∞	0	∞
	∞	5	10	∞	∞	0
1	0	∞	∞	∞	−1	∞
	1	0	∞	2	0	∞
	∞	2	0	∞	∞	−8
	−4	∞	∞	0	−5	∞
	∞	7	∞	∞	0	∞
	∞	5	10	∞	∞	0
2	0	∞	∞	∞	−1	∞
	1	0	∞	2	0	∞
	3	2	0	4	2	−8
	−4	∞	∞	0	−5	∞
	8	7	∞	9	0	∞
	6	5	10	7	5	0
3	0	∞	∞	∞	−1	∞
	1	0	∞	2	0	∞
	3	2	0	4	2	−8
	−4	∞	∞	0	−5	∞
	8	7	∞	9	0	∞
	6	5	10	7	5	0
4	0	∞	∞	∞	−1	∞
	−2	0	∞	2	−3	∞
	0	2	0	4	−1	−8
	−4	∞	∞	0	−5	∞
	5	7	∞	9	0	∞
	3	5	10	7	2	0
5	0	6	∞	8	−1	∞
	−2	0	∞	2	−3	∞
	0	2	0	4	−1	−8
	−4	2	∞	0	−5	∞
	5	7	∞	9	0	∞
	3	5	10	7	2	0
6	0	6	∞	8	−1	∞
	−2	0	∞	2	−3	∞
	−5	−3	0	−1	−6	−8
	−4	2	∞	0	−5	∞
	5	7	∞	9	0	∞
	3	5	10	7	2	0

Exercise 25.2-2

6

We set $w_{ij} = 1$ if (i, j) is an edge, and $w_{ij} = 0$ otherwise. Then we replace line 7 of EXTEND-SHORTEST-PATHS(L,W) by $l'_{ij} = l'_{ij} \vee (l_{ik} \wedge w_{kj})$. Then run the SLOW-ALL-PAIRS-SHORTEST-PATHS algorithm.

Exercise 25.2-3

See the modified version of the Floyd-Warshall algorithm:

Algorithm 3 MOD-FLOYD-WARSHALL(W)
---
n= W.rows
$D^0 = W$
π^0 is a matrix with nil in every entry
for i=1 to n **do**
    for j = 1 to n **do**
        if $i \neq j$ and $D^0_{i,j} < \infty$ **then**
            $\pi^0_{i,j} = i$
        end if
    end for
end for
for k=1 to n **do**
    let D^k be a new $n \times n$ matrix.
    let π^k be a new $n \times n$ matrix
    for i=1 to n **do**
        for j = 1 to n **do**
            if $d^{k-1}_{ij} \leq d^{k-1}_{i,k} + d^{k-1}_{k,j}$ **then**
                $d^k_{i,j} = d^{k-1}_{i,j}$
                $\pi^k_{i,j} = \pi^{k-1}_{i,j}$
            else
                $d^k_{i,j} = d^{k-1}_{i,k} + d^{k-1}_{k,j}$
                $\pi^k_{i,j} = \pi^{k-1}_{k,j}$
            end if
        end for
    end for
end for

In order to have that $\pi^{(k)}_{ij} = l$, we need that $d^{(k)}_{ij} \geq d^{(k)}_{il} + w_{lj}$. To see this fact, we will note that having $\pi^{(k)}_{ij} = l$ means that a shortest path from i to j last goes through l. A path that last goes through l corresponds to taking a chepest path from i to l and then following the single edge from l to j. However, This means that $d_{il} \leq d_{ij} - w_{ij}$, which we can rearrange to get the desired inequality. We can just continue following this inequality around, and if we ever get some cycle, $i_1, i_2, \ldots i_c$, then we would have that $d_{ii_1} \leq d_{ii_1} + w_{i_1 i_2} + w_{i_2 i_3} + \cdots + w_{i_c i_1}$. So, if we subtract the common term from both sides, we get that $0 \leq w_{i_c i_1} + \sum_{q=1}^{c-1} w_{i_q i_{q+1}}$. So, we have that we

would only have a cycle in the precedessor graph if we ahvt that there is a zero weight cycle in the original graph. However, we would never have to go around the weight zero cycle since the constructed path of shortest weight favors ones with a fewer number of edges because of the way that we handle the equality case in equation (25.7).

Exercise 25.2-4

Suppose we are updating $d_{ij}^(k)$. To be sure we get the same results from this algorithm, we need to check what happens to $d_{ij}^{(k-1)}$, $d_{ik}^{(k-1)}$ and $d_{kj}^{(k-1)}$. The $d_{ij}^{(k-1)}$ term will be unchanged. On the other hand, the other terms won't change because any shorest path from i to k which includes k necessarily includes it only once (since there are no negative-weight cycles) so it is the length of a shortest path using only vertices 1 through $k-1$. Thus, updating in place is okay.

Exercise 25.2-5

If we change the way that we handle the equality case, we willl still be generating a the correct values for the π matrix. This is because updating the π values to make paths that are longer but still tied for the lowest weight. Making $\pi_{ij} = \pi_{kj}$ means that we are making the shortest path from i to j passes through k at some point. This has the same cost as just going from i to j, since $d_{ij} = d_{ik} + d_{kj}$.

Exercise 25.2-6

If there was a negative-weight cycle, there would be a negative number occurring on the diagonal upon termination of the Floyd-Warshall algorithm.

Exercise 25.2-7
We can recursively compute the values of $\phi_{ij}^{(k)}$ by, letting it be $\phi_{ij}^{(k-1)}$ if $d_{ik}^{(k)} + d_{kj}^{(k)} \geq d_{ij}^{(k-1)}$, and otherwise, let it be k. This works correctly because it perfectly captures whether we decided to use vertex k when we were repeatedly allowing ourselves use of each vertex one at a time. To modify Floyd-Warshall to compute this, we would just need to stick within the innermost for loop, something that computes $\phi_{ij}^{(k)}$ by this recursive rule, this would only be a constant amount of work in this innermost for loop, and so would not cause the asymptotic runtime to increase. It is similar to the s table in matrix-chain multiplication because it is computed by a similar recurrence.

If we already have the n^3 values in $\phi_{ij}^{(k)}$ provided, then we can reconstruct the shortest path from i to j because we know that the largest vertex in the path from i to j is $\phi_{ij}^{(n)}$, call it a_1. Then, we know that the largest vertex in the path before a_1 will be $\phi_{ia_1}^{(a_1-1)}$ and the largest after a_1 will be $\phi_{a_1j}^{(a_1-1)}$. By

continuing to recurse until we get that the largest element showing up at some point is NIL, we will be able to continue subdividing the path until it is entirely constructed.

Exercise 25.2-8

Create an n by n matrix A filled with 0's. We are done if we can determine the vertices reachable from a particular vertex in $O(E)$ time, since we can just compute this for each $v \in V$. To do this, assign each edge weight 1. Then we have $\delta(v, u) \leq |E|$ for all $u \in V$. By Problem 24-4 (a) we can compute $\delta(v, u)$ in $O(E)$ for all $u \in V$. If $\delta(v, u) < \infty$, set $A_{ij} = 1$. Otherwise, leave it as 0.

Exercise 25.2-9

First, compute the strongly connected components of the directed graph, and look at it's component graph. This component graph is going to be acyclic and have at most as many vertices and at most as many edges as the original graph. Since it is acyclic, we can run our transitive closure algorithm on it. Then, for every edge (S_1, S_2) that shows up in the transitive closure of the component graph, we add an edge from each vertex in S_1 to a vertex in S_2. This takes time equal to $O(V + E^*)$. So, the total time required is $\leq f(|V|, |E|) + O(V + E)$.

Exercise 25.3-1

v	$h(v)$
1	-5
2	-3
3	0
4	-1
5	-6
6	-8

u	v	$\hat{w}(u, v)$
1	2	NIL
1	3	NIL
1	4	NIL
1	5	0
1	6	NIL
2	1	3
2	3	NIL
2	4	0
2	5	NIL
2	6	NIL
3	1	NIL
3	2	5
3	4	NIL
3	5	NIL
3	6	0
4	1	0
4	2	NIL
4	3	NIL
4	5	8
4	6	NIL
5	1	NIL
5	2	4
5	3	NIL
5	4	NIL
5	6	NIL
6	1	NIL
6	2	0
6	3	18
6	4	NIL
6	5	NIL

So, the $d_{i,j}$ values that we get are

$$
\begin{pmatrix}
0 & 6 & \infty & 8 & -1 & \infty \\
-2 & 0 & \infty & 2 & -3 & \infty \\
-5 & -3 & 0 & -1 & -6 & -8 \\
-4 & 2 & \infty & 0 & -5 & \infty \\
5 & 7 & \infty & 9 & 0 & \infty \\
3 & 5 & 10 & 7 & 2 & 0
\end{pmatrix}
$$

Exercise 25.3-2

This is only important when there are negative-weight cycles in the graph. Using a dummy vertex gets us around the problem of trying to compute $-\infty + \infty$ to find $\hat{w}$. Moreover, if we had instead used a vertex v in the graph instead of

the new vertex s, then we run into trouble if a vertex fails to be reachable from v.

Exercise 25.3-3

If all the edge weights are nonnegative, then the values computed as the shortest distances when running Bellman-Ford will be all zero. This is because when constructing G' on the first line of Johnson's algorithm, we place an edge of weight zero from s to every other vertex. Since any path within the graph has no negative edges, its cost cannot be negative, and so, cannot beat the trivial path that goes straight from s to any given vertex. Since we have that $h(u) = h(v)$ for every u and v, the reweighting that occurs only adds and subtracts 0, and so we have that $w(u, v) = \hat{w}(u, v)$

Exercise 25.3-4

This doesn't preserve shortest paths. Suppose we have a graph with vertices $a, b, c, d, e,$ and f and edges $(a, b), (b, c), (c, d), (d, e), (a, e), (a, f)$ with weights -1, -1, -1, -1, -1, 1, and -2. Professor Greenstreet would add 2 to every edge weight in the graph. Originally, the shortest path from a to e went through vertices b, c, and d. With the added weight, the new shortest path goes directly from a to e. Thus, property 1 of $\hat{w}$ is violated.

Exercise 25.3-5

By lemma 25.1, we have that the total weight of any cycles is unchanged as a result of the reweighting procedure. This can be seen in a way similar to how the last claim of lemma 25.1 was proven. Namely, we consider the cycle c as a path that has the same starting and ending vertices, so, by the first half od lemma 25.1, we have that

$$\hat{w}(c) = w(c) + h(v_0) - h(v_k) = w(c) = 0$$

This means that in the reweighted graph, we still have that the same cycle as before had a total weight of zero. Since there are no longer any negative weight edges after we reweight, this is precicely the second property of the reweighting procedure shown in the section. Since we have that the sum of all the edge weights in c is still equal to zero, but each of them individually has a nonnegative weight, it must be the case that each of them individually is equal to 0.

Exercise 25.3-6

Let G have vertices a, b, and c, and edge (b, c) with weight 1, and let $s = a$. When we try to compute $\hat{w}(b, c)$, it is undefined because $h(c) = h(b) = \infty$. Now suppose that G is strongly connected, and further that there are no negative-weight cycles. Since every vertex is reachable from every other, h is well-defined

for any choice of source vertex. We need only check that $\hat{w}$ satisfies properties 1 and 2. For the first property, let $p = \langle u = v_0, v_1, \ldots, v_k = v \rangle$ be a shortest path from u to v using the weight function w. Then we have

$$
\begin{aligned}
\hat{w}(p) &= \sum_{i=1}^{k} \hat{w}(v_{i-1}, v_i) \\
&= \sum_{i=1}^{k} w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i) \\
&= \left(\sum_{i=1}^{k} w(v_{i-1}, v_i) \right) + h(u) - h(v) \\
&= w(p) + h(u) - h(v).
\end{aligned}
$$

Since $h(u)$ and $h(v)$ are independent of p, we must have that w minimizes $w(p)$ if and only if $\hat{w}$ minimizes $\hat{w}(p)$. For the second property, the triangle inequality tells us that for any vertices $v, u \in V$ we have $\delta(v, u) \leq \delta(v, z) + w(z, u)$. Thus, if we define $h(u) = \delta(v, u)$ then we have $\hat{w}(z, u) = w(z, u) + h(z) - h(u) \geq 0$.

Problem 25-1

a. We can update the transitive closure in time $O(V^2)$ as follows. Suppose that we add the edge (x_1, x_2). Then, we will consider every pair of vertices (u, v). In order to of created a path between them, we would need some part of that path that goes from u to x_1 and some second part of that path that goes from x_2 to v. This means that we add the edge (u, v) to the transitive closure if and only if the transitive closure contains the edges (u, x_1) and (x_2, v). Since we only had to consider every pair of vertices once, the runtime of this update is only $O(V^2)$.

b. Suppose that we currently have two strongly connected components, each of size $|V|/2$ with no edges between them. Then their transitive closures computed so far will consist of two complete directed graphs on $|V|/2$ vertices each. So, there will be a total of $|V|^2/2$ edges adding the number of edges in each together.

Then, we add a single edge from one component to the other. This will mean that every vertex in the component the edge is coming from will have an edge going to every vertex in the component that the edge is going to. So, the total number of edges after this operation will be $|V|/2 + |V|/4$. So, the number of edges increased by $|V|/4$. Since each time we add an edge, we need to use at least constant time, since there is no cheap way to add many edges at once, the total amount of time needed is $\Omega(|V|^2)$.

c. We will have each vertex maintain a tree of vertices that have a path to it and a tree of vertices that it has a path to. The second of which is the transitive

closure at each step. Then, upon inserting an edge, (u,v), we will look at successive ancestors of u, and add v to their successor tree, just past u. If we ever don't insert an edge when doing this, we can stop exploring that branch of the ancestor tree. Similarly, we keep doing this for all of the ancestors of v. Since we are able to short circuit if we ever notice that we have already added an edge, we know that we will only ever reconsider the same edge at most n times, and, since the number of edges is $O(n^2)$, the total runtime is $O(n^3)$.

Problem 25-2

a. As in problem 6-2, the runtimes for a d-ary min heap are the same as for a d-ary max heap. The runtimes of INSERT, EXTRACT-MIN, and DECREASE-KEY are $O(\log_d n)$, $O(d \log_d n)$, and $O(\log_d n)$ respectively. If $d = \Theta(n^\alpha)$ these become $O(1/\alpha)$, $O(n^\alpha/\alpha)$, and $O(1/\alpha)$ respectively. The amortized costs for these operations in a FIbonacci heap are $O(1)$, $O(\lg n)$, and $O(1)$ respectively.

b. Choose $d = n^\varepsilon$. Then implement Dijkstra's algorithm using a d-ary min-heap. The analysis from part (a) tells us that the runtime will be $O(n/\varepsilon + 2n^{1+\varepsilon}/\varepsilon) = O(n^{1+\varepsilon}) = O(E)$.

c. Run the algorithm from part (b) once for each vertex of the graph.

d. Using the methods from section 25.3, create the graph G' with all nonnegative edge weights using the weight function $\hat{w}$, proceed as in part (c), then convert back to the original weight function.

Chapter 26

Michelle Bodnar, Andrew Lohr

May 5, 2017

Exercise 26.1-1

To see that the networks have the same maximum flow, we will show that every flow through one of the networks corresponds to a flow through the other. First, suppose that we have some flow through the network before applying the splitting procedure to the anti-symmetric edges. Since we are only changing one of any pair of anti-symmetric edges, for any edge that is unchanged by the splitting, we just have an identical flow going through those edges. Suppose that there was some edge (u, v) that was split because it had an anti-symmetric edge, and we had some flow, $f(u, v)$ in the original graph. Since the capacity of both of the two edges that are introduced by the splitting of that edge have the same capacity, we can set $f'(u, v) = f'(u, x) = f'(x, v)$. By constructing the new flow in this manor, we have an identical total flow, and we also still have a valid flow.

Similarly, suppose that we had some flow f' on the graph with split edges, then, for any triple of vertices u, x, v that correspond to a split edge, we must have that $f'(u, x) = f'(x, v)$ because the only edge into x is (u, x) and the only edge out of x is (x, v), and the net flow into and out of each vertex must be zero. We can then just set the flow on the unsplit edge equal to the common value that the flows on (u, x) and (x, v) have. Again, since we handle this on an edge by edge basis, and each substitution of edges maintains the fact that it is a flow of the same total, we have that the end result is also a valid flow of the same total value as the original.

Since we have shown that any flow in one digraph can be translated into a flow of the same value in the other, we can translate the maximum value flow for one of them to get that it's max value flow is $\leq$ to that of the other, and do it in the reverse direction as well to achieve equality.

Exercise 26.1-2

The capacity constraint remains the same. We modify flow conservation so that each s_i and t_i must satisfy the "flow in equals flow out" constraint, and we only exempt s and t. We define the value of a flow in the multiple-source, multiple-sink problem to be $\left| \sum_{i=1}^{m} \sum_{v \in V} f(s_i, v) - \sum_{v \in V} f(v, s_i) \right|$. Let $f_i = \sum_{v \in V} f(s_i, v) - \sum_{v \in V} f(v, s_i)$. In the single-source flow network, set

$f(s, s_i) = f_i$. This satisfies the capacity constraint and flow conservation, so it is a valid assignment. The flow for the multiple-source network in this case is $|f_1 + f_2 + \cdots + f_m|$. In the single-source case, since there are no edges coming into s, the flow is $\sum_{i=1}^{m} f(s, s_i)$. Since $f(s, s_i)$ is positive and equal to f_i, they are equivalent.

Exercise 26.1-3

Suppose that we are in the situation posed by the question, that is, that there is some vertex u that lies on no path from s to t. Then, suppose that we have for some vertex v, either $f(v, u)$ or $f(u, v)$ is nonzero. Since flow must be conserved at u, having any positive flow either leaving or entering u, there is both flow leaving and entering. Since u doesn't lie on a path from s to t, we have that there are two cases, either there is no path from s to u or(possibly and) there is no path from u to t. If we are in the second case, we construct a path with $c_0 = u$, and c_{i+1} is an successor of c_i that has $f(c_i, c_{i+1})$ being positive. Since the only vertex that is allowed to have a larger flow in than flow out is t, we have that this path could only ever terminate if it were to reach t, since each vertex in the path has some positive flow in. However, we could never reach t because we are in the case that there is no path from u to t. If we are in the former case that there is no path from s to u, then we similarly define $c_0 = u$, however, we let c_{i+1} be any vertex so that $f(c_{i+1}, c_i)$ is nonzero. Again, this sequence of vertices cannot terminate since we could never arrive at having s as one of the vertices in the sequence.

Since in both cases, we an always keep extending the sequence of vertices, we have that it must repeat itself at some point. Once we have some cycle of vertices, we can decrease the total flow around the cycle by an amount equal to the minimum amount of flow that is going along it without changing the value of the flow from s to t since neither of those two vertices show up in the cycle. However, by decreasing the flow like this, we decrease the total number of edges that have a positive flow. If there is still any flow passing though u, we can continue to repeat this procedure, decreasing the number of edges with a positive flow by at least one. Since there are only finitely many vertices, at some point we need to have that there is no flow passing through u. The flow obtained after all of these steps is the desired maximum flow that the problem asks for.

Exercise 26.1-4

Since f_1 and f_2 are flows, they satisfy the capacity constraint, so we have $0 \le \alpha f_1(u, v) + (1-\alpha) f_2(u, v) \le \alpha c(u, v) + (1-\alpha) c(u, v) = c(u, v)$, so the new flow satisfies the capacity constraint. Further, f_1 and f_2 satisfy flow conservation,

so for all $u \in V - \{s, t\}$ we have

$$\sum_{v \in V} \alpha f_1(v, u) + (1 - \alpha) f_2(v, u) = \alpha \sum_{v \in V} f_1(v, u) + (1 - \alpha) \sum_{v \in V} f_2(v, u)$$

$$= \alpha \sum_{v \in V} f_1(u, v) + (1 - \alpha) \sum_{v \in V} f_2(u, v)$$

$$= \sum_{v \in V} \alpha f_1(u, v) + (1 - \alpha) f_2(u, v).$$

Therefore the flows form a convex set.

Exercise 26.1-5

A linear programming problem consists of a set of variables, a linear function of those variables that needs to be maximized, and a a set of constraints. Our variables x_e will be the amount of flow across each edge e. The function to maximize is $\sum_{e \text{ leaving } s} x_e - \sum_{e \text{ entering } s} x_e$. The sum of these flows is exactly equal to the value of the flow from s to t. Now, we consider constraints. There are two types of constraints, capacity constraints and flow constraints. The capacity constraints are just $x_e \leq c(e)$ where c_e is the capacity of edge e. The flow constraints are that $\sum_{e \text{ leaving } v} x_e - \sum_{e \text{ entering } v} x_e = 0$ for all vertices $v \neq s, t$. Since this linear program captures all the same constraints, and wants to maximize the same thing, it is equivalent to the max flow problem.

Exercise 26.1-6

Use the map to create a graph where vertices represent street intersections and edges represent streets. Define $c(u, v) = 1$ for all edges (u, v). Since a street can be traversed, start off by creating a directed edge in each direction, then make the transformation to a flow problem with no antiparallel edges as described in the section. Make the home the source and the school the sink. If there exist at least two distinct paths from source to sink then the flow will be at least 2 because we could assign $f(u, v) = 1$ for each of those edges. However, if there is at most one distinct path from source to sink then there must exist a bridge edge (u, v) whose removal would disconnect s from t. Since $c(u, v) = 1$, the flow into u is at most 1. We may assume there are no edges into s or out from t, since it doesn't make sense to return home or leave school. By flow conservation, this implies that $f = \sum_{v \in V} f(s, v) \leq 1$. Thus, determining the maximum flow tells the Professor whether or not his children can go to the same school.

Exercise 26.1-7

We can capture the vertex constraints by splitting out each vertex into two, where the edge between the two vertices is the vertex capacity. More formally,

our new flow network will have vertices $\{0, 1\} \times V$. It has an edge between $1 \times v$ and $0 \times u$ if there is an edge (v, u) in the original graph, the capacity of such an edge is just $c(v, u)$. The edges of the second kind that the new flow network will have are from $0 \times v$ to $1 \times v$ for every v with capacity $l(v)$. This new flow network will have $2|V|$ vertices and have $|V| + |E|$ edges. Lastly, we can see that this network does capture the idea that the vertices have capacities $l(v)$. This is because any flow that goes through v in the original graph must go through the edge $(0 \times v, 1 \times v)$ in the new graph, in order to get from the edges going into v to the edges going out of v.

Exercise 26.2-1

To see that equation (26.6) equals (26.7), we will show that the terms that we are throwing into the sums are all zero. That is, we will show that if $v \in V \setminus (V_1 \cup V_2)$, then $f'(s, v) = f'(v, s) = 0$. Since $v \notin V_1$, then there is no edge from s to v, similarly, since $v \notin V_2$, there is no edge from v to s. This means that there is no edge connecting s and v in any way. Since flow can only pass along edges, we know that there can be no flow passing directly between s and v.

Exercise 26.2-2

The flow across the cut is $11 + 1 + 7 + 4 - 4 = 19$. The capacity of the cut is $16 + 4 + 7 + 4 = 31$.

Exercise 26.2-3
If we perform a breadth first search where we consider the neighbors of a vertex as they appear in the ordering $\{s, v_1, v_2, v_3, v_4, t\}$, the first path that we will find is s, v_1, v_3, t. The min capacity of this augmenting path is 12, so we send 12 units along it. We perform a BFS on the resulting residual network. This gets us the path s, v_2, v_4, t. The min capacity along this path is 4, so we send 4 units along it. Then, the only path remaining in the residual network is $\{s, v_2, v_4, v_3\}$ which has a min capacity of 7, since that's all that's left, we find it in our BFS. Putting it all together, the total flow that we have found has a value of 23.
Exercise 26.2-4

A minimum cut corresponding to the maximum flow is $S = \{s, v_1, v_2, v_4\}$ and $T = \{v_3, t\}$. The augmenting path in part (c) cancels flow on edge (v_3, v_2).

Exercise 26.2-5

Since the only edges that have infinite value are those going from the super-source or to the supersink, as long as we pick a cut that has the supersource and all the original sources on one side, and the other side has the supersink as well as all the original sinks, then it will only cut through edges of finite capacity. Then, by Corollary 26.5, we have that the value of the flow is bounded above

by the value of any of these types of cuts, which is finite.

Exercise 26.2-6

Begin by making the modification from multi-source to single-source as done in section 26.1. Next, create an extra vertex $\hat{s}_i$ for each i and place it between s and s_i. Explicitly, remove the edge from s to s_i and add edges $(s, \hat{s}_i)$ and $(\hat{s}_i, s_i)$. Similarly, create an extra vertex $\hat{t}_i$ for each vertex t_i and place it between t and t_i. Remove the edges (t_i, t) and add edges $(t_i, \hat{t}_i)$ and $(\hat{t}_i, t)$. Assign $c(\hat{s}_i, s_i) = p_i$ and $c(t_i, \hat{t}_i) = q_i$. If a flow which satisfies the constraints exists, it will assign $f(\hat{s}_i, s_i) = p_i$. By flow conservation, this implies that $\sum_{v \in V} f(s_i, v) = p_i$. Similarly, we must have $f(t_i, \hat{t}_i) = q_i$, so by flow conservation this implies that $\sum_{v \in V} f(v, t_i) = q_i$.

Exercise 26.2-7

To check that f_p is a flow, we make sure that it satisfies both the capacity constraints and the flow constraints. First, the capacity constraints. To see this, we recall our definition of $c_f(p)$, that is, it is the smallest residual capacity of any of the edges along the path p. Since we have that the residual capacity is always less than or equal to the initial capacity, we have that each value of the flow is less than the capacity. Second, we check the flow constraints, Since the only edges that are given any flow are along a path, we have that at each vertex interior to the path, the flow in from one edge is immediately canceled by the flow out to the next vertex in the path. Lastly, we can check that its value is equal to $c_f(p)$ because, while s may show up at spots later on in the path, it will be canceled out as it leaves to go to the next vertex. So, the only net flow from s is the initial edge along the path, since it (along with all the other edges) is given flow $c_f(p)$, that is the value of the flow f_p.

Exercise 26.2-8

Paths chosen by the while loop of line 3 go from s to t and are simple because capacities are always nonnegative. Thus, no edge into s will ever appear on an augmenting path, so such edges may as well never have existed.

Exercise 26.2-9

The augmented flow does satisfy the flow conservation property, since the sum of flow into a vertex and out of a vertex can be split into two sums each, one running over flow in f and the other running over flow in f', since we have the parts are equal separately, their sums are also equal.

The capacity constraint is not satisfied by this arbitrary augmentation of flows. To see this, suppose we only have the vertices s and t, and have a single edge from s to t of capacity 1. Then we could have a flow of value 1 from s to

t, however, augmenting this flow with itself ends up putting two units along the edge from s to t, which is greater than the capacity we can send.

Exercise 26.2-10

Suppose we already have a maximum flow f. Consider a new graph G where we set the capacity of edge (u, v) to $f(u, v)$. Run Ford-Fulkerson, with the modification that we remove an edge if its flow reaches its capacity. In other words, if $f(u, v) = c(u, v)$ then there should be no reverse edge appearing in residual network. This will still produce correct output in our case because we never exceed the actual maximum flow through an edge, so it is never advantageous to cancel flow. The augmenting paths chosen in this modified version of Ford-Fulkerson are precisely the ones we want. There are at most $|E|$ because every augmenting path produces at least one edge whose flow is equal to its capacity, which we set to be the actual flow for the edge in a maximum flow, and our modification prevents us from ever destroying this progress.

Exercise 26.2-11

To test edge connectivity, we will take our graph as is, pick an arbitrary s to be our source for the flow network, and then, we will consider every possible other selection of our sink t. For each of these flow networks, we will replace each (undirected) edge in the original graph with a pair of anti-symmetric edges, each of capacity 1.

We claim that the minimum value of all of these different considered flow networks' maximum flows is indeed the edge connectivity of the original graph. Consider one particular flow network, that is, a particular choice for t. Then, the value of the maximum flow is the same as the value of the minimum cut separating s and t. Since each of the edges have a unit capacity, the value of any cut is the same as the number of edges in the original graph that are cut by that particular cut. So, for this particular choice of t, we have that the maximum flow was the number of edges needed to be removed so that s and t are in different components. Since our end goal is to remove edges so that the graph becomes disconnected, this is why we need to consider all $n - 1$ flow networks. That is, it may be much harder for some choices of t than others to make s and t end up in different components. However, we know that there is some vertex that has to be in a different component than s after removing the smallest number of edges required to disconnect the graph. So, this value for the number of edges is considered when we have t be that vertex.

Exercise 26.2-12

Since every vertex lies on some path starting from s there must exist a cycle which contains the edge (v, s). Use a depth first search to find such a cycle with no edges of zero flow. Such a cycle must exist since f satisfies conservation of flow. Since the graph is connected this takes $O(E)$. Then decrement the flow

of every edge on the cycle by 1. This preserves the value of the flow so it is still maximal. It won't violate the capacity constraint because $f > 0$ on every edge of the cycle prior to decrementing. Finally, flow conservation isn't violated because we decrement both an incoming and outgoing edge for each vertex on the cycle by the same amount.

Exercise 26.2-13

Suppose that your given flow network contains $|E|$ edges, then, we were to modify all of the capacities of the edges by taking any edge that has a postive capacity and increasing its capacity by $\frac{1}{|E|+1}$. Doing this modification can't get us a set of edges for a min cut that isn't also a min cut for the unmodified graph because the difference between the value of the min cut and the next lowest cut vale was at least one because all edge weights were integers. This means that the new min cut value is going to be at most the original plus $\frac{|E|}{|E|+1}$. Since this value is more than the second smallest valued cut in the original flow network, we know that the choice of cuts we make in the new flow network is also a minimum cut in the original. Lastly, since we added a small constant amount to the value of each edge, our minimum cut would have the smallest possible number of edges, otherwise one with fewer would have a smaller value.

Exercise 26.3-1
First, we pick an augmenting path that passes through vertices 1 and 6. Then, we pick the path going through 2 and 8. Then, we pick the path going through 3 and 7. Then, the resulting residual graph has no path from s to t. So, we know that we are done, and that we are pairing up vertices $(1,6), (2,8)$, and $(3,7)$. This number of unit augmenting paths agrees with the value of the cut where you cut the edges $(s,3), (6,t)$, and $(7,t)$.

Exercise 26.3-2

We proceed by induction on the number of iterations of the while loop of Ford-Fulkerson. After the first iteration, since c only takes on integer values and $(u,v).f$ is set to 0, c_f only takes on integer values. Thus, lines 7 and 8 of Ford-Fulkerson only assign integer values to $(u,v).f$. Assume that $(u,v).f \in \mathbb{Z}$ for all (u,v) after the nth iteration. On the $(n+1)^{st}$ iteration $c_f(p)$ is set to the minimum of $c_f(u,v)$ which is an integer by the induction hypothesis. Lines 7 and 8 compute $(u,v).f$ or $(v,u).f$. Either way, these the the sum or difference of integers by assumption, so after the $(n+1)^{st}$ iteration we have that $(u,v).f$ is an integer for all $(u,v) \in E$. Since the value of the flow is a sum of flows of edges, we must have $|f| \in \mathbb{Z}$ as well.

Exercise 26.3-3

The length of an augmenting path can be at most $2 \min\{|L|, |R|\} + 1$. To

see that this is the case, we can construct an example which has an augmenting path of that length.

Suppose that the vertices of L are $\{\ell_1, \ell_2, \ldots \ell_{|L|}\}$, and of R are $\{r_1, r_2, \ldots, r_{|R|}\}$. For convenience, we will call $m = \min\{|L|, |R|\}$. Then, we will place the edges

$$\{(\ell_m, r_m - 1), (\ell_1, r_1), (\ell_1, r_m)\} \cup \left(\cup_{i=2}^{i=m-1}\{(\ell_i, r_i), (\ell_i, r_{i-1})\}\right)$$

Then, after augmenting with the shortest length path $\min\{|L|, |R|\} - 1$ times, we could end up having sent a unit flow along $\{(\ell_i, r_i)\}_{i=1,\ldots,m-1}$. At this point, there is only a single augmenting path, namely, $\{s, \ell_m, r_{m-1}, \ell_{m-1}, r_{m-2}, \ldots, \ell_2, r_1, \ell_1, r_m, t\}$. This path has the length $2m + 1$.

It is clear that any simple path must have length at most $2m + 1$, since the path must start at s, then alternate back and forth between L and R, and then end at t. Since augmenting paths must be simple, it is clear that our bound given for the longest augmenting path is tight.

Exercise 26.3-4

First suppose there exists a perfect matching in G. Then for any subset $A \subseteq L$, each vertex of A is matched with a neighbor in R, and since it is a matching, no two such vertices are matched with the same vertex in R. Thus, there are at least $|A|$ vertices in the neighborhood of A. Now suppose that $|A| \leq |N(A)|$ for all $A \subseteq L$. Run Ford-Fulkerson on the corresponding flow network. The flow is increased by 1 each time an augmenting path is found, so it will suffice to show that this happens $|L|$ times. Suppose the while loop has run fewer than L times, but there is no augmenting path. Then fewer than L edges from L to R have flow 1. Let $v_1 \in L$ be such that no edge from v_1 to a vertex in R has nonzero flow. By assumption, v_1 has at least one neighbor $v'_1 \in R$. If any of v_1's neighbors are connected to t in G_f then there is a path, so assume this is not the case. Thus, there must be some edge (v_2, v_1) with flow 1. By assumption, $N(\{v_1, v_2\}) \geq 2$, so there must exist $v'_2 \neq v'_1$ such that $v'_2 \in N(\{v_1, v_2\})$. If (v'_2, t) is an edge in the residual network we're done since v'_2 must be a neighbor of v_2, so $s, v_1, v'_1, v_2, v'_2, t$ is a path in G_f. Otherwise v'_2 must have a neighbor $v_3 \in L$ such that (v_3, v'_2) is in G_f. Specifically, $v_3 \neq v_1$ since (v_3, v'_2) has flow 1, and $v_3 \neq v_2$ since (v_2, v'_1) has flow 1, so no more flow can leave v_2 without violating conservation of flow. Again by our hypothesis, $N(\{v_1, v_2, v_3\}) \geq 3$, so there is another neighbor $v'_3 \in R$.

Continuing in this fashion, we keep building up the neighborhood v'_i, expanding each time we find that (v'_i, t) is not an edge in G_f. This cannot happen L times, since we have run the Ford-Fulkerson while-loop fewer than $|L|$ times, so there still exist edges into t in G_f. Thus, the process must stop at some vertex v'_k, and we obtain an augmenting path $s, v_1, v'_1, v_2, v'_2, v_3, \ldots, v_k, v'_k, t$, contradicting our assumption that there was no such path. Therefore the while loop runs at least $|L|$ times. By Corollary 26.3 the flow strictly increases each time by f_p. By Theorem 26.10 f_p is an integer. In particular, it is equal to 1. This implies that $|f| \geq |L|$. It is clear that $|f| \leq |L|$, so we must have $|f| = |L|$. By

Corollary 26.11 this is the cardinality of a maximum matching. Since $|L| = |R|$, any maximum matching must be a perfect matching.

Exercise 26.3-5

We convert the bipartite graph into a flow problem by making a new vertex for the source which has an edge of unit capacity going to each of the vertices in L, and a new vertex for the sink that has an edge from each of the vertices in R, each with unit capacity. We want to show that the number of edge between the two parts of the cut is at least L, this would get us by the max-flow-min-cut theorem that there is a flow of value at least $|L|$. The, we can apply the integrality theorem that all of the flow values are integers, meaning that we are selecting $|L|$ disjoint edges between L and R.

To see that every cut must have capacity at lest $|L|$, let S_1 be the side of the cut containing the source and let S_2 be the side of the cut containing the sink. Then, look at $L \cap S_1$. The source has an edge going to each of $L \cap (S_1)^c$, and there is an edge from $R \cap S_1$ to the sink that will be cut. This means that we need that there are at least $|L \cap S_1| - |R \cap S_1|$ many edges going from $L \cap S_1|$ to $R \cap S_2$. If we look at the set of all neighbors of $L \cap S_1$, we get that there must be at least the same number of neighbors in R, because otherwise we could sum up the degrees going from $L \cap S_1$ to R on both sides, and get that some of the vertices in R would need to have a degree higher than d. This means that the number of neighbors of $L \cap S_1$ is at least $L \cap S_1$, but we have that they are in S_1, but there are only $|R \cap S_1|$ of those, so, we have that the size of the set of neighbors of $L \cap S_1$ that are in S_2 is at least $|L \cap S_1| - |R \cap S_1|$. Since each of these neighbors has an edge crossing the cut, we have that the total number of edges that the cut breaks is at least $(|L| - |L \cap S_1|) + (|L \cap S_1| - |R \cap S_1|) + |R \cap S_1| = |L|$. Since each of these edges is unit valued, the value of the cut is at least $|L|$.

Exercise 26.4-1

When we run INITIALIZE-PREFLOW(G,s), $s.e$ is zero prior to the for loop on line 7. Then, for each of the vertices that s has an edge to, we decrease the value of $s.e$ by the capacity of that edge. This means that at the end, $s.e$ is equal to the negative of the sum of all the capacities coming out of s. This is then equal to the negative of the cut value of the cut that puts s on one side, and all the other vertices on the other. The negative of the value of the min cut is larger or equal to the negative of the value of this cut. Since the value of the max flow is the value of the min cut, we have that the negative of the value of the max flow is larger or equal to $s.e$.

Exercise 26.4-2

We must select an appropriate data structure to store all the information which will allow us to select a valid operation in constant time. To do this, we will need to maintain a list of overflowing vertices. By Lemma 26.14, a push or

a relabel operation always applies to an overflowing vertex. To determine which operation to perform, we need to determine whether $u.h = v.h + 1$ for some $v \in N(u)$. We'll do this by maintaining a list $u.high$ of all neighbors of u in G_f which have height greater than or equal to u. We'll update these attributes in the PUSH and RELABEL functions. It is clear from the pseudocode given for PUSH that we can execute it in constant time, provided we have maintain the attributes $\delta_f(u,v)$, $u.e$, $c_f(u,v)$, $(u,v).f$, and $u.h$. Each time we call PUSH(u,v) the result is that u is no longer overflowing, so we must remove it from the list. Maintain a pointer $u.overflow$ to u's position in the overflow list. If a vertex u is not overflowing, set $u.overflow = NIL$. Next, check if v became overflowing. If so, set $v.overflow$ equal to the head of the overflow list. Since we can update the pointer in constant time and delete from a linked list given a pointer to the element to be deleted in constant time, we can maintain the list in $O(1)$. The RELABEL operation takes $O(V)$ because we need to compute the minimum $v.h$ from among all $(u,v) \in E_f$, and there could be $|V| - 1$ many such v. We will also need to update $u.high$ during RELABEL. When RELABEL(u) is called, set $u.high$ equal to the empty list and for each vertex v which is adjacent to u, if $v.h = u.h + 1$, add u to the list $v.high$. Since this takes constant time per adjacent vertex we can maintain the attribute in $O(V)$ per call to relabel.

Exercise 26.4-3

To run RELABEL(u), we need to take the min a number of things equal to the out degree of u (and so taking this min will take time proportional to the out degree). This means that since each vertex will only be relabeled at most $O(|V|)$ many times, the total amount of work is on the order of $|V| \sum_{v \in V} outdeg(v)$. But the sum of all the out degrees is equal to the number of edges, so, we have the previous expression is on the order of $|V||E|$.

Exercise 26.4-4

In the proof of $2 \implies 3$ in Theorem 26.6 we obtain a minimum cut by letting $S = \{v \in V |$ there exists a path from s to v in $G_f\}$ and $T = V - S$. Given a flow, we can form the residual graph in $O(E)$. Then we just need to perform a depth first search to find the vertices reachable from s. This can be done in $O(V + E)$, and since $|E| \geq |V| - 1$ the whole procedure can be carried out in $O(E)$.

Exercise 26.4-5

First, construct the flow network for the bipartite graph as in the previous section. Then, we relabel everything in L. Then, we push from every vertex in L to a vertex in R, so long as it is possible. keeping track of those that vertices of L that are still overflowing can be done by a simple bit vector. Then, we relabel everything in R and push to the last vertex. Once these operations have been done, The only possible valid operations are to relabel the vertices of L

that weren't able to find an edge that they could push their flow along, so could possibly have to get a push back from R to L. This continues until there are no more operations to do. This takes time of $O(V(E + V))$.

Exercise 26.4-6

The number of relabel operations and saturating pushes is the same as before. An edge can handle at most k nonsaturating pushes before it becomes saturated, so the number of nonsaturating pushes is at most $2k|V||E|$. Thus, the total number of basic operations is at most $2|V|^2 + 2|V||E| + 2k|V||E| = O(kVE)$.

Exercise 26.4-7

This won't affect the asymptotic performance, in fact it will improve the bound obtained in lemma 16.20 to be that no vertex will ever have a height more than $2|V| - 3$. Since this lemma was the source of all the bounds later, they carry through, and are actually a little bit (not asymptotically) better (lower).

To see that it won't affect correctness of the algorithm. We notice that the reason that we needed the height to be as high as it was was so that we could consider all the simple paths from s to t. However, when we are done initializing, we have that the only overflowing vertices are the ones for which there is an edge to them from s. Then, we only need to consider all the simple paths from them to t, the longest such one involves $|V| - 1$ vertices, and, so, only $|V| - 2$ different edges, and so it only requires that there are $|V| - 2$ differences in heights, since the set $\{0, 1, \ldots, |V| - 3\}$ has $|V| - 2$ different values, this is possible.

Exercise 26.4-8

We'll prove the claim by induction on the number of push and relabel operations. Initially, we have $u.h = |V|$ if $u = s$ and 0 otherwise. We have $s.h - |V| = 0 \le \delta_f(s, s) = 0$ and $u.h = 0 \le \delta_f(u, t)$ for all $u \ne s$, so the claim holds prior to the first iteration of the while loop on line 2 of the GENERIC-PUSH-RELABEL algorithm. Suppose that the properties have been maintained thus far. If the next iteration is a nonsaturating push then the properties are maintained because the heights and existence of edges in the residual network are preserved. If it is a saturating push then edge (u, v) is removed from the residual network, which increases both $\delta_f(u, t)$ and $\delta_f(u, s)$, so the properties are maintained regardless of the height of u. Now suppose that the next iteration causes a relabel of vertex u. For all v such that $(u, v) \in E_f$ we must have $u.h \le v.h$. Let $v' = \min\{v.h | (u, v) \in E_f\}$. There are two cases to consider. First, suppose that $v'.h < |V|$. Then after the relabeling we have $u.h = 1 + v'.h \le 1 + \min_{(u,v) \in E_f} \delta_f(v, t) = \delta_f(u, t)$. Second, suppose that $v'.h \ge |V|$. Then after relabeling we have $u.h = 1 + v'.h \le 1 + |V| + \min_{(u,v) \in E_f} \delta_f(v, s) = \delta_f(u, s) + |V|$ which implies that $u.h - |V| \le \delta)f(u, s)$. Therefore the GENERIC-PUSH-

RELABEL procedure maintains the desired properties.

Exercise 26.4-9

What we should do is to, for successive backwards neighborhoods of t, relabel everything in that neighborhood. This will only take at most $O(VE)$ time (see 26.4-3). This also has the upshot of making it so that once we are done with it, every vertex's height is equal to the quantity $\delta_f(u,t)$. Then, since we begin with equality, after doing this, the inductive step we had in the solution to the previous exercise shows that this equality is preserved.

Exercise 26.4-10

Each vertex has maximum height $2|V|-1$. Since heights don't decrease, and there are $|V|-2$ vertices which can be overflowing, the maximum contribution of relabels to Φ over all vertices is $(2|V|-1)(|V|-2)$. A saturating push from u to v increases Φ by at most $v.h \le 2|V|-1$, and there are at most $2|V||E|$ saturating pushes, so the total contribution over all saturating pushes to Φ is at most $(2|V|-1)(2|V||E|)$. Since each nonsaturating push decrements Φ by at least on and Φ must equal zero upon termination, we must have that the number of nonsaturating pushes is at most

$$(2|V|-1)(|V|-2)+(2|V|-1)(2|V||E|) = 4|V|^2|E|+2|V|^2-5|V|+3-2|V||E|.$$

Using the fact that $|E| \ge |V|-1$ and $|V| \ge 4$ we can bound the number of saturating pushes by $4|V|^2|E|$.

Exercise 26.5-1

When we initialize the preflow, we have 26 units of flow leaving s. Then, we consider v_1 since it is the first element in the L list. When we discharge it, we increase it's height to 1 so that it can dump 12 of it's excess along its edge to vertex v_3, to discharge the rest of it, it has to increase it's height to $|V|+1$ to discharge it back to s. It was already at the front, so, we consider v_2. We increase its height to 1. Then, we send all of its excess along its edge to v_4. We move it to the front, which means we next consider v_1, and do nothing because it is not overflowing. Up next is vertex v_3. After increasing its height to 1, it can send all of its excess to t. This puts v_3 at the front, and we consider the non-overflowing vertices v_2 and v_1. Then, we consider v_4, it increases its height to 1, then sends 4 units to t. Since it still has an excess of 10 units, it increases its height once again. Then it becomes valid for it to send flow back to v_2 or to v_3. It considers v_4 first because of the ordering of its neighbor list. This means that 10 units of flow are pushed back to v_2. Since $v_4.h$ increased, it moves to the front of the list Then, we consider v_2 since it is the only still overflowing vertex. We increase its height to 3. Then, it is overflowing by 10 so it increases its height to 3 to send 6 units to v_4. It's height increased so it goes to the front

of the list. Then, we consider v_4, which is overflowing. it increases its height to 3, then it sends 6 units to v_3. Again, it goes to the front of the list. Up next is v_2 which is not overflowing, v_3 which is, so it increases it's height by 1 to send 4 units of flow to t. Then sends 2 units to v_4 after increasing in height. The excess flow keeps bobbing around the four vertices, each time requiring them to increase their height a bit to discharge to a neighbor only to have that neighbor increase to discharge it back until v_2 has increased in height enough to send all of it's excess back to s, this completes and gives us a maximum flow of 23.

Exercise 26.5-2

Initially, the vertices adjacent to s are the only ones which are overflowing. The implementation is as follows:

Algorithm 1 PUSH-RELABEL-QUEUE(G,s)

1: INITIALIZE-PREFLOW(G, s)
2: Initialize an empty queue q
3: **for** $v \in G.Adj[s]$ **do**
4:      $q.push(v)$
5: **end for**
6: **while** $q.head \neq NIL$ **do**
7:      DISCHARGE($q.head$)
8:      $q.pop()$
9: **end while**

Note that we need to modify the DISCHARGE algorithm to push vertices v onto the queue if v was not overflowing before a discharge but is overflowing after one. Between lines 7 and 8 of DISCHARGE(u), add the line "**if** $v.e > 0$, $q.push(v)$." This is an implementation of the generic push-relabel algorithm, so we know it is correct. The analysis of runtime is almost identical to that of Theorem 26.30. We just need to verify that there are at most $|V|$ calls to DISCHARGE between two consecutive relabel operations. Observe that after calling PUSH(u, v), Corollary 26.28 tells us that no admissible edges are entering v. Thus, once v is put into the queue because of the push, it can't be added again until it has been relabeled. Thus, at most $|V|$ vertices are added to the queue between relabel operations.

Exercise 26.5-3

If we change relabel to just increment the value of u, we will not be ruining the correctness of the Algorithm. This is because since it only applies when $u.h \leq v.h$, we won't be every creating a graph where h ceases to be a height function, since $u.h$ will only ever be increasing by exactly one whenever relabel is called, ensuring that $u.h + 1 \leq v.h$. This means that Lemmatae 26.15 and 26.16 will still hold. Even Corollary 26.21 holds since all it counts on is that

relabel causes some vertex's h value to increase by at least one, it will still work when we have all of the operations causing it to increase by exactly one. However, Lemma 26.28 will no longer hold. That is, it may require more than a single relabel operation to cause an admissible edge to appear, if for example, $u.h$ was strictly less than the h values of all its neighbors. However, this lemma is not used in the proof of Exercise 26.4-3, which bounds the number of relabel operations. Since the number of relabel operations still have the same bound, and we know that we can simulate the old relabel operation by doing (possibly many) of these new relabel operations, we have the same bound as in the original algorithm with this different relabel operation.

Exercise 26.5-4

We'll keep track of the heights of the overflowing vertices using an array and a series of doubly linked lists. In particular, let A be an array of size $|V|$, and let $A[i]$ store a list of the elements of height i. Now we create another list L, which is a list of lists. The head points to the list containing the vertices of highest height. The next pointer of this list points to the next nonempty list stored in A, and so on. This allows for constant time insertion of a vertex into A, and also constant time access to an element of largest height, and because all lists are doubly linked, we can add and delete elements in constant time. Essentially, we are implementing the algorithm of Exercise 26.5-2, but with the queue replaced by a priority queue with constant time operations. As before, it will suffice to show that there are at most $|V|$ calls to discharge between consecutive relabel operations.

Consider what happens when a vertex v is put into the priority queue. There must exist a vertex u for which we have called PUSH(u,v). After this, no admissible edge is entering v, so it can't be added to the priority queue again until after a relabel operation has occurred on v. Moreover, every call to DISCHARGE terminates with a PUSH, so for every call to DISCHARGE there is another vertex which can't be added until a relabel operation occurs. After $|V|$ DISCHARGE operations and no relabel operations, there are no remaining valid PUSH operations, so either the algorithm terminates, or there is a valid relabel operation which is performed. Thus, there are $O(V^3)$ calls to DISCHARGE. By carrying out the rest of the analysis of Theorem 26.30, we conclude that the runtime is $O(V^3)$.

Exercise 26.5-5

Suppose to try and obtain a contradiction that there were some minimum cut for which a vertex that had $v.h > k$ were on the sink side of that cut. For that minimum cut, there is a residual flow network for which that cut is saturated. Then, if there were any vertices that were also on the sink side of the cut which had an edge going to v in this residual flow network, since it's h value cannot be equal to k, we know that it must be greater than k since it could be only at most one less than v. We can continue in this way to let S be the set

of vertices on the sink side of the graph which have an h value greater than k. Suppose that there were some simple path from a vertex in S to s. Then, at each of these steps, the height could only decrease by at most 1, since it cannot get from above k to 0 withour going through k, we know that there is no path in the residual flow network going from a vertex in S to s. Since a minimal cut corresponds to disconnected parts of the residual graph for a maximum flow, and we know there is no path from S to s, there is a minimum cut for which S lies entirely on the source side of the cut. This was a contradiction to how we selected v,and so have shown the first claim.

Now we show that after updating the h values as suggested, we are still left with a height function. Suppose we had an edge (u, v) in the residual graph. We knew from before that $u, h \leq v.h + 1$. However, this means that if $u.h > k$, so must be $v.h$. So, if both were above k, we would be making them equal, causing the inequality to still hold. Also, if just $v.k$ were above k, then we have not decreased it's h value, meaning that the inequality also still must hold. Since we have not changed the value of $s.h$, and $t.h$, we have all the required properties to have a height function after modifying the h values as described.

Problem 26-1

a. This problem is identical to exercise 26.1-7.

b. Construct a vertex constrained flow network from the instance of the escape problem by letting our flow network have a vertex (each with unit capacity) for each intersection of grid lines, and have a bidirectional edge with unit capacity for each pair of vertices that are adjacent in the grid. Then, we will put a unit capacity edge going from s to each of the distinguished vertices, and a unit capacity edge going from each vertex on the sides of the grid to t. Then, we know that a solution to this problem will correspond to a solution to the escape problem because all of the augmenting paths will be a unit flow, because every edge has unit capacity. This means that the flows through the grid will be the paths taken. This gets us the escaping paths if the total flow is equal to m (we know it cannot be greater than m by looking at the cut which has s by itself). And, if the max flow is less than m, we know that the escape problem is not solvable, because otherwise we could construct a flow with value m from the list of disjoint paths that the people escaped along.

Problem 26-2

a. Set up the graph G' as defined in the problem, give each edge capacity 1, and run a maximum-flow algorithm. I claim that if (x_i, y_j) has flow 1 in the maximum flow and we set (i, j) to be an edge in our path cover, then the result is a minimum path cover. First observe that no vertex appears twice in the same path. If it did, then we would have $f(x_i, y_j) = f(x_k, y_j)$ for some $i \neq k \neq j$. However, this contradicts the conservation of flow, since the

15

capacity leaving y_j is only 1. Moreover, since the capacity from s to x_i is 1, we can never have two edges of the form (x_i, y_j) and (x_i, y_k) for $k \neq j$. We can ensure every vertex is included in some path by asserting that if there is no edge (x_i, y_j) or $(x_j, y_i$ for some j, then j will be on a path by itself. Thus, we are guaranteed to obtain a path cover. If there are k paths in a cover of n vertices, then they will consist of $n - k$ edges in total. Given a path cover, we can recover a flow by assigning edge (x_i, y_j) flow 1 if and only if (i, j) is an edge in one of the paths in the cover. Suppose that the maximum flow algorithm yields a cover with k paths, and hence flow $n - k$, but a minimum path cover uses strictly fewer than k paths. Then it must use strictly more than $n - k$ edges, so we can recover a flow which is larger than the one previously found, contradicting the fact that the previous flow was maximal. Thus, we find a minimum path cover. Since the maximum flow in the graph corresponds to finding a maximum matching in the bipartite graph obtained by considering the induced subgraph of G' on $\{1, 2, \ldots, n\}$, section 26.3 tells us that we can find a maximum flow in $O(VE)$.

b. This doesn't work for directed graphs which contain cycles. To see this, consider the graph on $\{1, 2, 3, 4\}$ which contains edges $(1, 2), (2, 3), (3, 1)$, and $(4, 3)$. The desired output would be a single path 4,3,1,2 but flow which assigns edges (x_1, y_2), (x_2, y_3), and (x_3, y_1) flow 1 is maximal.

Problem 26-3

a. Suppose to a contradiction that there were some $J_i \in T$, and some $A_k \in R_i$ so that $A_k \notin T$. However, by the definition of the flow network, there is an edge of infinite capacity going from A_k to J_i because $A_k \in R_i$. This means that there is an edge of infinite capacity that is going across the given cut. This means that the capacity of the cut is infinite, a contradiction to the given fact that the cut was finite capacity.

b. Though tempting, it doesn't suffice to just look at the experts that are on the s side of the cut. To see why this doesn't work, imagine there's one specialized skill areal, such as "Computer power switch operator', that is required for every job. Then, any finite cut that would include any job getting done would requiring that this expert be hired. However, since there is an infinite capacity edge coming from him to every other job, then all of the experts need for all the other jobs would also need to be hired. So, if we have this obiquitously required employee, any minimum cut would have to be all or nothing, but it is trivial to find a counterexample to this being optimal.

In order for this problem to be solvable, one must assume that for every expert you've hired, you do all of the jobs that he is required for. If this is the case, then let $S_k \subseteq [n]$ be the indices of the experts that lie on the source side of the cut, and let $S_i \subseteq [m]$ be the indices of jobs that lie on the source side of the cut, then the net revenue is just

$$\sum_{S_i} p_i - \sum_{S_k} c_k$$

To see this is minimum, transferring over some set of experts and tasks from the sink side to the source side causes the capacity to go down by the cost of those experts and go up by the revenue of those jobs. If the cut was minimal than this must be a positive change, so the revenue isn't enough to justify the hire, meaning that those jobs that were on the source side in the minimal cut are exactly the jobs to attempt.

c. Again, to get a solution, we must make the assumption that for every expert that is hired, all jobs that that expert is required for must be completed. Basically just run either the $O(V^3)$ relabel-to-front algorithm described in section 26.5 on the flow network, and hire the experts that are on the source side of the cut. By the previous part, we know that this gets us the best outcome. The number of edges in the flow network is m+n+r, and the number of vertices is 2+m+n, so the runtime is just $O((2+m+n)^3)$, so it's cubic in $max(m,n)$. There is no dependence on R using this algorithm, but this is reasonable since we have the inherent bound that $r < mn$, which is a lower order term.

Without this unstated assumption, I suspect that there isn't an efficient solution possible, but cannot think of what NP-complete problem you would use for the reduction.

(Of course if he just needed experts in topics contained in this book, he could of just hired either Michelle or me)

Problem 26-4

a. If there exists a minimum cut on which (u, v) doesn't lie then the maximum flow can't be increased, so there will exist no augmenting path in the residual network. Otherwise it does cross a minimum cut, and we can possibly increase the flow by 1. Perform one iteration of Ford-Fulkerson. If there exists an augmenting path, it will be found and increased on this iteration. Since the edge capacities are integers, the flow values are all integral. Since flow strictly increases, and by an integral amount each time, a single iteration of the while loop of line 3 of Ford-Fulkerson will increase the flow by 1, which we know to be maximal. To find an augmenting path we use a BFS, which runs in $O(V + E') = O(V + E)$.

b. If the edge's flow was already at least 1 below capacity then nothing changes. Otherwise, find a path from s to t which contains (u, v) using BFS in $O(V + E)$. Decrease the flow of every edge on that path by 1. This decreases total flow by 1. Then run one iteration of the while loop of Ford-Fulkerson in $O(V + E)$. By the argument given in part a, everything is integer valued and flow strictly increases, so we will either find no augmenting path, or will increase the flow by 1 and then terminate.

Problem 26-5

a. Since the capacity of a cut is the sum of the capacity of the edges going from a vertex on one side to a vertex on the other, it is less than or equal to the sum of the capacities of all of the edges. Since each of the edges has a capacity that is $\leq C$, if we were to replace the capacity of each edge with C, we would only be potentially increasing the sum of the capacities of all the edges. After so changing the capacities of the edges, the sum of the capacities of all the edges is equal to $C|E|$, potentially an overestimate of the original capacity of any cut, and so of the minimum cut.

b. Since the capacity of a path is equal to the minimum of the capacities of each of the edges along that path, we know that any edges in the residual network that have a capacity less than K cannot be used in such an augmenting path. Similarly, so long as all the edges have a capacity of at least K, then the capacity of the augmenting path, if it is found, will be of capacity at least K. This means that all that needs be done is remove from the residual network those edges whose capacity is less than K and then run BFS.

c. Since K starts out as a power of 2, and through each iteration of the while loop on line 4, it decreases by a factor of two until it is less than 1. There will be some iteration of that loop when $K = 1$. During this iteration, we will be using any augmenting paths of capacity at least 1 when running the loop on line 5. Since the original capacities are all integers, the augmenting paths at each step will be integers, which means that no augmenting path will have a capacity of less than 1. So, once the algorithm terminates, there will be no more augmenting paths since there will be no more augmenting paths of capacity at least 1.

d. Each time line 4 is executed we know that there is no augmenting path of capacity at least $2K$. To see this fact on the initial time that line 4 is executed we just note that $2K = 2 \cdot 2^{\lfloor \lg(C) \rfloor} > 2 \cdot 2^{\lg(C)-1} = 2^{\lg(C)} = C$. Then, since an augmenting path is limited by the capacity of the smallest edge it contains, and all the edges have a capacity at most C, no augmenting path will have a capacity greater than that. On subsequent times executing line 4, the loop of line 5 during the previous execution of the outer loop will of already used up and capacious augmenting paths, and would only end once there are no more.

Since any augmenting path must have a capacity of less than $2K$, we can look at each augmenting path p, and assign to it an edge e_p which is any edge whose capacity is tied for smallest among all the edges along the path. Then, removing all of the edges e_p would disconnect the residual network since every possible augmenting path goes through one of those edge. We know that there are at most $|E|$ of them since they are a subset of the edges. We also know that each of them has capacity at most $2K$ since that was the

18

value of the augmenting path they were selected to be tied for cheapest in. So, the total cost of this cut is $2K|E|$.

e. Each time that the inner while loop runs, we know that it adds an amount of flow that is at least K, since that's the value of the augmenting path. We also know that before we start that while loop, there is a cut of cost $\leq 2K|E|$. This means that the most flow we could possibly add is $2K|E|$. Combining these two facts, we get that the most cuts possible is $\frac{2K|E|}{K} = 2|E| \in O(|E|)$.

f. We only execute the outermost for loop $\lg(C)$ many times since $\lg(2^{\lfloor \lg(C) \rfloor}) \leq \lg(C)$. The inner while loop only runs $O(|E|)$ many times by the previous part. Finally, every time the inner for loop runs, the operation it does can be done in time $O(|E|)$ by part b. Putting it all together, the runtime is $O(|E|^2 \lg(C))$.

Problem 26-6

a. Suppose M is a matching and P is an augmenting path with respect to M. Then P consists of k edges in M, and $k+1$ edges not in M. This is because the first edge of P touches an unmatched vertex in L, so it cannot be in M. Similarly, the last edge in P touches an unmatched vertex in R, so the last edge cannot be in M. Since the edges alternate being in or not in M, there must be exactly one more edge not in M than in M. This implies that $|M \oplus P| = |M| + |P| - 2k = |M| + 2k + 1 - 2k = |M| + 1$ since we must remove each edge of M which is in P from both M and P. Now suppose $P_1, P_2, \ldots, P_k$ are vertex-disjoint augmenting paths with respect to M. Let k_i be the number of edges in P_i which are in M, so that $|P_i| = 2k + i + 1$. Then we have

$$M \oplus (P_1 \cup P_2 \cup \ldots \cup P_k) = |M| + |P_1| + \ldots + |P_k| - 2k_1 - 2k_2 - \ldots - 2k_k = |M| + k.$$

To see that we in fact get a matching, suppose that there was some vertex v which had at least 2 incident edges e and e'. They cannot both come from M, since M is a matching. They cannot both come from P since P is simple and every other edge of P is removed. Thus, $e \in M$ and $e' \in P \setminus M$. However, if $e \in M$ then $e \in P$, so $e \notin M \oplus P$, a contradiction. A similar argument gives the case of $M \oplus (P_1 \cup \ldots \cup P_k)$.

b. Suppose some vertex in G' has degree at least 3. Since the edges of G' come from $M \oplus M^*$, at least 2 of these edges come from the same matching. However, a matching never contains two edges with the same endpoint, so this is impossible. Thus every vertex has degree at most 2, so G' is a disjoint union of simple paths and cycles. If edge (u, v) is followed by edge (z, w) in a simple path or cycle then we must have $v = z$. Since two edges with the same endpoint cannot appear in a matching, they must belong alternately to M and M^*. Since edges alternate, every cycle has the same number of edges

19

in each matching and every path has at most one more edge in one matching than in the other. Thus, if $|M| \le |M^*|$ there must be at least $|M^*| - |M|$ vertex-disjoint augmenting paths with respect to M.

c. Every vertex matched by M must be incident with some edge in M'. Since P is augmenting with respect to M', the left endpoint of the first edge of P isn't incident to a vertex touched by an edge in M'. In particular, P starts at a vertex in L which is unmatched by M since every vertex of M is incident with an edge in M'. Since P is vertex disjoint from $P_1, P_2, \ldots, P_k$, any edge of P which is in M' must in fact be in M and any edge of P which is not in M' cannot be in M. Since P has edges alternately in M' and $E - M'$, P must in fact have edges alternately in M and $E - M$. Finally, the last edge of P must be incident to a vertex in R which is unmatched by M'. Any vertex unmatched by M' is also unmatched by M, so P is an augmenting path for M. P must have length at least l since l is the length of the shortest augmenting path with respect to M. If P had length exactly l then this would contradict the fact that $P_1 \cup \ldots \cup P_k$ is a maximal set of vertex disjoint paths of length l because we could add P to the set. Thus P has more than l edges.

d. Any edge in $M \oplus M'$ is in exactly one of M or M'. Thus, the only possible contributing edges from M' are from $P_1 \cup \ldots \cup P_k$. An edge from M can contribute if and only if it is not in exactly one of M and $P_1 \cup \ldots \cup P_k$, which means it must be in both. Thus, the edges from M are redundant so $M \oplus M' = (P_1 \cup \ldots \cup P_k)$ which implies $A = (P_1 \cup \ldots \cup P_k) \oplus P$.

Now we'll show that P is edge disjoint from each P_i. Suppose that an edge e of P is also an edge of P_i for some i. Since P is an augmenting path with respect to M', either $e \in M'$ or $e \in E - M'$. Suppose $e \in M'$. Since P is also augmenting with respect to M, we must have $e \in M$. However, if e is in M and M' then e cannot be in any of the P_i's by the definition of M'. Now suppose $e \in E - M'$. Then $e \in E - M$ since P is augmenting with respect to M. Since e is an edge of P_i, $e \in E - M'$ implies that $e \in M$, a contradiction.

Since P has edges alternately in M' and $E - M'$ and is edge disjoint from $P_1 \cup \ldots \cup P_k$, P is also an augmenting path for M, which implies $|P| \ge l$. Since every edge in A is disjoint we conclude that $|A| \ge (k+1)l$.

e. Suppose M^* is a matching with strictly more than $|M| + |V|/(l+1)$ edges. By part b there are strictly more than $|V|/(l+1)$ vertex-disjoint augmenting paths with respect to M. Each one of these contains at least l edges, so it is incident on $l + 1$ vertices. Since the paths are vertex disjoint, there are strictly more than $|V|(l+1)/(l+1)$ distinct vertices incident with these paths, a contradiction. Thus, the size of the maximum matching is at most $|M| + |V|/(l+1)$.

f. Consider what happens after iteration number $\sqrt{|V|}$. Let M^* be a maximal matching in G. Then $|M^*| \ge |M|$ so by part b, $M \oplus M^*$ contains at least $|M^*| - |M|$ vertex disjoint augmenting paths with respect to M. By part

c, each of these is also a an augmenting path for M. Since each has length $\sqrt{|V|}$, there can be at most $\sqrt{|V|}$ such paths, so $|M^*| - |M| \leq \sqrt{|V|}$. Thus, only $\sqrt{|V|}$ additional iterations of the repeat loop can occur, so there are at most $2\sqrt{|V|}$ iterations in total.

g. For each unmatched vertex in L we can perform a modified BFS to find the length of the shortest path to an unmatched vertex in R. Modify the BFS to ensure that we only traverse an edge if it causes the path to alternate between an edge in M and an edge in $E - M$. The first time an unmatched vertex in R is reached we know the length k of a shortest augmenting path. We can use this to stop our search early if at any point we have traversed more than that number of edges. To find disjoint paths, start at the vertices of R which were found at distance k in the BFS. Run a DFS backwards from these, which maintains the property that the next vertex we pick has distance one fewer, and the edges alternate between being in M and $E - M$. As we build up a path, mark the vertices as used so that we never traverse them again. This takes $O(E)$, so by part f the total runtime is $O(\sqrt{V}E)$.

Chapter 27

Michelle Bodnar, Andrew Lohr

April 12, 2016

Exercise 27.1-1

This modification is not going to affect the asymptotic values of the span work or parallelism. All it will do is add an amount of overhead that wasn't there before. This is because as soon as the $FIB(n-2)$ is spawned the spawning thread just sits there and waits, it does not accomplish any work while it is waiting. It will be done waiting at the same time as it would of been before because the $FIB(n-2)$ call will take less time, so it will still be limited by the amount of time that the $FIN(n-1)$ call takes.

Exercise 27.1-2

The computation dag is given in the image below. The blue numbers by each strand indicate the time step in which it is executed. The work is 29, span is 10, and parallelism is 2.9.


P-FIB(5) Computation DAG
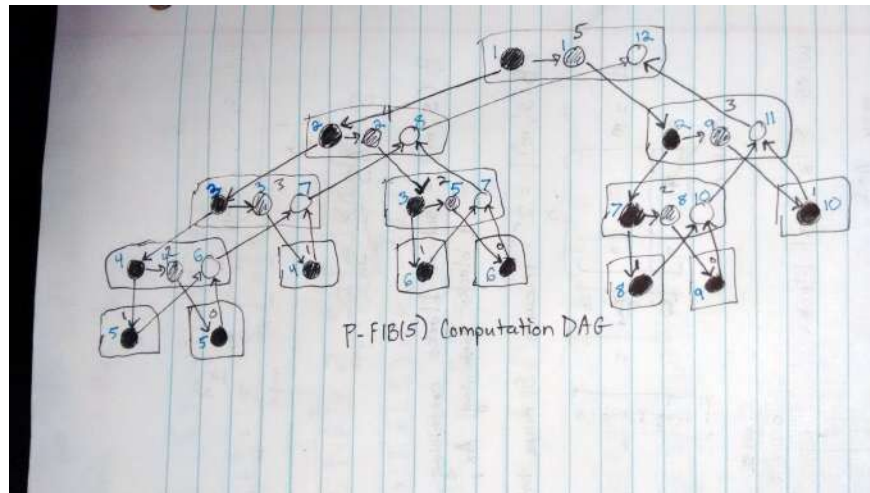
Exercise 27.1-3

Suppose that there are x incomplete steps in a run of the program. Since each of these steps causes at least one unit of work to be done, we have that there is at most $(T_1 - x)$ units of work done in the complete steps. Then, we suppose by contradiction that the number of complete steps is strictly greater than $\lfloor (T_1 - x)/P \rfloor$. Then, we have that the total amount of work done during the complete steps is $P \cdot (\lfloor (T_1 - x)/P \rfloor + 1) = P \lfloor (T_1 - x)/P \rfloor + P = (T_1 - x) - ((T_1 - x) \bmod P) + P > T_1 - x$. This is a contradiction because there are only $(T_1 - x)$ units of work done during complete steps, which is less than the amount we would be doing. Notice that since T_∞ is abound on the total number of both kinds of steps, it is a bound on the number of incomplete steps, x , so,

$$T_P \leq \lfloor (T_1 - x)/P \rfloor + x \leq \lfloor (T_1 - T_\infty)/P \rfloor + T_\infty$$

Where the second inequality comes by noting that the middle expression, as a function of x is monotonically increasing, and so is bounded by the largest value of x that is possible, namely T_∞.

Exercise 27.1-4

The computation is given in the image below. Let vertex u have degree k, and assume that there are m vertices in each vertical chain. Assume that this is executed on k processors. In one execution, each strand from among the k on the left is executed concurrently, and then the m strands on the right are executed one at a time. If each strand takes unit time to execute, then the total computation takes $2m$ time. On the other hand, suppose that on each time step of the computation, $k - 1$ strands from the left (descendants of u) are executed, and one from the right (a descendant of v), is executed. If each strand take unit time to executed, the total computation takes $m + m/k$. Thus, the ratio of times is $2m/(m + m/k) = 2/(1 + 1/k)$. As k gets large, this approaches 2 as desired.
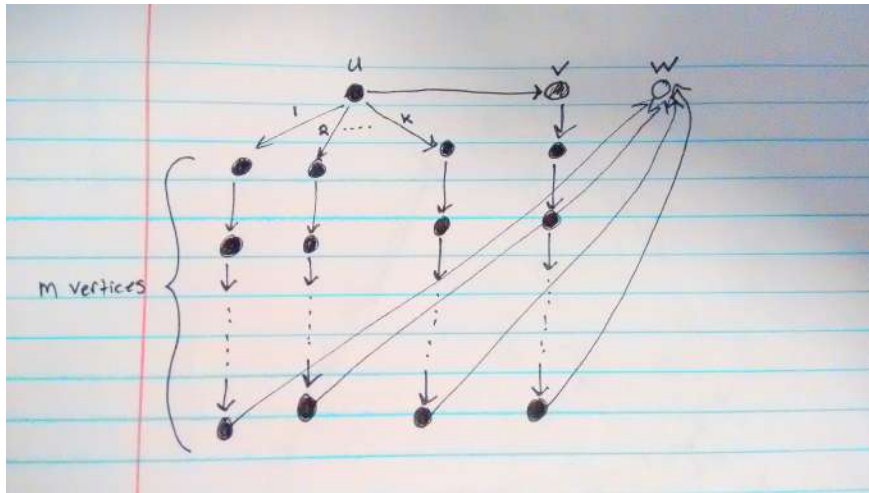
Exercise 27.1-5

The information from T_{10} applied to equation (27.5) give us that

$$42 \leq T_1 - T_\infty 10 + T_\infty$$

which tell us that

$$420 \leq T_1 + 9T_\infty$$

Subtracting these two equations, we have that $100 \leq 8T_\infty$.

If we apply the span law to $T_6 4$, we have that $10 \geq T_\infty$. Applying the work law to our measurement for T_4 gets us that $320 \geq T_1$. Now, looking at the result of applying (27.5) to the value of $T_1 0$, we get that

$$420 \leq T_1 + 9T_\infty \leq 320 + 90 = 410$$

a contradiction. So, one of the three numbers for runtimes must be wrong. However, computers are complicated things, and its difficult to pin down what can affect runtime in practice. It is a bit harsh to judge professor Karan too poorly for something that may of been outside her control (maybe there was just a garbage collection happening during one of the measurements, throwing it off).

Exercise 27.1-6

We'll parallelize the for loop of lines 6-7 in a way which won't incur races. With the algorithm $P - PROD$ given below, it will be easy to rewrite the code. For notation, let a_i denote the ith row of the matrix A.

Algorithm 1 P-PROD(a,x,j,j')

1: **if** $j == j'$ **then**
2:     **return** $a[j] \cdot x[j]$
3: **end if**
4: mid $= \left\lfloor \frac{j+j'}{2} \right\rfloor$
5: a' = spawn P-PROD(a,x,j,mid)
6: x' = P-PROD(a,x,mid+1,j')
7: sync
8: **return** a'+x'

Exercise 27.1-7

The work is unchanged from the serial programming case. Since it is flipping $\Theta(n^2)$ many entries, it does $\Theta(n^2)$ work. The span of it is $\Theta(\lg(n))$ this is because each of the parallel for loops can have its children spawned in time $\lg(n)$, so the total time to get all of the constant work tasks spawned is $2\lg(n) \in \Theta(\lg)$.

Algorithm 2 MAT-VEC(A,x)

1: $n = A.rows$
2: let y be a new vector of length n
3: **parallel for** $i = 1$ to n **do**
4:     $y_i = 0$
5: **end**
6: **parallel for** $i = 1$ to n **do**
7:     $y_i = \text{P-PROD}(a_i, x, 1, n)$
8: **end**
9: **return** y

Since the work of each task is $o(\lg(n))$, that doesn't affect the T_∞ runtime. The parallelism is equal to the work over the span, so it is $\Theta(n^2/\lg(n))$.

Exercise 27.1-8

The work is $\Theta(1 + \sum_{j=2}^{n} j - 1) = \Theta(n^2)$. The span is $\Theta(n)$ because in the worst case when $j = n$, the for-loop of line 3 will need to execute n times. The parallelism is $\Theta(n^2)/\Theta(n) = \Theta(n)$.
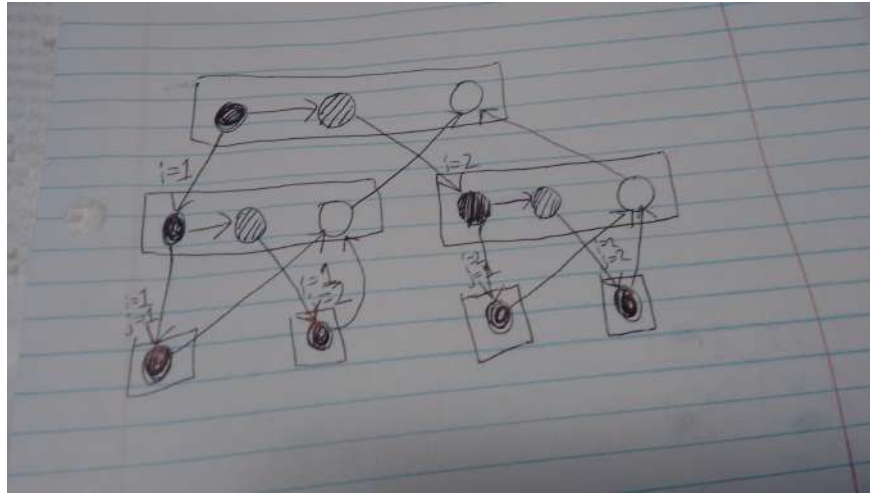
Exercise 27.1-9

We solve for P in the following equation obtained by setting $T_P = T'_P$.

$$\frac{T_1}{P} + T_\infty = \frac{T'_1}{P} + T'_\infty$$
$$\frac{2048}{P} + 1 = \frac{1024}{P} + 8$$
$$\frac{1024}{P} = 7$$
$$\frac{1024}{7} = P$$

So we get that there should be approximately 146 processors for them to have the same runtime.
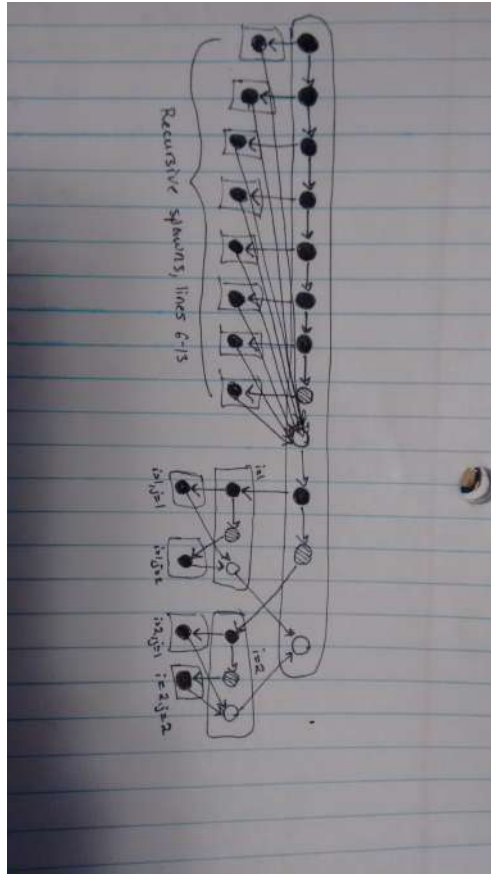
Exercise 27.2-1

See the computation dag in the image below. Assuming that each strand takes unnit time, the work is 13, the span is 6, and the parallelism is $\frac{13}{6}$

Exercise 27.2-2

See the computation dag in the image below. Assuming each strand takes unit time, the work is 30, the span is 16, and the parallelism is $\frac{15}{8}$.

Exercise 27.2-3

We perform a modification of the P-SQUARE-MATRIX-MULTIPLY algorithm. Basically, as hinted in the text, we will parallelize the innermost for loop in such a way that there aren't any data races formed. To do this, we will just define a parallelized dot product procedure. This means that lines 5-7 can be replaced by a single call to this procedure. P-DOT-PRODUCT computes the dot dot product of the two lists between the two bounds on indices.

Using this, we can use this to modify P-SQUARE-MATRIX-MULTIPLY

Since the runtime of the inner loop is $O(\lg(n))$, which is the depth of the recursion. Since the paralel for loops also take $O(\lg(n))$ time. So, since the runtimes are additive here, the total span of this procedure is $\Theta(\lg(n))$. The total work is still just $O(n^3)$ Since all the spawning and recursing call be replaced with the normal serial version once there aren't enough free processors to handle all of the spawned calls to P-DOT-PRODUCT.

Exercise 27.2-4

Algorithm 3 P-DOT-PROD(v,w,low,high)

---

  if low == high **then**
    return v[low] = v[low]
  end if
  mid = $\left\lfloor \frac{low+high}{2} \right\rfloor$
  x = spawn P-DOT-PROD(v,w,low,mid)
  y = P-DOT-PROD(v,w,mid+1,high)
  sync
  return x+y

Algorithm 4 MODIFIED-P-SQUARE-MATRIX-MULTIPLY

---

  n = A.rows
  let C be a new $n \times n$ matrix
  parallel for i=1 to n **do**
    parallel for j=1 to n **do**
      $c_{i,j}$ = P-DOT-PROD($A_{i,.}, B_{.,j}, 1, n$)
    end
  end
  return C

Assume that the input is two matrices A and B to be multiplied. For this algorithm we use the function P-PROD defined in exercise 21.7-6. For notation, we let A_i denote the ith row of A and A_i' denote the ith column of A. Here, C is assumed to be a p by r matrix. The work of the algorithm is $\Theta(prq)$, since this is the runtime of the serialization. The span is $\Theta(\log(p) + \log(r) + \log(q)) = \Theta(\log(pqr))$. Thus, the parallelism is $\Theta(pqr/\log(pqr))$, which remains highly parallel even if any of p, q, or r are 1.

Algorithm 5 MATRIX-MULTIPLY(A,B,C,p,q,r)

1: **parallel for** $i = 1$ to p **do**
2:   **parallel for** $j = 1$ to r **do**
3:     C_{ij} = P-PROD($A_i, B_j', 1, q$)
4:   **end**
5: **end**
6: **return** C

Exercise 27.2-5

Split up the region into four sections. Then, this amounts to finding the transpose the upper left and lower right of the two submatrices. In addition to that, you also need to swap the elements in the upper right with their transpose position in the lower left. This dealing with the upper right swapping only takes

time $O(\lg(n^2)) = O(\lg(n))$. In addition, there are two subproblems, each of half the size. This gets us the recursion:

$$T_\infty(n) = T_\infty(n/2) + \lg(n)$$

By the master theorem, we get that the total span of this procedure is $T_\infty \in O(\lg(n))$. The total work is still the usual $O(n^2)$.

Exercise 27.2-6

Since D^k cannot be computed without D^{k-1} we cannot parallelize the for loop of line 3 of Floyd-Warshall. However, the other two loops can be parallelized. The work is $\Theta(n^2)$, as in the serial case. The span is $\Theta(n \lg n)$. Thus, the parallelism is $\Theta(n/\lg n)$. The algorithm is as follows:

Algorithm 6 P-FLOYD-WARSHALL(W)

1: $n = W.rows$
2: $D^{(0)} = W$
3: **for** $k = 1$ to n **do**
4:      let $D^{(k)} = (d_{ij}^{(k)})$ be a new $n \times n$ matrix
5:      **parallel for** $i = 1$ to n **do**
6:          **parallel for** $j = 1$ to n **do**
7:              $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
8:          **end**
9:      **end**
10: **end for**
11: **return** $D^{(n)}$

Exercise 27.3-1

To coarsen the base case of P-MERGE, just replace the condition on line 2 with a check that $n < k$ for some base case size k. And instead of just copying over the particular element of A to the right spot in B, you would call a serial sort on the remaining segment of A and copy the result of that over into the right spots in B.

Exercise 27.3-2

By a slight modification of exercise 9.3-8 we can find we can find the median of all elements in two sorted arrays of total length n in $O(\lg n)$ time. We'll modify P-MERGE to use this fact. Let MEDIAN(T, p_1, r_1, p_2, r_2) be the function which returns a pair, q, where $q.pos$ is the position of the median of all the elements T which lie between positions p_1 and r_1, and between positions p_2 and r_2, and $q.arr$ is 1 if the position is between p_1 and r_1, and 2 otherwise. The first 8 lines of code are identical to those in P-MERGE given on page 800, so

we omit them here.

Algorithm 7 P-MEDIAN-MERGE($T, p_1, r_1, p_2, r_2, A, p_3$)

1: Run lines 1 through 8 of P-MERGE
2: $q = \text{MEDIAN}(T, p_1, r_1, p_2, r_2)$
3: **if** $q.arr == 1$ **then**
4:      $q_2 = \text{BINARY-SEARCH}(T[q.pos]), T, p_2, r_2)$
5:      $q_3 = p_3 + q.pos - p_1 + q_2 - p_2$
6:      $A[q_3] = T[q.pos]$
7:      **spawn** P-MEDIAN-MERGE($T, p_1, q.pos - 1, p_2, q_2 - 1, A, p_3$)
8:      P-MEDIAN-MERGE($T, q.pos + 1, r_1, q_2 + 1, r_2, A, p_3$)
9:      **sync**
10: **else**
11:      $q_2 = \text{BINARY-SEARCH}(T[q.pos], T, p_1, r_1)$
12:      $q_3 = p_3 + q.pos - p_2 + q_2 - p_1$
13:      $A[q_3] = T[q.pos]$
14:      **spawn** P-MEDIAN-MERGE($T, p_1, q_2 - 1, p_2, q.pos - 1, A, p_3$)
15:      P-MEDIAN-MERGE($T, q_2 + 1, r_1, q.pos + 1, r_2, A, p_3$)
16:      **sync**
17: **end if**

The work is characterized by the recurrence $T_1(n) = O(\lg n) + 2T_1(n/2)$, whose solution tells us that $T_1(n) = O(n)$. The work is at least $\Omega(n)$ since we need to examine each element, so the work is $\Theta(n)$. The span satisfies the recurrence $T_\infty(n) = O(\lg n) + O(\lg n/2) + T_\infty(n/2) = O(\lg n) + T_\infty(n/2) = \Theta(\lg^2 n)$, by exercise 4.6-2.

Exercise 27.3-3

Suppose that there are c different processors, and the array has length n and you are going to use its last element as a pivot. Then, look at each chunk of size $\lceil \frac{n}{c} \rceil$ of entries before the last element, give one to each processor. Then, each counts the number of elements that are less than the pivot. Then, we compute all the running sums of these values that are returned. This can be done easily by considering all of the subarrays placed along the leaves of a binary tree, and then summing up adjacent pairs. This computation can be done in time $\lg(\min\{c, n\})$ since it's the log of the number of leaves. From there, we can compute all the running sums for each of the subarrays also in logarithmic time. This is by keeping track of the sum of all more left cousins of each internal node, which is found by adding the left sibling's sum vale to the left cousin value of the parent, with the root's left cousin value initiated to 0. This also just takes time the depth of the tree, so is $\lg(\min\{c, n\})$. Once all of these values are computed at the root, it is the index that the subarray's elements less than the pivot should be put. To find the position where the subarray's elements larger

than the root should be put, just put it at twice the sum value of the root minus the left cousin value for that subarray. Then, the time taken is just $O(\frac{n}{c})$. By doing this procedure, the total work is just $O(n)$, and the span is $O(\lg(n))$, and so has parallelization of $O(\frac{n}{\lg(n)})$. This whole process is split across the several algoithms appearing here.

Algorithm 8 PPartition(L)

$c = \min\{c, n\}$
pivot $= \mathrm{L}[n]$
let Count be an array of length c
let $r_1, \ldots r_{c+1}$ be roughly evenly spaced indices to L with $r_1 = 1$ and $r_{c+1} = n$
for i=1 $\ldots$ c **do**
    Count[i] = spawn countnum($L[r_i, r_{i+1} - 1]$,pivot)
end for
sync
let T be a nearly complete binary tree whose leaves are the elements of Count whose vertices have the attributes sum and lc
for all the leaves, let their sum value be the corresponding entry in Count
ComputeSums(T.root)
T.root.lc $= 0$
ComputeCousins(T.root)
Let Target be an array of length n that the elements will be copied into
for i=1 $\ldots$ c **do**
    let cousin be the lc value of the node in T that corresponds to i
    spawn CopyElts(L,Target, cousin,$r_i, r_{i+1} - 1$)
end for
Target[n] = Target[T.root.sum]
Target[T.root.sum] = L[n]
return Target

Algorithm 9 CountNum(L,x)

---

ret $= 0$
for i=1 $\ldots$ L.length **do**
    if $L[i] < x$ **then**
        ret++
    end if
end for
return ret

Exercise 27.3-4

See the algorithm P-RECURSIVE-FFT. it parallelized over the two recursive calls, having a parallel for works because each of the iterations of the for loop

10

Algorithm 10 ComputeSums(v)

---

if v is an internal node **then**
    x = spawn ComputeSums(v.left)
    y = ComputeSums(v.right)
    sync
    v.sum = x+y
end if
return v.sum

Algorithm 11 ComputeCousins(v)

---

if $v \neq NIL$ **then**
    v.lc = v.p.lv
    if $v = v.p.right$ **then**
        v.lc += c.p.left.sum
    end if
    spawn ComputeCousins(v.left)
    ComputeCousins(v.right)
    sync
end if

Algorithm 12 CopyElts(L1, L2, lc,lb,ub)

---

counter1 = lc+1
counter2 = lb
for i=lb ... ub **do**
    if $L1[i] < x$ **then**
        L2[counter1++] = L1[i]
    else
        L2[counter2++] = L1[i]
    end if
end for

---

touch independent sets of variables. The span of the procedure is only $\Theta(\lg(n))$ giving it a parallelization of $\Theta(n)$

Algorithm 13 P-RECURSIVE-FFT(a)

$n = a.length$
if $n == 1$ **then**
    return a
end if
$\omega_n = e^{2\pi i/n}$
$\omega = 1$
$a^{[0]} = (a_0, a_2, \ldots, a_{n-2})$
$a^{[1]} = (a_1, a_3, \ldots, a_{n-1})$
$y^{[0]} = $ spawn P-RECURSIVE-FFT$(a^{[0]})$
$y^{[1]} = $ P-RECURSIVE-FFT$(a^{[1]})$
sync
parallel for $k = 0, \ldots, n/2 - 1$ **do**
    $y_k = y_k^{[0]} + \omega y_k^{[1]}$
    $y_{k+(n/2)} = y_k^{[0]} - \omega y_k^{[1]}$
    $\omega = \omega \omega_n$
end
return y

Exercise 27.3-5

Randomly pick a pivot element, swap it with the last element, so that it is in the correct format for running the procedure described in 27.3-3. Run partition from problem $27.3 - 3$. As an intermediate step, in that procedure, we compute the number of elements less than the pivot (T.root.sum), so keep track of that value after the end of PPartition. Then, if we have that it is less than k, recurse on the subarray that was greater than or equal to the pivot, decreasing the order statistic of the element to be selected by T.root.sum. If it is larger than the order statistic of the element to be selected, then leave it unchanged and recurse on the subarray that was formed to be less than the pivot. A lot of the analysis in section 9.2 still applies, except replacing the timer needed for partitioning with the runtime of the algorithm in problem 27.3-3. The work is unchanged from the serial case because when $c = 1$, the algorithm reduces to the serial algorithm for partitioning. For span, the $O(n)$ term in the equation half way down page 218 can be replaced with an $O(\lg(n))$ term. It can be seen with the substitution method that the solution to this is logarithmic

$$E[T(n)] \leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} C \lg(k) + O(\lg(n)) \leq O(\lg(n))$$

So, the total span of this algorithm will still just be $O(\lg(n))$.

Exercise 27.3-6

Let MEDIAN(A) denote a brute force method which returns the median element of the array A. We will only use this to find the median of small arrays, in particular, those of size at most 5, so it will always run in constant time. We also let $A[i..j]$ denote the array whose elements are $A[i], A[i+1], \ldots, A[j]$. The function P-PARTITION(A, x) is a multithreaded function which partitions A around the input element x and returns the number of elements in A which are less than or equal to x. Using a parallel for-loop, its span is logarithmic in the number of elements in A. The work is the same as the serialization, which is $\Theta(n)$ according to section 9.3. The span satisfies the recurrence $T_\infty(n) = \Theta(\lg n/5) + T_\infty(n/5) + \Theta(\lg n) + T_\infty(7n/10+6) \leq \Theta(\lg n) + T_\infty(n/5) + T_\infty(7n/10+6)$. Using the substitution method we can show that $T_\infty(n) = O(n^\varepsilon)$ for some $\varepsilon < 1$. In particular, $\varepsilon = .9$ works. This gives a parallelization of $\Omega(n^{.1})$.

Algorithm 14 P-SELECT(A,i)

1: **if** $n == 1$ **then**
2:     **return** $A[1]$
3: **end if**
4: Initialize a new array T of length $\lfloor n/5 \rfloor$
5: **parallel for** $i = 0$ to $\lfloor n/5 \rfloor - 1$ **do**
6:     $T[i+1] = \text{MEDIAN}(A[i\lfloor n/5 \rfloor..i\lfloor n/5 \rfloor + 4])$
7: **end**
8: **if** $n/5$ is not an integer **then**
9:     $T[\lfloor n/5 \rfloor] = \text{MEDIAN}(A[5\lfloor n/5 \rfloor..n])$
10: **end if**
11: $x = \text{P-SELECT}(T, \lceil n/5 \rceil)$
12: $k = \text{P-PARTITION}(A, x)$
13: **if** $k == i$ **then**
14:     **return** x
15: **else if** $i < k$ **then**
16:     P-SELECT($A[1..k-1], i$)
17: **else**
18:     P-SELECT($A[k+1..n], i-k$)
19: **end if**

Problem 27-1

a. See the algorithm Sum-Arrays(A,B,C). The parallelism is $O(n)$ since it's work is $n \lg(n)$ and the span is $\lg(n)$.

b. If grainsize is 1, this means that each call of Add-Subarray just sums a single pair of numbers. This means that since the for loop on line 4 will run n times, both the span and work will be $O(n)$. So, the parallelism is just $O(1)$.

Algorithm 15 Sum-Arrays(A,B,C)

$n = \left\lfloor \frac{A.length}{2} \right\rfloor$
if n=0 **then**
    C[1] = A[1]+B[1]
else
    spawn Sum-Arrays(A[1...n], B[1...n], C[1...n])
    Sum-Arrays(A[n+1 ... A.length],B[n+1 ... A.length],C[n+1 ... A.length])
    sync
end if

---

c. Let g be the grainsize. The runtime of the function that spawns all the other functions is $\left\lceil \frac{n}{g} \right\rceil$. The runtime of any particular spawned task is g. So, we want to minimize

$$\frac{n}{g} + g$$

To do this we pull out our freshman calculus hat and take a derivative, we get

$$0 = 1 - \frac{n}{g^2}$$

So, to solve this, we set $g = \sqrt{n}$. This minimizes the quantity and makes the span $O(n/g + g) = O(\sqrt{n})$. Resulting in a parallelism of $O(\sqrt{(n)})$.

Problem 27-2

a. Our algorithm P-MATRIX-MULTIPLY-RECURSIVE-SPACE(C,A,B) multiplies A and B, and adds their product to the matrix C. It is assumed that C contains all zeros when the function is first called.

b. The work is the same as the serialization, which is $\Theta(n^3)$. It can also be found by solving the recurrence $T_1(n) = \Theta(n^2) + 8T(n/2)$ where $T_1(1) = 1$. By the mater theorem, $T_1(n) = \Theta(n^3)$. The span is $T_\infty(n) = \Theta(1) + T_\infty(n/2) + T_\infty(n/2)$ with $T_\infty(1) = \Theta(1)$. By the master theorem, $T_\infty(n) = \Theta(n)$.

c. The parallelism is $\Theta(n^2)$. Ignoring the constants in the Θ-notation, the parallelism of the algorithm on 1000×1000 matrices is 1,000,000. Using P-MATRIX-MULTIPLY-RECURSIVE, the parallelism is 10,000,000, which is only about 10 times larger.

Problem 27-3

14

Algorithm 16 P-MATRIX-MULTIPLY-RECURSIVE-SPACE(C,A,B)

1: $n = A.rows$
2: **if** $n = 1$ **then**
3:     $c_11 = c_11 + a_11b_11$
4: **else**
5:     Partition A, B, and C into $n/2 \times n/2$ submatrices
6:     **spawn** P-MATRIX-MULTIPLY-RECURSIVE-SPACE(C_{11}, A_{11}, B_{11})
7:     **spawn** P-MATRIX-MULTIPLY-RECURSIVE-SPACE(C_{12}, A_{11}, B_{12})
8:     **spawn** P-MATRIX-MULTIPLY-RECURSIVE-SPACE(C_{21}, A_{21}, B_{11})
9:     **spawn** P-MATRIX-MULTIPLY-RECURSIVE-SPACE(C_{22}, A_{21}, B_{12})
10:     **sync**
11:     **spawn** P-MATRIX-MULTIPLY-RECURSIVE-SPACE(C_{11}, A_{12}, B_{21})
12:     **spawn** P-MATRIX-MULTIPLY-RECURSIVE-SPACE(C_{12}, A_{12}, B_{22})
13:     **spawn** P-MATRIX-MULTIPLY-RECURSIVE-SPACE(C_{21}, A_{22}, B_{21})
14:     **spawn** P-MATRIX-MULTIPLY-RECURSIVE-SPACE(C_{22}, A_{22}, B_{22})
15:     **sync**
16: **end if**

a. For the algorithm LU-DECOMPOSITION(A) on page 821, the inner for loops can be parallelized, since they never update values that are read on later runs of those loops. However, the outermost for loop cannot be parallelized because across iterations of it the changes to the matrices from previous runs are used to affect the next. This means that the span will be $\Theta(n \lg(n))$, work will still be $\Theta(n^3)$ and, so, the parallelization will be $\Theta(\frac{n^3}{n \lg(n)}) = \Theta(\frac{n^2}{\lg(n)})$.

b. The for loop on lines 7-10 is taking the max of a set of things, while recording the index that that max occurs. This for loop can therefor be replaced with a $\lg(n)$ span parallelized procedure in which we arrange the n elements into the leaves of an almost balanced binary tree, and we let each internal node be the max of its two children. Then, the span will just be the depth of this tree. This procedure can gracefully scale with the number of processors to make the span be linear, though even if it is $\Theta(n \lg(n))$ it will be less than the $\Theta(n^2)$ work later. The for loop on line 14-15 and the implicit for loop on line 15 have no concurrent editing, and so, can be made parallel to have a span of $\lg(n)$. While the for loop on lines 18-19 can be made parallel, the one containing it cannot without creating data races. Therefore, the total span of the naive parallelized algorithm will be $\Theta(n^2 \lg(n))$, with a work of $\Theta(n^3)$. So, the parallelization will be $\Theta(\frac{n}{\lg(n)})$. Not as parallized as part (a), but still a significant improvement.

c. We can parallelize the computing of the sums on lines 4 and 6, but cannot also parallize the for loops containing them without creating an issue of concurrently modifying data that we are reading. This means that the span will be $\Theta(n \lg(n))$, work will still be $\Theta(n^2)$, and so the parallelization will be $\Theta(\frac{n}{\lg(n)})$.

15

d. The recurrence governing the amount of work of implementing this procedure is given by

$$I(n) \le 2I(n/2) + 4M(n/2) + O(n^2)$$

However, the two inversions that we need to do are independent, and the span of parallelized matrix multiply is just $O(\lg(n))$. Also, the n^2 work of having to take a transpose and subtract and add matrices has a span of only $O(\lg(n))$. Therefore, the span satisfies the recurrence

$$I_\infty(n) \le I_\infty(n/2) + O(\lg(n))$$

This recurrence has the solution $I_\infty(n) \in \Theta(\lg^2(n))$ by exercise 4.6-2. Therefore, the span of the inversion algorithm obtained by looking at the procedure detailed on page 830. This makes the parallelization of it equal to $\Theta(M(n)/\lg^2(n))$ where $M(n)$ is the time to compute matrix products.

Problem 27-4

a. The algorithm below has $\Theta(n)$ work because its serialization satisfies the recurrence $T_1(n) = 2T(n/2) + \Theta(1)$ and $T(1) = \Theta(1)$. It has span $T_\infty(n) = \Theta(\lg n)$ because it satisfies the recurrence $T_\infty(n) = T_\infty(n/2) + \Theta(1)$ and $T_\infty(1) = \Theta(1)$.

Algorithm 17 P-REDUCE(x,i,j)

1: **if** $i == j$ **then**
2:     **return** $x[i]$
3: **else**
4:     $mid = \lfloor (i+j)/2 \rfloor$
5:     $x =$ **spawn** P-REDUCE(x, i, mid)
6:     $y =$ P-REDUCE$(x, mid + 1, j)$
7:     **sync**
8:     **return** $x \otimes y$
9: **end if**

---

b. The work of P-SCAN-1 is $T_1(n) = \Theta(n^2)$. The span is $T_\infty(n) = \Theta(n)$. The parallelism is $\Theta(n)$.

c. We'll prove correctness by induction on the number of recursive calls made to P-SCAN-2-AUX. If a single call is made then $n = 1$, and the algorithm sets $y[1] = x[1]$ which is correct. Now suppose we have an array which requires $n+1$ recursive calls. The elements in the first half of the array are accurately

computed since they require one fewer recursive calls. For the second half of the array,

$$y[i] = x[1] \otimes x[2] \otimes \ldots \otimes x[i] = (x[1] \otimes \ldots \otimes x[k]) \otimes (x[k+1] \otimes \ldots \otimes x[i]) = y[k] \otimes (x[k+1] \otimes \ldots \otimes x[i]).$$

Since we have correctly computed the parenthesized term with P-SCAN-2-AUX, line 8 ensures that we have correctly computed $y[i]$.

The work is $T_1(n) = \Theta(n \lg n)$ by the master theorem. The span is $T_\infty(n) = \Theta(\lg^2 n)$ by exercise 4.6-2. The parallelism is $\Theta(n/\lg n)$.

d. Line 8 of P-SCAN-UP should be filled in by $right \otimes t[k]$. Lines 5 and 6 of P-SCAN-DOWN should be filled in by v and $v \otimes t[k]$ respectively. Now we prove correctness. First I claim that if line 5 is accessed after l recursive calls, then

$$t[k] = x[k] \otimes x[k-1] \otimes \ldots \otimes x[k - \lfloor n/2^l \rfloor + 1]$$

and

$$right = x[k+1] \otimes x[k+2] \otimes \ldots x[k + \lfloor n/2^l \rfloor].$$

If $n = 2$ we make a single call, but no recursive calls, so we start our base case at $n = 3$. In this case, we set $t[2] = x[2]$, and $2 - \lfloor 3/2 \rfloor + 1 = 2$. We also have $right = x[3] = x[2+1]$, so the claim holds. In general, on the lth recursive call we set $t[k] = \text{P-SCAN-UP}(x, t, i, k)$, which is $t[\lfloor (i+k)/2 \rfloor] \otimes right$. By our induction hypothesis, $t[k] = x[(i+k)/2] \otimes x[(i+k)/2 - 1] \otimes \ldots \otimes x[(i+k)/2 - \lfloor n/2^{l+1} \rfloor + 1] \otimes x[(i+k)/2 + 1 \otimes \ldots \otimes x[(i+k)/2 + \lfloor n/2^{l+1} \rfloor]]$. This is equivalent to our claim since $(k-i)/2 = \lfloor n/2^{l+1} \rfloor$. A similar proof shows the result for $right$.

With this in hand, we can verify that the value v passed to P-SCAN-DOWN(v, x, t, y, i, j) satisfies $v = x[1] \otimes x[2] \otimes \ldots \otimes x[i-1]$. For the base case, if a single recursive call is made then $i = j = 2$, and we have $v = x[1]$. In general, for the call on line 5 there is nothing to prove because i doesn't change. For the call on line 6, we replace v by $v \otimes t[k]$. By our induction hypothesis, $v = x[1] \otimes \ldots \otimes x[i-1]$. By the previous paragraph, if we are on the lth recursive call, $t[k] = x[i] \otimes \ldots \otimes x[k - \lfloor n/2^l \rfloor + 1] = x[i]$ since on the lth recursive call, k and i must differ by $\lfloor n/2^l \rfloor$. Thus, the claim holds. Since we set $y[i] = v \otimes x[i]$, the algorithm yields the correct result.

e. The work of P-SCAN-UP satisfies $T_1(n) = 2T(n/2) + \Theta(1) = \Theta(n)$. The work of P-SCAN-DOWN is the same. Thus, the work of P-SCAN-3 satisfies $T_1(n) = \Theta(n)$. The span of P-SCAN-UP is $T_\infty(n) = T_\infty(n/2) + O(1) = \Theta(\lg n)$, and similarly for P-SCAN-DOWN. Thus, the span of P-SCAN-3 is $T_\infty(n) = \Theta(\lg n)$. The parallelism is $\Theta(n/\lg n)$.

Problem 27-5

a. Note that in this algorithm, the first call will be SIMPLE-STENCIL(A,A), and when there are ranges indexed into a matrix, what is gotten back is a view of the original matrix, not a copy. That is, changed made to the view will show up in the original. We can set up a recurrence for the work, which

---

Algorithm 18 $SIMPLE - STENCIL(A, A_2)$

---

let $n_1 \times n_2$ be the size of A_2.
let $m_i = \left\lfloor \frac{n_i}{2} \right\rfloor$ for $i = 1, 2$.
if $m_1 == 0$ **then**
    if $m_2 == 0$ **then**
        compute the value for the only position in A_2 based on the current
values in A.
    else
        $SIMPLE - STENCIL(A, A_2[1, 1 \ldots m_2])$
        $SIMPLE - STENCIL(A, A_2[1, m_2 + 1 \ldots n_3])$
    end if
else
    if $m_2 == 0$ **then**
        $SIMPLE - STENCIL(A, A_2[1 \ldots m_1, 1])$
        $SIMPLE - STENCIL(A, A_2[m_1 + 1 \ldots n_1, 1])$
    else
        $SIMPLE - STENCIL(A, A_2[1 \ldots m_1, 1 \ldots m_2])$
        spawn $SIMPLE - STENCIL(A, A_2[m_1 + 1 \ldots n_1, 1 \ldots m_2])$
        $SIMPLE - STENCIL(A, A_2[1 \ldots m_1, m_2 + 1 \ldots n_2])$
        sync
        $SIMPLE - STENCIL(A, A_2[m_1 + 1 \ldots n_1, m_2 + 1 \ldots n_2])$
    end if
end if

---

is just
$$W(n) = 4W(n/2) + \Theta(1)$$
which we can see by the master theorem has a solution which is $\Theta(n^2)$. For the span, the two middle subproblems are running at the same time, so,
$$S(n) = 3S(n/2) + \Theta(1)$$
Which has a solution that is $\Theta(n^{\lg(3)})$, also by the master theorem.

b. Just use the implementation for the third part with $b = 3$The work has the same solution of n^2 because it has the recurrence
$$W(n) = 9W(n/3) + \Theta(1)$$
The span has recurrence
$$S(n) = 5S(n/3) + \Theta(1)$$
Which has the solution $\Theta(n^{\log_3(5)})$

Algorithm 19 GEN-SIMPLE-STENCIL(A,A_2,b)

c.   let $n \times m$ be the size of A_2.

    if $(n \neq 0)\&\&(m \neq 0)$ **then**

        if $(n == 1)\&\&(m == 1)$ **then**

            compute the value at the only position in A_2

        else

            let $n_i = \lfloor \frac{in}{b} \rfloor$ for $i = 1, \ldots, b-1$

            let $m_i = \lfloor \frac{im}{b} \rfloor$ for $i = 1, \ldots, b-1$

            let $n_0 = m_0 = 1$

            for k=2, $\ldots$ b+1 **do**

                for i=1, $\ldots$ k-2 **do**

                    spawn $GEN\!-\!SIMPLE\!-\!STENCIL(A, A_2[n_{i-1} \ldots n_i, m_{k-i-1} \ldots m_{k-i}], b)$

                end for

                $GEN\!-\!SIMPLE\!-\!STENCIL(A, A_2[n_{i-1} \ldots n_i, m_{k-i-1} \ldots m_{k-i}], b)$

                sync

            end for

            for k=b+2, $\ldots$, 2b **do**

                for i=1,$\ldots$,2b-k **do**

                    spawn $GEN\!-\!SIMPLE\!-\!STENCIL(A, A_2[n_{b-k+i-1} \ldots n_{b-k+i}, m_{b-i-1} \ldots m_{b-i}], b)$

                end for

                $GEN\!-\!SIMPLE\!-\!STENCIL(A, A_2[n_{3b-2k} \ldots n_{3b-2k+1}i, m_{2k-2b} \ldots m_{2k-2b+1}], b)$

                sync

            end for

        end if

    end if

The recurrences we get are

$$W(n) = b^2 W(n/b) + \Theta(1)$$

$$S(n) = (2b - 1)W(n/b) + \Theta(1)$$

So, the work is $\Theta(n^2)$, and the span is $\Theta(n^{\lg_b(2b-1)})$. This means that the parallelization is $\Theta(n^{2-\lg_b(2b-1)})$. So, to show the desired claim, we only need to show that $2 - \log_b(2b - 1) < 1$

$$2 - \log_b(2b - 1) < 1$$
$$\log_b(2b) - \log_b(2b - 1) < 1$$
$$\log_b\left(\frac{2b}{2b - 1}\right) < 1$$
$$\frac{2b}{2b - 1} < b$$
$$2b < 2b^2 - b$$
$$0 < 2b^2 - 3b$$
$$0 < (2b - 3)b$$

This is clearly true because b is an integer greater than 2 and this right hand side only has zeroes at 0 and $\frac{3}{2}$ and is positive for larger b.

Algorithm 20 BETTER-STENCIL(A)

---

d.   **for** k=2,..., n+1 **do**
        for i=1, ...k-2 **do**
            spawn compute and update the entry at A[i,k-i]
        end for
        compute and update the entry at A[k-1,1]
        sync
    end for
    for k=n+2,...2n **do**
        for i=1, ...2n-k **do**
            spawn compute and update the entries along the diagonal which have
    indices summing to k
        end for
        sync
    end for

This procedure has span only equal to the length of the longest diagonal with is $O(n)$ with a factor of $\lg(n)$ thrown in. So, the parallelism is $O(n^2/(n \lg(n))) = O(n/\lg(n))$.

Problem 27-6

a. The work law becomes $E[T_P] \geq E[T_1]/P$. The span law becomes $E[T_P] \geq E[T_\infty]$. The greedy scheduler bound becomes $E[T_P] \leq E[T_1]/P + E[T_\infty]$.

b. We'll compute each.

$$E[T_1]/E[T_P] = \frac{100 + 10^9 \cdot .99}{.01 + 10^9 \cdot .99} \approx 1$$

$$E[T_1/T_p] = 100 + .99 = 100.99.$$

Since the algorithm almost always runs in the same amount of time, regardless of the increase in number of processors, and the speedup tells us how many times faster something runs on P processors than on 1, the expected speedup should be approximately 1. Thus, $E[T_1]/E[T_P]$ is the better definition.

c. As $P \to \infty$ the speedup should approach the parallelism, so it makes sense to use a definition which agrees with part b in the limit.

d. Assume that PARTITION is implemented as described in exercise 27.3-3, with work $\Theta(n)$ and span $\Theta(\lg n)$. However, we will not modify anything else about RANDOMIZED-PARTITION.

Algorithm 21 P-RANDOMIZED-QUICKSORT(A,p,r)

1: **if** $p < r$ **then**
2:     $q = $ RANDOMIZED-PARTITION(A, p, r)
3:     **spawn** P-RANDOMIZED-QUICKSORT$(a, P, Q - 1)$
4:     RANDOMIZED-QUICKSORT$(A, q + 1, r)$
5: **end if**

---

e. The work is just the runtime of the serialization, which we know to have expected time $O(n \lg n)$. For the span, our analysis will be similar to that of RANDOMIZED-SELECT from page 216. Let X_k be the indicator random variable which is equal to 1 if $A[p..q]$ has exactly k elements and 0 otherwise. Then we have

$$T_\infty(n) \leq \sum_{k=1}^{n} X_k \cdot T_\infty(\max(k - 1, n - k)) + \Theta(\lg n).$$

By linearity of expectation this implies that

$$E[T_\infty(n)] \leq \sum_{k=1}^{n} \frac{1}{n} E[T_\infty(\max(k - 1, n - k))] + \Theta(\lg n).$$

Since each even term appears twice we can write this as

$$E[T_\infty(n)] \leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} E[T_\infty(k)] + \Theta(\lg n).$$

We'll show that $E[T_\infty(n)] = O(n^{1-\varepsilon})$ for ε such that $\frac{2-2^{\varepsilon-1}}{2-\varepsilon} < 1$ by the substitution method. Suppose that $E[T_\infty(n)] \leq c_1 n^{1-\varepsilon}$. Then we have

$$
\begin{aligned}
E[T_\infty(n)] &\leq \frac{2c_1}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} k^{1-\varepsilon} + \Theta(\lg n) \\
&\leq \frac{2c_1}{n} \int_{k=\lfloor n/2 \rfloor}^{n} x^{1-\varepsilon} dx + \Theta(\lg n) \\
&= \frac{2c_1}{n} \left. \frac{x^{2-\varepsilon}}{2-\varepsilon} \right|_{k=\lfloor n/2 \rfloor}^{n} + \Theta(\lg n) \\
&= c_1 n^{1-\varepsilon} \left(\frac{2-2^{\varepsilon-1}}{2-\varepsilon} \right) + \Theta(\lg n).
\end{aligned}
$$

Since the dominating term is strictly less than $c_1 n^{1-\varepsilon}$, we can overcome the $\Theta(\lg n)$ term. Thus, $E[T_\infty(n)] = O(n^{1-\varepsilon})$, so we achieve sublinear expected time. The expected parallelization $\Omega(n \lg n / n^{1-\varepsilon}) = \Omega(n^\varepsilon \lg n)$.

22

Chapter 28

Michelle Bodnar, Andrew Lohr

May 5, 2017

Exercise 28.1-1

We get the solution:

$$\begin{pmatrix} 3 \\ 14 - 4 \cdot 3 \\ -7 - 5 \cdot (14 - 4 \cdot 3) + 6 \cdot 3 \end{pmatrix} = \begin{pmatrix} 3 \\ 2 \\ 1 \end{pmatrix}$$

Exercise 28.1-2

An LU decomposition of the matrix is given by

$$\begin{pmatrix} 4 & -5 & 6 \\ 8 & -6 & 7 \\ 12 & -7 & 12 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 2 & 1 \end{pmatrix} \begin{pmatrix} 4 & -5 & 6 \\ 0 & 4 & -5 \\ 0 & 0 & 4 \end{pmatrix}.$$

Exercise 28.1-3

First, we find the LUP decomposition of the given matrix

$$\begin{pmatrix} 1 & 5 & 4 \\ 2 & 0 & 3 \\ 5 & 8 & 2 \end{pmatrix}$$

we bring the 5 to the top, and then divide the first column by 5, and use Schur complements to change the rest of the matrix to get

$$\begin{pmatrix} 5 & 8 & 2 \\ .4 & -3.2 & 2.2 \\ .2 & 3.4 & 3.6 \end{pmatrix}$$

Then, we swap the third and second rows, and apply the Schur complement to get

$$\begin{pmatrix} 5 & 8 & 2 \\ .2 & 3.4 & 3.6 \\ .4 & -\frac{3.2}{3.4} & 2.2 + \frac{11.52}{3.4} \end{pmatrix}$$

This gets us the LUP decomposition that

$$L = \begin{pmatrix} 1 & 0 & 0 \\ .2 & 1 & 0 \\ .4 & -\frac{3.2}{3.4} & 1 \end{pmatrix}$$

$$U = \begin{pmatrix} 5 & 8 & 2 \\ 0 & 3.4 & 3.6 \\ 0 & 0 & 2.2 + \frac{11.52}{3.4} \end{pmatrix}$$

$$P = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

Using this, we can get that the solution must be

$$\begin{pmatrix} 1 & 0 & 0 \\ .2 & 1 & 0 \\ .4 & -\frac{3.2}{3.4} & 1 \end{pmatrix} \begin{pmatrix} 5 & 8 & 2 \\ 0 & 3.4 & 3.6 \\ 0 & 0 & 2.2 + \frac{11.52}{3.4} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 5 \\ 12 \\ 9 \end{pmatrix}$$

Which, by forward substitution,

$$\begin{pmatrix} 5 & 8 & 2 \\ 0 & 3.4 & 3.6 \\ 0 & 0 & 2.2 + \frac{11.52}{3.4} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 5 \\ 11 \\ 7 + \frac{35.2}{3.4} \end{pmatrix}$$

So, finally by back substitution,

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} -\frac{3}{19} \\ -\frac{1}{19} \\ \frac{49}{19} \end{pmatrix}$$

Exercise 28.1-4

The LUP decomposition of a diagonal matrix D is $D = IDI$ where I is the identity matrix.

Exercise 28.1-5

A LU decomposition of a permutation matrix is letting P be the inverse permutation matrix, and let both L and U be the identity matrix. Now, we need show that this representation is unique. We know that the permutation matrix A is non-singular. This means that U has nonzero elements all along its diagonal. Now, suppose that there were some nonzero element off of the diagonal in L, which is to say $L_{i,j} \neq 0$ for $i \neq j$. Then, look at row i in the product LU. This has a nonzero entry both at column j and at column i. Since it has more than one non-zero entry, it cannot be transformed into a permutation matrix by permuting the rows. Similarly, we have that U cannot have any off-diagonal elements. Lastly, since we know that both L and U are diagonal matrices, we

know that L is the identity. Since A only has ones as its nonzero entries, and $LU = U$. U must also only have ones as its nonzero entries. So, we have U is the identity. This means that $PA = I$, which means that $P = A^{-1}$. This completes showing that the given decomposition is unique.

Exercise 28.1-6

The zero matrix always has an LU decomposition by taking L to be any unit lower-triangular matrix and U to be the zero matrix, which is upper triangular.

Exercise 28.1-7

For LU decomposition, it is indeed necessary. If we didn't run the last run of the outermost for loop, u_{nn} would be left its initial value of zero instead of being set equal to a_{nn}. This can clearly produce incorrect results, because the LU decomposition of any non-singular matrix must have both L and U having positive determinant. However, if $u_{nn} = 0$, the determinant of U will be zero by problem D.2-2.

For LUP-decomposition, the iteration of the outermost for loop that occurs with $k = n$ will not change the final answer. Since π would have to be a permutation on a single element, it cannot be non-trivial. and the for loop on line 16 will not run at all.

Exercise 28.2-1

Showing that being able to multiply matrices in time $M(n)$ implies being able to square matrices in time $M(n)$ is trivial because squaring a matrix is just multiplying it by itself.

The more tricky direction is showing that being able to square matrices in time $S(n)$ implies being able to multiply matrices in time $O(S(n))$.

As we do this, we apply the same regularity condition that $S(2n) \in O(S(n))$. Suppose that we are trying to multiply the matrices, A and B, that is, find AB. Then, define the matrix

$$C = \begin{pmatrix} I & A \\ 0 & B \end{pmatrix}$$

Then, we can find C^2 in time $S(2n) \in O(S(n))$. Since

$$C^2 = \begin{pmatrix} I & A + AB \\ 0 & B \end{pmatrix}$$

Then we can just take the upper right quarter of C^2 and subtract A from it to obtain the desired result. Apart from the squaring, we've only done work that is $O(n^2)$. Since $S(n)$ is $\Omega(n^2)$ anyways, we have that the total amount of work we've done is $O(n^2)$.

Exercise 28.2-2

In this problem and the next, we'll follow the approach taken in *Algebraic Complexity Theory* by Burgisser, Claussen, and Shokrollahi. Let A be an $n \times n$ matrix. Without loss of generality we'll assume $n = 2^k$, and impose the regularity condition that $L(n/2) \leq cL(n)$ where $c < 1/2$ and $L(n)$ is the time it takes to find an LUP decomposition of an $n \times n$ matrix. First, decompose A as

$$A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix}$$

where A_1 is $n/2$ by n. Let $A_1 = L_1 U_1 P_1$ be an LUP decomposition of A_1, where L_1 is $n/2$ by $n/2$, U_1 is $n/2$ by n, and P_1 is n by n. Perform a block decomposition of U_1 and $A_2 P_1^{-1}$ as $U_1 = [\overline{U}_1 | B]$ and $A_2 P_1^{-1} = [C | D]$ where $\overline{U}_1$ and C are $n/2$ by $n/2$ matrices. Since we assume that A is nonsingular, $\overline{U}_1$ must also be nonsingular. Set $F = D - C\overline{U}_1^{-1} B$. Then we have

$$A = \begin{bmatrix} L_1 & 0 \\ C\overline{U}_1^{-1} & I_{n/2} \end{bmatrix} \begin{bmatrix} \overline{U}_1 & B \\ 0 & F \end{bmatrix} P_1.$$

Now let $F = L_2 U_2 P_2$ be an LUP decomposition of F, and let $\overline{P} = \begin{bmatrix} I_{n/2} & 0 \\ 0 & P_2 \end{bmatrix}$. Then we may write

$$A = \begin{bmatrix} L_1 & 0 \\ C\overline{U}_1^{-1} & L_2 \end{bmatrix} \begin{bmatrix} \overline{U}_1 & BP_2^{-1} \\ 0 & U_2 \end{bmatrix} \overline{P} P_1.$$

This is an LUP decomposition of A. To achieve it, we computed two LUP decompositions of half size, a constant number of matrix multiplications, and a constant number of matrix inversions. Since matrix inversion and multiplication are computationally equivalent, we conclude that the runtime is $O(M(n))$.

Exercise 28.2-3

From problem 28.2-2, we can find a LU-decomposition algorithm that only takes time $O(M(n))$. So, we run that algorithm and multiply together all of the entries along the diagonal of U, this will be the determinant of the original matrix.

We have no short answer for the second direction of this problem. Typically we strive to complete all the exercises independently as a way of gaining a greater understanding ourselves. For the second half of this problem, this was unfortunately not something that we achieved. After being stuck for several weeks, I asked a number of other graduate students, and even two professors, none were able to help with how the proof went. One did however refer me to the book Algebraic Complexity Theory by Burgisser, Claussen, and Shokrollahi. They have the proof in their section 16.4. This however is not at all self contained and so would be very long to try and include here as a solution. This was the

hardest exercise in the book. If you know a good proof of this fact, I would love to hear it, please let me know by emailing ajl213 "at" math.rutger.edu.
Exercise 28.2-4

Suppose we can multiply boolean matrices in $M(n)$ time, where we assume this means that if we're multiplying boolean matrices A and B, then $(AB)_{ij} = (a_{i1} \wedge b_{1j}) \vee \ldots \vee (a_{in} \wedge b_{nj})$. To find the transitive closure of a boolean matrix A we just need to find the nth power of A. We can do this by computing A^2, then $(A^2)^2$, then $((A^2)^2)^2)$, and so on. This requires only $\lg n$ multiplications, so the transitive closure can be computed in $O(M(n) \lg n)$.

For the other direction, first view A and B as adjacency matrices, and impose the regularity condition $T(3n) = O(T(n))$, where $T(n)$ is the time to compute the transitive closure of a graph on n vertices. We will define a new graph whose transitive closure matrix contains the boolean product of A and B. Start by placing $3n$ vertices down, labeling them $1, 2, \ldots, n, 1', 2', \ldots, n', 1'', 2'', \ldots, n''$. Connect vertex i to vertex j' if and only if $A_{ij} = 1$. Connect vertex j' to vertex k'' if and only if $B_{jk} = 1$. In the resulting graph, the only way to get from the first set of n vertices to the third set is to first take an edge which "looks like" an edge in A, then take an edge which "looks like " an edge in B. In particular, the transitive closure of this graph is:

$$\begin{bmatrix} I & A & AB \\ 0 & I & B \\ 0 & 0 & I \end{bmatrix}.$$

Since the graph is only of size $3n$, computing its transitive closure can be done in $O(T(3n)) = O(T(n))$ by the regularity condition. Therefore multiplying matrices and finding transitive closure are are equally hard.

Exercise 28.2-5

It does not work necessarily over the field of two elements. The problem comes in in applying theorem D.6 to conclude that $A^T A$ is positive definite. In the proof of that theorem they obtain that $||Ax||^2 \geq 0$ and only zero if every entry of Ax is zero. This second part is not true over the field with two elements, all that would be required is that there is an even number of ones in Ax. This means that we can only say that $A^T A$ is positive semi-definite instead of the positive definiteness that the algorithm requires.

Exercise 28.2-6

We may again assume that our matrix is a power of 2, this time with complex entries. For the moment we assume our matrix A is Hermitian and positive-definite. The proof goes through exactly as before, with matrix transposes replaced by conjugate transposes, and using the fact that Hermitian positive-definite matrices are invertible. Finally, we need to justify that we can obtain the

same asymptotic running time for matrix multiplication as for matrix inversion when A is invertible, but not Hermitian positive-definite. For any nonsingular matrix A, the matrix A^*A is Hermitian and positive definite, since for any x we have $x^*A^*Ax = \langle Ax, Ax \rangle > 0$ by the definition of inner product. To invert A, we first compute $(A^*A)^{-1} = A^{-1}(A^*)^{-1}$. Then we need only multiply this result on the right by A^*. Each of these steps takes $O(M(n))$ time, so we can invert any nonsingluar matrix with complex entries in $O(M(n))$ time.

Exercise 28.3-1

To see this, let e_i be the vector that is zeroes except for a one in the ith position. Then, we consider the quantity $e_i^T A e_i$ for every i. $A e_i$ takes each row of A and pulls out the ith column of it, and puts those values into a column vector. Then, multiplying that on the left by e_i^T, pulls out the ith row of this quantity, which means that the quantity $e_i^T A e_i$ is exactly the value of $A_{i,i}$. So, we have that by positive definiteness, since e_i is nonzero, that quantity must be positive. Since we do this for every i, we have that every entry along the diagonal must be positive.

Exercise 28.3-2

Let $x = -by/a$. Since A is positive-definite we must have

$$
\begin{aligned}
0 &< [xy]^T A \begin{bmatrix} x \\ y \end{bmatrix} \\
&= [xy]^T \begin{bmatrix} ax + by \\ bx + cy \end{bmatrix} \\
&= ax^2 + 2bxy + cy^2 \\
&= cy^2 - \frac{b^2 y^2}{a} \\
&= (c - b^2/a)y^2.
\end{aligned}
$$

Thus, $c - b^2/a > 0$ which implies $ca - b^2 > 0$, since $a > 0$.

Exercise 28.3-3

Suppose to a contradiction that there were some element a_{ij} with $i \neq j$ so that a_{ij} were a largest element. We will use e_i to denote the vector that is all zeroes except for having a 1 at position i. Then, we consider the value $(e_i - e_j)^T A(e_i - e_j)$. When we compute $A(e_i - e_j)$ this will return a vector which is column i minus column j. Then, when we do the last multiplication, we will get the quantity which is the ith row minus the jth row. So,

$$
\begin{aligned}
(e_i - e_j)^T A(e_i - e_j) &= a_{ii} - a_{ij} - a_{ji} + a_{jj} \\
&= a_{ii} + a_{jj} - 2a_{ij} \leq 0
\end{aligned}
$$

Where we used symmetry to get that $a_{ij} = a_{ji}$. This result contradicts the fact that A was positive definite. So, our assumption that there was a element tied for largest off the diagonal must of been false.

Exercise 28.3-4

The claim clearly holds for matrices of size 1 because the single entry in the matrix is positive the only leading submatrix is the matrix itself. Now suppose the claim holds for matrices of size n, and let A be an $(n+1) \times (n+1)$ symmetric positive-definite matrix. We can write A as

$$A = \left[\begin{array}{c|c} A' & w \\ \hline v & c \end{array} \right].$$

Then A' is clearly symmetric, and for any x we have $x^T A' x = [x 0] A \left[\begin{array}{c} x \\ 0 \end{array} \right] > 0$, so A' is positive-definite. By our induction hypothesis, every leading submatrix of A' has positive determinant, so we are left only to show that A has positive determinant. By Theorem D.4, the determinant of A is equal to the determinant of the matrix

$$B = \left[\begin{array}{c|c} c & v \\ \hline w & A' \end{array} \right].$$

Theorem D.4 also tells us that the determinant is unchanged if we add a multiple of one column of a matrix to another. Since $0 < e_{n+1}^T A e_{n+1} = c$, we can use multiples of the first column to zero out every entry in the first row other than c. Specifically, the determinant of B is the same as the determinant of the matrix obtained in this way, which looks like

$$C = \left[\begin{array}{c|c} c & 0 \\ \hline w & A'' \end{array} \right].$$

By definition, $\det(A) = c \det(A'')$. By our induction hypothesis, $\det(A'') > 0$. Since $c > 0$ as well, we conclude that $\det(A) > 0$, which completes the proof.

Exercise 28.3-5
When we do an LU decomposition of a positive definite symmetric matrix, we never need to permute the rows. This means that the pivot value being used from the first operation is the entry in the upper left corner. This gets us that for the case $k = 1$, it holds because we were told to define $det(A_0) = 1$, getting us, $a_{11} = det(A_1)/det(A_0)$. When Diagonalizing a matrix, the product of the

pivot values used gives the determinant of the matrix. So, we have that the determinant of A_k is a product of the kth pivot value with all the previous values. By induction, the product of all the previous values is $det(A_{k-1})$. So, we have that if x is the kth pivot value, $det(A_k) = xdet(A_{k-1})$, giving us the desired result that the k th pivot value is $det(A_k)/det(A_{k-1})$.

Exercise 28.3-6

First we form the A matrix

$$A = \begin{bmatrix} 1 & 0 & e \\ 1 & 2 & e^2 \\ 1 & 3\lg 3 & e^3 \\ 1 & 8 & e^4 \end{bmatrix}.$$

We'll compute the pseudoinverse using Wolfram Alpha, then multiply it by y, to obtain the coefficient vector

$$c = \begin{bmatrix} .411741 \\ -.20487 \\ .16546 \end{bmatrix}.$$

Exercise 28.3-7

$$AA^+A = A((A^TA)^{-1}A^T)A$$
$$= A(A^TA)^{-1}(A^TA)$$
$$= A$$

$$A^+AA^+ = ((A^TA)^{-1}A^T)A((A^TA)^{-1}A^T)$$
$$= (A^TA)^{-1}(A^TA)(A^TA)^{-1}A^T$$
$$= (A^TA)^{-1}A^T$$
$$= A^+$$

$$(AA^+)^T = (A(A^TA)^{-1}A^T)^T$$
$$= A((A^TA)^{-1})^TA^T$$
$$= A((A^TA)^T)^{-1}A^T$$
$$= A(A^TA)^{-1}A^T$$
$$= AA^+$$

$$(AA^+)^T = ((A^T A)^{-1} A^T A)^T$$
$$= ((A^T A)^{-1} (A^T A))^T$$
$$= I^T$$
$$= I$$
$$= (A^T A)^{-1} (A^T A)$$
$$= A^+ A$$

Problem 28-1

a. By applying the procedure of the chapter, we obtain that

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix}$$

$$U = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$P = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

b. We first do back substitution to obtain that

$$Ux = \begin{pmatrix} 5 \\ 4 \\ 3 \\ 2 \\ 1 \end{pmatrix}$$

So, by forward substitution, we have that

$$x = \begin{pmatrix} 5 \\ 9 \\ 12 \\ 14 \\ 15 \end{pmatrix}$$

9

c. We will set $Ax = e_i$ for each i, where e_i is the vector that is all zeroes except for a one in the ith position. Then, we will just concatenate all of these solutions together to get the desired inverse.

equation	solution
$Ax_1 = e_1$	$x_1 = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$
$Ax_2 = e_2$	$x_2 = \begin{pmatrix} 1 \\ 2 \\ 2 \\ 2 \\ 2 \end{pmatrix}$
$Ax_3 = e_3$	$x_3 = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 3 \\ 3 \end{pmatrix}$
$Ax_4 = e_4$	$x_4 = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 4 \end{pmatrix}$
$Ax_5 = e_5$	$x_5 = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{pmatrix}$

This gets us the solution that

$$A^{-1} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 2 & 2 & 2 \\ 1 & 2 & 3 & 3 & 3 \\ 1 & 2 & 3 & 4 & 4 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix}$$

d. When performing the LU decomposition, we only need to take the max over at most two different rows, so the loop on line 7 of LUP-DECOMPOSITION drops to $O(1)$. There are only some constant number of nonzero entries in each row, so the loop on line 14 can also be reduced to being $O(1)$. Lastly, there are only some constant number of nonzero entries of the form a_{ik} and a_{kj}. since the square of a constant is also a constant, this means that the nested for loops on lines 16-19 also only take time O(1) to run. Since the

for loops on lines 3 and 5 both run $O(n)$ times and take $O(1)$ time each to run(provided we are smart to not consider a bunch of zero entries in the matrix), the total runtime can be brought down to $O(n)$.

Since for a Tridiagonal matrix, it will only ever have finitely many nonzero entries in any row, we can do both the forward and back substitution each in time only $O(n)$.

Since the asymptotics of performing the LU decomposition on a positive definite tridiagonal matrix is $O(n)$, and this decomposition can be used to solve the equation in time $O(n)$, the total time for solving it with this method is $O(n)$. However, to simply record the inverse of the tridiagonal matrix would take time $O(n^2)$ since there are that many entries, so, any method based on computing the inverse of the matrix would take time $\Omega(n^2)$ which is clearly slower than the previous method.

e. The runtime of our LUP decomposition algorithm drops to being $O(n)$ because we know there are only ever a constant number of nonzero entries in each row and column, as before. Once we have an LUP decomposition, we also know that that decomposition have both L and U having only a constant number of non-zero entries in each row and column. This means that when we perform the forward and backward substitution, we only spend a constant amount of time per entry in x, and so, only takes $O(n)$ time.

Problem 28-2

a. We have $a_i = f_i(0) = y_i$ and $b_i = f'_i(0) = f'(x_i) = D_i$. Since $f_i(1) = a_i + b_i + c_i + d_i$ and $f'_i(1) = b_i + 2c_i + 3d_i$ we have $d_i = D_{i+1} - 2y_{i+1} + 2y_i + D_i$ which implies $c_i = 3y_{i+1} - 3y_i - D_{i+1} - 2D_i$. Since each coefficient can be computed in constant time from the known values, we can compute the $4n$ coefficients in linear time.

b. By the continuity constraints we have $f''_i(1) = f''_{i+1}(0)$ which implies that $2c_i + 6d_i = 2c_{i+1}$, or $c_i + 3d_i = c_{i+1}$. Using our equations from above, this is equivalent to

$$D_i + 2D_{i+1} + 3y_i - 3y_{i+1} = 3y_{i+2} - 3y_{i+1} - D_{i+2} - 2D_{i+1}.$$

Rearranging gives the desired equation

$$D_i + 4D_{i+1} + D_{i+2} = 3(y_{i+2} - y_i).$$

c. The condition on the left endpoint tells us that $f''_0(0) = 0$, which implies $2c_0 = 0$. By part a, this means $3(y_1 - y_0) = 2D_0 + D_1$. The condition on the right endpoint tells us that $f''_{n-1}(1) = 0$, which implies $c_{n-1} + 3d_{n-1} = 0$. By part a, this means $3(y_n - y_{n-1}) = D_{n-1} + 2D_n$.

d. The matrix equation has the form $AD = Y$, where A is symmetric and tridiagonal. It looks like this:

$$
\begin{bmatrix}
2 & 1 & 0 & 0 & \cdots & 0 \\
1 & 4 & 1 & 0 & \cdots & 0 \\
0 & \ddots & \ddots & \ddots & \cdots & \vdots \\
\vdots & \cdots & 1 & 4 & 1 & 0 \\
0 & \cdots & 0 & 1 & 4 & 1 \\
0 & \cdots & 0 & 0 & 1 & 2
\end{bmatrix}
\begin{bmatrix}
D_0 \\
D_1 \\
D_2 \\
\vdots \\
D_{n-1} \\
D_n
\end{bmatrix}
=
\begin{bmatrix}
3(y_1 - y_0) \\
3(y_2 - y_0) \\
3(y_3 - y_1) \\
\vdots \\
3(y_n - y_{n-2}) \\
3(y_n - y_{n-1})
\end{bmatrix}.
$$

e. Since the matrix is symmetric and tridiagonal, Problem 28-1 e tells us that we can solve the equation in $O(n)$ time by performing an LUP decomposition. By part a of this problem, once we know each D_i we can compute each f_i in $O(n)$ time.

f. For the general case of solving the nonuniform natural cubic spline problem, we require that $f(x_{i+1}) = f_i(x_{i+1} - x_i) = y_{i+1}$, $f'(x_{i+1}) = f'_i(x_{i+1} - x_i) = f'_{i+1}(0)$ and $f''(x_{i+1}) = f''_i(x_{i+1} - x_i) = f''_{i+1}(0)$. We can still solve for each of a_i, b_i, c_i, and d_i in terms of y_i, y_{i+1}, D_i, and D_{i+1}, so we still get a tridiagonal matrix equation. The solution will be slightly messier, but ultimately it is solved just like the simpler case, in $O(n)$ time.

Chapter 29

Michelle Bodnar, Andrew Lohr

April 12, 2016

Exercise 29.1-1

$$c = \begin{pmatrix} 2 \\ -3 \\ 3 \end{pmatrix}$$

$$A = \begin{pmatrix} 1 & 1 & -1 \\ -1 & -1 & 1 \\ 1 & -2 & 2 \end{pmatrix}$$

$$b = \begin{pmatrix} 7 \\ -7 \\ 4 \end{pmatrix}$$

with $m = n = 3$

Exercise 29.1-2

One solution is $(x_1, x_2, x_3) = (6, 1, 0)$ with objective value 9. Another is $(5, 2, 0)$ with objective value 4. A third is $(4, 3, 0)$ with objective value -1.

Exercise 29.1-3

$$N = \{1, 2, 3\}$$
$$B = \{4, 5, 6\}$$
$$A = \begin{pmatrix} 1 & 1 & -1 \\ -1 & -1 & 1 \\ 1 & -2 & 2 \end{pmatrix}$$
$$b = \begin{pmatrix} 7 \\ -7 \\ 4 \end{pmatrix}$$

$$c = \begin{pmatrix} 2 \\ -3 \\ 3 \end{pmatrix}$$

And $v = 0$.

Exercise 29.1-4

$$
\begin{aligned}
\text{maximize} \quad & -2x_1 - 2x_2 - 7x_3 + x_4 \\
\text{subject to} \quad & -x_1 + x_2 - x_4 \leq -7 \\
& x_1 - x_2 + x_4 \leq 7 \\
& -3x_1 + 3x_2 - x_3 \leq -24 \\
& x_1, x_2, x_3, x_4 \geq 0
\end{aligned}
$$

Exercise 29.1-5

First, we will multiply the second and third inequalities by minus one to make it so that they are all $\leq$ inequalities. We will introduce the three new variables x_4, x_5, x_6, and perform the usual procedure for rewriting in slack form

$$
\begin{aligned}
x_4 &= 7 - x_1 - x_2 + x_3 \\
x_5 &= -8 + 3x_1 - x_2 \\
x_6 &= -x_1 + 2x_2 + 2x_3 \\
x_1, x_2, x_3, x_4, x_5, x_6 &\geq 0
\end{aligned}
$$

where we are sill trying to maximize $2x_1 - 6x_3$. The basic variables are x_4, x_5, x_6 and the nonbasic variables are x_1, x_2, x_3.

Exercise 29.1-6

By dividing the second constraint by 2 and adding to the first, we have $0 \leq -3$, which is impossible. Therefore there linear program is unfeasible.

Exercise 29.1-7

For any number $r > 1$, we can set $x_1 = 2r$ and $x_2 = r$. Then, the restaints become

$$
\begin{aligned}
-2r + r = -r &\leq -1 \\
-2r - 2r = -4r &\leq -2 \\
2r, r &\geq 0
\end{aligned}
$$

All of these inequalities are clearly satisfied because of our initial restriction in selecting r. Now, we look to the objective function, it is $2r - r = r$. So, since

we can select r to be arbitrarily large, and still satisfy all of the constraints, we can achieve an arbitrarily large value of the objective function.

Exercise 29.1-8

In the worst case, we have to introduce 2 variables for every variable to ensure that we have nonnegativity constraints, so the resulting program will have $2n$ variables. If each constraint is an equality, we would have to double the number of constraints to create inequalities, resulting in $2m$ constraints. Changing minimization to maximization and greater-than signs to less-than signs don't affect the number of variables or constraints, so the upper bound is $2n$ on variables and $2m$ on constraints.

Exercise 29.1-9

Consider the linear program where we want to maximize $x_1 - x_2$ subject to the constraints $x_1 - x_2 \leq 1$ and $x_1, x_2 \geq 0$. clearly the objective value can never be greater than one, and it is easy to achieve the optimal value of 1, by setting $x_1 = 1$ and $x_2 = 0$. Then, this feasible region is unbounded because for any number r, we could set $x_1 = x_2 = r$, and that would be feasible because the difference of the two would be zero which is ≤ 1.

If we further wanted it so that there was a single solution that achieved the finite optimal value, we could add the requirements that $x_1 \leq 1$.

Exercise 29.2-1

The objective is already in normal form. However, some of the constraints are equality constraints instead of $\leq$ constraints. This means that we need to rewrite them as a pair of inequality constraints, the overlap of whose solutions is just the case where we have equality. we also need to deal with the fact that most of the variables can be negative. To do that, we will introduce variables for the negative part and positive part, each of which need be positive, and we'll just be sure to subtract the negative part. d_s need not be changed in this way since it can never be negative since we are not assuming the existence of negative weight cycles.

$$d_v^+ - d_v^- - d_u^+ + d_u^- \leq w(u,v) \text{for every edge } (u,v)$$
$$d_s \leq 0$$

all variables are positive

Exercise 29.2-2

$$\begin{aligned}
\text{maximize} \quad & d_y \\
\text{subject to} \quad & d_t \le d_s + 3 \\
& d_x \le d_t + 6 \\
& d_y \le d_s + 5 \\
& d_y \le d_t + 2 \\
& d_z \le d_x + 2 \\
& d_t \le d_y + 1 \\
& d_x \le d_y + 4 \\
& d_z \le d_y + 1 \\
& d_s \le d_z + 1 \\
& d_x \le d_z + 7 \\
& d_2 = 0
\end{aligned}$$

Exercise 29.2-3

We will follow a similar idea to the way to when we were finding the shortest path between two particular vertices.

$$\begin{aligned}
\text{maximize} \quad & \sum_{v \in V} d_v \\
\text{subject to} \quad & d_v \le d_u + w(u,v) \text{for each edge (u,v)} \\
& d_s = 0
\end{aligned}$$

The first type of constraint makes sure that we never say that a vertex is further away than it would be if we just took the edge corresponding to that constraint. Also, since we are trying to maximize all of the variables, we will make it so that there is no slack anywhere, and so all the d_v values will correspond to lengths of shortest paths to v. This is because the only thing holding back the variables is the information about relaxing along the edges, which is what determines shortest paths.

Exercise 29.2-4

$$\begin{aligned}
\text{maximize} \quad & f_{sv_1} + f_{sv_2} \\
\text{subject to} \quad & f_{sv_1} \le 16 \\
& f_{sv_2} \le 14 \\
& f_{v_1v_3} \le 12 \\
& f_{v_2v_1} \le 4 \\
& f_{v_2v_4} \le 14 \\
& f_{v_3v_2} \le 9 \\
& f_{v_3t} \le 20 \\
& f_{v_4v_3} \le 7 \\
& f_{v_4t} \le 4 \\
& f_{sv_1} + f_{v_2v_1} = f_{v_1v_3} \\
& f_{sv_2} + f_{v_3v_2} = f_{v_2v_1} + f_{v_2v_4} \\
& f_{v_1v_3} + f_{v_4v_3} = f_{v_3v_2} + f_{v_3t} \\
& f_{v_2v_4} = f_{v_4v_3} + f_{v_4t} \\
& f_{uv} \ge 0 \text{ for } u, v \in \{s, v_1, v_2, v_3, v_4, t\}
\end{aligned}$$

Exercise 29.2-5

All we need to do to bring the number of constraints down from $O(V^2)$ to $O(V + E)$ is to replace the way we index the flows. Instead of indexing it by a pair of vertices, we will index it by an edge. This won't change anything about the analysis because between pairs of vertices that don't have an edge between them, there definitely won't be any flow. Also, it brings the number of constraints of the first and third time down to $O(E)$ and the number of constraints of the second kind stays at $O(V)$.

$$\begin{aligned}
\text{maximize} \quad & \sum_{\text{edges } e \text{ coming out of } s} f_e - \sum_{\text{edges } e \text{ going into } s} f_s \\
\text{subject to} \quad & f_{(u,v)} \le c(u, v) \text{ for each edge (u,v)} \\
& \sum_{\text{edges } e \text{ leaving } u} f_e = \sum_{\text{edges } e \text{ entering } u} f_e \text{ for each edge } u \in V - \{s, t\} \\
& f_e \ge 0 \text{ for each edge } e
\end{aligned}$$

Exercise 29.2-6

Recall from section 26.3 that we can solve the maximum-bipartite-matching problem by viewing it as a network flow problem, where we append a source s and sink t, each connected to every vertex is L and R respectively by an edge with capacity 1, and we give every edge already in the bipartite graph capacity

1. The integral maximum flows are in correspondence with maximum bipartite matchings. In this setup, the linear programming problem to solve is as follows:

$$\text{maximize} \quad \sum_{v \in L} f_{sv}$$
$$\text{subject to} \quad f_{uv} \leq 1 \text{ for each } u, v \in \{s\} \cup L \cup R \cup \{t\} = V$$
$$\sum_{v \in V} f_{vu} = \sum_{v \in V} f_{uv} \text{ for each } u \in L \cup R$$
$$f_{uv} \geq 0 \text{ for each } u, v \in V$$

Exercise 29.2-7

As in the minimum cost flow problem, we have constraints for the edge capacities, for the conservation of flow, and nonegativity. The difference is that the restraint that before we required exactly d units to flow, now, we require that for each commodity, the right amount of that commodity flows. the conservation equalities will be applied to each different type of commodity independently. If we super script f that will denote the type of commodity the flow is describing, if we do not superscript it, it will denote the aggregate flow.

We want to minimize

$$\sum_{u,v \in V} a(u,v) f_{uv}$$

The capacity constraints are that

$$\sum_{i \in [k]} \sum_{u,v \in V} f^i_{uv} \leq c(u,v)$$

The conservation constraints are that for every $i \in [k]$, for every $u \in V \setminus \{s_i, t_i\}$.

$$\sum_{v \in V} f^i_{u,v} = \sum_{v \in V} f^i_{v,u}$$

Now, the constraints that correspond to requiring a certain amount of flow are that for every $i \in [k]$.

$$\sum_{v \in V} f^i_{s_i,v} - \sum_{v \in V} f^i_{v,s_i} = d$$

Now, we put in the constraint that makes sure what we called the aggregate flow is actually the aggregate flow, so, for every $u, v \in V$,

$$f_{u,v} = \sum_{i \in [k]} f^i_{u,v}$$

Finally, we get to the fact that all flows are nonnegative, for every $u, v \in V$,

$$f_{u,v} \geq 0$$

Exercise 29.3-1

We subtract equation (29.81) from equation (29.79). This gets us

$$0 = v - v' + \sum_{j \in N} (c_j - c_j') x_j$$

which can be rearranged to

$$\sum_{j \in N} c_j' x_j = (v - v') + \sum_{j \in N} c_j x_j$$

Then, by applying Lemma 29.3, we get that for every j, we have $c_j = c_j'$ and also, $(v - v') = 0$, so $v = v'$.

Exercise 29.3-2

The only time v is updated in PIVOT is line 14, so it will suffice to show that $c_e \hat{b}_e \geq 0$. Prior to making the call to PIVOT, we choose an index e such that $c_e > 0$, and this is unchanged in PIVOT. We set $\hat{b}_e$ in line 3 to be b_l / a_{le}. The loop invariant proved in Lemma 29.2 tells us that $b_l \geq 0$. The if-condition of line 6 of SIMPLEX tells us that only the noninfinite δ_i must have $a_{ie} > 0$, and we choose l to minimize δ_l, so we must have $a_{le} > 0$. Thus, $c_e \hat{b}_e \geq 0$, which implies v can never decrease.

Exercise 29.3-3

To show that the two slack forms are equivalent, we will show both that they have equal objective functions, and their sets of feasible solutions are equal.

First, we'll check that their sets of feasible solutions are equal. Basically all we do to the constraints when we pivot is take the non-basic variable, e, and solve the equation corresponding to the basic variable l for e. We are then taking that expression and replacing e in all the constraints with this expression we got by solving the equation corresponding to l. Since each of these algebraic operations are valid, the result of the sequence of them is also algebraically equivalent to the original.

Next, we'll see that the objective functions are equal. We decrease each c_j by $c_e \hat{a}_{ej}$, which is to say that we replace the non-basic variable we are making basic with the expression we got it was equal to once we made it basic.

Since the slack form returned by PIVOT, has the same feasible region and an equal objective function, it is equivalent to the original slack form passed in.

Exercise 29.3-4

First suppose that the basic solution is feasible. We set each $x_i = 0$ for $1 \leq i \leq n$, so we have $x_{n+i} = b_i - \sum_{j=1}^{n} a_{ij}x_j = b_i$ as a satisfied constraint. Since we also require $x_{n+i} \geq 0$ for all $1 \leq i \leq m$, this implies $b_i \geq 0$. Now suppose $b_i \geq 0$ for all i. In the basic solution we set $x_i = 0$ for $1 \leq i \leq n$ which satisfies the nonnegativity constraints. We set $x_{n+i} = b_i$ for $1 \leq i \leq m$ which satisfies the other constraint equations, and also the nonnegativity constraints on the basic variables since $b_i \geq 0$. Thus, every constraint is satisfied, so the basic solution is feasible.

Exercise 29.3-5

First, we rewrite the linear program into it's slack form, we want to maximize $18x_1 + 12.5x_2$, given the constraints

$$x_3 = 20 - x_1 - x_2$$
$$x_4 = 12 - x_1$$
$$x_5 = 16 - x_2$$
$$x_1, x_2, x_3, x_4, x_5 \geq 0$$

Then, we take the initial basic solution, we get that $x_1 = x_2 = 0$ and $x_3 = 20$, $x_4 = 12$, and $x_5 = 16$, with a value of the objective function of 0. Now, we pick x_1 as our non basic variable in the simplex method. Of all of our Δ values, we get that the smallest corresponds to x_4, so, we pivot to x_4 from x_1. This gets us that we want to maximize $216 - 18x_4 + 12.5x_2$ subject to the constraints:

$$x_3 = 8 + x_4 - x_2$$
$$x_1 = 12 + x_4$$
$$x_5 = 16 - x_2$$
$$x_1, x_2, x_3, x_4, x_5 \geq 0$$

Then, we need to select x_2 as our non-basic variable, which gets us that we should pivot to x_3, which gets us that the objective is $316 - 5.5x_4 - 12.5x_3$ and the constraints are

$$x_2 = 8 + x_4 - x_3$$
$$x_1 = 12 + x_4$$
$$x_5 = 8 - x_4 + x_3$$
$$x_1, x_2, x_3, x_4, x_5 \geq 0$$

We now stop since no more non-basic variables appear in the objective with a positive coefficient. Our solution is $(12, 8, 0, 0, 8)$ and has a value of 316. Going back to the standard form we started with, we just disregard the values of x_3

through x_5 and have the solution that $x_1 = 12$ and $x_2 = 8$. We can check that this is both feasible and has the objective achieve 316.

Exercise 29.3-6

The first step is to convert the linear program into slack form. We'll introduce the slack variables x_3 and x_4. We have:

$$z = 5x_1 - 3x_2$$
$$x_3 = 1 - x_1 + x_2$$
$$x_4 = 2 - 2x_1 - x_2.$$

The nonbasic variables are x_1 and x_2. Of these, only x_1 has a positive coefficient in the objective function, so we must choose $x_e = x_1$. Both equations limit x_1 by 1, so we'll choose the first one to rewrite x_1 with. Using $x_1 = 1 - x_3 + x_2$ we obtain the new system

$$z = 5 - 5x_3 + 2x_2$$
$$x_1 = 1 - x_3 + x_2$$
$$x_4 = 2x_3 - 2x_2.$$

Now x_2 is the only nonbasic variable with positive coefficient in the objective function, so we set $x_e = x_2$. The last equation limits x_2 by 0 which is most restrictive, so we set $x_2 = x_3 - \frac{x_4}{2}$. Rewriting, our new system becomes

$$z = 5 - 3x_3 - x_4$$
$$x_1 = 1 - \frac{x_4}{2}$$
$$x_2 = x_3 - \frac{x_4}{2}.$$

Every nonbasic variable now has negative coefficient in the objective function, so we take the basic solution $(x_1, x_2, x_3, x_4) = (1, 0, 0, 0)$. The objective value this achieves is 5.

Exercise 29.3-7

First, we convert this equation to the slack form. Doing so doesn't change the objective, but the constraints become

$$z = -x_1 - x_2 - x_3$$
$$x_4 = -10000 + 2x_1 + 7.5x_2 + 3x_3$$
$$x_5 = -30000 + 20x_1 + 5x_2 + 10x_3$$
$$x_1, x_2, x_3, x_4, x_5 \geq 0$$

Also, since the objective is to minimize a given function, we'll change it over to maximizing the negative of that function. In particular maximize $-x_1 - x_2 - x_3$. Now, we note that the initial basic solution is not feasible, because it would leave x_4 and x_5 being negative. This means that finding an initial solution requires using the method of section 29.5. The auxiliary linear program in slack form is

$$z = -x_0$$
$$x_4 = -10000 + 2x_1 + 7.5x_2 + 3x_3 + x_0$$
$$x_5 = -30000 + 20x_1 + 5x_2 + 10x_3 + x_0$$
$$x_0, x_1, x_2, x_3, x_4, x_5 \geq 0$$

We choose x_0 as the entering variable and x_5 as the leaving variable, since it is the basic variable whose value in the basic solution is most negative. After pivoting, we have the slack form

$$z = -30000 + 20x_1 + 5x_2 + 10x_3 - x_5$$
$$x_0 = 30000 - 20x_1 - 5x_2 - 10x_3 + x_5$$
$$x_4 = 20000 - 18x_1 + 2.5x_2 - 7x_3 + x_5$$
$$x_0, x_1, x_2, x_3, x_4, x_5 \geq 0$$

The associated basic solution is feasible, so now we just need to repeatedly call PIVOT until we obtain an optimal solution to L_{aux}. We'll choose x_2 as our entering variable. This gives

$$z = -x_0$$
$$x_2 = 6000 - 4x_1 - 2x_3 + x_5/5 - x_0/5$$
$$x_4 = 35000 - 28x_1 - 12x_3 + (3/2)x_5 - x_0/2$$
$$x_0, x_1, x_2, x_3, x_4, x_5 \geq 0$$

This slack form is the final solution to the auxiliary problem. Since this solution has $x_0 = 0$, we know that our initial problem was feasible. Furthermore, since $x_0 = 0$, we can just remove it from the set of constraints. We then restore the original objective function, with appropriate substitutions made to include only the nonbasic variables. This yields

$$z = -6000 + 3x_1 + x_3 - x_5/5$$
$$x_2 = 6000 - 4x_1 - 2x_3 + x_5/5$$
$$x_4 = 35000 - 28x_1 - 12x_3 + (3/2)x_5$$
$$x_1, x_2, x_3, x_4, x_5 \geq 0$$

This slack form has a feasible basic solution, and we can return it to SIM-PLEX. We choose x_1 as our entering variable. This gives

$$z = -2250 - (2/7)x_3 - (3/28)x_4 - (11/280)x_5$$
$$x_1 = 1250 - (3/7)x_3 - (1/28)x_4 + (3/56)x_5$$
$$x_2 = 1000 - (2/7)x_3 + (1/7)x_4 - (1/70)x_5$$
$$x_1, x_2, x_3, x_4, x_5 \geq 0.$$

At this point, all coefficients in the objective function are negative, so the basic solution is an optimal solution. This solution is $(x_1, x_2, x_3) = (1250, 1000, 0)$.

Exercise 29.3-8

Consider the simple program

$$z = -x_1$$
$$x_2 = 1 - x_1.$$

In this case we have $m = n = 1$, so $\binom{m+n}{n} = \binom{2}{1} = 2$, however, since the only coefficients of the objective function are negative, we can't make any other choices for basic variable. We must immediately terminate with the basic solution $(x_1, x_2) = (0, 1)$, which is optimal.

Exercise 29.4-1

By just transposing A, swapping b and c, and switching the maximization to a minimization, we want to minimize $20y_1 + 12y_2 + 16y_3$ subject to the constraints

$$y_1 + y_2 \geq 18$$
$$y_1 + y_3 \geq 12.5$$
$$y_1, y_2, y_3 \geq 0$$

Exercise 29.4-2

By working through each aspect of putting a general linear program into standard form, as outlined on page 852, we can show how to deal with transforming each into the dual individually. If the problem is a minimization instead of a maximization, replace c_j by $-c_j$ in (29.84). If there is a lack of nonnegativity constraint on x_j we duplicate the jth column of A, which corresponds to duplicating the jth row of A^T. If there is an equality constraint for b_i, we convert it to two inequalities by duplicating then negating the ith column of A^T, duplicating then negating the ith entry of b, and adding an extra y_i variable. We handle the greater-than-or-equal-to sign $\sum_{i=1}^{n} a_{ij} x_j \geq b_i$ by negating i^{th} column of A^T and negating b_i. Then we solve the dual problem of minimizing $b^T y$ subject to $A^T y \geq c$ and $y \geq 0$.

Exercise 29.4-3

First, we'll convert the linear program for maximum flow described in equation (29.47)-(29.50) into standard form. The objective function says that c is a vector indexed by a pair of vertices, and it is positive one if s is the first index and negative one if s is the second index (zero if it is both). Next, we'll modify the constraints by switching the equalities over into inequalities to get

$$
\begin{aligned}
f_{u,v} &\leq c(u,v) && \text{for each } u, v \in V \\
\sum_{u \in V} f_{vu} &\leq \sum_{u \in V} f_{uv} && \text{for each } v \in V - \{s, t\} \\
\sum_{u \in V} f_{vu} &\geq \sum_{u \in V} f_{uv} && \text{for each } v \in V - \{s, t\} \\
f_{u,v} &\geq 0 && \text{for each } u, v \in V
\end{aligned}
$$

Then, we'll convert all but the last set of the inequalities to be $\leq$ by multiplying the third line by -1.

$$
\begin{aligned}
f_{u,v} &\leq c(u,v) && \text{for each } u, v \in V \\
\sum_{u \in V} f_{vu} &\leq \sum_{u \in V} f_{uv} && \text{for each } v \in V - \{s, t\} \\
\sum_{u \in V} -f_{vu} &\leq \sum_{u \in V} -f_{uv} && \text{for each } v \in V - \{s, t\} \\
f_{u,v} &\geq 0 && \text{for each } u, v \in V
\end{aligned}
$$

Finally, we'll bring all the variables over to the left to get

$$f_{u,v} \leq c(u,v) \qquad\qquad \text{for each } u,v \in V$$

$$\sum_{u \in V} f_{vu} - \sum_{u \in V} f_{uv} \leq 0 \qquad\qquad \text{for each } v \in V - \{s,t\}$$

$$\sum_{u \in V} -f_{vu} - \sum_{u \in V} -f_{uv} \leq 0 \qquad\qquad \text{for each } v \in V - \{s,t\}$$

$$f_{u,v} \geq 0 \qquad\qquad \text{for each } u,v \in V$$

Now, we can finally write down our A and b. A will be a $|V|^2 \times |V|^2 + 2|V| - 4$ matrix built from smaller matrices A_1 and A_2 which correspond to the three types of constraints that we have (of course, not counting the non-negativity constraints). We will let $g(u,v)$ be any bijective mapping from $V \times V$ to $[|V|^2]$. We'll also let h be any bijection from $V - \{s,t\}$ to $[|V| - 2]$

$$A = \begin{pmatrix} A_1 \\ A_2 \\ -A_2 \end{pmatrix}$$

Where A_1 is defined as having its row $g(u,v)$ be all zeroes except for having the value 1 at at the $g(u,v)$th entry. We define A_2 to have it's row $h(u)$ be equal to 1 at all columns j for which $j = g(v,u)$ for some v and equal to -1 at all columns j for which $j = g(u,v)$ for some v. Lastly, we mention that b is defined as having it's jth entry be equal to $c(u,v)$ if $j = g(u,v)$ and zero if $j > |V|^2$.

Now that we have placed the linear program in standard form, we can take its dual. We want to minimize $\sum_{i=1}^{|V|^2 + 2|V| - 2} b_i y_i$ given the constraints that all the y values are non-negative, and $A^T y \geq c$.

Exercise 29.4-4

First we need to put the linear programming problem into standard form, as follows:

$$\text{maximize} \quad \sum_{(u,v)\in E} -a(u,v)f_{uv}$$

$$\text{subject to} \quad f_{uv} \leq c(u,v) \text{ for each } u,v \in V$$

$$\sum_{v\in V} f_{vu} - \sum_{v\in V} f_{uv} \leq 0 \text{ for each } u \in V - \{s,t\}$$

$$\sum_{v\in V} f_{uv} - \sum_{v\in V} f_{vu} \leq 0 \text{ for each } u \in V - \{s,t\}$$

$$\sum_{v\in V} f_{sv} - \sum_{v\in V} f_{vs} \leq d$$

$$\sum_{v\in V} f_{vs} - \sum_{v\in V} f_{sv} \leq -d$$

$$f_{uv} \geq 0.$$

We now formulate the dual problem. Let the vertices be denoted $v_1, v_2, \ldots, v_n, s, t$ and the edges be $e_1, e_2, \ldots, e_k$. Then we have $b_i = c(e_i)$ for $1 \leq i \leq k$, $b_i = 0$ for $k+1 \leq i \leq k+2n$, $b_{k+2n+1} = d$, and $b_{k+2n+2} = -d$. We also have $c_i = -a(e_i)$ for $1 \leq i \leq k$. For notation, let $j.left$ denote the tail of edge e_j and $j.right$ denote the head. Let $\chi_s(e_j) = 1$ if e_j enters s, set it equal to -1 if e_j leaves s, and set it equal to 0 if e_j is not incident with s. The dual problem is:

$$\text{minimize} \quad \sum_{i=1}^{k} c(e_i)y_i + dy_{k+2n+1} - dy_{k+2n+2}$$

$$\text{subject to} \quad y_j + y_{k+e_j.right} - y_{k+j.left} - y_{k+n+e_j.right} + y_{k+n+e_j.left}$$
$$- \chi_s(e_j)y_{k+2n+1} + \chi_s(e_j)y_{k+2n+2} \geq -a(e_j)$$

where j runs between 1 and k. There is one constraint equation for each edge e_j.

Exercise 29.4-5

Suppose that our original linear program is in standard form for some A, b, c. Then, the dual of this is to minimize $\sum_{i=1}^{m} b_i y_i$ subject to $A^T y \geq c$ This can be rewritten as wanting to maximize $\sum_{i=1}^{m} (-b_i)y_i$ subject to $(-A)^T y \leq -c$. Since this is a standard form, we can take its dual easily, it is minimize $\sum_{j=1}^{n} (-c_j)x_j$ subject to $(-A)x \geq -b$. This is the same as minimizing $\sum_{j=1}^{n} c_j x_j$ subject to $Ax \leq b$, which was the original linear program.

Exercise 29.4-6

Corollary 26.5 from Chapter 26 can be interpreted as weak duality.

Exercise 29.5-1

For line 5, first let (N, B, A, b, c, v) be the result of calling PIVOT on L_{aux} using x_0 as the entering variable. Then repeatedly call PIVOT until an optimal solution to L_{aux} is obtained, and return this to (N, B, A, b, c, v). To remove x_0 from the constraints, set $a_{i,0} = 0$ for all $i \in B$, and set $N = N \backslash \{0\}$. To restore the original objective function of L, for each $j \in N$ and each $i \in B$, set $c_j = c_j - c_i a_{ij}$.

Exercise 29.5-2

In order to enter line 10 of INITIALIZE-SIMPLEX and begin iterating the main loop of SIMPLEX, we must have recovered a basic solution which is feasible for L_{aux}. Since $x_0 \geq 0$ and the objective function is $-x_0$, the objective value associated to this solution (or any solution) must be negative. Since the goal is to aximize, we have an upper bound of 0 on the objective value. By Lemma 29.2, SIMPLEX correctly determines whether or not the input linear program is unbounded. Since L_{aux} is not unbounded, this can never be returned by SIMPLEX.

Exercise 29.5-3

Since it is in standard form, the objective function has no constant term, it is entirely given by $\sum_{i=1}^{n} c_i x_i$, which is going to be zero for any basic solution. The same thing goes for its dual. Since there is some solution which has the objective function achieve the same value both for the dual and the primal, by the corollary to the weak duality theorem, that common value must be the optimal value of the objective function.

Exercise 29.5-4

Consider the linear program in which we wish to maximize x_1 subject to the constraint $x_1 < 1$ and $x_1 \geq 0$. This has no optimal solution, but it is clearly bounded and has feasible solutions. Thus, the Fundamental theorem of linear programming does not hold in the case of strict inequalities.

Exercise 29.5-5

The initial basic solution isn't feasible, so we will need to form the auxiliary linear program:

15

$$\text{maximize} \quad -x_0$$
$$\text{subject to } x_1 - x_2 - x_0 \leq 8$$
$$-x_1 - x_2 - x_0 \leq -3$$
$$-x_1 + 4x_2 - x_0 \leq 2$$
$$x_1, x_2, x_0 \geq 0.$$

Then we write this linear program in slack form:

$$z = -x_0$$
$$x_3 = 8 - x_1 + x_2 + x_0$$
$$x_4 = -3 + x_1 + x_2 + x_0$$
$$x_5 = 2 + x_1 - 4x_2 + x_0$$
$$x_1, x_2, x_3, x_4, x_5, x_0 \geq 0.$$

Next we make one call to PIVOT where x_0 is the entering variable and x_4 is the leaving variable. This gives:

$$z = -3 + x_1 + x_2 - x_4$$
$$x_0 = 3 - x_1 - x_2 + x_4$$
$$x_3 = 11 - 2x_1 + x_4$$
$$x_5 = 5 - 5x_2 + x_4$$
$$x_1, x_2, x_3, x_4, x_5, x_0 \geq 0.$$

The basic solution is feasible, so we repeatedly call PIVOT to get the optimal solution to L_{aux}. We'll choose x_1 to be our entering variable and x_0 to be the leaving variable. This gives

$$z = -x_0$$
$$x_1 = 3 - x_0 - x_2 + x_4$$
$$x_3 = 5 + 2x_0 + 2x_2 - x_4$$
$$x_5 = 5 - 5x_2 + x_4$$
$$x_1, x_2, x_3, x_4, x_5, x_0 \geq 0.$$

The basic solution is now optimal for L_{aux}, so we return this slack form to SIMPLEX, set $x_0 = 0$, and update the objective function which yields

$$z = 3 + 2x_2 + x_4$$
$$x_1 = 3 - x_2 + x_4$$
$$x_3 = 5 + 2x_2 - x_4$$
$$x_5 = 5 - 5x_2 + x_4$$
$$x_1, x_2, x_3, x_4, x_5, x_0 \geq 0.$$

We'll choose x_2 as our entering variable, which makes x_5 our leaving variable. PIVOT then gives

$$z = 5 + (7/5)x_4 - (2/5)x_5$$
$$x_2 = 1 + (1/5)x_4 - (1/5)x_5$$
$$x_1 = 2 + (4/5)x_4 + (1/5)x_5$$
$$x_3 = 7 - (3/5)x_4 - (2/5)x_5$$
$$x_1, x_2, x_3, x_4, x_5, x_0 \geq 0.$$

We'll choose x_4 as our entering variable, which makes x_3 our leaving variable. PIVOT then gives

$$z = (64/3) - (7/3)x_3 - (4/3)x_5$$
$$x_4 = (35/3) - (5/3)x_3 - (2/3)x_5$$
$$x_2 = (10/3) - (1/3)x_3 - (1/3)x_5$$
$$x_1 = (34/3) - (4/3)x_3 - (1/3)x_5$$
$$x_1, x_2, x_3, x_4, x_5, x_0 \geq 0.$$

Now all coefficients in the objective function are negative, so the basic solution is the optimal solution. It is $(x_1, x_2) = (34/3, 10/3)$.

Exercise 29.5-6

The initial basic solution isn't feasible, so we will need to form the auxiliary linear program:

$$\begin{aligned}
\text{maximize} \quad & -x_0 \\
\text{subject to} \quad & x_1 + 2x_2 - x_0 \leq 4 \\
& -2x_1 - 6x_2 - x_0 \leq -12 \\
& x_2 - x_0 \leq 1 \\
& x_1, x_2, x_0 \geq 0.
\end{aligned}$$

Then we write this linear program in slack form:

$$z = -x_0$$
$$x_3 = 4 - x_1 - 2x_2 + x_0$$
$$x_4 = -12 + 2x_1 + 6x_2 + x_0$$
$$x_5 = 1 - x_2 + x_0.$$

Next we make one call to PIVOT where x_0 is the entering variable and x_4 is the leaving variable. This gives:

$$z = -12 + 2x_1 + 6x_2 - x_4$$
$$x_3 = 16 - 3x_1 - 8x_2 + x_4$$
$$x_0 = 12 + x_4 - 2x_1 - 6x_2$$
$$x_5 = 13 - 2x_1 - 8x_2 + x_4.$$

The basic solution is $(x_0, x_1, x_2, x_3, x_4, x_5) = (12, 0, 0, 16, 0, 13)$ which is feasible for the auxiliary program. Now we need to run SIMPLEX to find the optimal objective value to L_{aux}. Let x_1 be our next entering variable. It is most constrained by x_3, which will be our leaving variable. After PIVOT, the new linear program is

$$z = -4/3 + (2/3)x_2 - (2/3)x_3 - (1/3)x_4$$
$$x_1 = 16/3 - (8/3)x_2 - (1/3)x_3 + (1/3)x_4$$
$$x_0 = 4/3 - (2/3)x_2 + (2/3)x_3 + (1/3)x_4$$
$$x_5 = 7/3 - (8/3)x_2 + (2/3)x_3 + (1/3)x_4.$$

Every coefficient in the objective function is negative, so we take the basic solution $(x_0, x_1, x_2, x_3, x_4, x_5) = (4/3, 16/3, 0, 0, 0, 7/3)$ which is also optimal. Since $x_0 \neq 0$, the original linear program must be unfeasible.

Exercise 29.5-7

The initial basic solution isn't feasible, so we will need to form the auxiliary linear program:

$$\text{maximize} \quad -x_0$$
$$\text{subject to} \quad -x_1 + x_2 - x_0 \le -1$$
$$-x_1 - x_2 - x_0 \le -3$$
$$-x_1 + 4x_2 - x_0 \le 2$$
$$x_1, x_2, x_0 \ge 0.$$

Then we write this linear program in slack form:

$$z = -x_0$$
$$x_3 = -1 + x_1 - x_2 + x_0$$
$$x_4 = -3 + x_1 + x_2 + x_0$$
$$x_5 = 2 + x_1 - 4x_2 + x_0$$
$$x_1, x_2, x_3, x_4, x_5, x_0 \ge 0.$$

Next we make one call to PIVOT where x_0 is the entering variable and x_4 is the leaving variable. This gives:

$$z = -3 + x_1 + x_2 - x_4$$
$$x_0 = 3 - x_1 - x_2 + x_4$$
$$x_3 = 2 - 2x_2 + x_4$$
$$x_5 = 5 - 5x_2 + x_4$$
$$x_1, x_2, x_3, x_4, x_5, x_0 \ge 0.$$

Let x_1 be our entering variable. Then x_0 is our leaving variable, and we have

$$z = -x_0$$
$$x_1 = 3 - x_0 - x_2 + x_4$$
$$x_3 = 2 - 2x_2 + x_4$$
$$x_5 = 5 - 5x_2 + x_4$$
$$x_1, x_2, x_3, x_4, x_5, x_0 \ge 0.$$

The basic solution is feasible, and optimal for L_{aux}, so we return this and run SIMPLEX. Updating the objective function and setting $x_0 = 0$ gives

$$z = 3 + 2x_2 + x_4$$
$$x_1 = 3 - x_2 + x_4$$
$$x_3 = 2 - 2x_2 + x_4$$
$$x_5 = 5 - 5x_2 + x_4$$
$$x_1, x_2, x_3, x_4, x_5, x_0 \geq 0.$$

We'll choose x_2 as our entering variable, which makes x_3 our leaving variable. This gives

$$z = 5 - x_3 + 2x_4$$
$$x_2 = 1 - (1/2)x_3 + (1/2)x_4$$
$$x_1 = 2 + (1/2)x_3 + (1/2)x_4$$
$$x_5 = (5/2)x_3 - (3/2)x_4$$
$$x_1, x_2, x_3, x_4, x_5, x_0 \geq 0.$$

Next we use x_4 as our entering variable, which makes x_5 our leaving variable. This gives

$$z = 5 + (7/3)x_3 - (4/3)x_5$$
$$x_4 = (5/3)x_3 - (2/3)x_5$$
$$x_2 = 1 + (1/3)x_3 - (1/3)x_5$$
$$x_1 = 2 + (4/3)x_3 - (1/3)x_5$$
$$x_1, x_2, x_3, x_4, x_5, x_0 \geq 0.$$

Finally, we would like to choose x_3 as our entering variable, but every coefficient on x_3 is positive, so SIMPLEX returns that the linear program is unbounded.

Exercise 29.5-8

We first put the linear program in standard form:

$$\begin{aligned}
\text{maximize} \quad & x_1 + x_2 + x_3 + x_4 \\
\text{subject to} \quad & 2x_1 - 8x_2 - 10x_4 \leq -50 \\
& -5x_1 - 2x_2 \leq -100 \\
& -3x_1 + 5x_2 - 10x_3 + 2x_4 \leq -25 \\
& x_1, x_2, x_3, x_4 \geq 0.
\end{aligned}$$

The initial basic solution isn't feasible, so we will need to form the auxiliary linear program. It is given below in slack form:

$$z = -x_0$$
$$x_5 = -50 - 2x_1 + 8x_2 + 10x_4 + x_0$$
$$x_6 = -100 + 5x_1 + 2x_2 + x_0$$
$$x_7 = -25 + 3x_1 - 5x_2 + 10x_3 - 2x_4 + x_0.$$

The index of the minimum b_i is 2, so we take x_0 to be our entering variable and x_6 to be our leaving variable. The call to PIVOT on line 8 yields

$$z = -100 + 5x_1 + 2x_2 - x_6$$
$$x_5 = 50 - 7x_1 + 8x_2 + 10x_4 + x_6$$
$$x_0 = 100 - 5x_1 - 2x_2 + x_6$$
$$x_7 = 75 - 2x_1 - 7x_2 + 10x_3 - 2x_4 + x_6.$$

Next we'll take x_2 to be our entering variable and x_5 to be our leaving variable. The call to PIVOT yields

$$z = -225/2 + (27/4)x_1 - (10/4)x_4 + (1/4)x_5 - (5/4)x_6$$
$$x_2 = -50/8 + (7/8)x_1 - (10/8)x_4 + (1/8)x_5 - (1/8)x_6$$
$$x_0 = 225/2 - (27/4)x_1 + (10/4)x_4 - (1/4)x_5 + (5/4)x_6$$
$$x_7 = 475/4 - (65/8)x_1 + 10x_3 + (54/8)x_4 - (7/8)x_5 + (15/8)x_6.$$

The work gets rather messy, but INITIALIZE-SIMPLEX does eventually give a feasible solution to the linear program, and after running the simplex method we find that $(x_1, x_2, x_3, x_4) = (175/11, 225/22, 125/44, 0)$ is an optimal solution to the original linear programming problem.

Exercise 29.5-9

1. One option is that $r = 0$, $s \geq 0$ and $t \leq 0$. Suppose that $r > 0$, then, if we have that s is non-negative and t is non-positive, it will be as we want.

2. We will split into two cases based on r. If $r = 0$, then this is exactly when t is non-positive and s is non-negative. The other possible case is that r is negative, and t is positive. In which case, because r is negative, we can always get rx as small as we want so s doesn't matter, however, we can never make rx positive so it can never be $\geq t$.

3. Again, we split into two possible cases for r. If $r = 0$, then it is when t is non-negative and s is non-positive. The other possible case is that r is positive, and s is negative. Since r is positive, rx will always be non-negative, so it cannot be $\leq s$. But since r is positive, we have that we can always make rx as big as we want, in particular, greater than t.

4. If we have that $r = 0$ and t is positive and s is negative. If r is nonzero, then we can always either make rx really big or really small depending on the sign of r, meaning that either the primal or the dual would be feasable.

Problem 29-1

a. We just let the linear inequalities that we need to satisfy be our set of constraints in the linear program. We let our function to maximize just be a constant. The solver for linear programs would fail to detect any feasible solution if the linear constraints were not feasible. If the linear programming solver returns any solution at all, we know that the linear constraints are feasible.

b. Suppose that we are trying to solve the linear program in standard form with some particular A, b, c. That is, we want to maximize $\sum_{j=1}^{n} c_j x_j$ subject to $Ax \leq b$ and all entries of the x vector are non-negative. Now, consider the dual program, that is, we want to minimize $\sum_{i=1}^{m} b_i y_i$ subject to $A^T y \geq c$ and all the entries in the y vector are nonzero. We know by Corollary 29.9, if x and y are feasible solutions to their respective problems, then, if we have that their objective functions are equal, then, they are both optimal solutions.

We can force their objective functions to be equal. To do this, let c_k be some nonzero entry in the c vector. If there are no nonzero entries, then the function we are trying to optimize is just the zero function, and it is exactly a feasibility question, so we we would be done. Then, we add two linear inequalities to require $x_k = \frac{1}{c_k} \left(\sum_{i=1}^{m} b_i y_i - \sum_{j=1}^{n} c_j x_j \right)$. This will require that whatever values the variables take, their objective functions will be equal. Lastly, we just throw these in with the inequalities we already had. So, the constraints will be:

$$Ax \leq b$$
$$A^T y \geq c$$
$$x_k \leq \frac{1}{c_k} \left(\sum_{i=1}^{m} b_i y_i - \sum_{j=1}^{n} c_j x_j \right)$$
$$x_k \geq \frac{1}{c_k} \left(\sum_{i=1}^{m} b_i y_i - \sum_{j=1}^{n} c_j x_j \right)$$
$$x_1, x_2, \ldots, x_n, y_1, y_2, \ldots y_m \geq 0$$

22

We have a number of variables equal to $n + m$ and a number of constraints equal to $2+2n+2m$, so both are polynomial in n and m. Also, any assignment of variables which satisfy all of these constraints will be a feasible solution to both the problem and its dual that cause the respective objective functions to take the same value, and so, must be an optimal solution to both the original problem and its dual. This of course assumes that the linear inequality feasibility solver doesn't merely say that the inequalities are satisfiable, but actually returns a satisfying assignment.

Lastly, it is necessary to note that if there is some optimal solution x, then, we can obtain an optimal solution for the dual that makes the objective functions equal by theorem 29.10. This ensures that the two constraints we added to force the objectives of the primal and the dual to be equal don't cause us to change the optimal solution to the linear program.

Problem 29-2

a. An optimal solution to the LP program given in (29.53) - (29.57) is $(x_1, x_2, x_3) = (8, 4, 0)$. An optimal solution to the dual is $(y_1, y_2, y_3) = (0, 1/6, 2/3)$. It is then straightforward to verify that the equations hold.

b. First suppose that complementary slackness holds. Then the optimal objective value of the primal problem is, if it exists,

$$\sum_{k=1}^{n} c_k x_k = \sum_{k=1}^{n}\sum_{i=1}^{m} a_{ik} y_i x_k$$
$$= \sum_{i=1}^{m}\sum_{k=1}^{n} a_{ik} x_k y_i$$
$$= \sum_{i=1}^{m} b_i y_i$$

which is precisely the optimal objective value of the dual problem. If any x_j is 0, then those terms drop out of them sum, so we can safely replace c_k by whatever we like in those terms. Since the objective values are equal, they must be optimal. An identical argument shows that if an optimal solution exists for the dual problem then any feasible solution for the primal problem which satisfies the second equality of complementary slackness must also be optimal.

Now suppose that x and y are optimal solutions, but that complementary slackness fails. In other words, there exists some j such that $x_j \neq 0$ but $\sum_{i=1}^{m} a_{ij} y_i > c_j$, or there exists some i such that $y_i \neq 0$ but $\sum_{j=1}^{n} a_{ij} x_j < b_i$. In the first case we have

$$\sum_{k=1}^{n} c_k x_k < \sum_{k=1}^{n}\sum_{i=1}^{m} a_{ik} y_i x_k$$
$$= \sum_{i=1}^{m}\sum_{k=1}^{n} a_{ik} x_k y_i$$
$$= \sum_{i=1}^{m} b_i y_i.$$

This implies that the optimal objective value of the primal solution is strictly less than the optimal value of the dual solution, a contradiction. The argument for the second case is identical. Thus, x and y are optimal solutions if and only if complementary slackness holds.

c. This follows immediately from part b. If x is feasible and y satisfies conditions 1, 2, and 3, then complementary slackness holds, so x and y are optimal. On the other hand, if x is optimal, then the dual linear program must have an optimal solution y as well, according to Theorem 29.10. Optimal solutions are feasible, and by part b x and y satisfy complementary slackness. Thus, conditions 1, 2, and 3 hold.

Problem 29-3

a. The proof for weak duality goes through identically. Nowhere in it does it use the integrality of the solutions.

b. Consider the linear program given in standard form by $A = (1)$, $b = \left(\frac{1}{2}\right)$ and $c = (2)$. The highest we can get this is 0 since that's that only value that x can be. Now, consider the dual to this, that is, we are trying to minimize $\frac{x}{2}$ subject to the constraint that $x \geq 2$. This will be minimized when $x = 2$, so, the smallest solution we can get is 1.

Since we have just exhibited an example of a linear program that has a different optimal solution as it's dual, the duality theorem does not hold for integer linear program.

c. The first inequality comes from looking at the fact that by adding the restriction that the solution must be integer valued, we obtain a set of feasible solutions that is a subset of the feasible solutions of the original primal linear program. Since, to get IP, we are taking the max over a subset of the things we are taking a max over to get P, we must get a number that is no larger. The third inequality is similar, except since we are taking min over a subset, the inequality goes the other way.

The middle equality is given by Theorem 29.10.

Problem 29-4

Suppose that both systems are solvable, let x denote a solution to the first system, and y denote a solution to the second. Taking transposes we have $x^T A^T \leq 0^T$. Right multiplying by y gives $x^T c = x^T A^T y \leq 0^T$, which is a contradiction to the fact that $c^T x > 0$. Thus, both systems cannot be simultaneously solved. Now suppose that the second system fails. Consider the following linear program:

$$\text{maximize } 0x \text{ subject to } A^T y = c \text{ and } y \geq 0$$

and its corresponding dual program

$$\text{minimize } -c^T x \text{ subject to } Ax \leq 0.$$

Since the second system fails, the primal is infeasible. However, the dual is always feasible by taking $x = 0$. If there were a finite solution to the dual, then duality says there would also be a finite solution to the primal. Thus, the dual must be unbounded. Thus, there must exist a vector x which makes $-c^T x$ arbitrarily small, implying that there exist vectors x for which $c^T x$ is strictly greater than 0. Thus, there is always at least one solution.

Problem 29-5

a. This is exactly the linear program given in equations (29.51) - (29.52) except that the equation on the third line of the constraints should be removed, and for the equation on the second line of the constraints, u should be selected from all of V instead of from $V \setminus \{s, t\}$.

b. If $a(u, v) > 0$ for every pair of vertices, then, there is no point in sending any flow at all. So, an optimal solution is just to have no flow. This obviously satisfies the capacity constraints, it also satisfies the conservation constraints because the flow into and out of each vertex is zero.

c. We assume that the edge (t, s) is not in E because that would be a silly edge to have for a maximum flow from s to t. If it is there, remove it and it won't decrease the maximum flow. Let $V' = V$ and $E' = E \cup \{(t, s)\}$. For the edges of E' that are in E, let the capacity be as it is in E and let the cost be 0. For the other edge, we set $c(t, s) = \infty$ and $a(t, s) = -1$. Then, if we have any circulation in G', it will be trying to get as much flow to go across the edge (t, s) in order to drive the objective function lower, the other flows will have no affect on the objective function. Then, by Kirchhoff's current law (a.k.a. common sense), the amount going across the edge (t, s) is the same as the total flow in the rest of the network from s to t. This means that maximizing the flow across the edge (t, s) is also maximizing the flow from s to t. So, all we need to do to recover the maximum flow for the original network is to keep the same flow values, but throw away the edge (t, s).

25

d. Suppose that s is the vertex that we are computing shortest distance from. Then, we make the circulation network by first starting with the original graph, giving each edge a cost of whatever it was before and infinite capacity. Then, we place an edge going from every vertex that isn't s to s that has a capacity of 1 and a cost of $-|E|$ times the largest cost appearing among all the edge costs already in the graph. Giving it such a negative cost ensures that placing other flow through the network in order to get a unit of flow across it will cause the total cost to decrease. Then, to recover the shortest path for any vertex, start at that vertex and then go to any vertex that is sending a unit of flow to it. Repeat this until you've reached s.

Chapter 30

Michelle Bodnar, Andrew Lohr

April 20, 2016

Exercise 30.1-1

$$(56)x^6 - 8x^5 + (8 - 42)x^4 + (-80 + 6 + 21)x^3 + (-3 - 6)x^2 + (60 + 3)x - 30$$

which is

$$56x^6 - 8x^5 - 34x^4 - 53x^3 - 9x^2 + 63x - 30$$

Exercise 30.1-2

Let A be the matrix with 1's on the diagonal, $-x_0$'s on the super diagonal, and 0's everywhere else. Let q be the vector $(r, q_0, q_1, \ldots, x_{n-2})$. If $a = (a_0, a_1, \ldots, a_{n-1})$ then we need to solve the matrix equation $Aq = a$ to compute the remainder and coefficients. Since A is tridiagonal, Problem 28-1 (e) tells us how to solve this equation in linear time.

Exercise 30.1-3

For each pair of points, $(p, A(p))$, we can compute the pair $(\frac{1}{p}, A^{rev}(\frac{1}{p}))$. To do this, we note that $A^{rev}(\frac{1}{p}) = \sum_{j=0}^{n-1} a_{n-1-j} \left(\frac{1}{p}\right)^j = \sum_{j=0}^{n-1} a_j \left(\frac{1}{p}\right)^{n-1-j} = p^{1-n} \sum_{j=0}^{n-1} a_j p^j = p^{1-n} A(p)$ since we know what $A(p)$ is, we can compute $A^{rev}(\frac{1}{p})$ of course, we are using the fact that $p \neq 0$ because we are dividing by it. Also, we know that each of these points are distinct, because $\frac{1}{p} = \frac{1}{p'}$ implies that $p = p'$ by cross multiplication. So, since all the x values were distinct in the point value representation of A, they will be distinct in this point value representation of A^{rev} that we have made.

Exercise 30.1-4

Suppose that just $n-1$ point-value pairs uniquely determine a polynomial P which satisfies them. Append the point value pair (x_{n-1}, y_{n-1}) to them, and let P' be the unique polynomial which agrees with the n pairs, given by Theorem 30.1. Now append instead (x_{n-1}, y'_{n-1}) where $y_{n-1} \neq y'_{n-1}$, and let P'' be the

polynomial obtained from these points via Theorem 30.1. Since polynomials coming from n pairs are unique, $P' \neq P''$. However, P' and P'' agree on the original $n-1$ point-value pairs, contradicting the fact that P was determined uniquely.

Exercise 30.1-5

First, we show that we can compute the coefficient representation of $\prod_j (x - x_j)$ in time $\Theta(n^2)$. We will do it by recursion, showing that multiplying $\prod_{j<k}(x - x_j)$ by $(x - x_k)$ only takes time $O(n)$, since this only needs to be done n times, this gets is total runtime of $O(n)$. Suppose that $\sum_{i=0}^{k-1} k_i x^i$ is a coefficient representation of $\prod_{j<k}(x - x_j)$. To multiply this by $(x - x_k)$, we just set $(k+1)_i = k_{i-1} - x_k k_i$ for $i = 1, \ldots k$ and $(k+1)_0 = -x_k \cdot k_0$. Each of these coefficients can be computed in constant time, since there are only linearly many coefficients, then, the time to compute the next partial product is just $O(n)$.

Now that we have a coefficient representation of $\prod_j (x - x_j)$, we need to compute, for each k $\prod_{j-k}(x - x_j)$, each of which can be computed in time $\theta(n)$ by problem 30.1-2. Since the polynomial is defined as a product of things containing the thing we are dividing by, we have that the remainder in each case is equal to 0. Lets call these polynomials f_k. Then, we need only compute the sum $\sum_k y_k \frac{f_k(x)}{f_k(x_k)}$. That is, we compute $f(x_k)$ each in time $\Theta(n)$, so all told, only $\Theta(n^2)$ time is spent computing all the $f(x_k)$ values. For each of the terms in the sum, dividing the polynomial $f_k(x)$ by the number $f_k(x_k)$ and multiplying by y_k only takes time $\Theta(n)$, so total it takes time $\Theta(n^2)$. Lastly, we are adding up n polynomials, each of degree bound $n-1$, so the total time taken there is $\Theta(n^2)$.

Exercise 30.1-6

If we wish to compute P/Q but Q takes on the value zero at some of these points, then we can't carry out the "obvious" method. However, as long as all point value pairs (x_i, y_i) we choose for Q are such that $y_i \neq 0$, then the approach comes out exactly as we would like.

Exercise 30.1-7

For the set A, we define the polynomial f_A to have a coefficient representation that has a_i equal zero if $i \notin A$ and equal to 1 if $i \in A$. Similarly define f_B. Then, we claim that looking at $f_C := f_A \cdot f_B$ in coefficient form, we have that the ith coefficient, c_i is exactly equal to the number of times that i is realized as a sum of elements from A and B. Since we can perform the polynomial multiplication in time $O(n \lg(n))$ by the methods of this chapter, we can get the final answer in time $O(n \lg(n))$. To see that f_C has the nice property described, we'll look at the ways that we could end up having a term of x^i appear. Each contribution to that coefficient must come from there being some k so that $a_k \neq 0$ and $b_{i-k} \neq 0$, because the powers of x attached to each are additive when we

multiply. Since each of these contributions is only ever 1, the final coefficient is counting the total number of such contributions, therefore counting the number of $k \in A$ such that $i - k \in B$, which is exactly what we claimed f_C was counting.

Exercise 30.2-1

$$\omega_n^{n/2} = \left(e^{2\pi i/n}\right)^{n/2} = e^{\pi i} = -1 = \omega_2$$

Exercise 30.2-2

The DFT is $(6, -2i - 2, -2, 2i - 2)$.

Exercise 30.2-3

We want to evaluate both of the functions at the fourth roots of unity, that is, $\pm 1, \pm i$. We have an initial call of $RECURSIVE-FFT((-10, 1, -1, 7, 0, 0, 0, 0))$. This causes a call of $RECURSIVE-FFT((-10, -1, 0, 0))$, which evaluates to $(-11, -10-i, -9, -10+i)$. It also causes a call of $RECURSIVE-FFT((1, 7, 0, 0))$ which returns $(8, 1+7i, -6, 1-7i)$. Now, in evaluating the original function call, we have $y_0 = -11 + 8 = -3$, $y_4 = -19$. Then, we change ω to $\omega_8 = \frac{1+i}{\sqrt{2}}$, and have $y_1 = -10 - i + \frac{1+i}{\sqrt{2}}(1 + 7i) = -10 - i - 3\sqrt{2} + 4\sqrt{2}i$ and $y_5 = -10 - i + 3\sqrt{2} - 4\sqrt{2}$. At the next value of k, we get $y_2 = -9 - 6i$ and $y_6 = -9 + 6i$. Lastly, we compute $y_3 = -10 + i + 3\sqrt{2} + 4\sqrt{2}i$ and $y_7 = -10 + i - 3\sqrt{2} - 4\sqrt{2}i$. So, the vector returned is $(-3, -10 - i - 3\sqrt{2} + 4\sqrt{2}i, -9 - 6i, -10 + i + 3\sqrt{2} + 4\sqrt{2}i, -19, -10 - i + 3\sqrt{2} - 4\sqrt{2}i, -9 + 6i, -10 + i - 3\sqrt{2} - 4\sqrt{2}i)$. Similarly, if we wanted to compute the FFT of the other polynomial, we'd get the FFT of B is given by $(5, 3 - 7\sqrt{2} + \sqrt{2}i, 3 - 14i, 3 + 7\sqrt{2} + \sqrt{2}i, 1, 3 + 7\sqrt{2} - \sqrt{2}i, 3 + 14i, 3 - 7\sqrt{2} - \sqrt{2}i)$. Then, we just multiply together these point value representations to get that the product of A and B has the point value representation of

$$
\begin{aligned}
(&- 15, \\
&4 + 62\sqrt{2} + (-65 + 9\sqrt{2})i, \\
&- 111 + 108i, \\
&4 - 62\sqrt{2} + (65 + 9\sqrt{2})i, \\
&- 19, \\
&4 - 62\sqrt{2} + (-65 + -9\sqrt{2})i, \\
&- 111 - 108i, \\
&4 + 62\sqrt{2} + (65 - 9\sqrt{2})i)
\end{aligned}
$$

Interpolating this polynomial using equation (30.11), we get

$$a_0 = \frac{1}{8} \sum_{k=0}^{7} y_k = -30$$

$$a_1 = \frac{1}{8} \sum_{k=0}^{7} y_k \omega_8^{-k} = 63$$

$$a_2 = \frac{1}{8} \sum_{k=0}^{7} y_k \omega_8^{-2k} = -9$$

$$a_3 = \frac{1}{8} \sum_{k=0}^{7} y_k \omega_8^{-3k} = -53$$

$$a_4 = \frac{1}{8} \sum_{k=0}^{7} y_k \omega_8^{-4k} = -34$$

$$a_5 = \frac{1}{8} \sum_{k=0}^{7} y_k \omega_8^{-5k} = -8$$

$$a_6 = \frac{1}{8} \sum_{k=0}^{7} y_k \omega_8^{-6k} = 56$$

$$a_7 = \frac{1}{8} \sum_{k=0}^{7} y_k \omega_8^{-7k} = 0$$

Giving us the polynomial

$$56x^6 - 8x^5 - 34x^4 - 53x^3 - 9x^2 + 63x - 30$$

The same as in problem 30.1-1.

Exercise 30.2-4

Exercise 30.2-5

To show that our algorithm for n being a power of 3 works, we will first prove an analogue of the halving lemma. In particular, for n a power of 3, that the cube of the nth complex roots of unity are the $n/3$ complex $(n/3)$th roots of unity. First, we note that by the cancellation lemma, $(\omega_n^k)^3 = \omega_{n/3}^k$.

$$(\omega_n^{k+n/3})^3 = \omega_n^{3k+n}$$
$$= (\omega_n^k)^3$$

Now, we write $A(x) = \sum_{j=0}^{n-1} a_j x^j$, and define $A^{[i]} = \sum_{j=0}^{n/3-1} a_{i+3j} x^j$ for $i =$

Algorithm 1 RECURSIVE-FFT-INV(y)

1: $n = y.length$
2: **if** $n == 1$ **then**
3:    **return** y/n
4: **end if**
5: $w_n = e^{2\pi i/n}$
6: $w = 1/n$
7: $y^{[0]} = (y_0, y_2, \ldots, y_{n-2})$
8: $y^{[1]} = (y_1, y_3, \ldots, y_{n-1})$
9: $a^{[0]} = \text{RECURSIVE-FFT-INV}(y^{[0]})$
10: $a^{[1]} = \text{RECURSIVE-FFT-INV}(y^{[1]})$
11: **for** $k = 0$ to $n/2 - 1$ **do**
12:    $a_k = a_k^{[0]} + w a_k^{[1]}$
13:    $a_{k+(n/2)} = a_k^{[0]} - w a_k^{[1]}$
14:    $w = w w_n$
15: **end for**
16: **return** a

$1, 2, 3.$ Then, we can see that

$$A(x) = A^{[0]}(x^3) + x A^{[1]}(x^3) + x^2 A^{[2]}(x^3)$$

The recurrence we get is

$$T(n) = 3T(n/3) + \Theta(n)$$
$$= \Theta(n \lg(n))$$

Exercise 30.2-6

It will suffice to show that each power of omega is distinct and that our new definition satisfies all the relevant properties of omega that made DFT and inverse DFT work in the first place. To start, note that $2^t, 2^{2t}, \ldots, 2^{nt/2}$ are all distinct and even since they are each less than m. Further, for $1 \leq k < n/2$ we have that $2^{kt} 2^{nt/2}$ is equivalent to -2^{kt}, which is equivalent to $2^{nt/2} + 1 - 2^{kt}$ which is odd, so we conclude that all powers of ω are distinct, $\omega^n = 2^n t = (-1)^2 = 1$, $\omega^{k+n/2} = -\omega^k$. Finally, $\sum_{i=0}^{n-1} \omega^i = \sum_{i=0}^{n-1} 2^{ti} = (1 - 2^{tn})/(1 - 2^t) = 0$. Thus, our new definition of ω satisfies all the requisite properties to make the DFT work, and since the values are distinct it is well-defined and behaves identically.

Exercise 30.2-7

We just do a bunch of multiplications. More seriously, let $P_{i,0}(x) = (x - z_{i-1})$ for $i = 1, \ldots, n$. Then, we compute the following products, $P_{i,k} = P_{i,k-1} \cdot P_{2i,k-1}$, for any $i \leq n/(2^k)$. If we ever index outside of the already defined $P_{i,k}$ values, we pretend that the value we get is 1. Then, our final answer will

Algorithm 2 POW3FFT(a)

---

n = a.length
if n==1 **then**
    return a
end if
$\omega_n = e^{2\pi i/n}$
$\omega = 1$
$a^{[0]} = (a_0, a_3, a_6 \ldots, a_{n-3})$
$a^{[1]} = (a_1, a_4, a_7 \ldots, a_{n-2})$
$a^{[2]} = (a_2, a_5, a_8 \ldots, a_{n-1})$
$y^{[0]} = POW3FFT(a^{[0]})$
$y^{[1]} = POW3FFT(a^{[1]})$
$y^{[2]} = POW3FFT(a^{[2]})$
for k=0,1…n/2-1 **do**
    $y_k = y_k[0] + \omega y_k^{[1]} + \omega^2 y_k^{[2]}$
    $y_{k+n/3} = y_k[0] + \omega_3 \omega y_k^{[1]} + \omega_3^2 \omega^2 y_k^{[2]}$
    $y_{k+2n/3} = y_k[0] + \omega_3^2 \omega y_k^{[1]} + \omega_3 \omega^2 y_k^{[2]}$
    $\omega = \omega \omega_n$
end for
return y

---

be $P_{1,\lfloor \lg(n) \rfloor + 1}$. We have that obtaining a polynomial in this way that has the recurrence where we n represents the time required to do it for a polynomial of degree bound n that has zeroes at n given points.

$$T(n) = 2T(n/2) + \Theta(n \lg(n))$$

Which, we know by exercise 4.6-2, has a solution of $T \in \Theta(n \lg^2(n))$.

Exercise 30.2-8

Define a polynomial $P(x)$ of degree bound $2n$ by $P(x) = \sum_{j=0}^{2n-1} b_j x^j$ where $b_j = a_j z^{j^2/2}$ if $j \leq n-1$ and 0 for $j \geq n$. Define $Q(x) = \sum_{j=0}^{2n-1} c_j x^j$ where $c_j = z^{-j^2}/2$. We can compute their product in time $O(2n \lg 2n) = O(n \lg n)$. If we let d_k be the coefficient on x^k in their product, for $k \geq n$ we have

$$d_k = \sum_{j=0}^{k} b_j c_{k-j} = \sum_{j=0}^{n-1} \left(a_j z^{j^2/2} \right) \left(z^{-(k-j)^2}/2 \right).$$

By setting $y_k = z^{k^2/2} d_k$ in linear time, we can compute the chirp transform in $O(n \lg n)$.

Exercise 30.3-1

6

By calling BIT-REVERSE-COPY, we get that $A = (0, 4, 3, 7, 2, 5, -1, 9)$. after the first pass of the outermost, loop, when $s = 1$, we have that the array is $A = (4, -4, 10, -4, 7, -3, 8, -10)$. The value at the end of the next iteration of the outermost loop is $(14, -3 - 3i, -8, -3 - 3i, 15, 3 + 4i, -13, 3 - 4i)$. Then, on the last iteration, we get our final answer of

$$A = (19,$$
$$-4 - 4i + \frac{7}{\sqrt{2}} - \frac{13}{\sqrt{2}}i,$$
$$-6 + i,$$
$$-4 + 4i - \frac{7}{\sqrt{2}} - \frac{13}{\sqrt{2}}i,$$
$$-1,$$
$$-4 - 4i - \frac{7}{\sqrt{2}} + \frac{13}{\sqrt{2}}i,$$
$$-6 - i,$$
$$-4 + 4i + \frac{7}{\sqrt{2}} + \frac{13}{\sqrt{2}}i)$$

Exercise 30.3-2

We can consider ITERATIVE-FFT as working in two steps, the copy step (line 1) and the iteration step (lines 4 through 14). To implement in inverse iterative FFT algorithm, we would need to first reverse the iteration step, then reverse the copy step. We can do this by implementing the INVERSE-INTERATIVE-FFT algorithm exactly as we did with the FFT algorithm, but this time creating the iterative version of RECURSIVE-FFT-INV as given in exercise 30.2-4.

Exercise 30.3-3

It computes a twiddle factor for each iteration of the innermost for loop. Since there are n/m iterations of the loop on line 6 and, for each $m/2$ iterations of the innermost loop, there are a total of $n/(2^s) \cdot 2^{s-1} = n/2$ twiddle factors. If we, before line 6 compute all of the powers $< m/2$ of ω_m, we won't have to do any computation of them later on. These are the only twiddle factors that will show up, and so, we only compute $m/2 = s^2/2 = 2^{s-1}$ of them.

Exercise 30.3-4

First observe that if the input to a butterfly operation is all zeros, then so is the output. Our initial input will be $(a_0, a_1, \ldots, a_{n-1})$ where $a_i = 0$ if $0 \le i \le n/2 - 1$, and i otherwise. By examining the diagram on page 919, we

see that the zeros will propogate as half the input to each butterfly operation in the final stage. Thus, if any of the output values are 0 we know that one of the y_k's is zero, where $k \geq n/2 - 1$. If this is the case, then the faulty butterfly added occurs along the wire connected to y_k. If none of the output values are zero, the faulty butterfly adder must occur as one which only deals with the first $n/2 - 1$ wires. In this case, we rerun the test with input such that $a_i = i$ if $0 \leq i \leq n/2 - 1$ and 0 otherwise. This time, we will know for sure which of the first $n/2 - 1$ wires touches the faulty butterfly adder. Since only $\lg n$ adders touch a given wire, we have reduced the problem of size n to a problem of size $\lg n$. Thus, at most $2\lg^\star n = O(\lg^\star n)$ tests are required.

Problem 30-1

a. Similar to problem 4.2-7,

$$(a + b)(c + d) = ac + cb + ad + cd$$

So, we compute that product, we also compute ac and cd. This gets us the the product of the two polynomials is

$$acx^2 + ((a + b)(c + d) - ac - cd)x + cd$$

b. Assume that n is a power of two, if it isn't, then just bump it up to the nearest power of two, since the degree bound can be higher than the degree of the polynomials. Suppose that we want to multiply the polynomials $A_1(x) = \sum_{j=1}^{n-1} a_{j,1}x^j$ and $A_2(x) = \sum_{j=1}^{n-1} a_{j,2}x^j$.

In the first method, we'll set $H_i(x) = \sum_{j=\frac{n}{2}}^{n-1} a_{j,i}x^j$ and $L_i(x) = \sum_{j=0}^{n/2-1} a_{j,i}x^j$ for $i = 1, 2$. Then, we have that $A_i(x) = H_i(x)x^{n/2} + L_i(x)$ for $i = 1, 2$. Then, by the method of the first part of this problem, we have that

$$
\begin{aligned}
A_1(x) \cdot A_2(x) = {}& (H_1(x) \cdot H_2(x))x^n \\
& + ((H_1(x) + L_1(x)) \cdot (H_2(x) + L_2(x)) - H_1(x) \cdot H_2(x) - L_1(x) \cdot L_2(x))x^{n/2} \\
& + L_1(x) \cdot L_2(x)
\end{aligned}
$$

So, the runtime of this procedure for degree bound n is, by the master theorem:

$$
\begin{aligned}
HL(n) &= 3HL(n/2) + \Theta(n) \\
&= \Theta(n^{\lg(3)})
\end{aligned}
$$

8

Now, for the second method, we write $O_i(x) = \sum_{j=0}^{n/2-1} a_{2j+1,i}x^j$ and $E_i(x) = \sum_{j=0}^{n/2-1} a_{2j,i}x^j$ for $i = 1, 2$. Then, we have that for both values of i, $A_i = xO_i(x^2) + E_i(x^2)$. So,

$$
\begin{aligned}
A_1(x) \cdot A_2(x) = {} & x^2(O_1(x^2) \cdot O_2(x^2)) \\
& + x((O_1(x^2) + E_1(x^2))(O_2(x^2) + E_1(x^2)) - O_1(x^2) \cdot O_2(x^2) - E_1(x^2) \cdot E_2(x^2)) \\
& + E_1(x^2) \cdot E_2(x^2)
\end{aligned}
$$

So, again, we only need to do three multiplies, each with a degree bound of half. So, the runtime for this, call it $OE(n)$ is

$$
\begin{aligned}
OE(n) &= 3OE(n/2) + \Theta(n) \\
&= \Theta(n^{\lg(n)})
\end{aligned}
$$

c. Suppose that we want to multiply two integers $A_1 = \sum_{k=0}^{\lfloor \lg(A_1) \rfloor} a_{k,1}2^k$ and $A_2 = \sum_{k=0}^{\lfloor \lg(A_2) \rfloor} a_{k,2}2^k$. Then, we'll associate to these polynomials $f_i = \sum_{k=0}^{\lfloor \lg(A_i) \rfloor} a_{k,i}x^i$. Then, we exactly have that $f_i(2) = A_i$. So, to find $A_1 \cdot A_2$, all we need do is multiply the polynomials f_1 and f_2 and evaluate their product at 2. Since both of their degrees are bounded by n, we can multiply them in time $\Theta(n^{\lg(3)})$ by the previous part. evaluating them also only takes linear time, so doesn't change the total runtime.

Problem 30-2

a. The sum of two Toeplitz matrices is Toeplitz, but the product is not.

b. Let A be a Toeplitz matrix. We can use a vector of length $2n-1$ to represent it, given by: $(c_0, \ldots, c_{2n-2}) = (a_{n,1}, a_{n-1,1}, \ldots, a_{2,1}, a_{1,1}, a_{1,2}, \ldots, a_{a,n})$. To add two Toeplitz matrices, simply add their associated vectors.

c. We can interpret this as the multiplication of two polynomials. Specifically, let $P(x) = c_0 + c_1 x + \ldots + c_{2n-1}x^{2n-2}$. Let $(b_0, b_1, \ldots, b_{n-1})$ be the vector of length n by which we wish to multiply, and let y_k denote the kth entry of the vector which results from the multiplication. Let $Q(x) = b_0 x^{n-1} + b_1 x^{n-2} + \ldots + b_{n-1}$. Then the coefficient on $x^{n-k+n-1}$ in $P(x)Q(x)$ is given by $\sum_{i=0}^{n-1} c_{n-k+i}b_i = \sum_{i=0}^{n-1} a_{k,i}b_i = y_k$. Since we can multiply the polynomials in $O(n \lg n)$ and the needed results are just some of the coefficients, we can multiply a Toeplitz matrix by an n-vector in $O(n \lg n)$.

d. We can view matrix multiplication as simply multiplication by an n vector carried out n times. If b_j is the jth column of the second matrix, then Ab_j is the jth column of the resulting matrix. By part c, this can be done in $O(n^2 \lg n)$ time, which is asymptotically faster than even Strassen's algorithm.

Problem 30-3

a.

$$
\begin{aligned}
y_{k_1,\ldots,k_d} &= \sum_{j_1=0}^{n_1-1} \cdots \sum_{j_d=0}^{n_d-1} a_{j_1,\ldots j_d} \omega_{n_1}^{j_1 k_1} \cdots \omega_{n_d}^{j_d k_d} \\
&= \sum_{j_d=0}^{n_d-1} \cdots \sum_{j_1=0}^{n_1-1} a_{j_1,\ldots j_d} \omega_{n_1}^{j_1 k_1} \cdots \omega_{n_d}^{j_d k_d} \\
&= \sum_{j_d=0}^{n_d-1} \cdots \sum_{j_2=0}^{n_2-1} \left(\sum_{j_1=0}^{n_1-1} a_{j_1,\ldots j_d} \omega_{n_1}^{j_1 k_1} \right) \omega_{n_2}^{j_2 k_2} \cdots \omega_{n_d}^{j_d k_d}
\end{aligned}
$$

So, the thing inside the parentheses is a one dimensional Fourier transform that must be computed for every possible term of the outer sums, that is, must be computed $n_2 n_3 \cdots n_d = n/n_1$ times because in each term the a values might be different. Once that is computed we've decreased the number of sums by 1. This means that by keeping on applying this, we can keep decreasing the number of dimensions until the problem is solved. We are actually only needing to do $n/(\prod_{i \leq k} n_i)$ of the DFT's along dimension k instead of the larger number stated in the problem of n/n_i.

b. We can exchange the order of summation however we please because none of the indices of summation ever appear in the bounds for a different summation sign.

c. The time to do each DFT along the kth dimension is $O(n_k \lg(n_k))$, since we only need to do it at most $n/(\prod_{j<k} n_j)$ times, the runtime of all of them in dimension k is at most $O(n/(\prod_{j<k} n_j) \lg(n_k))$. Also, note that we may assume that all of the n_k values are at least two, because otherwise doing that DFT would be trivial. So, the total time is on the order of

$$
\sum_{k=1}^{d} n/\left(\prod_{j<k} n_j \right) \lg(n_k) \leq \lg(n) \sum_{k=1}^{d} n/\left(\prod_{j<k} n_j \right)
$$

$$
\leq \lg(n) \sum_{k=1}^{d} n/2^{k-1}
$$

$$
< n \lg(n)
$$

Which is independent of d.

Problem 30-4

a. It is straightforward to check that $A^{(t)}(x_0) = t!b_t$. Since $A^{(t+1)}(x_0) = (t + 1)A^{(t)}(x_0)b_{t+1}/b_t$, we can compute each next term in constant time from the previous, so computing $A^{(t)}(x_0)$ for $t = 0, 1, \ldots, n-1$ takes $O(n)$ time.

b. We just need to perform an inverse FFT procedure on the n point-value pairs $A(x_0 + w_n^k) = \sum_{j=0}^{n-1} b_j w_n^{kj}$. This takes $O(n \lg n)$.

c. Let $\chi(x) = 1$ if $x \geq 0$ and 0 otherwise. Using the binomial theorem we have

$$
\begin{aligned}
A(x_0 + w_n^k) &= \sum_{j=0}^{n-1} a_j \sum_{r=0}^{j} \binom{j}{r} w_n^{kr} x_0^{j-r} \\
&= \sum_{j=0}^{n-1} a_j \sum_{r=0}^{n-1} \frac{j!}{r!(j-r)!} w_n^{kr} x_0^{j-r} \chi(j-r) \\
&= \sum_{r=0}^{n-1} \frac{w_n^{kr}}{r!} \sum_{j=0}^{n-1} a_j j! \frac{x_0^{j-r} \chi(j-r)}{(j-r)!} \\
&= \sum_{r=0}^{n-1} \frac{w_n^{kr}}{r!} \sum_{j=0}^{n-1} f(j) g(r-j).
\end{aligned}
$$

d. Let $s(r) = \sum_{j=0}^{n-1} f(j)g(r-j)$ in $O(n)$ time. Letting $b_j = \frac{s(r)}{r!}$. Then we need only compute the DFT of the coefficient vector $(b_0, b_1, \ldots, b_{n-1})$, which can be done in $O(n \lg n)$. By part b, once we have these evaluations we can compute the derivatives in $O(n \lg n)$ time as well.

Problem 30-5

a. First, note that because the degree of $x - z$ is one, $A(x)mod(x - z)$ will be a constant. By the definition of modular arithmetic for polynomials(or any Euclidean Domain), there is some polynomial $f(x)$ so that $A(x) = f(x)(x - z) + (A(x) \mod (x - z))$. So, if we evaluate this expression at z, the first term goes to zero, and we have that $A(z) = A(x) \mod (x - z)$.

b. $P_{kk}(x) = (x - x_k)$ so, by the previous part, $Q_{kk} = A(x_k)$. The degree of $P_{0,n-1}$ is equal to n which is higher than the degree of A. Therefore modding out by it doesn't change the value value of $A(x)$ at all. That is, if we were to write $A(x) = f(x)P_{0,n-1} + Q_{0,n-1}$, the only acceptable value of $f(x)$ is zero, otherwise there would be a too high degree term on the right.

c. Suppose we write $A(x) = f_1(x)P_{ij}(x) + Q_{ij}(x)$. Then, we take $Q_{ij} = f_2(x)P_{ik} + (Q_{ij}(x) \mod P_{ik})$. Since we have that P_{ik} is a product over a smaller set of irreducible factors than P_{ij}, we can write $A(x) = (f_1(x) \prod_{\ell=k+1}^{j}(x - x_\ell) + f_2)P_{ik} + (Q_{ij}(x) \mod P_{ik})$. Since we can write it as a remainder of A

11

after dividing by P_{ik}, we have that $A \mod P_{ik} = Q_{ij}(x) \mod P_{ik}$, which is to say, $Q_{ik} = Q_{ij}(x) \mod P_{ik}$.

We basically do the same thing to show the other equality. Suppose that $Q_{ij} = f_3 P_{kj} + (Q_{ij}(x) \mod P_{kj})$, then $A(x) = (f_1 \prod_{\ell=i}^{\ell=k-1}(x-x_\ell) + f_3)P_{kj} + (Q_{ij}(x) \mod P_{kj})$ and so, $Q_{kj} = A \mod P_{kj} = (Q_{ij}(x) \mod P_{kj})$.

d. Initially, we know what the value of $Q_{0,n-1}$ is, since it is just $A(x)$. Suppose that n is a power of 2, since it makes the analysis easier to do, if it is not a power of two, then bump up the degree bound to the nearest value of 2, since we have that $(2n) + \lg^2(2n) \in O(n \lg^2(n))$, doing this increase of the degree bound will not change the asymptotics of the algorithm. Since we have that the number of points we are evaluating at is equal to the degree bound, just pick arbitrary points to pad the original set of points with and then disregard their values once they are computed. The idea is to cut in half $0, \ldots, n-1$, by computing $Q_{0,n/2-1}$ and $Q_{n/2,n-1}$ using the rule from the previous part until you arrive at having to compute Q_{ii} for some i, which by part b is equal to $A(x_i)$. Since the computing of each of the Q values before the end is only a matter of computing two polynomial remainders, the time to do this is given by the recurrence

$$T(n) = 2T(n/2) + \Theta(n \lg(n))$$

Even though the master theorem doesn't apply to this recurrence, exercise 4.6-2 tells us that $T \in \Theta(n \lg^2(n))$.

Problem 30-6

a. By the prime number theorem, the number of primes between 1 and N is approximately $\frac{1}{\ln(N)}$. This means that between 1 and $n \lg(n)$, we can expect that a random number will be prime with probability $\frac{1}{\lg(n \lg(n))} = \frac{1}{\lg(n)+\lg(\lg(n))} \approx \frac{1}{\lg(n)}$. Also, we are considering $\lg(n)$ numbers between 1 and $n \lg(n) + 1$ that are one more than a multiple of n, so, the expected number of prime numbers of that form with $k \leq \lg(n)$ is one, so, by the poisson paradigm, we would think that with probability at least about $\frac{1}{e}$, there is a prime number of that form with $k \leq \lg(n)$.

Sine the value of p is expected to be about $n \lg(n)$, it has a length of about $\lg(n \lg(n)) = \lg(n) + \lg(\lg(n))$.

b.

c.

d.

12

Chapter 31

Michelle Bodnar, Andrew Lohr

May 9, 2017

Exercise 31.1-1

By the given equation, we can write $c = 1 \cdot a + b$, with $0 \geq b < a$. By the definition of remainders given just below the division theorem, this means that b is the remainder when c is divided by a, that is $b = c \mod a$.

Exercise 31.1-2

Suppose that there are only finitely many primes $p_1, p_2, \ldots, p_k$. Then $p = p_1 p_2 \cdots p_k + 1$ isn't prime, so there must be some p_i which divides it. However, $p_i \cdot (p_1 \cdots p_{i-1} p_{i+1} \cdots p_k) < p$ and $p \cdot (p_1 \cdots p_{i-1} p_{i+1} \cdots p_k + 1) > p$, so p_i can't divide p. Since this holds for any choice of i, we obtain a contradiction. Thus, there are infinitely many primes.

Exercise 31.1-3

$a|b$ means there exists $k_1 \in \mathbb{Z}$ so that $k_1 a = b$. $b|c$ means there exists $k_2 \in \mathbb{Z}$ so that $k_2 b = c$. This means that $(k_1 k_2)a = c$. Since the integers are a ring, $k_1 k_2 \in \mathbb{Z}$, so, we have that $a|c$.

Exercise 31.1-4

Let $g = \gcd(k, p)$. Then $g|k$ and $g|p$. Since p is prime, $g = p$ or $g = 1$. Since $0 < k < p$, $g < p$. Thus, $g = 1$.

Exercise 31.1-5

By Theorem 31.2, since $gcd(a, n) = 1$, there exist integers p, q so that $pa + qn = 1$, so, $bpa + bqn = b$. Since $n|ab$, there exists an integer k so that $kn = ab$. This means that $knp + pqn = (k + q)pn = b$. Since n divides the left hand side, it must divide the right hand side as well.

Exercise 31.1-6

Observe that $\binom{p}{k} = \frac{p!}{k!(p-k)!} = \frac{p(p-1)\cdots(p-k+1)}{k!}$. Let $q = (p-1)(p-2)\cdots(p-k+1)$. Since p is prime, $k! \nmid p$. However, we know that $\binom{p}{k}$ is an integer because it is also a counting formula. Thus, $k!$ divides pq. By Corollary 31.5, $k! | q$. Write $q = ck!$. Then we have $\binom{p}{k} = pc$, which is divisible by p.

By the binomial theorem and the fact that $p | \binom{p}{k}$ for $0 < k < p$,

$$(a+b)^p = \sum_{k=0}^{p} \binom{p}{k} a^k b^{p-k} \equiv a^p + b^p (\mathrm{mod}\ p).$$

Exercise 31.1-7

First, suppose that $x = yb + (x \mod b)$, $(x \mod b) = za + ((x \mod b) \mod a)$, and $ka = b$. Then, we have $x = yka + (x \mod b) = (yk + z)a + ((x \mod b) \mod a)$. So, we have that $x \mod a = ((x \mod b) \mod a)$.

For the second part of the problem, suppose that $x \mod b = y \mod b$. Then, by the first half of the problem, applied first to x and then to b, $x \mod a = (x \mod b) \mod a = (y \mod b) \mod a = y \mod a$. So, $x \equiv y \mod a$.

Exercise 31.1-8

We can test in time polynomial in β whether or not a given β-bit number is a perfect kth power. Then, since two to the βth power is longer than the number we are given, we only need to test values of k between 2 and β, thereby keeping the runtime polynomial in β.

To check whether a given number n is a perfect kth power, we will be using a binary search like technique. That is, if we want to find the k-th root of a number, we initially know that it lies somewhere between 0 and the number itself. We then look at the number of the current range of number under consideration, raise it to the kth power in time polynomial in β. We can do this by the method of repeated squaring discussed in section 31.6. Then, if we get a number larger than the given number when we perform repeated squaring, we know that the true kth root is in the lower half of the range in consideration, if it is equal, it is the midpoint, if larger, it is the upper half. Since each time, we are cutting down the range by a factor of two, and it is initially a range of length $\Theta(2^\beta)$, the number of times that we need to raise a number the the kth power is $\Theta(\beta)$. Putting it all together, with the $O(\beta^3)$ time exponentiation, we get that the total runtime of this procedure is $O(\beta^5)$.

Exercise 31.1-9

For (31.6), we see that a and b in theorem 31.2 which provides a characterization of gcd appear symmetrically, so swapping the two won't change anything.

For (31.7), theorem 31.2 tells us that gcd's are defined in terms of integer linear combinations. If we had some integer linear combination involving a and

b, we can changed that into one involving (-a) and b by replacing the multiplier of a with its negation.

For (31.8), by repeatedly applying (31.6) and (31.7), we can get this equality for all four possible cases based on the signs of both a and b.

For(31.9), consider all integer linear combinations of a and 0, the thing we multiply by will not affect the final linear combination, so, really we are just taking the set of all integer multiples of a and finding the smallest element. We can never decrease the absolute value of a by multiplying by an integer ($|ka| = |k||a|$), so, the smallest element is just what is obtained by multiplying by 1, which is $|a|$.

For (31.10), again consider possible integer linear combinations $na + mka$, we can rewrite this as $(n + km)a$, so it has absolute value $|n + km||a|$. Since the first factor is an integer, we can't have it with a value less than 1 and still have a positive final answer, this means that the smallest element is when the first factor is 1, which is achievable by setting $n = 1, m = 0$.

Exercise 31.1-10

Consider the prime factorization of each of a, b, and c, written as $a = p_1 p_2 \ldots p_k$ where we allow repeats of primes. The gcd of b and c is just the product of all p_i such that p_i appears in both factorizations. If it appears multiple times, we include it as many times as it appears on the side with the fewest occurrences of p_i. (Explicitly, see equation 31.13 on page 934). To get the gcd of $\gcd(b, c)$ and a, we do this again. Thus, the left hand side is just equal to the intersection (with appropriate multiplicities) of the products of prime factors of a, b, and c. For the right hand side, we consider intersecting first the prime factors of a and b, and then the prime factors of c, but since intersections are associative, so is the gcd operator.

Exercise 31.1-11

Suppose to a contradiction that we had two different prime decomposition. First, we know that the set of primes they both consist of are equal, because if there were any prime p in the symmetric difference, p would divide one of them but not the other. Suppose they are given by $(e_1, e_2, \ldots, e_r)$ and $(f_1, f_2, \ldots, f_r)$ and suppose that $e_i < f_i$ for some position. Then, we either have that $p_i^{e_i+1}$ divides a or not. If it does, then the decomposition corresponding to $\{e_i\}$ is wrong because it doesn't have enough factors of p_i, otherwise, the one corresponding to $\{f_i\}$ is wrong because it has too many.

Exercise 31.1-12

Standard long division runs in $O(\beta^2)$, and one can easily read off the remainder term at the end.

Exercise 31.1-13

First, we bump up the length of the original number until it is a power of two, this will not affect the asymptotics, and we just imagine padding it with zeroes on the most significant side, so it does not change its value as a number. We split the input binary integer, and split it into two segments, a less significant half ℓ and an more significant half m, so that the input is equal to $m2^{\beta/2} + \ell$. Then, we recursively convert m and ℓ to decimal. Also, since we'll need it later, we compute the decimal versions of all the values of 2^{2^i} up to 2^{β}. There are only $\lg(\beta)$ of these numbers, so, the straightforward approach only takes time $O(\lg^2(\beta))$ so will be overshadowed by the rest of the algorithm. Once we've done that, we evaluate $m2^{\beta/2} + \ell$, which involves computing the product of two numbers and adding two numbers, so, we have the recurrence

$$T(\beta) = 2T(\beta/2) + M(\beta/2)$$

Since we have trouble separating M from linear by a n^ϵ for some epsilon, the analysis gets easier if we just forget about the fact that the difficulty of the multiplication is going down in the subcases, this concession gets us the runtime that $T(\beta) \in O(M(\beta)\lg(\beta))$ by master theorem.

Note that there is also a procedure to convert from binary to decimal that only takes time $\Theta(\beta)$, instead of the given algorithm which is $\Theta(M(\beta)\lg(\beta)) \in \Omega(\beta\lg(\beta))$ that is rooted in automata theory. We can construct a deterministic finite transducer between the two languages, then, since we only need to take as many steps as there are bits in the input, the runtime will be linear. We would have states to keep track of the carryover from each digit to the next.

Exercise 31.2-1

First, we show that the expression given in equation (31.13) is a common divisor. To see that we just notice that

$$a = (\prod_{i=1}^{r} p_i^{e_i - \min(e_i, f_i)}) \prod_{i=1}^{r} p_i^{\min(e_i, f_i)}$$

and

$$b = (\prod_{i=1}^{r} p_i^{f_i - \min(e_i, f_i)}) \prod_{i=1}^{r} p_i^{\min(e_i, f_i)}$$

Since none of the exponents showing up are negative, everything in sight is an integer.

Now, we show that there is no larger common divisor. We will do this by showing that for each prime, the power can be no higher. Suppose we had some common divisor d of a and b. First note that d cannot have a prime factor that doesn't appear in both a or b, otherwise any integer times d would also have that factor, but being a common divisor means that we can write both a and b as an integer times d. So, there is some sequence $\{g_i\}$ so that $d = \prod_{i=1}^{r} p_i^{g_i}$.

4

Now, we claim that for every i, $g_i \leq \min(e_i, f_i)$. Suppose to a contradiction that there was some i so that $g_i > \min(e_i, f_i)$. This means that d either has more factors of p_i than a or than b. However, multiplying integers can't cause the number of factors of each prime to decrease, so this is a contradiction, since we are claiming that d is a common divisor. Since the power of each prime in d is less than or equal to the power of each prime in c, we must have that $d \leq c$. So, c is a GCD.

Exercise 31.2-2

We'll create a table similar to that of figure 31.1:

a	b	$\lfloor a/b \rfloor$	d	x	y
899	493	1	29	-6	11
493	406	1	29	5	-6
406	87	4	29	-1	5
87	58	1	29	1	-1
58	29	2	29	0	1
29	0	-	29	1	0

Thus, $(d, x, y) = (29, -6, 11)$.

Exercise 31.2-3

Let c be such that $a = cn + (a \mod n)$. If $k = 0$, it is trivial, so suppose $k < 0$. Then, EUCLID(a+kn,n) goes to line 3, so returns $EUCLID(n, a \mod n)$. Similarly, $EUCLID(a, n) = EUCLID((a \mod n) + cn, n) = EUCLID(n, a \mod n)$. So, by correctness of the Euclidean algorithm,

$$
\begin{aligned}
gcd(a + kn, n) &= EUCLID(a + kn, n) \\
&= EUCLID(n, a \mod n) \\
&= EUCLID(a, n) \\
&= gcd(a, n)
\end{aligned}
$$

Exercise 31.2-4

Algorithm 1 ITERATIVE-EUCLID(a,b)

1: **while** $b > 0$ **do**
2:    $(a, b) = (b, a \mod b)$
3: **end while**
4: **return** a

Exercise 31.2-5

We know that for all k, if $b < F_{k+1} < \phi^{k+1}/\sqrt{5}$, then it takes fewer than k steps. If we let $k = \log_\phi b + 1$, then, since $b < \phi^{\log_\phi b + 2}/\sqrt{5} = \frac{\phi^2}{\sqrt{5}} \cdot b$, we have that it only takes $1 + \log_\phi(b)$ steps.

We can improve this bound to $1+\log_\phi(b/\gcd(a,b))$. This is because we know that the algorithm will terminate when it reaches $\gcd(a,b)$. We will emulate the proof of lemma 31.10 to show a slightly different claim that Euclid's algorithm takes k recursive calls, then $a \geq \gcd(a,b)F_{k+2}$ and $b \geq \gcd(a,b)F_{k+!}$. We will similarly do induction on k. If it takes one recursive call, and we have $a > b$, we have $a \geq 2\gcd(a,b)$ and $b = \gcd(a,b)$.

Now, suppose it holds for $k-1$, we want to show it holds for k. The first call that is made is of EUCLID$(b, a \mod b)$. Since this then only needs $k-1$ recursive calls, we can apply the inductive hypothesis to get that $b \geq \gcd(a,b)F_{k+1}$ and $a \mod b \geq \gcd(a,b)F_k$. Since we had that $a > b$, we have that $a \geq b + (a \mod b) \geq \gcd(a,b)(F_{k+1}+F_k) = \gcd(a,b)F_{k+2}$ completing the induction.

Since we have that we only need k steps so long as $b < \gcd(a,b)F_{k+1} < gcd(a,b)\phi^{k+1}$. we have that $\log_\phi(b/\gcd(a,b)) < k+1$. This is satisfied if we set $k = 1 + \log_\phi(b/\gcd(a,b))$

Exercise 31.2-6

Since $F_{k+1} \mod F_k = F_{k-1}$ we have $\gcd(F_{k+1},F_k) = \gcd(F_k, F_{k-1})$. Since $\gcd(2,1) = 1$ we have that $\gcd(F_{k+1},F_k) = 1$ for all k. Moreover, since F_k is increasing, we have $\lfloor F_{k+1}/F_k \rfloor = 1$ for all k. When we reach the base case of the recursive calls to EXTENDED-EUCLID, we return $(1,1,0)$. The following returns are given by $(d,x,y) = (d',y',x'-y')$. We will show that in general, EXTENDED-EUCLID(F_{k+1},F_k) returns $(1, (-1)^{k+1}F_{k-2}, (-1)^k F_{k-1})$. We have already shown d is correct, and handled the base case. Now suppose the claim holds for $k-1$. Then EXTENDED-EUCLID(F_{k+1},F_k) returns $(d',y',x'-y')$ where $d' = 1$, $y' = (-1)^{k-1}F_{k-2}$, $x' = (-1)^k F_{k-3}$, so the algorithm returns

$$(1, (-1)^{k-1}F_{k-2}, (-1)^k F_{k-3} - (-1)^{k-1}F_{k-2}) = (1, (-1)^{k+1}F_{k-2}, (-1)^k(F_{k-3}+F_{k-2})$$
$$= (1, (-1)^{k+1}F_{k-2}, (-1)^k F_{k-1})$$

as desired.

Exercise 31.2-7

To show it is independent of the order of its arguments, we prove the following swap property, for all a,b,c, $\gcd(a,\gcd(b,c)) = \gcd(b,\gcd(a,c))$. By applying these swaps in some order, we can obtain an arbitrary ordering on the variables (the permutation group is generated by the set of adjacent transpositions). Let a_i be the power of the ith prime in the prime factor decomposition of a, similarly define b_i and c_i. Then, we have that

$$\gcd(a, \gcd(b,c)) = \prod_i p_i^{\min(a_i, \min(b_i, c_i))}$$

$$= \prod_i p_i^{\min(a_i, b_i, c_i)}$$

$$= \prod_i p_i^{\min(b_i, \min(a_i, c_i))}$$

$$= \gcd(b, \gcd(a,c))$$

To find the integers $\{x_i\}$ as described in the problem, we use a similar approach as for EXTENDED-EUCLID.

Exercise 31.2-8

From the gcd interpretation given in equation (31.13), it is clear that $\operatorname{lcm}(a_1, a_2) = \frac{a_1 \cdot a_2}{\gcd(a_1, a_2)}$. More generally, $\operatorname{lcm}(a_1, \ldots, a_n) = \frac{a_1 \cdots a_n}{\gcd(\cdots(\gcd(\gcd(a_1, a_2), a_3), \ldots), a_n)}$. We can compute the denominator recursively using the two-argument gcd operation. The algorithm is given below, and runs in $O(n)$ time.

Algorithm 2 LCM$(a_1, \ldots, a_n)$

$x = a_1$
$g = a_1$
for $i = 2$ to n **do**
$\quad x = x \cdot a_i$)
$\quad g = \gcd(g, a_i)$
end for
return x/g

Exercise 31.2-9

For two numbers to be relatively prime, we need that the set of primes that occur in both of them are disjoint. Multiplying two numbers results in a number whose set of primes is the union of the two numbers multiplied. So, if we let $p(n)$ denote the set of primes that divide n. By testing that $\gcd(n_1 n_2, n_3 n_4) = \gcd(n_1 n_3, n_2 n_4) = 1$. We get that $(p(n_1) \cup p(n_2)) \cap (p(n_3) \cup (n_4)) = (p(n_1) \cup p(n_3)) \cap (p(n_2) \cup (n_4)) = \emptyset$. Looking at the first equation, it gets us that $p(n_1) \cap p(n_3) = p(n_1) \cap p(n_4) = p(n_2) \cap p(n_3) = p(n_2) \cap p(n_4) = \emptyset$. The second tells, among other things, that $p(n_1) \cap p(n_2) = p(n_3) \cap p(n_4) = \emptyset$. This tells us that the sets of primes of any two elements are disjoint, so all elements are relatively prime.

A way to view this that generalizes more nicely is to consider the complete graph on n vertices. Then, we select a partition of the vertices into two parts. Then, each of these parts corresponds to the product of all the numbers corresponding to the vertices it contains. We then know that the numbers that

any pair of vertices that are in different parts of the partition correspond to will be relatively prime, because we can distribute the intersection across the union of all the prime sets in each partition. Since partitioning into two parts is equivalent to selecting a cut, the problem reduces to selecting $\lg(k)$ cuts of K_n so that every edge is cut by one of the cuts. To do this, first cut the vertex set in as close to half as possible. Then, for each part, we recursively try to cut in in close to half, since the parts are disjoint, we can arbitrarily combine cuts on each of them into a single cut of the original graph. Since the number of time you need to divide n by two to get 1 is $\lfloor \lg(n) \rfloor$, we have that that is the number of times we need to take gcd.

Exercise 31.3-1

$$
\begin{array}{c|cccc}
+_4 & 0 & 1 & 2 & 3 \\
\hline
0 & 0 & 1 & 2 & 3 \\
1 & 1 & 2 & 3 & 0 \\
2 & 2 & 3 & 0 & 1 \\
3 & 3 & 0 & 1 & 2 \\
\end{array}
$$

$$
\begin{array}{c|cccc}
\cdot_5 & 1 & 2 & 3 & 4 \\
\hline
1 & 1 & 2 & 3 & 4 \\
2 & 2 & 4 & 1 & 3 \\
3 & 3 & 1 & 4 & 2 \\
4 & 4 & 3 & 2 & 1 \\
\end{array}
$$

Then, we can see that these are equivalent under the mapping $\alpha(0) = 1$, $\alpha(1) = 3$, $\alpha(2) = 4$, $\alpha(3) = 2$.

Exercise 31.3-2

The subgroups of $\mathbb{Z}_9$ and $\{0\}$, $\{0, 3, 6\}$, and $\mathbb{Z}_9$ itself. The subgroups of $\mathbb{Z}_{13}^*$ are $\{1\}$, $\{1, 3, 9\}$, $\{1, 3, 4, 9, 10, 12\}$, $\{1, 5, 8, 12\}$, $\{1, 12\}$ and $\mathbb{Z}_{13}^*$ itself.

Exercise 31.3-3

Since S was a finite group, every element had a finite order, so, if $a \in S'$, there is some number of times that you can add it to itself so that you get the identity, since adding any two things in S' gets us something in S', we have that S' has the identity element. Associativity is for free because is is a property of the binary operation, no the space that the operation draws it's arguments from. Lastly, we can see that it contains the inverse of every element, because we can just add the element to itself a number of times equal to one less than its order. Then, adding the element to that gets us the identity.

Exercise 31.3-4

The only prime divisor of p^e is p. From the definition of the Euler phi function, we have

$$\phi(p^e) = p^e \left(1 - \frac{1}{p}\right) = p^{e-1}(p - 1).$$

Exercise 31.3-5

To see this fact, we need to show that the given function is a bijection. Since the two sets have equal size, we only need to show that the function is onto. To see that it is onto, suppose we want an element that maps to x. Since $\mathbb{Z}_n^*$ is a finite Abelian group by theorem 31.13, we can take inverses, in particular, there exists an element a^{-1} so that $aa^{-1} = 1 \mod n$. This means that $f_a(a^{-1}x) = aa^{-1}x \mod n = (aa^{-1} \mod n)(x \mod n) = x \mod n$. Since we can find an element that maps to any element of the range and the sizes of domain and range are the same, the function is a bijection. Any bijection from a set to itself is a permutation by definition.

Exercise 31.4-1

First, we run extended Euclid on $35, 50$ and get the result $(5, -7, 10)$. Then, our initial solution is $-7 * 10/5 = -14 = 36$. Since $d = 5$, we have four other solutions, corresponding to adding multiples of $50/5 = 10$. So, we also have that our entire solution set is $x = \{6, 16, 26, 36, 46\}$.

Exercise 31.4-2

If $ax \equiv ay \mod n$ then $a(x - y) \equiv 0 \mod n$. Since $\gcd(a, n) = 1$, n doesn't divide a unless $n = 1$, in which case the claim is trivial. By Corollary 31.5, since n divides $a(x - y)$, n divides $x - y$. Thus, $x \equiv y \mod n$. To see that the condition $\gcd(a, n)$ is necessary, let $a = 3$, $n = 6$, $x = 6$, and $y = 2$. Note that $\gcd(a, n) = \gcd(3, 6) = 3$. Then $3 \cdot 6 \equiv 3 \cdot 2 \mod 6$, but $6 \not\equiv 2 \mod 6$.

Exercise 31.4-3

it will work. It just changes the initial value, and so changes the order in which solutions are output by the program. Since the program outputs all values of x that are congruent to $x_0 \mod n/b$, if we shift the answer by a multiple of n/b by this modification, we will not be changing the set of solutions that the procedure outputs.

Exercise 31.4-4

The claim is clear if $a \equiv 0$ since we can just factor out an x. For $a \neq 0$, let $g(x) = g_0 + g_1 x + \ldots + g_{t-1} x^{t-1}$. In order for $f(x)$ to equal $(x - a)g(x)$ we must have $g_0 = f_0(-a)^{-1}$, $g_i = (f_i - g_{i-1})(-a)^{-1}$ for $1 \leq i \leq t - 1$ and

9

$g_{t-1} = f_t$. Since p is prime, $\mathbb{Z}_p^* = \{1, 2, \ldots, p-1\}$ so every element, including $-a$, has a multiplicative inverse. It is easy to satisfy each of these equations as we go, until we reach the last two, at which point we need to satisfy both $g_{t-1} = (f_{t-1} - g_{t-2})(-a)^{-1}$ and $g_{t-1} = f_t$. More compactly, we need $f_t = (f_{t-1} - g_{t-2})(-a)^{-1}$. We will show that this happens when a is a zero of f.

First, we'll proceed inductively to show that for $0 \leq k \leq t-1$ we have $a^{k+1} g_k = -\sum_{i=0}^k f_i a^i$. For the base case we have $a g_0 = -f_0$, which holds. Assume the claim holds for $k-1$. Then we have

$$
\begin{aligned}
a^{k+1} g_k &= a^{k+1}(f_k - g_{k-1})(-a)^{-1} \\
&= -a^k f_k + a^k g_{k-1} \\
&= -a^k f_k - \sum_{i=0}^{k-1} f_i a^i \\
&= \sum_{i=0}^k f_i a^i
\end{aligned}
$$

which completes the induction step. Now we show that we can satisfy the equation given above. It will suffice to show that $-a^t f_t = a^{t-1}(f_{t-1} - g_{t-2})$.

$$
\begin{aligned}
a^{t-1}(f_{t-1} - g_{t-2}) &= a^{t-1} f_{t-1} - a^{t-1} g_{t-2} \\
&= a^{t-1} f_{t-1} + \sum_{i=0}^{t-2} f_i a^i \\
&= \sum_{i=0}^{t-1} f_i a^i \\
&= -a^t f_t
\end{aligned}
$$

where the second equality is justified by our earlier claim and the last equality is justified because a is a zero of f. Thus, we can find such a g.

It is clear that a polynomial of degree 1 can have at most 1 distinct zero modulo p since the equation $x = -a$ has at most 1 solution by Corollary 31.25. Now suppose the claim holds for $t > 1$. Let f be a degree $t+1$ polynomial. If f has no zeros then we're done. Otherwise, let a be a zero of f. Write $f(x) = (x-a)g(x)$. Then by the induction hypothesis, since g is of degree t, g has at most t distinct zeros modulo p. The zeros of f are a and the zeros of g, so f has at most $t+1$ distinct zeros modulo p.

Exercise 31.5-1

These equations can be viewed as a single equation in the ring $\mathbb{Z}_5^+ \times Z_11^+$, in particular $(x_1, x_2) = (4, 5)$. This means that x needs to be the element in $\mathbb{Z}_55^+$ that corresponds to the element $(4, 5)$. To do this, we use the process described in the proof of Theorem 31.27. We have $m_1 = 11$, $m_2 = 5$, $c_1 = 11(11^{-1} \mod 5) = 11$, $c_2 = 5(5^{-1} \mod 11) = 45$. This means that the corresponding solution is $x = 11 \cdot 4 + 45 \cdot 5 \mod 55 = 44 + 225 \mod 55 = 269 \mod 55 = 49 \mod 55$. So, all numbers of the form $49 + 55k$ are a solution.

Exercise 31.5-2

Since $9 \cdot 8 \cdot 7 = 504$, we'll be working mod 504. We also have $m_1 = 56$, $m_2 = 63$, and $m_3 = 72$. We compute $c_1 = 56(5) = 280$, $c_2 = 63(7) = 441$, and $c_3 = 72(4) = 288$. Thus, $a = 280 + 2(441) + 3(288) \mod 504 = 10 \mod 504$. Thus, the desired integers x are of the form $x = 10 + 504k$ for $k \in \mathbb{Z}$.

Exercise 31.5-3

Suppose that $x \equiv a^{-1} \mod n$. Also, $x_i \equiv x \mod n_i$ and $a_i \equiv a \mod n_i$. What we then want to show is that $x_i \equiv a_i^{-1} \mod n_i$. That is, we want that $a_i x_i \equiv 1 \mod n_i$. To see this, we just use equation 31.30. To get that $ax \mod n$ corresponds to $(a_1 x_1 \mod n_1, \ldots, a_k x_k \mod n_k)$. This means that 1 corresponds to $(1 \mod n_1, \ldots, 1 \mod n_k)$. This is telling us exactly what we needed, in particular, that $a_i x_i \equiv 1 \mod n_i$.

Exercise 31.5-4

Let $f(x) = f_0 + f_1 x + \ldots + f_d x^d$. Using the correspondence of Theorem 31.27, $f(x) \equiv 0 \mod n$ if and only if $\sum_{i=0}^{d} f_{i_j} x_j^i \equiv 0 \mod n_j$ for $j = 1$ to k. The product formula arises by constructing a zero of f via choosing $x_1, x_2, \ldots, x_k$ such that $f(x_j) \equiv 0 \mod n_j$ for each j, then letting x be the number associated to $(x_1, \ldots, x_k)$.

Exercise 31.6-1

element	order
1	1
2	10
3	5
4	5
5	5
6	10
7	10
8	10
9	5
10	2

The smallest primitive root is 2, and has the following values for $ind_{11,2}(x)$

x	$ind_{11,2}(x)$
1	10
2	1
3	8
4	2
5	4
6	9
7	7
8	3
9	6
10	5

Exercise 31.6-2

To perform modular exponentiation by examining bits from right to left, we'll keep track of the value of a^{2^i} as we go. See below for an implementation:

Algorithm 3 MODULAR-EXPONENTIATION-R-to-L(a, b, n)

$c = 0$
$d = 1$
$s = a$
let $\langle b_k, b_{k-1}, \ldots, b_0 \rangle$ be the binary representation of b
for $i = 0$ to k **do**
    if $b_i == 1$ **then**
        $d = s \cdot d \mod n$
        $c = 2^i + c$
    end if
    $s = s \cdot s \mod n$
end for
return d

Exercise 31.6-3

Since we know $\phi(n)$, we know that $a^{\phi(n)} \equiv 1 \mod n$ by Euler's theorem. This tells us that $a^{-1} = a^{\phi(n)-1}$ because $aa^{-1} = \equiv aa^{\phi(n)-1} \equiv a^{\phi(n)} \equiv 1 \mod n$. Since when we multiply this expression times a, we get the identity, it is the inverse of a. We can then compute $n^{\phi(n)}$ efficiently, since $\phi(n) < n$, so can be represented without using more bits than was used to represent n.

Exercise 31.7-1

For the secret key's value of e we compute the inverse of $d = 3 \mod \phi(n) = 280$. To do this, we first compute $\phi(280) = \phi(2^3)\phi(7)\phi(5) = 4 \cdot 6 \cdot 4 = 96$. Since

12

any number raised to this will be one mod 280, we will raise it to one less than this. So, we compute

$$
\begin{aligned}
3^{95} &\equiv 3(3^2)^{47} \\
&\equiv 3(9(9^2)^{23}) \\
&\equiv 3(9(81(81^2)^{11})) \\
&\equiv 3(9(81(121(121^2)^5))) \\
&\equiv 3(9(81(121(81(81^2)^2) \\
&\equiv 3 \cdot 9 \cdot 81 \cdot 121 \cdot 81 \cdot 81 \\
&\equiv 3 \cdot 9 \cdot 121 \\
&\equiv 187 \mod 280
\end{aligned}
$$

Now that we know our value of e, we compute the encryption of $M = 100$ by computing $100^{187} \mod 319$, which comes out to an encrypted message of 122

Exercise 31.7-2

We know $ed = 1 \mod \phi(n)$. Since $d < \phi(n)$ and $e = 3$, we have $3d - 1 = k(p-1)(q-1)$ for $k = 1$ or 2. We have $k = 1$ if $3d - 1 < n$ and $k = 2$ if $3d - 1 > n$. Once we've determined k, $p + q = n - (3d-1)/k + 1$, so we can now solve for $p + q$ in time polynomial in β. Replacing $q - 1$ by $(p+q) - p - 1$ in our earlier equation lets us solve for p in time polynomial in β since we need only perform addition, multiplication, and division on numbers bounded by n.

Exercise 31.7-3

$$
\begin{aligned}
P_A(M_1)P_A(M_2) &\equiv M_1^e M_2^e \\
&\equiv (M_1 M_2)^e \\
&\equiv P_A(M_1 M_2) \mod n
\end{aligned}
$$

So, if the attacker can correctly decode $\frac{1}{100}$ of the encrypted messages, he does the following. If the message is one that he can decrypt, he is happy, decrypts it and stops. If it is not one that he can decrypt, then, he picks a random element in $\mathbb{Z}_m$, say x encrypts it with the public key, and multiplies that by the encrypted text, he then has a $\frac{1}{100}$ chance to be able to decrypt the new message. He keeps doing this until he can decrypt it. The number of steps needed follows a geometric distribution with a expected value of 100. Once he's stumbled upon one that he could decrypt, he multiplies by the inverses of all the elements that he multiplied by along the way. This recovers the final answer, and also can be done efficiently, since for every x, x^{n-2} is x^{-1} by Lagrange's theorem.

Exercise 31.8-1
Suppose that we can write $n = \prod_{i=1}^{k} p_i^{e_i}$, then, by the Chinese remainder theorem , we have that $\mathbb{Z}_n \cong \mathbb{Z}_{p_1^{e_1}} \times \cdots \times \mathbb{Z}_{p_1^{e_1}}$. Since we had that n was not a prime power, we know that $k \geq 2$. This means that we can take the elements $x = (p_1^{e_1} - 1, 1, \ldots, 1)$ and $y = (1, p_2^{e_2} - 1, 1, \ldots, 1)$. Since multiplication in the product ring is just coordinate wise, we have that the squares of both of these elements is the all ones element in the product ring, which corresponds to 1 in $\mathbb{Z}_n$. Also, since the correspondence from the Chinese remainder theorem was a bijection, since x and y are distinct in the product ring, they correspond to distinct elements in $\mathbb{Z}_n$. Thus, by taking the elements corresponding to x and y under the Chinese remainder theorem bijection, we have that we have found two squareroots of 1 that are not the identity in $\mathbb{Z}_n$. Since there is only one trivial non-identity squareroot in $\mathbb{Z}_n$, one of the two must be non-trivial. It turns out that both are non-trivial, but that's more than the problem is asking.

Exercise 31.8-2

Let $c = \gcd(\cdots (\gcd(\gcd(\phi(p_1^{e_1}), \phi(p_2^{e_2})), \phi(p_3^{e_3})), \ldots), \phi(p_r^{e_r}))$. Then we have $\lambda(n) = \phi(e_1^{e_1}) \cdots \phi(p_r^{e_r})/c = \phi(n)/c$. Since the lcm is an integer, $\lambda(n) | \phi(n)$.
Suppose p is prime and $p^2 | n$. Since $\phi(p^2) = p^2(1 - \frac{1}{p}) = p^2 - p = p(p-1)$, we have that p must divide $\lambda(n)$. however, since p divides n, it cannot divide $n - 1$, so we cannot have $\lambda(n) | n - 1$.
Now, suppose that is the product of fewer than 3 primes, that is $n = pq$ for some two distinct primes $p < q$. Since both p and q were primes, $\lambda(n) = lcm(\phi(p), \phi(q)) = lcm(p-1, q-1)$. So, mod $q - 1$, $\lambda(n) \equiv 0$, however, since $n - 1 = pq - 1 = (q-1)(p) + p - 1$, we have that $n - 1 \equiv p - 1 \mod q - 1$. Since $\lambda(n)$ has a factor of $q - 1$ that $n - 1$ does not, meaning that $\lambda(n) \nmid n - 1$.

Exercise 31.8-3

First, we prove the following lemma. For any integers a, b, n, $\gcd(a, n) \cdot \gcd(b, n) \geq \gcd(ab, n)$. Let $\{p_i\}$ be an enumeration of the primes, then, by Theorem 31.8, there is exactly one set of powers of these primes so that $a = \prod_i p_i^{a_i}$, $b = \prod_i p_i^{b_i}$, and $n = \prod_i p_i^{n_i}$.

$$\gcd(a, n) = \prod_i p_i^{\min(a_i, n_i)}$$
$$\gcd(b, n) = \prod_i p_i^{\min(b_i, n_i)}$$
$$\gcd(ab, n) = \prod_i p_i^{\min(a_i + b_i, n_i)}$$

We combine the first two equations to get:

$$
\begin{aligned}
\gcd(a,n) \cdot \gcd(b,n) &= \left(\prod_i p_i^{\min(a_i,n_i)} \right) \cdot \left(\prod_i p_i^{\min(b_i,n_i)} \right) \\
&= \prod_i p_i^{\min(a_i,n_i)+\min(b_i,n_i)} \\
&\geq \prod_i p_i^{\min(a_i+b_i,n_i)} \\
&= \gcd(ab,n)
\end{aligned}
$$

Since x is a non-trivial squareroot, we have that $x^2 \equiv 1 \mod n$, but $x \neq 1$ and $x \neq n - 1$. Now, we consider the value of $\gcd(x^2 - 1, n)$. By theorem 31.9, this is equal to $\gcd(n, x^2 - 1 \mod n) = \gcd(n, 1 - 1) = \gcd(n, 0) = n$. So, we can then look at the factorization of $x^2 - 1 = (x + 1)(x - 1)$ to get that

$$
\gcd(x+1,n)\gcd(x-1,n) \geq n
$$

However, we know that since x is a nontrivial squareroot, we know that $1 < x < n - 1$ so, neither of the factors on the right can be equal to n. This means that both of the factors on the right must be nontrivial.

Exercise 31.9-1

The Pollard-Rho algorithm would first detect the factor of 73 when it considers the element 84, when we have x_{12} because we then notice that $\gcd(814 - 84, 1387) = 73$.

Exercise 31.9-2

Create an array A of length n. For each x_i, if $x_i = j$, store i in $A[j]$. If j is the first position of A which must have its entry rewritten, set t to be the entry originally stored in that spot. Then count how many additional x_i's must be computed until $x_i = j$ again. This is the value of u. The running time is $\Theta(t+u)$.

Exercise 31.9-3

Assuming that p^e divides n, by the same analysis as sin the chapter, it will take time $\Theta(p^{e/2})$. To see this, we look at what is happening to the sequence mod p^n.

$$x'_{i+1} = x_{i+1} \mod p^e$$
$$= f_n(x_i) \mod p^e$$
$$= ((x^2 - 1) \mod n) \mod p^e$$
$$= (x^2 - 1) \mod p^e$$
$$= (x'_i)^2 - 1 \mod p^e$$
$$= f_{p^e}(x'_i)$$

So, we again are having the birthday paradox going on, but, instead of hoping for a repeat from a set of size p, we are looking at all the equivalence classes mod p^e which has size p^e, so, we have that the expected number of steps before getting a repeat in that size set is just the squareroot of its size, which is $\Theta(\sqrt{p^e}) = \Theta(p^{e/2})$.

Exercise 31.9-4

Taking the idea suggested in the exercise statement, suppose that we store the product of a batch of B consecutive values of x_i. We should also recall that the number of recursive calls that GCD makes is on the order of the log of the smaller of the two numbers. Since we don't have control over what value between 0 and n is taken by x_i, we should consider that our product of consecutive terms is larger than n. So, we will make $O(\beta)$ many recursive calls. We will compute The B many values of x_i all before computing GCD, or checking against k. However, this means that we need to change line Line 11 of POLLARD-RHO to checking ig $i > k$, and setting y equal to x_k, which will still be remembered as it has to of been in the last B computed. The part that needs justification is that

$$\gcd \left(\prod_{k=0}^{B-1} y - x_{i+k}, n \right)$$

will be non-trivial iff one of $\gcd(y - x_{i+k}, n)$ are nontrivial. We need show that for all a, b, we have $1 < \gcd(ab, n)$ if and only if $1 < \gcd(a, n)$ or $1 < \gcd(b, n)$. Once we have that then we will be able to easily show what we originally needed by induction. For the "if" direction, suppose that, without loss of generality, $d = \gcd(a, n)$ and $1 < d < n$. This means that $d|n$, and $d|a$, but $a|ab$, so, by 31.1-3, we have $d|ab$. Any divisor is a lower bound for the gcd, so we have that $1 < \gcd(ab, n)$. For the "only if" direction, suppose that we have $\gcd(ab, n) = d > 1$. Then, there is some prime number dividing d. This prime has to belong to either a or b. So, we have a positive divisor for one of a or b. Since the prime also divides n, we again get a lower bound on either $\gcd(a, n)$ or $\gcd(b, n)$. In picking B, we need to balance the cost of computing gcd more times with the cost of computing it on larger number. Since we want to think of x_i being (pseudo) randomly distributed, the only control we have on the value of $y - x_i$ is that it is at most n. This means that the value of P in our algorithm can

be as large as n^B. There was also some extra cost in multiplying together the values of x_i. The fastest we can multiply to m bit numbers is $O(m \lg(m))$, so, to compute P, we take up time $\sum_m O(m \lg(m)) = O(Bm \lg(m))$. We would then have to balance this cost against the fact that regardless of B, there will be at most $O(\lg(n))$ many recursive calls to compute the gcd. Another consideration, not on runtime, but on space complexity is that we need to store the b most recent values of x_i, so that when we try to set y to a new value, the x_k that it is referring to is still stored.

Algorithm 4 BATCH-POLLARD-RHO(n,B)

$i = 1$
$x_1 = \text{RANDOM}(0, n - 1)$
$y = x_1$
$k = B$
while TRUE **do**
    $P = 1$
    for $k = 1$ to $k = B$ **do**
        $x_{i+k} = (x_{i+k-1}^2 - 1) \mod n$
        $P = P \cdot (y - x_{i+k})$
    end for
    $i = i + B$
    $d = \gcd(P, n)$
    if $d \neq 1$ and $d \neq n$ **then**
        print d
    end if
    if $i >= k$ **then**
        $y = x_k$
        $k = 2k$
    end if
end while

Problem 31-1

a. If a and b are both even, then we can write them as $a = 2(a/2)$ and $b = 2(b/2)$ where both factors in each are integers. This means that, by Corollary 31.4, $\gcd(a, b) = 2 \gcd(a/2, b/2)$.

b. If a is odd, and b is even, then we can write $b = 2(b/2)$, where $b/2$ is an integer, so, since we know that 2 does not divide a, the factor of two that is in b cannot be part of the gcd of the two numbers. This means that we have $\gcd(a, b) = \gcd(a, b/2)$. More formally, suppose that $d = \gcd(a, b)$. Since d is a common divisor, it must divide a, and so, it must not have any even factors. This means that it also divides a and $b/2$. This means that $\gcd(a, b) \leq \gcd(a, b/2)$. To see the reverse, suppose that $d' = \gcd(a, b/2)$, then it is also a divisor of a and b, since we can just double whatever we need

to multiply it by to get $b/2$. Since we have inequalities both ways, we have equality.

c. If a and b are both odd, then, first, in analog to theorem 31.9, we show that $\gcd(a,b) = \gcd(a - b, b)$. Let d and d' be the gcd's on the left and right respectively. Then, we have that there exists n_1, n_2 so that $n_1 a + n_2 b = d$, but then, we can rewrite to get $n_1(a - b) + (n_1 + n_2)b = d$. This gets us $d \geq d'$. To see the reverse, let n_1', n_2' so that $n_1'(a - b) + n_2'b = d'$. We rewrite to get $n_1'a + (n_2' - n_1')b = d'$, so we have $d' \geq d$. This means that $\gcd(a,b) = \gcd(a - b, b) = \gcd(b, a - b)$. From there, we fall into the case of part b. This is because the first argument is odd, and the second is the difference of two odd numbers, hence is even. This means we can halve the second argument without changing the quantity. So, $\gcd(a, b) = \gcd(b, (a - b)/2) = \gcd((a - b)/2, b)$.

d. See the algorithm BINARY-GCD(a,b)

Algorithm 5 BINARY-GCD(a,b)
___

  if $a \mod 2 \equiv 1$ **then**
    if $b \mod 2 \equiv 1$ **then**
      return BINARY-GCD$((a - b)/2, b)$
    else
      return BINARY-GCD$(a, b/2)$
    end if
  else
    if $b \mod 2 \equiv 1$ **then**
      return BINARY-GCD$(a/2, b)$
    else
      return 2·BINARY-GCD$(a/2, b/2)$
    end if
  end if
___

Problem 31-2

a. We can imagine first writing a and b in their binary representations, and then performing long division as usual on these numbers. Each time we compute a term of the quotient we need to perform a multiplication of a and that term, which takes $\lg b$ bit operations, followed by a subtraction from the first b terms of a which takes $\lg b$ bit operations. We repeat this once for each digit of the quotient which we compute, until the remainder is smaller than b. There are $\lg q$ bits in the quotient, plus the final check of the remainder. Since each requires $\lg b$ bit operations to perform the multiplication and subtraction, the method requires $O((1 + \lg q)\lg b)$ bit operations.

b. The reduction requires us to compute $a \mod b$. By carrying out ordinary "paper and pencil" long division we can compute the remainder. The time to do this, by part a, is bounded by $k(1 + \lg q)(\lg b)$ for some constant k. In the worst case, q has $\lg a - \lg b$ bits. Thus, we get the bound $k(1 + \lg a - \lg b)(\lg b)$. Next, we compute $\mu(a, b) - \mu(b, a \mod b) = (1 + \lg a)(1 + \lg b) - (1 + \lg b)(1 + \lg(a \mod b)$. This is smallest when $a \mod b$ is small, so we have a lower bound of $(1 + \lg a)(1 + \lg b) - (1 + \lg b)$. Assuming $\lg b \geq 1$, we can take $c = k$ to obtain the desired inequality.

c. As shown in part (b), EUCLID takes at most $c(\mu(a, b) - \mu(b, a \bmod b))$ operations on the first recursive call, at most $c(\mu(b, a \mod b) - \mu(a \mod b, b \mod (a \mod b))$ operations on the second recursive call, and so on. Summing over all recursive calls gives a telescoping series, resulting in $c\mu(a, b) + O(1) = O(\mu(a, b))$. When applies to two β-bit inputs the runtime is $O(\mu(a, b)) = O((1 + \beta)(1 + \beta)) = O(\beta^2)$.

Problem 31-3

a. Mirroring the proof in chapter 27, we first notice that in order to solve $FIB(n)$, we need to compute $FIB(n - 1)$ and $FIB(n - 2)$. This means that the recurrence it satisfies is

$$T(n) = T(n - 1) + T(n - 2) + \Theta(1)$$

We find it's solution using the substitution method. Suppose that the $\Theta(1)$ is bounded above by c_2 and bounded below by c_1. Then, we'll inductively assume that $T(k) \leq cF_k - c_2 k$ for $k < n$. Then,

$$
\begin{aligned}
T(n) &= T(n - 1) + T(n - 2) \\
&\leq cF_{n-1} - c_2(n - 1) + cF_{n-2} - c_2(n - 2) + c_2 \\
&= cF_n - c_2 n + (4 - n)c_2 \\
&\leq cF_n - c_2 n
\end{aligned}
$$

Where the last inequality only holds if we have that $n \geq 4$, but since small values can just be absorbed into the constants, we are allowed to assume this.

To show that $T \in \Omega(F_n)$, we again use the substitution method. Suppose that $T(k) \geq cF_k + c_1 k$ for $k < n$. Then.

$$
\begin{aligned}
T(n) &= T(n - 1) + T(n - 2) \\
&\geq cF_{n-1} + c_1(n - 1) + cF_{n-2} + c_1(n - 2) + c_1 \\
&= cF_n + c_1 n + (n - 4)c_1 \\
&\geq cF_n - c_1 n
\end{aligned}
$$

Again, this last inequality only holds if we have $n \geq 4$, but small cases can be absorbed into the constants, we may assume that $n \geq 4$.

b. This problem is the same as exercise 15.1-5.

c. For this problem, we assume that all integer multiplications and additions can be done in unit time. We will show first that

$$\left(\begin{array}{cc} 0 & 1 \\ 1 & 1 \end{array} \right)^k = \left(\begin{array}{cc} F_{k-1} & F_k \\ F_k & F_{k+1} \end{array} \right)$$

Where we start We will proceed by induction. Then,

$$\left(\begin{array}{cc} 0 & 1 \\ 1 & 1 \end{array} \right)^{k+1} = \left(\begin{array}{cc} 0 & 1 \\ 1 & 1 \end{array} \right) \left(\begin{array}{cc} 0 & 1 \\ 1 & 1 \end{array} \right)^k$$

$$= \left(\begin{array}{cc} 0 & 1 \\ 1 & 1 \end{array} \right) \left(\begin{array}{cc} F_{k-1} & F_k \\ F_k & F_{k+1} \end{array} \right)$$

$$= \left(\begin{array}{cc} F_k & F_{k-1} + F_k \\ F_{k-1} + F_k & F_k + F_{k+1} \end{array} \right)$$

$$= \left(\begin{array}{cc} F_k & F_{k+1} \\ F_{k+1} & F_{k+2} \end{array} \right)$$

completing the induction. Then, we just show that we can compute the given matrix to the power $n - 2$ in time $O(\lg(n))$, and look at it's bottom right entry. We will use a technique similar to section 31.6, that is, we will use the idea of iterated squaring in order to obtain high powers quickly. First, we should note that using 8 multiplications and 4 additions, we can multiply any two square matrices. This means that matrix multiplications can be done in constant time, so, we only need to bound the number of those in our algorithm. Run the algorithm MATRIX-POW(A,n-2) and extract the bottom left argument. We can see that this algorithm only takes time $O(\lg(n))$ because in each step, we are halving the value of n, and within each step, we are only performing a constant amount of work, so the solution to

$$T(n) = T(n/2) + \Theta(1)$$

is $O(\lg(n))$ by the master theorem.

Algorithm 6 MATRIX-POW(A,n)

---

if $n\%2 = 1$ **then**
    return $A \cdot$MATRIX-POW$(A^2, \frac{n-1}{2})$
else
    return MATRIX-POW$(A^2, n/2)$
end if

---

d. Here, we replace the assumption of unit time additions and multiplications with having it take time $\Theta(\beta)$ to add and $\Theta(\beta^2)$ to multiply two β bit numbers. For the naive approach, We are adding a number which is growing exponentially each time, so, the recurrence becomes

$$T(n) = T(n-1) + T(n-2) + \Theta(n)$$

Which has the same solution 2^n. Which can be seen by a substitution argument. Suppose that $T(k) \le c2^k$ for $k < n$. Then,

$$
\begin{aligned}
T(n) &= T(n-1) + T(n-2) + \Theta(\lg(n)) \\
&\le c(\frac{1}{2} + \frac{1}{4})2^n + \Theta(\lg(n)) \\
&= c2^n - c2^{n-2} + \Theta(\lg(n)) \\
&\le c2^k
\end{aligned}
$$

Since we had that it was $\Omega(2^n)$ in the case that the term we added was $\Theta(1)$, and we have upped this term to $\Theta(\lg(n))$, we still have that $T(n) \in \Omega(2^n)$. This means that $T(n) \in \Theta(2^n)$.

Now, considering the memoized version. We have that our solution has to satisfy the recurrence

$$M(n) = M(n-1) + \Theta(n)$$

This clearly has a solution of $\sum_{i=2}^{n} n \in \Theta(n^2)$ by equation (A.11) where it is trivial to obtain $\int x \, dx$.

Finally, we reanalyze our solution to part (c). For this, we have that we are performing a constant number of both additions and multiplications. This means that, because we are multiplying numbers that have value on the order of ϕ^n, hence have order n bits, our recurrence becomes

$$P(n) = P(n/2) + \Theta(n^2)$$

Which has a solution of $\Theta(n^2)$.

Though it is not asked for, we can compute Fibonacci in time only $\Theta(n \lg(n))$ because multiplying integers with β bits can be done in time $\beta \lg(\beta)$ using the fast Fourier transform methods of the previous chapter.

Problem 31-4

a. Since p is prime, Theorem 31.32 implies that $\mathbb{Z}_p^*$ is cyclic, so it has a generator g. Thus, $g^2, g^4, \ldots, g^{p-1}$ are all distinct. Moreover, each one is clearly a

quadratic residue so there are at least $(p-1)/2$ residues. Now suppose that a is a quadratic residue and $a = g^{2k+i}$ for some i. Then we must also have $a = x^2$ for some x, and $x = g^m$ for some m, since g is a generator. Thus, $g^{2m} = g^{2k+i}$. By the discrete logarithm theorem we must have $2m = 2k+1$ mod $\phi(p)$. However, this is impossible since $\phi(p) = p-1$ which is even, but $2m$ and $2k+1$ differ by an odd amount. Thus, precisely the elements of the form g^{2i} for $i = 1, 2, \ldots, (p-1)/2$ are quadratic residues.

b. If a is a quadratic residue modulo p then there exists x such that $a^{(p-1)/2} = (x^2)^{(p-1)/2} = x^{p-1} = 1 = \left(\frac{a}{p}\right)$. On the other hand, suppose a is not a quadratic residue. Then $a = g^{2i+1}$ for some i and $a^{(p-1)/2} = \left(g^{2i+1}\right)^{(p-1)/2} = \left(g^{(p-1)/2}\right)^{2i+1} = (-1)^{2i+1} = -1 \mod p$. To see why $g^{(p-1)/2} = -1$, recall that Theorem 31.34 tells us that $g = \pm 1$. Since g is a generator, powers of g are distinct. Since $g^{p-1} = 1$, we must have $g^{(p-1)/2} = -1$.

To determine whether a given number a is a quadratic residue modulo p, we simply compute $a^{(p-1)/2} \mod p$ and check if it is 1 or -1. We can do this using the MODULAR-EXPONENTIATION function, and the number of bit operations is $O((\lg p)^3)$.

c. If $p = 4k+3$ and a is a quadratic residue then $a^{2k+1} = 1 \mod p$. Then we have $(a^{k+1})^2 = a^{2k+2} = aa^{2k+1} = a$, so a^{k+1} is a square root of a. To find the square root, we use the MODULAR-EXPONENTIATION algorithm which has $O((\lg p)^3)$ bit operations.

d. Run the algorithm of part b repeatedly until a non-quadratic residue is found. Since only half the elements of $\mathbb{Z}_p^*$ are residues, after k runs this approach will find a nonresidue with probability $1 - 2^{-k}$. The expected number of runs is $\sum_{k=1}^{\infty} k \cdot 2^{-k} = 2$, so the expected number of bit operations is $O((\lg p)^3)$.

Chapter 32

Michelle Bodnar, Andrew Lohr

April 12, 2016

Exercise 32.1-1

We let (i, j) denote that the algorithm checks index i of the text against index j of the pattern. We'll let p(s) indicate that the matching algorithm reported an occurrence with a shift of s. The algorithm has the following execution on $T = 000010001010001$ and $P = 0001$.

$s = 0$	$(0, 0)$	$(1, 1)$	$(2, 2)$	$(3, 3)$	
$s = 1$	$(1, 0)$	$(2, 1)$	$(3, 2)$	$(4, 3)$	$p(1)$
$s = 2$	$(2, 0)$	$(3, 1)$	$(4, 2)$		
$s = 3$	$(3, 0)$	$(4, 1)$			
$s = 4$	$(4, 0)$				
$s = 5$	$(5, 0)$	$(6, 1)$	$(7, 2)$	$(8, 3)$	$p(5)$
$s = 6$	$(6, 0)$	$(7, 1)$	$(8, 2)$		
$s = 7$	$(7, 0)$	$(8, 1)$			
$s = 8$	$(8, 0)$				
$s = 9$	$(9, 0)$	$(10, 1)$			
$s = 10$	$(10, 0)$				
$s = 11$	$(11, 0)$	$(12, 1)$	$(13, 2)$	$(14, 3)$	$p(11)$

Exercise 32.1-2

We know that one occurrence of P in T cannot overlap with another, so we don't need to double-check the way the naive algorithm does. If we find an occurrence of P_k in the text followed by a nonmatch, we can increment s by k instead of 1. It can be modified in the following way:

Exercise 32.1-3

For any particular value of s, the probability that the ith character will need to be looked at when doing the string comparison is the probability that the first $i - 1$ characters matched, which is $\frac{1}{d^{i-1}}$. So, by linearity of expectation, summing up over all s and i, we have that the expected number of steps is, by equation (A.5),

Algorithm 1 DISTINCT-NAIVE-STRING-MATCHER(T, P)

$n = T.length$
$m = P.length$
$k = 0$
while $s \leq n - m$ **do**
    $i = 1$
  if $T[s] == P[1]$ **then**
      $k = s$
      $i = 0$
      while $T[k + i] == P[i]$ and $i < m$ **do** $i = i + 1$
        if $i == m$ **then**
          Print "Pattern occurs with shift" k
        end if
      end while
  end if
    $s = s + i$
end while

$$(n - m + 1) \sum_{i=1}^{m} \frac{1}{d^{i-1}} = (n - m + 1) \left(\frac{(d^{-m} - 1)}{(d^{-1} - 1)} \right)$$

$$= (n - m + 1)(\frac{1 - d^{-m}}{1 - d^{-1}})$$

$$\leq (n - m + 1)\frac{1}{1 - d^{-1}}$$

$$\leq (n - m + 1)\frac{1}{1 - \frac{1}{2}}$$

$$= 2(n - m + 1)$$

Exercise 32.1-4

We can decompose a pattern with $g - 1$ gap characters into the form $a_1 \diamond a_2 \diamond \cdots \diamond a_g$. Since we only care whether or not the pattern appears somewhere, it will suffice to look for the first occurrence of a_1, followed by the first occurrence of a_2 which comes anywhere after a_1, and so on. If the pattern P has length m and the text has length n then the runtime of the naive strategy is $O(nm)$.

Exercise 32.2-1

Since the string 26 only appears once in the text, to find the number of spurious hits, we will find the total number of hits and subtract 1. when we compute the hash of the pattern we get $(20 + 6) \mod 11 \equiv 26 \mod 11 \equiv 4 \mod 11$.

We get the following hashes of various shift values:

s	t_s
0	9
1	3
2	8
3	4
4	4
5	4
6	4
7	10
8	9
9	2
10	3
11	1
12	9
13	2
14	5

Since there were 4 hits, three of them must of been spurious.

Exercise 32.2-2

We first tackle the case where each of the k patterns has the same length m. We first compute the number associated to each pattern in $O(m)$ time, which contributes $O(km)$ to the runtime. In line 10 we make k checks, one for each pattern, which contributes $O(n - m + 1)km$ time. Thus, the total runtime is $O(km(n - m + 1))$. For patterns of different lengths $l_1, l_2, \ldots, l_k$, keep an array A such that $A[i]$ holds the number associated to the first l_i digits of T, mod q. We'll have to update each of these each time the outer for-loop is executed, but it will allow us to immediately compare any of the pattern numbers to the appropriate text number, as done in line 10.

Exercise 32.2-3

We use the same idea of maintaining a hash of the pattern and computing running hashes of the text. However, updating the hash at each step can take time as long as $\Theta(m)$ because the number of entries which are both entering and leaving the hashed window is $2m$, and you have to at least look at all of them as they come in and leave. This would get us a total expected runtime (occurring not too many spurious hits of $(n - m + 1)^2 \cdot m$, and still the same worst case as the trivial algorithm, which is $(n - m + 1)^2 \cdot m^2$.

In order to compute this hash, we will be giving each entry of the window a power of d that unique to its position. The entry in row i, column j will be multiplied by $d^{m^2 - mi + j}$. Then, moving to the right, we multiply the value of the hash by d, subtract off the scaled entries that were in the left column, and

3

add in the entries that are in the right column, also appropriately scaled by what row they are in. Similarly for shifting the window up or down. Again, all of this arithmetic is done mod some large prime.

Exercise 32.2-4

Suppose $A(x) = B(x)$. Then $\sum_{i=0}^{n-1}(a_i - b_i)x^i \equiv 0 \mod q$. Since q is prime, Exercise 31.4-4 tells us that this equation has at most $n-1$ solutions modulo q. Since each of the q choices for x is equally likely, the probability that we pick one of the potential $n-1$ which make the equation hold is $(n-1)/q < (n-1)/1000n < 1/1000$. However, if the files are the same then $a_i = b_i$ for all i, so $A(x) = B(x)$.

Exercise 32.3-1

The states will be $\{0, 1, 2, 3, 4, 5, 6\}$ and having a transition function given by

state	a	b
0	1	0
1	2	0
2	2	3
3	4	0
4	2	5
5	1	0

The sequence of states for T is $0, 1, 2, 2, 3, 4, 5, 1, 2, 3, 4, 2, 3, 4, 5, 1, 2, 3$, and so finds two occurrences of the pattern, one at $s = 1$ and another at $s = 9$.

Exercise 32.3-2

See picture below for the 22 state automaton.
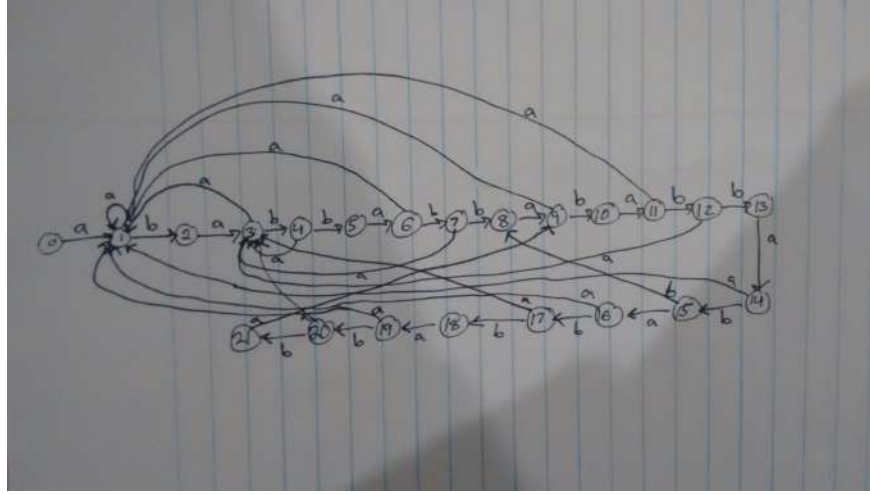
Exercise 32.3-3

The state transition function looks like a straight line, with all other edges going back to either the initial vertex (if it is not the first letter of the patter) or the second vertex (if it is the first letter of the pattern). If it were to go back to any later state, that would mean that some suffix of what we had constructed so far(which was a prefix of P) was a prefix of the copy of P that we are next trying to find.

Exercise 32.3-4

We can construct the automaton as follows: Let P_k be the longest prefix which both P and P' have in common. Create the automaton F for P_k as usual. Add an arrow labeled $P[k+1]$ from state k to a chain of states $k+1, k+2, \ldots, |P|$, and draw the appropriate arrows, so that $\delta(q,a) = \sigma(P_q a)$. Next, add an arrow labeled $P'[k+1]$ from state k to a chain of states $(k+1)', (k+2)', \ldots, |P'|'$. Draw the appropriate arrows so that $\delta(q,a) = \sigma(P'_q a)$.

Exercise 32.3-5

To create a DFA that worked with gap characters, construct the DFA so that it has $|P|+1$ states. let m be the number of gap characters Suppose that the positions of all the gap characters within the pattern p are given by g_i, and let $g_0 = 0$, $g_m = |P| + 1$. Let the segment of pattern occurring after gap character i but before the $i+1$ gap character be called P_i. Then, we will imagine that we are trying to match each of these patterns in sequence, but if we have trouble matching some particular pattern, then we can not undo the success we enjoyed in matching earlier patterns.

More concretely, suppose that we have $(Q_i, q_{i,0}, A_i, \Sigma_i, \delta_i)$ is the DFA corre-

sponding to the pattern P_i. Then, we will construct our DFA so that $Q = \sqcup_i Q_i$, $q_0 = q_{0,0}$, $A = A_{m+1}$, $\Sigma = \cup_i \Sigma_i$, and δ is described below. If we are at state $q \in Q_i$ and see character a, if $q \notin A_i$, we just go to the state proscribed by $\delta_i(q, a)$. If, however, we have that $q \in A_i$, then, $\delta(q, a) = \delta_{i+1}(q_{i+1,0}, a)$. This construction achieves the description given in English above.

Exercise 32.4-1

The prefix function is:

i	$\pi(i)$
1	0
2	0
3	1
4	2
5	0
6	1
7	2
8	0
9	1
10	2
11	0
12	1
13	2
14	3
15	4
16	5
17	6
18	7
19	8

Exercise 32.4-2

The largest $\pi^*[q]$ can be is $q - 1$. This is tight because if P consists of a the letter a repeated m times, the $\pi^*[q] = q - 1$ for all q.

Exercise 32.4-3

Suppose that at position i, you have the value of the prefix function is $\pi[i]$. Then, if $i - \pi[i] \geq |P|$, this means that there is an occurrence of $|P|$ starting at position $i - \pi[i]$.

A simpler way to achieve a similar result is to expand the alphabet by one, making it so that some character c does not occur in either P or T, then compute the prefix array of PcT, then, the prefix function is bounded by $|P|$ and any time that it reaches that bound, we have that there is an occurrence of the pattern, since we know that any prefix containing the c cannot be a proper

suffix of any other prefix.

Exercise 32.4-4

To show that the running time of KMP-MATCHER is $O(n)$, we'll show that the total number of executions of the while loop of line 6 is $O(n)$. Observe that for each iteration for the for loop of line 5, q increases by at most 1, in line 9. This is because $\pi(q) < q$. On the other hand, the while loop decreases q. Since q can never be negative, we must decrease q fewer than n times in total, so the while loop executes at most $n - 1$ times. Thus, the total runtime is $O(n)$.

Exercise 32.4-5

Basically, each time that we have to execute line 7, we have that we are decreasing q by at least 1. Since the only place that we ever increase the value of q is on line 9, and then we are only increasing it by 1, each of these runs of line 5 are paid for by the times we ran 9.

Using this as our motivation, we let the potential function be proportional to the value of q. This means that when we execute line 9, we pay a constant amount to raise up the potential function. And when we run line 7, we decrease the potential function which reduces the ammortized cost of an iteration of the while loop on line 6 to a zero amortized cost. The only other time that we change the value of q is on line 12, which only gets run once per execution of the outer for loop anyways and the amortization is in our favor there.

Since the ammortized cost under this potential function of each iteration of the outermost for loop is constant, and that loop runs n times, the total cost of the algorithm is $\Theta(n)$.

Exercise 32.4-6

We'll prove this by induction on the number of recursive calls to π'. The behavior is identical to π if we are in the first or third cases (base cases) of the definition of π', so the behavior is correct for a single call. Otherwise, $\pi'[q] = \pi'[\pi[q]]$. The conditions of case 2 imply that the while loop of line 6 will execute an additional time after the update of line 7, so it is equivalent to setting $q = \pi[\pi[q]]$, and then continuing with the while loop as usual. Since π' recurses on $\pi[q]$ one fewer times than on q, its behavior is correct on $\pi[q]$, proving that the modified algorithm is correct. KMP-MATCHER already runs asymptotically as fast as possible, so this doesn't constitute a runtime improvement in the worst case. However, every time a recursive call to π' is made, we circumvent having to check $P[q + 1]$ against $T[i]$.

Exercise 32.4-7

If the lengths to T and T' are different, they are obviously not cyclic rotations of each other, so suppose that $|T| = T'$. Let our text be TT and our pattern

be T'. If and only if T' occurs in TT, then the two given strings are cyclic rotations of each other. This can be done in linear time by the Knuth Morris Pratt algorithm.

To see that being cyclic rotations means that T' occurs in TT, suppose that T' is obtained from T by cyclically shifting the right character to the left s times. This means that the $|T| - s$ prefix of T is a suffix of T', and the s suffix of T is a prefix of T'. This means that T' occurs in TT with a shift of $|T| - s$.

Now, suppose that T' occurs in TT with a shift of s. This means that the s suffix of T is a prefix of T', it also means that the $|T| - s$ characters left over in T' are a prefix of T. So, they are cyclic rotations of each other.

Exercise 32.4-8

We have $\delta(q, a) = \sigma(P_q a)$, which is the length of the longest prefix of P which is a suffix of $P_q a$. Let this be k. Then $P[1] = P[q - k + 2]$, $P[2] = P[q - k + 3], \ldots, P[k - 1] = P[q], P[k] = a$. On the other hand, $\delta(\pi[q], a) = \sigma(P_{\pi[q]} a)$. It is clear that $\sigma(P_q a) \geq \sigma(P_{\pi[q]} a)$. However, $\pi[q] \geq k - 1$ and since $P[k] = a$, we must have $\sigma(P_{\pi[q]} a) \geq k$. Thus, they are equal. If $q \neq m$ and $P[q + 1] = a$ then $\delta(q, a) = q + 1$. We can now compute the transition function δ in $O(m|\Sigma|)$ in the algorithm TRANSITION-FUNCTION, given below.

Algorithm 2 TRANSITION-FUNCTION(P, Σ)

$\pi = $ COMPUTE-PREFIX-FUNCTION(P)
for $a \in \Sigma$ **do**
    $\delta(0, a) = 0$
end for
$\delta(0, P[1]) = 1$
for $a \in \Sigma$ **do**
    for $q = 1$ to m **do**
        if $q == m$ or $P[q + 1] \neq a$ **then**
            $\delta(q, a) = \delta(\pi[q], a)$
        else
            $\delta(q, a) = q + 1$
        end if
    end for
end for

Problem 32-1

a. First, compute the prefix array for the given pattern in linear time using the method of section 32.4. Then, Suppose that $\pi[i] = i - k$. If $k | i$, we know that k is the length of the primitive root, so, the word has a repetition factor of $\frac{i}{k}$. We also know that there is no smaller repetition factor in this case because otherwise we could include one more power of that root.

Now, suppose that we have k not dividing i. We will show that we can only have the trivial repetition factor of 1. Suppose we had some repetition $y^r = P_i$. Then, we know that $\pi[i] \geq y^{r-1}$. however, if we have it strictly greater than this, this means that we can write the y's themselves as powers because we have them aligning with themselves.

Since the only difficult step of this was finding the prefix function, which takes linear time, the runtime of this procedure is linear.

b. To determine the probability that there is a repetition factor of r, we assign whatever we want to the first $\frac{i}{r}$ letters, and after that, all of the other letters are determined. This means that position i has a repetition factor of r with probability $\frac{1}{2^{i\frac{(r-1)}{r}}}$ for every r dividing i. By applying a union bound to this, we have that the probability that P_i has a repetition factor of more than r is bounded by

$$\sum_{r'>r, r'|i} \frac{1}{2^{i\frac{(r'-1)}{r'}}} = \frac{1}{2^i} \sum_{r'>r, r'|i} 2^{i/r'}$$

$$= \frac{1}{2^i} \sum_{j=1}^{\lfloor \frac{i/r}{} \rfloor} 2^j$$

$$\leq \frac{2^{i/r}}{2^{i-1}}$$

$$= 2^{i/r-i+1}$$

Then, applying the union bound over all values of i, we have the probability that a repetition factor of at least r is bounded by

$$\sum_{i=0}^{m} 2^{i/r-i+1} = 2 \sum_{i=1}^{m} 2^{(\frac{1}{r}-1)i}$$

$$= 2 \frac{1 - 2^{\frac{m+1}{r}-m-1}}{1 - 2^{(\frac{1}{r}-1)}}$$

$$\leq 2 \frac{1}{2^{(\frac{1}{r}-1)}}$$

This shrinks quickly enough in r, that the expected value is finite, and since there is no m in the expression, we have the expected value is bounded by a constant.

c. This algorithm correctly finds the occurrences of the pattern P for reasons similar to the Knuth Morris Pratt algorithm. That is, we know that we will only increase $q \, \rho^*(P)$ many times before we have to bite the bullet and

9

increase s, and s can only be increased at most $n - m$ many times. It will definitely find every possible occurrence of the pattern because it searches for every time that the primitive root of P occurs, and it must occur for P to occur.

Chapter 33

Michelle Bodnar, Andrew Lohr

April 12, 2016

Exercise 33.1-1

Suppose first that the cross product is positive. We shall consider the angles that both of the vectors make with the positive x axis. This is given by $tan^{-1}(y/x)$ for both. Since the cross product is positive, we have that $0 < x_1 y_2 - x_2 y_1$, which means that $\frac{y_1}{x_1} < \frac{y_2}{x_2}$. however, since arctan is a monotone function, this means that the angle that p_2 makes with the positive x axis is greater than the angle that p_1 does, which means that you need to move in a clockwise direction to get from p_2 to p_1.

If the cross product is negative, this means that we have $\frac{y_1}{x_1} > \frac{y_2}{x_2}$ which means that the angle that p_1 makes is greater, which means that we need to go counter clockwise from p_2 to get to p_1.

Exercise 33.1-2

If the segment $\overline{p_i p_j}$ is vertical, then p_k could be colinear with p_i and p_j, but lie directly below them. Then $x_i = x_j = x_k$, so if we don't also check the y values we won't catch that p_k is not on the segment.

Exercise 33.1-3

The beauty of the fact that our sorting algorithms from earlier in the book were only comparison based is that if we can implement a comparison operation between any two elements that operates in constant time, then, we can use the earlier comparison based sorting algorithms as a black box to work on our data.

So, what we need to do is, given two indices i and j, decide whether the polar angle of p_i with respect to p_0 is larger or smaller than the polar angle of p_j with respect to p_0. This can be done with a single cross product. That is, we look at the cross product, $(p_1 - p_0) \times (p_2 - p_0)$. This is positive if we need to turn left from p_1 to get to p_2. That is, if it is positive, then the polar angle is greater for p_2 than from p_1. We similarly know that we are in the reverse situation if we have that this cross product is negative. The only tricky thing is that we could have two distinct elements p_i and p_j so that the cross product is still zero. The problem statement is unclear how to resolve these sorts of ties, because they have the same polar angle. We could just pick some arbitrary

property of the points to resolve ties, such as we pick the point that is farther away from p_0 to be larger. Since we have a total ordering on the points that can be queried in constant time, we can use it in our $O(n \lg(n))$ algorithms from earlier on in the book.

Exercise 33.1-4

By Exercise 33.1-3 we can sort n points according to their polar angles with respect to a given point in $O(n \lg n)$ time. If points p_i, p_j, and p_k are colinear, then at two one of the following is true: (1) p_j and p_k have the same polar angle with respect to p_i, (2) p_i and p_k have the same polar angle with respect to p_j, or (3) p_i and p_j have the same polar angle with respect to p_k. Thus, it will suffice to do as follows: For each point p, compute the polar angle of all other points with respect to p. If there are any duplicates, those points are colinear. Since we must do this for each point, the algorithm has runtime $O(n^2 \lg n)$.

Exercise 33.1-5

Because, as stated in this definition of convex polynomials, we cannot have a vertex of a convex polygon be a convex combination of any two points of the boundary of the polynomial. This means that as we enter a particular vertex, we cannot have that it is colinear with the next vertex. Professor Amundsen's algorithm just rejects if both left and right turns are made. However, it should also reject if there is ever any vertex where no turn is made, because that vertex would then be a convex combination of the next and previous vertices.

Exercise 33.1-6

It will suffice to check whether or not the line segments $\overline{p_0 p_3}$ and $\overline{p_1 p_2}$ intersect, where $p_3 = (\max(x_1, x_2), y_0)$. We can do this in $O(1)$.

Exercise 33.1-7

Starting from the point p_0, pick an arbitrary direction, and consider the ray coming out in that direction. Instead of just counting the intersections with the sides of the polygon, we'll also count all the vertices that the ray intersects. for each side that it intersects, if it intersects the vertices at both sides, then we don't count that edge, because that means that the ray passes along that side. Lastly, if the ray passes through any vertex where both sides touching that vertex aren't of the previous type, we flip the parity of the count. Lastly, we say it is inside if the final count is odd. See the algorithm DETERMINE-INSIDE(P,p).

Exercise 33.1-8

Without loss of generality, assume that the interior of the polygon is to the right of the first segment $\overline{p_0 p_1}$. We will examine segments of the polygon one at a

Algorithm 1 DETERMINE-INSIDE(P,p), P is a polygon, and p is a point

Let S be the set of sides that the right horizontal ray intersects
Let T be the set of vertices that the right horizontal ray intersects
Let U be an empty set of sides
count = 0
for $s \in S$ **do**
    let p_1 and p_2 be the vertices at either side of s
    if $p_1 \in T$ and $p_2 \in T$ **then**
        put s in U
    end if
    count++
end for
for $x \in T$ **do**
    let y and z be the sides that x is touching
    if $y \in U$ or $z \in U$ **then**
        count++
    end if
end for
if count is odd **then**
    return inside
else
    return outside
end if

time. At any point, if we are at segment $\overline{p_i p_{i+1}}$ and if the next segment $\overline{p_{i+1}p_{i+2}}$ of the polygon turns right then, then we can compute the are of $\triangle p_i p_{i+1} p_{i+2}$, and reduce the problem to that of finding the area of the polygon without p_{i+1}, adding the area just computed. On the other hand, if we turn left then we need to compute the area of the polygon without p_{i+1}, but we need to subtract the area of $\triangle p_i p_{i+1} p_{i+2}$. Since we can compute the area of a triangle given its vertices in constant time, the runtime satisfies $T(n) = T(n-1) + O(1)$, so $T(n) = O(n)$.

Exercise 33.2-1

Suppose you split your set of lines into two equal sets, each of size $n/2$. Then, we will make half of them horizontal and close together, each above the next. That is, we'll put a horizontal line at $y = \frac{1}{k}$ for $k = 1, \ldots, n/2$ extending from -1 to 1. For the other half, we'll put lines along $x = \frac{1}{k}$ for $k = 1, \ldots, n/2$, extending from -1 to 1. Then, we'll have every line from the first set intersect every line from the second set. Therefore the total number of intersections is $\frac{n^2}{4}$, which is $\Theta(n^2)$.

Exercise 33.2-2

First suppose that a and b do not intersect. Let a_s, a_f, b_s, b_f denote the left and right endpoints of a and b respectively. Without loss of generality, let b have the leftmost left endpoint. If $\overline{b_s a_s}$ is to the right of $\overline{b_s b_f}$, then a is below b. Otherwise a is above b. Now suppose that a and b intersect. To decide which segment is on top, we need to determine whether the intersection occurs to the left or right of x. Assume that each point has x and y attributes. For example, $a_s = (a_s.x, a_s.y)$. The equation of the line through segment a is $y = m_1(x - a_s.x) + a_s.y$ where $m_1 = \frac{a_f.y - a_s.y}{a_f.x - a_s.x}$. The equation of the line through segment b is $y = m_2(x - b_s.x) + b_s.y$ where $m_2 = \frac{b_f.y - b_s.y}{b_f.x - b_s.x}$. Setting these equal to each other gives

$$x = \frac{b_s.y - m_2 b_s.x - a_s.y + m_1 a_s.x}{m_1 - m_2}.$$

Let $x = x_0$ be the equation of the sweep line at which we want to test the relationship between a and b. We need to determine whether or not $x < x_0$, but without using division. To do this, we'll need to clear denominators. $x < x_0$ is equivalent to

$$b_s.y - m_2 b_s.x - a_s.y + m_1 a_s.x < (m_1 - m_2)x_0$$

which is equivalent to this gross mess, which fortunately requires only addition, subtraction, multiplication, and comparison, so it is numerically stable:

$$(a_f.x - a_s.x)(b_f.x - b_s.x)(b_s.y - a_s.y) - (a_f.x - a_s.x)(b_f.y - b_s.y)b_s.x + (b_f x - b_s.x)(a_f.y - a_s.y)a_s.x$$
$$< (b_f.x - b_s.x)(a_f.y - a_s.y)x_0 - (a_f.x - a_s.x)(b_f.y - b_s.y)x_0.$$

Exercise 33.2-3

It looks like the moral of this book is that the only time that a professor can be right is when he's disagreeing with another professor. Professor Dixon is correct.

It will not necessarily print the leftmost intersection first. The intersection that it prints first will be the pair of lines such that both lines have their endpoints show up first in the lexicographical ordering on line 2. An example is, suppose we have the lines $\{\{(0, 1000), (2, 2000)\}, \{(0, 1001), (2, 1001)\}, \{(0, 0), (1, 2)\}, \{(0, 2), (1, 0)\}\}$. Then, the first two lines have the leftmost intersection, but the intersection between the last two lines will be printed out first.

The procedure will not necessarily display all intersections, in particular, suppose that we have the line segments $\{\{(0, 0), (4, 0)\}, \{(0, 1), (4, -2)\}, \{(0, 2), (4, -2)\}, \{(0, 3), (4, -1)\}\}$. There are intersections of the first line segment with each of the other line segments at 1,2, and 3. However, we cannot detect the intersection at 2 because the line segment from $(0, 2)$ to $(4, -2)$ is not adjacent to the horizontal line segment in the red-black tree either when we process left endpoints or right endpoints.

Exercise 33.2-4

An n vertex polygon $\langle p_0, p_1, \ldots, n_{n-1} \rangle$ is simple if and only if the only intersections of the segments $\overline{p_0 p_1}, \overline{p_1 p_2}, \ldots, \overline{p_{n-1} p_0}$ of the boundary are between consecutive segments $\overline{p_i p_{i+1}}$ and $\overline{p_{i+1} p_{i+2}}$ at the point p_{i+1}. We run the usual ANY-SEGMENTS-INTERSECT algorithm on the segments which make up the boundary of the polygon, with the modification that if an intersection is found, we first check if it is an acceptable one. If so, we ignore it and proceed. Since we can check this in $O(1)$, the runtime is the same as ANY-SEGMENTS-INTERSECT.

Exercise 33.2-5

Construct the set of line segments which correspond to all the sides of both polygons, then just use the algorithm from this section to see if any pair of them intersect. If we are in the fringe case that some segment is vertical, just rotate the whole picture by some epsilon. This won't change whether or not there is an intersection.

Exercise 33.2-6

We can use a modified version of the intersecting-segments algorithm to solve this problem. We'll first associate left and right endpoints to each disk. If disk D has radius r and center (x, y), define its left endpoint to be $(x - r, y)$ and its right endpoint to be $(x + r, y)$. Begin by ordering the endpoints of the disks first by left-right position. If two endpoints have the same x-coordinate, then covertical left endpoints come before right endpoints. Within these, order by y-coordinates from low to high. We'll use the same event point schedule as for the intersecting segments problem. Maintain a sweep-line status that gives

the relative order of the segments of the disks, where the segment associated to each disk is the segment formed by its left and right endpoints. When we encounter a left endpoint, we add the associated disk to the sweep-line status. When we encounter a right endpoint, we delete the disk from the sweep-line status. Consider the first time two disks become consecutive in the ordering. Let their centers be (x_1, y_1) and (x_2, y_2), and their radii be r_1 and r_2. Check if $(x_2 - x_1)^2 + (y_2 - y_1)^2 \leq (r_1 + r_2)^2$. If yes, then the two circles intersect. Otherwise they don't. Since we can check this in $O(1)$, and there are only $2n$ points which are added, we make at most $4n$ checks in total. Sorting the points takes $O(n \lg n)$, so the total runtime is $O(n \lg n) + O(n) = O(n \lg n)$.

Exercise 33.2-7

We preform a slight modification to what Professor Mason suggested in exercise 33.2-3. Once we have found an intersection, we then keep considering elements further and further away in the red black tree until we no longer have an intersection. Since all the tree operations only take time $O(\lg(n))$, and we are only doing an additional one on top of the original algorithm for each of the intersections that we found, we have that the additional runtime is $O(k \lg(n))$ so, the total runtime is $O((n + k) \lg(n))$.

Exercise 33.2-8

Suppose that at least 3 segments intersect at the same point. It is clear that if ANY-SEGMENTS-INTERSECT returns true, then it must be correct. Now we will show that it must return true if there is an intersection, even if it occurs as an intersection of 3 or more segments. Suppose that there is at least one intersection, and that p is the leftmost intersection point, breaking ties by choosing the point with the lowest y-coordinate. Let $a_1, a_2, \ldots, a_k$ be the segments that intersect at p. Since no intersections occur to the left of p, the order given by T is correct at all points to the left of p. Let z be the first sweep line at which some pair a_i, a_j is consecutive in T. Then z must occur at or to the left of p. Let i be the smallest number such that there exists a j such that a_i and a_j are consecutive in T, and assume that we choose the smallest j possible, and let q be the event point at which a_i and a_j become consecutive in the total preorder. If p is on z, then we must have $q = p$, and at this point the intersection is detected. As in the proof of correctness given in the section, the ordering of our endpoints allows us to detect this even if p is the left endpoint of a_i and the right endpoint of a_j. If p is not on z then q is to the left of p, and when we process q no other intersections have occurred, so the ordering in T is correct, and the algorithm correctly identifies the intersection between a_i and a_j.

Exercise 33.2-9

In the original statement of the problem, we are putting points with lower y-coordinates first. This means that when we are processing our vertical segment,

we want its lower bound to of already been processed by the time we process any of the left endpoints of other lines that may intersect the given line. Also, we don't want to remove the segment until we have already processed all the right endpoints of the lines that may of intersected it, which means we want it's upper bound to be dealt with in the second pass (the right endpoint pass). Again, since we process lower y-values first, this means that we have it added to our tree before we process anything it could intersect and have it removed after processing everything it could intersect.

If one or both of the segments are vertical at x in exercise 33.2-2, then testing whether they intersect is just a matter of looking to see if the other line is less than the upper bound and more than the lower bound at the given x value. otherwise we just see if it's more than the upper bound or less than the lower bound to see which direction the inequality should go.

Exercise 33.3-1

To see this, note that p_1 and p_m are the points with the lowest and highest polar angle with respect to p_0. By symmetry, we may just show it for p_1 and we would also have it for p_m just by reflecting the set of points across a vertical line. To a contradiction, suppose we have the convex hull doesn't contain p_1. Then, let p be the point in the convex hull that has the lowest polar angle with respect to p_0. If p is on the line from p_0 to p_1, we could replace it with p_1 and have a convex hull, meaning we didn't start with a convex hull. If we have that it is not on that line, then there is no way that the convex hull given contains p_1, also contradicting the fact that we had selected a convex hull.

Exercise 33.3-2

Let our n numbers be $a_1, a_2, \ldots, a_n$ and f be a strictly convex function, such as e^x. Let $p_i = (a_i, f(a_i))$. Compute the convex hull of $p_1, p_2, \ldots, p_n$. Then every point is in the convex hull. We can recover the numbers themselves by looking at the x-coordinates of the points in the order returned by the convex-hull algorithm, which will necessarily be a cyclic shift of the numbers in increasing order, so we can recover the proper order in linear time. In an algorithm such as GRAHAM-SCAN which starts with the point with minimum y-coordinate, the order returned actually gives the numbers in increasing order.

Exercise 33.3-3

Suppose that p and q are the two furthest apart points. Also, to a contradiction, suppose, without loss of generality that p is on the interior of the convex hull. Then, construct the circle whose center is q and which has p on the circle. Then, if we have that there are any vertices of the convex hull that are outside this circle, we could pick that vertex and q, they would have a higher distance than between p and q. So, we know that all of the vertices of the convex hull lie inside the circle. This means that the sides of the convex

hull consist of line segments that are contained within the circle. So, the only way that they could contain p, a point on the circle is if it was a vertex, but we supposed that p wasn't a vertex of the convex hull, giving us our contradiction.

Exercise 33.3-4

We simply run GRAHAM-SCAN but without sorting the points, so the runtime becomes $O(n)$. To prove this, we'll prove the following loop invariant: At the start of each iteration of the for loop of lines 7-10, stack S consists of, from bottom to top, exactly the vertices of $\mathrm{CH}(Q_{i-1})$. The proof is quite similar to the proof of correctness. The invariant holds the first time we execute line 7 for the same reasons outline in the section. At the start of the ith iteration, S contains $\mathrm{CH}(Q_{i-1})$. Let p_j be the top point on S after executing the while loop of lines 8-9, but before p_i is pushed, and let p_k be the point just below p_j on S. At this point, S contains $\mathrm{CH}(Q_j)$ in counterclockwise order from bottom to top. Thus, when we push p_i, S contains exactly the vertices of $\mathrm{CH}(Q_j \cup \{p_i\})$.

We now show that this is the same set of points as $\mathrm{CH}(Q_i)$. Let p_t be any point that was popped from S during iteration i and p_r be the point just below p_t on stack S at the time p_t was popped. Let p be a point in the kernel of P. Since the angle $\angle p_r p_t p_i$ makes a nonelft turn and P is star shaped, p_t must be in the interior or on the boundary of the triangle formed by p_r, p_i, and p. Thus, p_t is not in the convex hull of Q_i, so we have $\mathrm{CH}(Q_i - \{p_t\}) = \mathrm{CH}(Q_i)$. Applying this equality repeatedly for each point removed from S in the while loop of lines 8-9, we have $\mathrm{CH}(Q_j \cup \{p_i\}) = \mathrm{CH}(Q_i)$.

When the loop terminates, the loop invariant implies that S consists of exactly the vertices of $CH(Q_m)$ in counterclockwise order, proving correctness.

Exercise 33.3-5

Suppose that we have a convex hull computed from the previous stage $\{q_0, q_1, \ldots, q_m\}$, and we want to add a new vertex, p in and keep track of how we should change the convex hull. First, process the vertices in a clockwise manner, and look for the first time that we would have to make a non-left to get to p. This tells us where to start cutting vertices out of the convex hull. To find out the upper bound on the vertices that we need to cut out, turn around, start processing vertices in a clockwise manner and see the first time that we would need to make a non-right. Then, we just remove the vertices that are in this set of vertices and replace the with p. There is one last case to consider, which is when we end up passing ourselves when we do our clockwise sweep. Then we just remove no vertices and add p in in between the two vertices that we had found in the two sweeps. Since for each vertex we add we are only considering each point in the previous step's convex hull twice, the runtime is $O(nh) = O(n^2)$ where h is the number of points in the convex hull.

Exercise 33.3-6

Algorithm 2 ONLINE-CONVEX-HULL

---

let $P = \{p_0, p_1, \ldots p_m\}$ be the convex hull so far listed in counterclockwise order.
let p be the point we are adding
i=1
while going from p_{i-1} to p_i to p is a left turn and $i \neq 0$ **do**
    i++
end while
if i==0 **then**
    return P
end if
j=i
while going from p_{i+1} to p_i to p is a right turn and $j \geq i$ **do**
    j–
end while
if $j < i$ **then**
    insert p between p_j and p_i
else
    replace $p_i, \ldots p_j$ with p.
end if

First sort the points from left to right by x coordinate in $O(n \lg n)$, breaking ties by sorting from lowest to highest. At the ith step of the algorithm we'll compute $C_i = \text{CH}(\{p_1, \ldots, p_i\})$ using C_{i-1}, the convex hull computed in the previous step. In particular, we know that the rightmost of the first i points will be in C_i. The point which comes before p_i in a clockwise ordering will be the first point q of C_{i-1} such that $\overline{qp_i}$ does not intersect the interior of C_{i-1}. The point which comes after p_i will be the last point q' in a clockwise ordering of the vertices of C_{i-1} such that $\overline{p_iq'}$ does not intersect the interior of C_{i-1}. We can find each of these points in $O(\lg n)$ using a binary search. Here, assume that q and q' are given as the positions of the points in the clockwise ordering). This follows because we're searching a set of points which already forms a convex hull, so the segments $\overline{p_jp_i}$ will intersect the interior of C_{i-1} for the first k_1 points, not intersect for the next k_2 points, and intersect for the last k_3 points, where any of k_1, k_2, or k_3 could be 0. Once found, we can delete every point between q and q'. Since a point is deleted at most once and we store things in a red-black tree, the total runtime of all deletions is $O(n \lg n)$. Since we insert a total of n points, each taking $O(\lg n)$, the total runtime is thus $O(n \lg n)$. See the algorithm below:
Exercise 33.4-1

The flaw in his plan is pretty obvious, in particular, when we select line l, we may be unable perform an even split of the vertices. So, we don't neccesarily have that both the left set of points and right set of points have fallen to roughly half. For example, suppose that the points are all arranged on a vertical

---

Algorithm 3 INCREMENTAL-METHOD($p_1, p_2, \ldots, p_n$)

---
if $n \leq 3$ **then**
    return $(p_1, \ldots, p_n)$
end if
Use Merge Sort to sort the points by increasing x-coordinate, breaking ties by requiring increasing y-coordinate
Initialize an red-black tree C of size 3 with entries p_1, p_2, and p_3
for $i = 4$ to n **do**
    Let q be the result of binary searching for the first point of C_{i-1} such that $\overline{qp_i}$ doesn't intersect the interior of C_{i-1}
    Let q' be the result of binary searching for the last point of C_{i-1} such that $\overline{q'p_i}$ doesn't intersect the interior of C_{i-1}
        Delete $q + 1, q + 2, \ldots, q' - 1$ from C
        Insert p_i into C
end for

---

line, then, when we recurse on the the left set of points, we haven't reduced the problem size AT ALL, let alone by a factor of two. There is also the issue in this setup that you may end up asking about a set of size less than two when looking at the right set of points.

Exercise 33.4-2

Since we only care about the shortest distance, the distance δ' must be strictly less than δ. The picture in Figure 33.11(b) only illustrates the case of a nonstrict inequality. If we exclude the possibility of points whose x coordinate differs by exactly δ from l, then it is only possible to place at most 6 points in the $\delta \times 2\delta$ rectangle, so it suffices to check on the points in the 5 array positions following each point in the array Y'.

Exercise 33.4-3

In the analysis of the algorithm, most of it goes through just based on the triangle inequality. The only main point of difference is in looking at the number of points that can be fit into a $\delta \times 2\delta$ rectangle. In particular, we can cram in two more points than the eight shown into the rectangle by placing points at the centers of the two squares that the rectangle breaks into. This means that we need to consider points up to 9 away in Y' instead of 7 away. This has no impact on the asymptotics of the algorithm and it is the only correction to the algorithm that is needed if we switch from L_2 to L_1.

Exercise 33.4-4

We can simply run the divide and conquer algorithm described in the sec-

tion, modifying the brute force search for $|P| \leq 3$ and the check against the next 7 points in Y' to use the L_∞ distance. Since the L_∞ distance between two points is always less than the euclidean distance, there can be at most 8 points in the $\delta \times 2\delta$ rectangle which we need to examine in order to determine whether the closest pair is in that box. Thus, the modified algorithm is still correct and has the same runtime.

Exercise 33.4-5

We select the line l so that it is roughly equal, and then, we won't run into any issue if we just pick an arbitrary subset of the vertices that are on the line to go to one side or the other. Since the analysis of the algorithm allowed for both elements from P_L and P_R to be on the line, we still have correctness if we do this. To determine what values of Y belong to which of the set can be made easier if we select our set going to P_L to be the lowest however many points are needed, and the P_R to be the higher points. Then, just knowing the index of Y that we are looking at, we know whether that point belonged to P_L or to P_R.

Exercise 33.4-6

In addition to returning the distance of the closest pair, the modify the algorithm to also return the points passed to it, sorted by y-coordinate, as Y. To do this, merge Y_L and Y_R returned by each of its recursive calls. If we are at the base case, when $n \leq 3$, simply use insertion sort to sort the elements by y-coordinate directly. Since each merge takes linear time, this doesn't affect the recursive equation for the runtime.

Problem 33-1

a. We need just iteratively apply Jarvis march. The first march takes time $O(n|CH(Q_1)|)$, the next time $O(nCH(Q_2))$, and so on. So, since each point in Q appears in exactly one convex hull, as we take off successive layers, we have

$$\sum_i O(n|CH(Q_i)|) = O(n \sum_i |CH(Q_i)|) = O(n^2)$$

b. Suppose that the elements $r_1, r_2, r_3, \ldots r_\ell$ are the points that we are asked to sort. We will construct an instance of the convex layers problem, whose solution will tell us what the sorted order of $\{r_i\}$ is. Since we can't comparison sort quickly, and this would provide a solution of sorting based on a convex layers algorithm, it would mean that we cannot find a convex layers algorithm that takes time less than $\Omega(n \lg(n))$.

Suppose that all the $\{r_i\}$ are positive. If they aren't, we can in linear time find the one with the smallest value and subtract that value minus one from

each of them. We will select our 4ℓ points to be

$$P = \{(r_i, 0)\} \cup \{(0, \pm i)|i = 1, 2, \ldots \ell\} \cup \{(-i, 0)|i = 1, 2, \ldots \ell\}$$

Note that all of the points in this set are on the coordinate axes. So, every layer will contain one point that lies on each of the four half axes coming out of the origin. Looking at the points that lie on the positive x axis, they will correspond to the original points that we wanted to sort. Also, by looking at the outermost layer and going inwards, we are reading off the points $\{r_i\}$ in order of decreasing value. Since we have only increased the size of the problem by a constant factor, we haven't changed the asymptotics. In particular, if we had some magic algorithm for convex layers that was $o(n \lg(n))$, we would then have an algorithm that was $o(n \lg(n))$.

See also the solution to 33.3-2

Problem 33-2

a. Suppose that $y_i \leq y_{i+1}$ for some i. Let p_i be the point associated to y_i. In layer i, p_i is the leftmost point, so the x-coordinate of every other point in layer i is greater than the x-coordinate of p_i. Moreover, no other point in layer i can have y coordinate greater than p_i, since that would imply it dominates p_i. Let q_{i+1} be the point of layer $i + 1$ with y-coordinate y_{i+1}. If q_{i+1} is to the left of p_i, then q_{i+1} cannot be dominated by any point in L_i since every point in L_i is to the right of and below p_i. Moreover, if q_{i+1} is to the right of p_i then q_{i+1} dominates p_i, which can't happen. Thus, q_{i+1} cannot be weakly to the left or right of p_i, a contradiction. Thus $y_i > y_{i+1}$.

b. First suppose $j \leq k$. Then for $1 \leq i \leq j - 1$ we have that (x, y) is dominated by the point in layer i with y-coordinate y_i, so (x, y) is not in any of these layers. Since (x, y) is the leftmost point and $y_j < y$, and all other points in layer j have lower y coordinate, no point in layer j dominates (x, y). Moreover, since it is leftmost, no other point can be dominated by (x, y). Thus, (x, y) is in L_j, as well as all other points previously in L_j. The other layers are unaffected since we no longer consider (x, y) when computing them. Thus the layers of Q' are identical to the maximal layers of Q, except that $L_j = L_j \cup (x, y)$.

Now suppose $j = k + 1$. Then (x, y) is dominated by each point in layer i with y-coordinate y_i, so it can't be in any of the first k layers. This implies that it is in a layer of its own, $L_{k+1} = \{(x, y)\}$.

c. First sort the points by x coordinate, with the highest coordinate first. Process the points one at a time. For each point, find the layer in which it belongs as described in part b, creating a new layer if necessary. We can maintain lists of the layers in sorted order by y coordinate of the leftmost element of each list. In doing so, we can decide which list each new point belongs to in $O(\lg n)$ time. Since there are n points to process, the runtime

after sorting is $O(n \lg n)$. The initial sorting takes $O(n \lg n)$, so the total runtime is $O(n \lg n)$.

d. We'll have to modify our approach to deal with points having the same x- or y-coordinate. In particular, if two points have the same x-coordinate then when we go to place the second one, the old algorithm would have us put it in the same layer as the first one. We'll compensate for this as follows. Suppose we wish to add the point (x, y). Let j be the minimum index such that $y_j < y$. If the x-coordinate of the leftmost point of L_j is equal to x, then we need to create a new list L' which lives between L_{j-1} and L_j. Using red-black trees we can update the information in $O(\lg n)$ time. Otherwise, we add (x, y) to L_j as usual. If $j = k + 1$, then create a new layer L_{k+1}. Two points having the same y-coordinate doesn't actually cause any difficulty because of the strict inequality required for the check described in part b.

Problem 33-3

a. Take a convex hull of the set of all the ghostbusters and the ghosts. If the convex hull doesn't consist of either all ghosts or all busters, we can just pick an edge of the convex hull that joins a buster and a ghost, Since all of the other points lie on the same side of that line, the number of ghosts and busters will be n-1 and so will be equal.

So, assume that the convex hull does not contain one of both types. Since there is symmetry between ghosts and ghostbusters, suppose the convex hull is entirely made of ghostbusters. Pick an arbitrary ghostbuster on the convex hull, and that he's facing somewhere inside the convex hull. Have him/her initially pointing his proton pack just to the left the person furthest to his right and have him slowly start turning left. We know that initially there are more ghostbusters than ghosts to his right. We also know that by the time he is just to the right of the person furthest to his left there are more ghosts to his right than ghostbusters. This means at some point he must of gone from having more ghostbusters to his right to having more ghosts to his right. In order to have this happen he had to of just passed a ghost. So, he is then paired up with that ghost.

b. We just keep iterating the first part of this procedure, applying it separately to all the ghosts and ghostbusters to each of the sides of the line. We have that no beam will cross because the beams for each stays entirely on that side of the line. This gives us, for some $n \le k > 0$, the recurrence

$$T(n) = T(n - k) + T(k - 1) + n \lg(n)$$

This has the worst case when either k is really tiny or really close to n. Therefore, the worst case solution to this recurrence is $O(n^2 \lg(n))$.

Problem 33-4

13

a. Let a be given by endpoints (a_x, a_y, a_z) and (a'_x, a'_y, a'_z) and b be given by endpoints (b_x, b_y, b_z) and (b'_x, b'_y, b'_z). Compute, using cross products, whether or not segments $\overline{(a_x, a_y)(a'_x a'_y)}$ and $\overline{(b_x, b_y)(b'_x, b'_y)}$ intersect in constant time, as described earlier in the chapter. If they do, then either a or b is above the other one. If not, then they are unrelated. If they are related, we need to determine which of a and b are on top. In this case, there exist λ_1 and λ_2 such that

$$a_x + \lambda_1(a'_x - a_x) = b_x + \lambda_2(b'_x - b_x)$$

and

$$a_y + \lambda_1(a'_y - a_y) = b_y + \lambda_2(b'_y - b_y).$$

In other words, we get intersection when we project to the xy-plane. We can solve for λ_1 and λ_2. This requires division at first blush, but we shall see in a moment that this isn't necessary. In particular, a is above b if and only if $a_z + \lambda_1(a'_z - a_z) \geq b_z + \lambda_1(b'_z - b_z)$. By multiplying both sides by $(a'_x - a_x)(b'_y - b_y - (a'_y - a_y)(b'_x - b_x))$ we clear all denominators, so we need only perform addition, subtraction, multiplication, and comparison to determine whether a is on top. Moreover, we can do this in constant time.

b. Make a graph whose vertices are each of the n points. Find each pair of overlapping sticks. If a is above b, then draw a directed edge from a to b. Then perform a topological sort to determine an ordering of picking up the sticks. If such an ordering exists, then we use it. Otherwise there is no legal way to pick up the sticks. Since there could be as many as $O(n^2)$ instances of a point a being above a point b, there could be $\Theta(n^2)$ edges in the graph, so the runtime is $O(n^2)$.

Problem 33-5

a. Pick one point on one of the convex hulls, and look at the point on the other that has the lowest polar angle. Then, start marching counter clockwise around the first hull until it would require a non-right turn to go to the point selected before. Do the same thing, picking a point and looking at the point on the second polygon with highest polar angle, and keep marching in a clockwise direction until getting to the particular point would require a non-right. Cut out all the vertices between these two places we stopped inclusive. In their place put the vertices of the other convex polygon that are between to two selected vertices of it, inclusive.

b. Let P_1 be the first $\lceil n/2 \rceil$ points, and let P_2 be the second $\lfloor n/2 \rfloor$ points. Since the original set of points were selected independently from the sparse distribution, both the sets P_1 and P_2 were selected from a sparse distribution. This means that we have that $|CH(P_1)| \in O(n^{1-\epsilon})$ and also, $|CH(P_2)| \in O(n^{1-\epsilon})$. Then, by applying the procedure from part a, we have the recurrence $T(n) \leq 2T(n/2) + |CH(P_1)| + |CH(P_2)| = 2T(n/2) + O(n^{1-\epsilon})$.

By applying the master theorem, we see that this recurrence has solution $T(n) \in O(n)$.

Chapter 34

Michelle Bodnar, Andrew Lohr

April 12, 2016

Exercise 34.1-1

Showing that LONGEST-PATH-LENGTH being polynomial implies that LONGEST-PATH is polynomial is trivial, because we can just compute the length of the longest path and reject the instance of LONGEST-PATH if and only if k is larger than the number we computed as the length of the longest path.

Since we know that the number of edges in the longest path length is between 0 and $|E|$, we can perform a binary search for it's length. That is, we construct an instance of LONGEST-PATH with the given parameters along with $k = \frac{|E|}{2}$. If we hear yes, we know that the length of the longest path is somewhere above the halfway point. If we hear no, we know it is somewhere below. Since each time we are halving the possible range, we have that the procedure can require $O(\lg(|E|))$ many steps. However, running a polynomial time subroutine $\lg(n)$ many times still gets us a polynomial time procedure, since we know that with this procedure we will never be feeding output of one call of LONGEST-PATH into the next.

Exercise 34.1-2

The problem LONGST-SIMPLE-CYCLE is the relation that associates each instance of a graph with the longest simple cycle contained in that graph. The decision problem is, given k, to determine whether or not the instance graph has a simple cycle of length at least k. If yes, output 1. Otherwise output 0. The language corresponding to the decision problem is the set of all $\langle G, k \rangle$ such that $G = (V, E)$ is an undirected graph, $k \geq 0$ is an integer, and there exists a simple cycle in G consisting of at least k edges.

Exercise 34.1-3

A formal encoding of the adjacency matrix representation is to first encode an integer n in the usual binary encoding representing the number of vertices. Then, there will be n^2 bits following. The value of bit m will be 1 if there is an edge from vertex $\lfloor m/n \rfloor$ to vertex $(m\%n)$, and zero if there is not such an edge.

1

An encoding of the adjacency list representation is a bit more finessed. We'll be using a different encoding of integers, call it $g(n)$. In particular, we will place a 0 immediately after every bit in the usual representation. Since this only doubles the length of the encoding, it is still polynomially related. Also, the reason we will be using this encoding is because any sequence of integers encoded in this way cannot contain the string 11 and must contain at least one zero. Suppose that we have a vertex with edges going to the vertices indexed by $i_1, i_2, i_3, \ldots, i_k$. Then, the encoding corresponding to that vertex is $g(i_1)11g(i_2)11 \cdots 11g(i_k)1111$. Then, the encoding of the entire graph will be the concatenation of all the encodings of the vertices. As we are reading through, since we used this encoding of the indices of the vertices, we won't ever be confused about where each of the vertex indices end, or when we are moving on to the next vertex's list.

To go from the list to matrix representation, we can read off all the adjacent vertices, store them, sort them, and then output a row of the adjacency matrix. Since there is some small constant amount of space for the adjacency list representation for each vertex in the graph, the size of the encoding blows up by at most a factor of $O(n)$, which means that the size of the encoding overall is at most squared.

To go in the other direction, it is just a matter of keeping track of the positions in a given row that have ones, encoding those numerical values in the way described, and doing this for each row. Since we are only increasing the size of the encoding by a factor of at most $O(\lg(n))$(which happens in the dense graph case), we have that both of them are polynomially related.

Exercise 34.1-4

This isn't a polynomial-time algorithm. Recall that the algorithm from Exercise 16.2-2 had running time $\Theta(nW)$ where W was the maximum weight supported by the knapsack. Consider an encoding of the problem. There is a polynomial encoding of each item by giving the binary representation of its index, worth, and weight, represented as some binary string of length $a = \Omega(n)$. We then encode W, in polynomial time. This will have length $\Theta(\lg W) = b$. The solution to this problem of length $a+b$ is found in time $\Theta(nW) = \Theta(a * 2^b)$. Thus, the algorithm is actually exponential.

Exercise 34.1-5

We show the first half of this exercise by induction on the number of times that we call the polynomial time subroutine. If we only call it zero times, all we are doing is the polynomial amount of extra work, and therefore we have that the whole procedure only takes polynomial time.

Now, suppose we want to show that if we only make $n + 1$ calls to the polynomial time subroutine. Consider the execution of the program up until just before the last call. At this point, by the inductive hypothesis, we have only taken a polynomial amount of time. This means that all of the data that

we have constructed so far fits in a polynomial amount of space. This means that whatever argument we pass into the last polynomial time subroutine will have size bounded by some polynomial. The time that the last call takes is then the composition of two polynomials, and is therefore a polynomial itself. So, since the time before the last call was polynomial and the time of the last call was polynomial, the total time taken is polynomial in the input. This proves the claim of the first half of the input.

To see that it could take exponential time if we were to allow polynomially many calls to the subroutine, it suffices to provide a single example. In particular, let our polynomial time subroutine be the function that squares its input. Then our algorithm will take an integer x as input and then square it $\lg(x)$ many times. Since the size of the input is $\lg(x)$, this is only linearly many calls to the subroutine. However, the value of the end result will be $x^{2^{\lg(x)}} = x^x = 2^{x\lg(x)} = 2^{\lg(x)2^{\lg(x)}} \in \omega(2^{2^{\lg(x)}})$. So, the output of the function will require exponentially many bits to represent, and so the whole program could not of taken polynomial time.

Exercise 34.1-6

Let $L_1, L_2 \in P$. Then there exist algorithms A_1 and A_2 which decide L_1 and L_2 in polynomial time. We will use these to determine membership in the given languages. An input x is in $L_1 \cup L_2$ if and only if either A_1 or A_2 returns 1 when run on input x. We can check this by running each algorithm separately, each in polynomial time. To check if x is in $L_1 \cap L_2$, again run A_1 and A_2, and return 1 only if A_1 and A_2 each return 1. Now let $n = |x|$. For $i = 1$ to n, check if the first i bits of x are in L_1 and the last $n - i$ bits of x are in L_2. If this is ever true, then $x \in L_1 L_2$ so we return 1. Otherwise return 0. Each check is performed in time $O(n^k)$ for some k, so the total runtime is $O(n(n^k + n^k)) = O(n^{k+1})$ which is still polynomial. To check if $x \in \overline{L_1}$, run A_1 and return 1 if and only if A_1 returns 0. Finally, we need to determine if $x \in L_1^*$. To do this, for $i = 1$ to n, check if the first i bits of x are in L_1, and the last $n - i$ bits are in L_1^*. Let $T(n)$ denote the running time for input of size n, and let cn^k be an upper bound on the time to check if something of length n is in L_1. Then $T(n) \leq \sum_{i=1}^{n} cn^k T(n-i)$. Observe that $T(1) \leq c$ since a single bit is in L_1^* if and only if it is in L_1. Now suppose $T(m) \leq c'm^{k'}$. Then we have:

$$T(n) \leq cn^{k+1} + \sum_{i=0}^{n-1} c'i^{k'} \leq cn^{k+1} + c'n^{k'+1} = O(n^{\max k, k'}).$$

Thus, by induction, the runtime is polynomial for all n. Since we have exhibited polynomial time procedures to decide membership in each of the languages, they are all in P, so P is closed under union, intersection, concatenation, complement, and Kleene star.

Exercise 34.2-1

To verify the language, we should let the certificate be the mapping f from the vertices of G_1 to the vertices of G_2 that is an isomorphism between the graphs. Then all the verifier needs to do is verify that for all pairs of vertices u and v, they are adjacent in G_1 if and only if $f(u)$ is adjacent to $f(v)$ in G_2. Clearly it is possible to produce an isomorphism if and only if the graphs are isomorphic, as this is how we defined what it means for graphs to be isomorphic.

Exercise 34.2-2

Since G is bipartite we can write its vertex set as the disjoint union $S \sqcup T$, where neither S nor T is empty. Since G has an odd number of vertices, exactly one of S and T has an odd number of vertices. Without loss of generality, suppose it is S. Let $v_1, v_2, \ldots, v_n$ be a simple cycle in G, with $v_1 \in S$. Since n is odd, we have $v_1 \in S$, $v_2 \in T$, $\ldots$, $v_{n-1} \in T$, $v_n \in S$. There can be no edge between v_1 and v_n since they're both in S, so the cycle can't be Hamiltonian.

Exercise 34.2-3

Suppose that G is hamiltonian. This means that there is a hamiltonian cycle. Pick any one vertex v in the graph, and consider all the possibilities of deleting all but two of the edges passing through that vertex. For some pair of edges to save, the resulting graph must still be hamiltonian because the hamiltonian cycle that existed originally only used two edges. Since the degree of a vertex is bounded by the number of vertices minus one, we are only less than squaring that number by looking at all pairs ($\binom{n-1}{2} \in O(n^2)$). This means that we are only running the polynomial tester polynomially many independent times, so the runtime is polynomial. Once we have some pair of vertices where deleting all the others coming off of v still results in a hamiltonian graph, we will remember those as special, and ones that we will never again try to delete. We repeat the process with both of the vertices that are now adjacent to v, testing hamiltonicity of each way of picking a new vertex to save. We continue in this process until we are left with only $|V|$ edge, and so, we have just constructed a hamiltonian cycle.

Exercise 34.2-4

This is much like Exercise 34.1-6. Let L_1 and L_2 be languages in NP, with verification algorithms A_1 and A_2. Given input x and certificate y for a language in $L_1 \cup L_2$, we define A_3 to be the algorithm which returns 1 if either $A_1(x, y) = 1$ or $A_2(x, y) = 1$. This is a polynomial verification algorithm for $L_1 \cup L_2$, so NP is closed under unions. For intersection, define A_3 to return 1 if and only if $A_1(x, y)$ and $A_2(x, y)$ return 1. For concatenation, define A_3 to loop through $i = 1$ to n, checking each time if $A_1(x[1..i], y[1..i]) = 1$ and $A_1(x[i+1..n], y[i+1..n]) = 1$. If so, terminate and return 1. If the loop ends, return 0. This still takes polynomial time, so NP is closed under concatenation. Finally, we need to check Kleene star. Define A_3 to loop through $i = 1$ to n, each time checking if $A_1(x[1..i], y[1..i]) = 1$,

and $y[i + 1..n]$ is a certificate for $x[i + 1..n]$ being in L_1^*. Let $T(n)$ denote the running time for input of size n, and let cn^k be an upper bound on the time to verify that y is a certificate for x. Then $T(n) \leq \sum_{i=1}^{n} cn^k T(n-i)$. Observe that $T(1) \leq c$ since we can verify a certificate for a problem of length 1 in constant time. Now suppose $T(m) \leq c'm^{k'}$. Then we have:

$$T(n) \leq cn^{k+1} + \sum_{i=0}^{n-1} c'i^{k'} \leq cn^{k+1} + c'n^{k'+1} = O(n^{\max k, k'}).$$

Thus, by induction, the runtime of A_3 is polynomial. Note that we only needed to deal with the runtime recursion with respect to the length of x, since it is assumed $|y| = O(|x|^c)$ for some constant c. Therefore NP is closed under Kleene star.

A proof for closure under complement breaks down, however. If a certificate y is given for input x and $A_1(x, y)$ returns false, this doesn't tell us that y is a certificate for x being in the complement of L_1. It merely tells us that y didn't prove $x \in L_1$.

Exercise 34.2-5

Suppose that we know that the length of the certificates to the verifier are bounded by n^{k-1}, we know it has to be bounded by some polynomial because the verifier can only look at polynomially many bits because it runs in polynomial time. Then, we try to run the verifier on every possible assignment to each bit of the certificates of length up to that much. Then, the runtime of this will be a polynomial times $2^{n^{k-1}}$ which is little oh of 2^{n^k}.

Exercise 34.2-6

The certificate in this case would be a list of vertices $v_1, v_2, \ldots, v_n$, starting with u and ending with v, such that each vertex of G is listed exactly once and $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq n - 1$. Since we can check this in polynomial time, HAM-PATH belongs to NP.

Exercise 34.2-7

For a directed acyclic graph, we can compute the topological sort of the graph by the method of section 22.4. Then, looking at this sorting of the vertices, we say that there is a Hamiltonian path if as we read off the vertices, each is adjacent to the next, and they are not if there is any pair of vertices so that one is not adjacent to the next.

If we are in the case that each is adjacent to the next, then the topological sort itself gives us the Hamiltonian path. However, if there is any pair of vertices so that one is not adjacent to the next, this means that that pair of vertices do not have any paths going from one to the other. This would clearly imply that

there was no Hamiltonian path, because the Hamiltonian path would be going from one of them to the other.

To see the claim that a pair of vertices u and v that are adjacent in a topological sort but are not adjacent have no paths going from one to the other, suppose to a contradiction there were such a path from u to v. If there were any vertices along this path they would have to be after u since they are descendants of u and they would have to be before v because they are ancestors of v. This would contradict the fact that we said u and v were adjacent in a topological sort. Then the path would have to be a single edge from u to v, but we said that they weren't adjacent, and so, we have that there is no such path.

Exercise 34.2-8

Let L' be the complement of TAUTOLOGY. Then L' consists of all boolean formulas ϕ such that there exists an assignment $y_1, y_2, \ldots, y_k$ of 0 and 1 to the input variables which causes ϕ to evaluate to 0. A certificate would be such an assignment. Since we can check what an assignment evaluates to in polynomial time in the length of the input, we can verify a certificate in polynomial time. Thus, $L' \in$ NP which implies TAUTOLOGY $\in$ co-NP.

Exercise 34.2-9

A language is in $coNP$ if there is a procedure that can verify that an input is not in the language in polynomial time given some certificate. Suppose that for that language we a procedure that could compute whether an input was in the language in polynomial time receiving no certificate. This is exactly what the case is if we have that the language is in P. Then, we can pick our procedure to verify that an element is not in the set to be running the polynomial time procedure and just looking at the result of that, disregarding the certificate that is given. This then shows that any language that is in P is in $coNP$, giving us the inclusion that we wanted.

Exercise 34.2-10

Suppose NP $\neq$ co-NP. Let $L \in$ NP $\setminus$ co-NP. Since $P \subset$ NP $\cap$ co-NP, and $L \notin$ NP $\cap$ co-NP, we have $L \notin P$. Thus, P $\neq$ NP.

Exercise 34.2-11

As the hint suggests, we will perform a proof by induction. For the base case, we will have 3 vertices, and then, by enumeration, we can see that the only Hamiltonian graph on three vertices is K_3. For any connected graph on three vertices, the longest the path connecting them can be is 2 edges, and so we will have $G^3 = K_3$, meaning that the graph G was Hamiltonian.

Now, suppose that we want to show that the graph G on $n + 1$ vertices has the property that G^3 is Hamiltonian. Since the graph G is connected we know

that there is some spanning tree by Chapter 23. Then, let v be any internal vertex of that tree. Suppose that if we were to remove the vertex v, we would be splitting up the original graph in the connected components $V_1, V_2, \ldots, V_k$, sorted in increasing order of size. Suppose that the first ℓ_1 of these components have a single vertex. Suppose that the first ℓ_2 of these components have fewer than 3 vertices. Then, let v_i be the vertex of V_i that is adjacent to v in the tree. For $i > \ell_1$, let x_i be any vertex of V_i that is distance two from the vertex v. By induction, we have Hamiltonian cycles for each of the components $V_{\ell_2+1}, \ldots, V_k$. In particular, there is a Hamiltonian path from v_i to x_i. Then, for each i and j there is an edge from x_j to v_i, because there is a path of length three between them passing through v. This means that we can string together the Hamiltonian paths from each of the components with $i > \ell_1$. Lastly, since $V_1, \ldots, V_\ell$ all consist of single vertices that are only distance one from v, they are all adjacent in G^3. So, after stringing together the Hamiltonian paths for $i > \ell_1$, we just visit all of the single vertices in $v_1, v_2, \ldots, v_{\ell_1}$ in order, then, go to v and then to the vertex that we started this path at, since it was selected to be adjacent to v, this is possible. Since we have constructed a Hamiltonian cycle, we have completed the proof.

Exercise 34.3-1

The formula in figure 34.8b is

$$((x_1 \vee x_2) \wedge (\neg(\neg x_3))) \wedge (\neg(\neg x_3) \vee ((x_1) \wedge (\neg x_3) \wedge (x_2))) \wedge ((x_1) \wedge (\neg x_3) \wedge (x_2))$$

We can cancel out the double negation to get that this is the same expression as

$$((x_1 \vee x_2) \wedge (x_3)) \wedge ((x_3) \vee ((x_1) \wedge (\neg x_3) \wedge (x_2))) \wedge ((x_1) \wedge (\neg x_3) \wedge (x_2))$$

Then, the first clause can only be true if x_3 is true. But the last clause can only be true if $\neg x_3$ is true. This would be a contradiction, so we cannot have both the first and last clauses be true, and so the boolean circuit is not satisfiable since we would be taking the and of these two quantities which cannot both be true.

Exercise 34.3-2

Suppose $L_1 \leq_P L_2$ and let f_1 be the polynomial time reduction function such that $x \in L_1$ if and only if $f_1(x) \in L_2$. Similarly, suppose $L_2 \leq_P L_3$ and let f_2 be the polynomial time reduction function such that $x \in L_2$ if and only if $f_2(x) \in L_3$. Then we can compute $f_2 \circ f_1$ in polynomial time, and $x \in L_1$ if and only if $f_2(f_1(x)) \in L_3$. Therefore $L_1 \leq_P L_3$, so the $\leq_P$ relation is transitive.

Exercise 34.3-3

Suppose first that we had some polynomial time reduction from L to $\bar{L}$. This means that for every x there is some $f(x)$ so that $x \in L$ iff $f(x) \in \bar{L}$. This means that $x \in \bar{L}$ iff $x \notin L$ iff $f(x) \notin \bar{L}$ iff $f(x) \in L$. So, our poly-time computable function for the reduction is the same one that we had from $L \leq_P \bar{L}$. We can do an identical thing for the other direction.

Exercise 34.3-4

We could have instead used as a certificate a satisfying assignment to the input variables in the proof of Lemma 34.5. We construct the two-input, polynomial time algorithm A to verify CIRCUIT-SAT as follows. The first input is a standard encoding of a boolean combinatiorial circuit C, and the second is a satisfying assignment of the input variables. We need to compute the output of each logic gate until the final one, and then check whether or not the output of the final gate is 1. This is more complicated than the approach taken in the text, because we can only evaluate the output of a logic gate once we have successfully determined all input values, so the order in which we examine the gates matters. However, this can still be computed in polynomial time by essentially performing a breath-first search on the circuit. Each time we reach a gate via a wire we check whether or not all of its inputs have been computed. If yes, evaluate that gate. Otherwise, continue the search to find other gates, all of whose inputs have been computed.

Exercise 34.3-5
We do not have any loss of generality by this assumption. This is because since we bounded the amount of time that the program has to run to be polynomial, there is no way that the program can access more than an polynomial amount of space. That is, there is no way of moving the head of the turning machine further than polynomially far in only polynomial time because it can move only a single cell at a time.

Exercise 34.3-6

Suppose that $\emptyset$ is complete for P. Let $L = \{0, 1\}^*$. Then L is clearly in P, and there exists a polynomial time reduction function f such that $x \in \emptyset$ if and only if $f(x) \in L$. However, it's never true that $x \in \emptyset$, so this means it's never true that $f(x) \in L$, a contradiction since every input is in L. Now suppose $\{0, 1\}^*$ is complete for P, let $L' = \emptyset$. Then L' is in P and there exists a polynomial time reduction function f'. Then $x \in \{0, 1\}^*$ if and only if $f'(x) \in L'$. However x is always in $\{0, 1\}^*$, so this implies $f'(x) \in L'$ is always true, a contradiction because no binary input is in L'.

Finally, let L be some language in P which is not $\emptyset$ or $\{0, 1\}^*$, and let L' be any other language in P. Let $y_1 \notin L'$ and $y_2 \in L'$. Since $L \in P$, there

exists a polynomial time algorithm A which returns 0 if $x \notin L$ and 1 if $x \in L$. Define $f(x) = y_1$ if $A(x)$ returns 0 and $f(x) = y_2$ if $A(x)$ returns 1. Then f is computable in polynomial time and $x \in L$ if and only if $f(x) \in L'$. Thus, $L' \leq_P L$.

Exercise 34.3-7

Since L is in NP, we have that $\bar{L} \in coNP$ because we could just run our verification algorithm to verify that a given x is not in the complement of L, this is the same as verifying that x is in L. Since every coNP language has its complement in NP, suppose that we let S be any language in $coNP$ and let $\bar{S}$ be its compliment. Suppose that we have some polynomial time reduction f from $\bar{S} \in NP$ to L. Then, consider using the same reduction function. We will have that $x \in S$ iff $x \notin \bar{S}$ iff $f(x) \notin L$ iff $f(x) \in \bar{L}$. This shows that this choice of reduction function does work. So, we have shown that the compliment of any NP complete problem is also NP complete. To see the other direction, we just negate everything, and the proof goes through identically.

Exercise 34.3-8

To prove that a language L is NP-hard, one need not actually construct the polynomial time reduction algorithm F to compute the reduction function f for every $L' \in NP$. Rather, it is only necessary to prove that such an algorithm exists. Thus, it doesn't matter that F doesn't know A. If L' is in NP, we know that A exists. If A exists, dependent on k and that big-oh constant, we know that F exists and runs in polynomial time, and this is sufficient to prove CIRCUIT-SAT is NP-hard.

Exercise 34.4-1

Suppose that it is a circuit on two inputs, and then, we have n rounds of two and gates each, both of which take both of the two wires from the two gates from the previous round. Since the formulas for each round will consist of two copies of the formulas from the previous round, it will have an exponential size formula.

Exercise 34.4-2

To make this more readable, we'll just find the 3-CNF formula for each term listed in the AND of clauses for ϕ' on page 1083, including the auxiliary variables p and q as necessary.

$$y = (y \lor p \lor q) \land (y \lor p \lor \neg q) \land (y \lor \neg p \lor q) \land (y \lor \neg p \lor \neg q)$$
$$(y_1 \leftrightarrow (y_2 \land \neg x_2)) = (\neg y_1 \lor \neg y_2 \lor \neg x_2) \land (\neg y_1 \lor y_2 \lor \neg x_2) \land (\neg y_1 \lor y_2 \lor x_2) \land (y_1 \lor \neg y_2 \lor x_2)$$
$$(y_2 \leftrightarrow (y_3 \lor y_4)) = (\neg y_2 \lor y_3 \lor y_4) \land (y_2 \lor \neg y_3 \lor \neg y_4) \land (y_2 \lor \neg y_3 \lor y_4) \land (y_2 \lor y_3 \lor \neg y_4)$$
$$(y_3 \leftrightarrow (x_1 \rightarrow x_2)) = (\neg y_3 \lor \neg x_2 \lor x_2) \land (y_3 \lor \neg x_1 \lor \neg x_2) \land (y_1 \lor x_1 \lor \neg x_2) \land (y_3 \lor x_1 \lor x_2)$$
$$(y_4 \leftrightarrow \neg y_5) = (\neg x_4 \lor \neg y_5 \lor q) \land (\neg x_4 \lor \neg y_5 \lor \neg p) \land (x_4 \lor y_5 \lor p) \land (x_4 \lor y_5 \lor \neg p)$$
$$(y_5 \leftrightarrow (y_6 \lor x_4)) = (\neg y_5 \lor y_6 \lor x_4) \land (y_5 \lor \neg y_6 \lor \neg x_4) \land (y_5 \lor \neg y_6 \lor x_4) \land (y_5 \lor y_6 \lor \neg x_4)$$
$$(y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3)) = (\neg y_6 \lor \neg x_1 \lor \neg x_3) \land (\neg y_6 \lor x_1 \lor x_3) \land (y_6 \lor \neg x_1 \lor x_3) \land (y_6 \lor x_1 \lor \neg x_3).$$

Exercise 34.4-3

The formula could have $\Omega(n)$ free variables, then, the truth table corresponding to this formula would a number of rows that is $\Omega(2^n)$ since it needs to consider every possible assignment to all the variables. This then means that the reduction as described is going to incrase the size of the problem exponentially.

Exercise 34.4-4

To show that the language $L = $ TAUTOLOGY is complete for co-NP, it will suffice to show that $\overline{L}$ is NP-complete, where $\overline{L}$ is the set of all boolean formulas for which there exists an assignment of input variables which makes it false. We showed in Exercise 34.2-8 that TAUTOLOGY $\in$ co-NP, which implies $\overline{L} \in$ NP. Thus, we need only show that $\overline{L}$ is NP-hard. We'll give a polynomial time reduction from the problem of determining satisfiability of boolean formulas to determining whether or not a boolean formula fails to be a tautology. In particular, given a boolean formula ϕ, the negation of ϕ has a satisfying assignment if and only if ϕ has an assignment which causes it to evaluate to 0. Thus, our function will simply negate the input formula. Since this can be done in polynomial time in the length of the input, boolean satisfiability is polynomial time reducible to $\overline{L}$. Therefore $\overline{L}$ is NP-complete. By Exercise 34.3-7, this implies TAUTOLOGY is complete for co-NP.

Exercise 34.4-5

Since the problem is in disjunctive normal form, we can write it as $\lor_i \phi_i$ where each ϕ_i looks like the and of a bunch of variables and their negations. Then, we know that the formula is satisfiable if and only if any one if the ϕ_i are satisfiable. If a ϕ_i contains both a variable and its negation, then it is clearly not satisfiable, as one of the two must be false. However, if each variable showing up doesn't have its negation showing up, then we can just pick the appropriate value to assign to each variable. This is a property that can be checked in linear time, by just keeping two bit vectors of length equal to the number of variables, one representing if the variable has shown up negated and one for if the variable

has shown up without having been negated.

Exercise 34.4-6

Let A denote the polynomial time algorithm which returns 1 if input x is a satisfiable formula, and 0 otherwise. We'll define an algorithm A' to give a satisfying assignment. Let $x_1, x_2, \ldots, x_m$ be the input variables. In polynomial time, A' computes the boolean formula x' which results from replacing x_1 with true. Then, A' runs A on this formula. If it is satisfiable, then we have reduced the problem to finding a satisfying assignment for x' with input variables $x_2, \ldots, x_m$, so A' recursively calls itself. If x' is not satisfiable, then we set x_1 to false, and need to find a satisfying assignment for the formula x' obtained by replacing x_1 in x by false. Again, A' recursively calls itself to do this. If $m = 1$, A' takes a polynomial-time brute force approach by evaluating the formula when x_m is true, and when x_m is false. Let $T(n)$ denote the runtime of A' on a formula whose encoding is of length n. Note that we must have $m \leq n$. Then we have $T(n) = O(n^k) + T(n')$ for some k and $n' = |x'|$, and $T(1) = O(1)$. Since we make a recursive call once for each input variable there are m recursive calls, and the input size strictly decreases each time, so the overall runtime is still polynomial.

Exercise 34.4-7

Suppose that the original formula was $\wedge_i(x_i \vee y_i)$, and the set of variables were $\{a_i\}$. Then, consider the directed graph which has a vertex corresponding both to each variable, and each negation of a variable. Then, for each of the clauses $x \vee y$, we will place an edge going from $\neg x$ to y, and an edge from $\neg y$ to x. Then, anytime that there is an edge in the directed graph, that means if the vertex the edge is coming from is true, the vertex the edge is going to has to be true. Then, what we would need to see in order to say that the formula is satisfiable is a path from a vertex to the negation of that vertex, or vice versa. The naive way of doing this would be to run all pairs shortest path, and see if there is a path from a vertex to its negation. This however takes time $O(n^2 \lg(n))$, and we were charged with making the algorithm as efficient as possible. First, run the procedure for detecting strongly connected components, which takes linear time. For every pair of variable and negation, make sure that they are not in the same strongly connected component. Since our construction was symmetric with respect to taking negations, if there were a path from a variable to its negation, there would be a path going from its negation to itself as well. This means that we would detect any path from a variable to its negation, just by checking to see if they are contained in the same connected component or not.

Exercise 34.5-1

To do this, first, notice that it is in NP, where the certificate is just the injection from G_1 into G_2 so that G_1 is isomorphic to its image.

Now, to see that it is NP complete, we will do a reduction to clique. That

is, to detect if a graph has a clique of size k, just let G_1 be a complete graph on k vertices and let G_2 be the original graph. If we could solve the subgraph isomorphism problem quickly, this would allow us to solve the clique problem quickly.

Exercise 34.5-2

A certificate would be the n-vector x, and we can verify in polynomial time that $Ax \leq b$, so 0-1 integer linear programming (01LP) is in NP. To prove that 01LP is NP-hard, we show that 3-CNF-SAT $\leq_P$ 01LP. Let ϕ be 3-CNF formula with n input variables and k clauses. We construct an instance of 01LP as follows. Let A be a $k + 2n$ by $2n$ matrix. For $1 \leq i \leq k$, set entry $A(i, j)$ to -1 if $1 \leq j \leq n$ and clause C_i contains the literal x_j. Otherwise set it to 0. For $n + 1 \leq j \leq 2n$, set entry $A(i, j)$ to -1 if clause C_i contains the literal $\neg x_{j-n}$, and 0 otherwise. When $k + 1 \leq i \leq k + n$, set $A(i, j) = 1$ if $i - k = j$ or $i - k = j - n$, and 0 otherwise. When $k + n + 1 \leq i \leq k + 2n$, set $A(i, j) = -1$ if $i - k - n = j$ or $i - k - n = j - n$, and 0 otherwise. Let b be a $k + 2n$-vector. Set the first k entries to -1, the next n entries to 1, and the last n entries to -1. It is clear that we can construct A and b in polynomial time.

We now show that ϕ has a satisfying assignment if and only if there exists a 0-1 vector x such that $Ax \leq b$. First, suppose ϕ has a satisfying assignment. For $1 \leq i \leq n$, f x_i is true, make $x[i] = 1$ and $x[n + i] = 0$. If x_i is false, set $x[i] = 0$ and $x[n + i] = 1$. Since clause C_i is satisfied, there must exist some literal in it which makes it true. If it is x_j, then $x[j] = 1$ and $A(i, j) = -1$, so we get a contribution of -1 to the ith row of b. Since every entry in the upper k by $2n$ submatrix of A is nonpositive and every entry of x is nonnegative, there can be no positive contributions to the ith row of b. Thus, we are guaranteed that the ith row of Ax is at most -1. The same argument applies if the literal $\neg x_j$ makes clause i true. For $1 \leq m \leq n$, at most one of x_m and $\neg x_m$ can be true, so at most one of $x[m]$ and $x[m + n]$ can be true. When we multiply row $k + m$ by x, we get the number of 1's among $x[m]$ and $x[m + n]$. Thus, the $(k + m)^{th}$ row of b is at most 1, as required. Finally, when we multiply row $k + n + m$ of A by x, we get negative 1 times the number of 1's among $x[m]$ and $x[m + n]$. Since this is at least 1, the $(k + n + m)^{th}$ row of b is at most -1. Therefore all inequalities are satisfied, so x is a 0-1 solution to $Ax = b$.

Next we must show that any 0-1 solution to $Ax = b$ provides a satisfying assignment. Let x be such a 0-1 solution. The inequalities ensured by the last $2n$ rows of b guarantee that exactly one of $x[m]$ and $x[n + m]$ is set equal to 1 for $1 \leq m \leq n$. In particular, this means that each x_i is either true or false, but not both. Let this be our candidate satisfying assignment. By the construction of A, we get a contribution of -1 to row i of Ax every time a literal in C_i is true, based on our candidate assignment, and a 0 contribution every time a literal is false. Since row i of b is -1, this guarantees at least one literal which is true per our assignment in each clause. Since this holds for each of the k clauses, the candidate assignment is in fact a satisfying assignment. Therefore 01LP is NP-complete.

12

Exercise 34.5-3

We will try to show a reduction to the 0-1 integer programming problem. To see this, we will take our A from the 0-1 integer programming problem, and tack on a copy of the $n \times n$ identity matrix to its bottom, and tack on n ones to the end of b from teh 0-1 integer programming problem. This has the effect of adding the restrictions that every entry of x must be at most 1. However, since, for every i, we needed x_i to be an integer anyways, this only leaves the option that $x_i = 0$ or $x_i = 1$. This means that by adding these restrictions, we have that any solution to this system will be a solution to the 0-1 integer programming problem given by A and b.

Exercise 34.5-4

We can solve the problem using dynamic programming. Suppose there are n integers in S. Create a t by n table to solve the problem just as in the solution to the 0-1 knapsack problem described in Exercise 16.2-2. This has runtime $O(tn \lg t)$, since without loss of generality we may assume that every integer in S is less than or equal to t, otherwise we know it won't be included in the solution, and we can check for this in polynomial time. The extra $\lg t$ term comes from the fact that each addition takes $O(\lg t)$ time. Moreover, we can assume that S contains at most t^2 integers. If t is expressed in unary then the length of the problem is at most $O(t + t^2 \lg t) = O(t^3)$, since we express the integers in S in binary. The time to solve it is $O(t^4)$. Thus, the time to compute the solution is polynomial in the length of the input.

Exercise 34.5-5

We will be performing a reduction from the subset sum problem. Suppose that S and t are our set and target from our subset sum problem. Let x be equal to $\sum_{s \in S} s$. Then, we will add the elements $x + t, 2x - t$. Once we have added the elements, note that the sum of all of the elements in the new set S' will be $4x$. We also know that we cannot have both of the new elements that we added be on same side of the partition, because they add up to $3x$ which is three times all the other elements combined. Now, this set of elements will be what we pass into our set partition solver. Note that since the total is $4x$, each side will add up to $2x$. This means that if we look at the elements that on the same side as, but not equal to $2x - t$, they must add up to t. Since they were also members of the original set S, this means that they are a subset with the desired sum, solving the original instance of subset sum. Since it was proved in the section that subset sum is NP-complete, this proves that the set-partition problem is NP hard.

To see that it is in NP, just let the certificate be the set of elements of S that forms one side of the partition. It is linear time to add them up and make sure that they are exactly half the sum of all the elements in S.

Exercise 34.5-6

We'll show that the hamiltonian-path problem HAM-PATH is NP-complete. First, a certificate would be a list $\{v_1, v_2, \ldots, v_n\}$ of the vertices of the path, in order. We can check in polynomial time whether or not $\{v_i, v_{i+1}\}$ is an edge for $1 \leq i \leq n-1$. Thus, HAM-PATH is in NP.

Next, we'll show that HAM-PATH is NP-complete by showing that HAM-CYCLE $\leq_P$ HAM-PATH. Let $G = (V, E)$ be any graph. We'll construct a graph G' as follows. For each edge $e_i \in E$, let G_{e_i} denote the graph with vertex set V and edge set $E - e_i$. Let e_i have the form $\{u_i, v_i\}$. Now, G' will contain one copy of G_{e_i} for each $e_i \in E$. Additionally, G' will contain a vertex x connected to u_1, an edge from v_i to u_{i+1} for $1 \leq i \leq |E|-1$, and a vertex y and edge from $v_{|E|}$ to y. It is clear that we can construct G' from G in time polyomial in the size of G.

If G has a Hamiltonian cycle, then G_{e_i} has a Hamiltonian path starting at u_i and ending at v_i for each i. Thus, G' has a Hamiltonian cycle from x to y, obtained by taking each of these paths one after another. On the other hand, suppose G fails to have a Hamiltonian cycle. Since x and y have degree 1, the only way G' can have a Hamiltonian path is if it starts at x and ends at y. Moreover, since $\{v_1, u_2\}$ is a cut edge, it must be in the Hamiltonian path if it exists. Since we can not traverse this edge a second time, any Hamiltonian path must start with a Hamiltonian path from x to v_1. However, this means there is a Hamiltonian path from u_1 to v_1. Since $\{u_1, v_1\}$ is an edge in G, this implies there is a Hamiltonian cycle in G, a contradiction. Thus, G has a Hamiltonian cycle if and only if G' has a Hamiltonian path. Therefore HAM-PATH is NP-hard, so HAM-PATH is in fact NP-complete.

Exercise 34.5-7

The related decision problem is to, given a graph G and integer k decide if there is a simple cycle of length at least k in the graph G. To see that this problem is in NP, just let the certificate be the cycle itself. It is really easy just to walk along this cycle, keeping track of what vertices you've already seen, and making sure they don't get repeated.

To see that it is NP-hard, we will be doing a reduction to Hamilton cycle. Suppose we have a graph G and want to know if it is Hamilton. We then create an instance of the decision problem asking if the graph has a simply cycle of length at least $|V|$ vertices. If it does then there is a Hamiltonian cycle. If there is not, then there cannot be any Hamiltonian cycle.

Exercise 34.5-8

A certificate would be an assignment to input variables which causes exactly half the clauses to evaluate to 1, and the other half to evaluate to 0. Since we can check this in polynomial time, half 3-CNF is in NP. To prove that it's

NP-hard, we'll show that 3-CNF-SAT $\leq_p$ HALF-3-CNF. Let ϕ be any 3-CNF formula with m clauses and input variables $x_1, x_2, \ldots, x_n$. Let T be the formula $(y \vee y \vee \neg y)$, and let F be the formula $(y \vee y \vee y)$. Let $\phi' = \phi \wedge T \wedge \ldots \wedge T \wedge F \wedge \ldots \wedge F$ where there are m copies of T and $2m$ copies of F. Then ϕ' has $4m$ clauses and can be constructed from ϕ in polynomial time. Suppose that ϕ has a satisfying assignment. Then by setting $y = 0$ and the x_i's to the satisfying assignment, we satisfy the m clauses of ϕ and the m T clauses, but none of the F clauses. Thus, ϕ' has an assignment which satisfies exactly half of its clauses. On the other hand, suppose there is no satisfying assignment to ϕ. The m T clauses are always satisfied. If we set $y = 0$ then the total number of clauses satisfies in ϕ' is strictly less than $2m$, since each of the $2m$ F clauses is false, and at least one of the ϕ clauses is false. If we set $y = 1$, then strictly more than half the clauses of ϕ' are satisfied, since the $3m$ T and F clauses are all satisfied. Thus, ϕ has a satisfying assignment if and only if ϕ' has an assignment which satisfies exactly half of its clauses. We conclude that HALF-3-CNF is NP-hard, and hence NP-complete.

Problem 34-1

a) The related decision problem should be to, given a graph and a number k decide whether or not there is some independent set of size at least k. If we take the compliment of the given graph, then it will have a clique of size at least k if and only if the original graph has an independent set of size at least k. This is because if we take any set of vertices in the original graph, then it will be an independent set if and only if there are no edges between those vertices. However, in the compliment graph, this means that between every one of those vertices, there is an edge, which means they form a clique. So, to decide independent set, just decide clique in the compliment.

b) We know that since all independent sets are subsets of the set of vertices, then the size of the largest independent set will be an integer in the range $1..|V|$. Then, we will perform a binary search on this space of valid sizes of the largest independent set. That is, we pick the middle element, ask if there is an independent set of that size, if there is, we know we are in the upper half of this range of values for the size of the largest independent set, if not, then we are in the lower half. The total runtime of this procedure to find the size of the largest independent set will only be a factor of $\lg(|V||)$ higher than the solution to the decision problem. Call the size of the largest independent set k.

Now, for every pair of vertices, try adding an edge, and check if the procedure from before determines that the size of the largest independent set has decreased. If it hasn't that means that that pair of vertices doesn't prevent us from attaining an independent set of the given size. That is, we aren't in the case that there is only one maximal set of the given size and that pair of vertices belongs to it. So, add that edge to the graph, and continue in this

fashion for every pair of vertices. Once we are done, the size of the largest independent set will be the same, and we will have that every edge is filled in except for those going between an independent set of the given size. So, we just list off all the vertices whose degree is less than $|V| - 1$ as being members of our independent set.

c) Since every vertex has degree 2, and so self edges are allowed, the graph must look like a number of disjoint cycles. We can then consider the independent set problem separately for each of the cycles. If we have an even cycle, the largest independent set possible is half the vertices, by selecting them to be alternating. If it is an odd cycle, then we can do half rounded down, since when we get back to the start, we are in the awkward place where there are two unselected vertices between two selected vertices. It's easy to see that these are tight, because there is so little freedom in selecting an independent set in a cycle. So, to calculate the size of the smallest independent set, look at the sizes of each cycle c_i, then, the largest independent set will have size $\lfloor \frac{c_i}{2} \rfloor$.

d) First, find a maximal matching. This can be done in time $O(VE)$ by the methods of section 26.3. let $f(x)$ be defined for all vertices that were matched, and let it evaluate to the point that is paired with x in the maximal matching. Then, we do the following procedure. Let S_1 be the unpaired points, Let $S_2 = f(N(S_1)) \setminus S_1$, where we extend f to sets of vertices by just letting it be the set containing all the pairs of the given points. Similarly, define $S_{i+1} = f(N(S_i)) \setminus \left(\cup_{j=1}^{i} S_j \right)$. First, we need to show that this is well defined. That means that we want to make sure that we always have that every neighbor of S_i is paired with something. Since we could get from an unpaired point to something in S_i by taking a path that is alternating from being an edge in the matching and an edge not in the matching, starting with one that was not, if we could get to an unpaired point from S_i, that last edge could be tacked onto this path, and it would become an augmenting path, contradicting maximality of the original matching. Next, we can note that we never have an element in some S_i adjacent to an element in some S_j. Suppose there were, then we could take the path from an unpaired vertex to a vertex in S_i, add the edge to the element in S_j and then take the path from there to an unpaired vertex. This forms an augmenting path, which would again contradict maximality. The process of computing the $\{S_i\}$ must eventually terminate by becoming $\emptyset$ because they are selected to be disjoint and there are only finitely many vertices. Any vertices that are neither in an S_i or adjacent to one consist entirely of a perfect matching, that has no edges going to picked vertices. This means that the best we can do is to just pick everything from one side of the remaining vertices. This whole procedure of picking vertices takes time at most $O(E)$, since we consider going along each edge only twice. This brings the total runtime to $O(VE)$.

Thanks to John Chiarelli, a fellow graduate student at Rutgers, for helpful discussion of this part of the problem.

Problem 34-2

a. We can solve this problem in polynomial time as follows. Let a denote the number of coins of denomination x and b denote the number of coins of denomination y. Then we must have $a + b = n$. In order to divide the money exactly evenly, we need to know if there is a way to make $(ax + by)/2$ out of the coins. In other words, we need to determine whether there exist nonnegative integers c and d less than or equal to a and b respectively such that $cx + dy = (ax + by)/2$. There are $(a+1)(b+1) \leq (n+1)^2$ many such linear combinations. We can compute each one in time polynomial in the length of the input numbers, and there are polynomially many combinations to compute, so we can just check all combinations to see if we find one that works.

b. We can solve this problem in polynomial time as follows. Start by arranging the coins from largest to smallest. If there are an even number of the current largest coin, distribute them evenly to Bonnie and Clyde. Otherwise, give the extra one to Bonnie and then only give coins to Clyde until the difference has been resolved. This clearly runs in polynomial time, so we just need to show that this will always yield an even division if such a division is possible. Suppose that for some input of coins which can be divided evenly, the algorithm fails. Then there must exist a last time at which there were an odd number of a denomination 2^i, so that Bonnie got ahead and had 2^i more dollars than Clyde. At this point, we start giving coins only to Clyde. Since every denomination decrease cuts the amount in half, it will never be the case that Clyde had strictly less than Bonnie, was given an additional coin, and then had an amount strictly greater than Bonnie. Thus, the sum of all coins of size less than 2^i must not exceed 2^i. Since we assumed the coins can be divided evenly, there exists $b_0, b_1, \ldots$ and $c_0, c_1, \ldots$ such that we assign Bonnie b_i coins of value 2^i and Clyde c_i coins of value 2^i, and both receive an equal amount. Now remove all coins of value smaller than 2^i. Bonnie now has $\sum_{k=i}^{\infty} b_k 2^k$ dollars and Clyde has $\sum_{k=i}^{\infty} c_k 2^k$ dollars. Moreover, we know that there is an uneven distribution of wealth at this point, and since every coin has value at least 2^i, the difference is at least 2^i. Since the sum of the smaller coins is strictly less than 2^i, there is no way to distribute the smaller coins to fix the difference, a contradiction since we started with an even split! Thus, the proposed algorithm is correct.

c. This problem is NP-complete. First, an assignment of each check to either Bonnie or Clyde represents a certificate which can be checked in polynomial time by adding up the amounts on each of Bonnies checks, and ensuring that it is equal to the sum of the amounts on each of Clyde's checks. Next we'll show this problem, SPLIT-CHECKS is NP-hard by showing that SET-PARTITION $\leq_P$ SPLIT-CHECKS. Let S be a set of numbers. We can think of each one as giving the value of a check. If there exists a set $A \subset S$ such that $\sum_{x \in A} x = \sum_{x \in (S-A)} x$, then we can assign each check in A to Bonnie

and each check in $S - A$ to Clyde to get an equal division. On the other hand, if there is a way to evenly assign checks then we may just take A to be the set of checks given to Bonnie, so by contrapositive, if we can't find a set partition which evenly splits the set then we can't evenly divide the checks. Thus, the problem is NP-hard, so it is NP-complete.

d. An assignment of each check to either Bonnie or Clyde represents a certificate, and we can check in polynomial time whether or not the total amounts given to Bonnie and Clyde differ by at most 100. Thus, the problem is in NP. Next, we'll show it's NP-hard by showing SET-PARTITION $\leq_P$ SPLIT-CHECKS-100. Let $S = \{s_1, s_2, \ldots, s_n\}$ be a set, and let $xS = \{xs_1, xs_2, \ldots, xs_n\}$. Choose $x = \frac{101}{\min_{i,j} s_i - s_j}$. Then the difference between any two elements in xS is more than 100. If there exists $A \subset S$ such that $\sum_{s \in A} s = \sum_{s \in S-A} s$, then we give Bonnie all the checks in xA and Clyde all the checks in $x(S - A)$, for a perfectly even split of the money, which means the difference is less than 100. On the other hand, suppose there exists no such A. Then for any way of splitting the elements of S into two sets, their sums will differ by at least the minimum difference of elements in S. Thus, any way of splitting the checks in xS will result in a difference of at least 101, so there is no way to split them so that the difference is at most 100. Therefore SET-PARTITION $\leq_P$ SPLIT-CHECKS-100, so the problem is NP-hard. Since we showed it is in NP, it is also NP-complete.

Problem 34-3

a) To two color a graph, we will do it a connected component at a time, so, suppose that the graph is a single component. Pick a vertex and color arbitrarily, and color that vertex that color. Then, we repeatedly find a vertex that has a colored neighbor and color it the other color. If we are ever in the case that a vertex has neighbors of both colors, then the graph is not 2-colorable. This procedure is able to 2-color if the graph is 2-colorable, since the only point where our hand isn't forced is at the beginning when we pick a vertex and a color, but this choice is only a false one because of the symmetry of the two colors. If it finds it is 2-colorable, it also outputs a valid 2-coloring.

b) The equivalent decision problem is to, given a graph G and an integer k say if there is a coloring that uses at most k colors. The easy direction is showing that if the original problem is solvable in poly thime, then the decision problem is solvable in poly time. To do this, just compute the minimum number of colors needed and output true if this is $\leq k$.

The other direction is a bit harder. Suppose that we can solve the decision problem in polynomial time, then, we will try to show how we can actually compute the minimum number of colors needed. A trivial bound on the number of colors needed is the number of vertices, because if each vertex has it's own color, then the coloring has to be valid. So, we perform a binary

search on the number of colors, starting with the range $1..|V|$, halving it each time until we are down to a single possible number of colors needed in order to color the graph. This will only add a log factor to the runtime of the decision problem, and so will run in polynomial time.

c) For this problem, we need to show that if we can solve the decision problem quickly, then we can decide the language 3-COLOR quickly. This is just a matter of running the decision procedure with the same graph and with $k = 3$. This gets us the reduction we need to show that 3-COLOR being NP-complete implies the decision problem is NP-hard. The decision problem is in NP because we can just have the certificate explicitly be the coloring of the vertices of the graph.

d) When we restrict the graph to the vertices $x_i, \neg x_i, RED$, then we will obtain a K_3 because of the literal edges. This means that all three colors must show up in it. Since there is already $c(RED)$, then the other two must be $c(TRUE)$ and $c(FALSE)$. No matter whether we choose x_i or $\neg x_i$ to be $c(TRUE)$, we can just select the other one to be $c(FALSE)$, this gets us that, if we only care about the literal edges, we always have a 3 coloring regardless of whether we want each x_i to be true or false.

e) For convenience, we will call the vertices a, b, c, d, e from the figure, where we are reading from top to bottom and left to right for vertices that are horizontal from one another. Since we are trying to check that it is 3 colorable if and only if at least one of x, y, z are $c(TRUE)$, we can negate the only if direction. That is, we suppose they are all colored $c(FALSE)$ and show that the graph is not 3 colorable.

Suppose $c(x) = c(y) = c(z) = c(FALSE)$. Then, we have that the only possibility for vertex e is to be $c(RED)$. This means the only possibility for c is to be $c(FALSE)$. However, this means that $c(a) \neq c(x) = c(FALSE)$, $c(d) \neq c(y) = c(FALSE)$, and $c(b) \neq c(c) = c(FALSE)$. So, we have a contradiction because any K_3 must have one of each color, and none of the vertices in this K_3 can be $c(FALSE)$. This shows that the graph is not 3-colorable.

For the other direction, we do not negate. So, we assume there is a vertex colored $c(TRUE)$ and we show that the graph is 3-colorable. We will split into the following cases. Note that because x and y play a symmetric role, we can reduce the number of cases from 7 to 5

x	y	z	a	b	c	d	e
$c(TRUE)$	$c(TRUE)$	$c(TRUE)$	$c(FALSE)$	$c(TRUE)$	$c(FALSE)$	$c(RED)$	$c(RED)$
$c(FALSE)$	$c(TRUE)$	$c(TRUE)$	$c(RED)$	$c(TRUE)$	$c(FALSE)$	$c(FALSE)$	$c(RED)$
$c(FALSE)$	$c(FALSE)$	$c(TRUE)$	$c(RED)$	$c(FALSE)$	$c(RED)$	$c(TRUE)$	$c(FALSE)$
$c(TRUE)$	$c(TRUE)$	$c(FALSE)$	$c(FALSE)$	$c(TRUE)$	$c(FALSE)$	$c(RED)$	$c(RED)$
$c(FALSE)$	$c(TRUE)$	$c(FALSE)$	$c(TRUE)$	$c(RED)$	$c(FALSE)$	$c(FALSE)$	$c(RED)$

Then, in every case where at least one of the inputs is true, there is an assignment of colors to the other vertices that produces a valid 3-coloring.

f) Suppose we are given any instance of 3-CNF-SAT, we will be using exactly the same construction as described in the problem for the reduction. First, we note that each of the vertices corresponding to a variable and its negation must only have the colors $c(TRUE)$ and $c(FALSE)$, and exactly one can be $c(TRUE)$ because of the literal edges. This means that each of the clause vertices will only be colorable if we assign a color of true to one of the variable vertices that are in the clause. This means that if we set each variable that has $c(x_i) = c(TRUE)$ true and each variable that has $c(\neg x_i) = c(TRUE)$ to be false, we will of obtained an assignment that makes at least one of the entries in each clause true, and so, is a satisfying assignment of the formula. Since 3-CNF-SAT is NP-complete, this means that 3-COLOR is NP-hard.

To see that it is in NP, just let the certificate be the coloring. Checking the coloring can be done in linear time.

Problem 34-4

a. For fixed k, does there exist a permutation $\sigma \in S_n$ such that if we run the tasks in the order $a_{\sigma(1)}, \ldots, a_{\sigma(n)}$, the total profit is at least k?

b. It is clear that the decision problem is in NP because we can view a certificate as a permutation of the tasks. We then check the tasks in that order to see if they have finished by their deadlines and what profit we incur, finally comparing this to k.

c. Suppose that a particular task a_i is the first to be performed. Then we solve the subproblem of deciding whether there exists a solution to the problem with tasks $a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n$, each with their respective profits and times, deadlines such that $d_j = d_j - t_i$, and with total profit at least $k - p_i$. To find this out, we make a lookup table which keeps track of the optimal schedule computed bottom-up.

d. There are 2^n possible profits that could be made, based on whether or not we finish each task by its deadline. We can binary search these for the one with maximum profit which satisfies the decision problem, which we can determine in polynomial time by part c. Since binary search takes $\lg(2^n) = n$, we need only run the polynomial time algorithm of part c. n times before we find the maximal k which solves the decision problem, and thus solves the optimization problem.

Chapter 35

Michelle Bodnar, Andrew Lohr

September 17, 2017

Exercise 35.1-1

We could select the graph that consists of only two vertices and a single edge between them. Then, the approximation algorithm will always select both of the vertices, whereas the minimum vertex cover is only one vertex. more generally, we could pick our graph to be k edges on a graph with $2k$ vertices so that each connected component only has a single edge. In these examples, we will have that the approximate solution is off by a factor of two from the exact one.

Exercise 35.1-2

It is clear that the edges picked in line 4 form a matching, since we can only pick edges from E', and the edges in E' are precisely those which don't share an endpoint with any vertex already in C, and hence with any already-picked edge. Moreover, this matching is maximal because the only edges we don't include are the ones we removed from E'. We did this because they shared an endpoint with an edge we already picked, so if we added it to the matching it would no longer be a matching.

Exercise 35.1-3

We will construct a bipartite graph with $V = R \cup L$. We will try to construct it so that R is uniform, not that R is a vertex cover. However, we will make it so that the heuristic that the professor (professor who?) suggests will cause us to select all the vertices in L, and show that $|L| > 2|R|$.

Initially start off with $|R| = n$ fixed, and L empty. Then, for each i from 2 up to n, we do the following. Let $k = \lfloor \frac{n}{i} \rfloor$. Select S a subset of the vertices of R of size ki, and so that all the vertices in $R - S$ have a greater or equal degree. Then, we will add k vertices to L, each of degree i, so that the union of their neighborhoods is S. Note that throughout this process, the furthest apart the degrees of the vertices in R can be is 1, because each time we are picking the smallest degree vertices and increasing their degrees by 1. So, once this has been done for $i = n$, we can pick a subset of R whose degree is one less than the rest of R (or all of R if the degrees are all equal), and for each vertex in

1

that set we picked, add a single vertex to L whose only neighbor is the one in R that we picked. This clearly makes R uniform.

Also, lets see what happens as we apply the heuristic. Note that each vertex in R only has at most a single vertex of each degree in L. This means that as we remove vertices from L along with their incident edges, we will always have that the highest degree vertex remaining in L is greater or equal to the highest degree vertex in R. This means that we can always, by this heuristic, continue to select vertices from L instead of R. So, when we are done, we have selected, by this heuristic, that our vertex cover is all of L.

Lastly, we need to show that L is big enough relative to R. The size of L is greater or equal to $\sum_{i=2}^{n} \lfloor \frac{n}{i} \rfloor$ where we ignore any of the degree 1 vertices we may of added to L in our last step. This sum can be bounded below by the integral $\int_{i=2}^{n+1} \frac{n}{x} dx$ by formula (A.12). Elementary calculus tells us that this integral evaluates to $n(\lg(n+1) - 1)$, so, we can clearly select $n > 8$ to make it so that

$$
\begin{aligned}
2|R| = 2n \\
< n(\lg(8+1) - 1) \\
\le n(\lg(n+1) - 1) \\
= \int_{i=2}^{n+1} \frac{n}{x} dx \\
\le \sum_{i=2}^{n} \lfloor \frac{n}{i} \rfloor \\
\le |L|
\end{aligned}
$$

Since we selected L as our vertex cover, and L is more than twice the size of the vertex cover R, we do not have a 2- approximation using this heuristic. In fact, we have just shown that this heuristic is not a ρ approximation for any constant ρ.

Exercise 35.1-4

If a tree consists of a single node, we immediately return the empty cover and are done. From now on, assume that $|V| \ge 2$. I claim that for any tree T, and any leaf node v on that tree, there exists an optimal vertex cover for T which doesn't contain v. To see this, let $V' \subset V$ be an optimal vertex cover, and suppose that $v \in V'$. Since v has degree 1, let u be the vertex connected to v. We know that u can't also be in V', because then we could remove v and still have an vertex cover, contradicting the fact that V' was claimed to be optimal, and hence smallest possible. Now let V'' be the set obtained by removing v from V' and adding u to V'. Every edge not incident to u or v has an endpoint in V', and thus in V''. Moreover, every edge incident to u is taken care of because $u \in V''$. In particular, the edge from v to u is still okay because $u \in V''$. Therefore V'' is a vertex cover, and $|V''| = |V'|$, so it is

optimal. We may now write down the greedy approach, given in the algorithm GREEDY-VERTEX-COVER:

Algorithm 1 GREEDY-VERTEX-COVER(G)

1: $V' = \emptyset$
2: let L be a list of all the leaves of G
3: **while** $V \neq \emptyset$ **do**
4:     **if** $|V| == 1$ or 0 **then**
5:         **return** V'
6:     **end if**
7:     let v be a leaf of $G = (V, E)$, and $\{v, u\}$ be its incident edge
8:     $V = V - v$
9:     $V' = V' \cup u$
10:     **for** each edge $\{u, w\} \in E$ incident to u **do**
11:         **if** $d_w == 1$ **then**
12:             $L.insert(w)$
13:         **end if**
14:         $E = E - \{u, w\}$
15:     **end for**
16:     $V = V - u$
17: **end while**

As it stands, we can't be sure this runs in linear time since it could take $O(n)$ time to find a leaf each time we run line 7. However, with a clever implementation we can get around this. Keep in mind that a vertex either starts as a leaf, or becomes a leaf when an edge connecting it to a leaf is removed from it. We'll maintain a list L of current leaves in the graph. We can find all leaves initially in linear time in line 2. Assume we are using an adjacency list representation. In line 4 we check if $|V|$ is 1 or 0. If it is, then we are done. Otherwise, the tree has at least 2 vertices which implies that it has a leaf. We can get our hands on a leaf in constant time in line 7 by popping the first element off of the list L, which maintains our leaves. We can find the edge $\{u, v\}$ in constant time by examining v's adjacency list, which we know has size 1. As described above, we know that v isn't contained in an optimal solution. However, since $\{u, v\}$ is an edge and at least one of its endpoints must be contained in V', we must necessarily add u. We can remove v from V in constant time, and add u to V' in constant time. Line 10 will only execute once for each edge, and since G is a tree we have $|E| = |V| - 1$. Thus, the total contribution of the for-loop of line 10 is linear. We never need to consider edges adjacent to u since $u \in V'$. We check the degrees as we go, adding a vertex to L if we discover it has degree 1, and hence is a leaf. We can update the attribute d_w for each w in constant time, so this doesn't affect the runtime. Finally, we remove u from V and never again need to consider it.

3

Exercise 35.1-5

It does not imply the existence of an approximation algorithm for the maximum size clique. Suppose that we had in a graph of size n, the size of the smallest vertex cover was k, and we found one that was size $2k$. This means that in the compliment, we found a clique of size $n - 2k$, when the true size of the largest clique was $n - k$. So, to have it be a constant factor approximation, we need that there is some λ so that we always have $n - 2k \geq \lambda(n-k)$. However, if we had that k was close to $\frac{n}{2}$ for our graphs, say $\frac{n}{2} - \epsilon$, then we would have to require $2\epsilon \geq \lambda\frac{n}{2} + \epsilon$. Since we can make ϵ stay tiny, even as n grows large, this inequality cannot possibly hold.

Exercise 35.2-1

Suppose that $c(u,v) < 0$ and w is any other vertex in the graph. Then, to have the triangle inequality satisfied, we need $c(w,u) \leq c(w,v) + c(u,v)$. Now though, subtract $c(u,v)$ from both sides, we get $c(w,v) \geq c(w,u) - c(u,v) > c(w,u) + c(u,v)$ so, it is impossible to have $c(w,v) \leq c(w,u) + c(u,v)$ as the triangle inequality would require.

Exercise 35.2-2

Let m be some value which is larger than any edge weight. If $c(u,v)$ denotes the cost of the edge from u to v, then modify the weight of the edge to be $H(u,v) = c(u,v) + m$. (In other words, H is our new cost function). First we show that this forces the triangle inequality to hold. Let u, v, and w be vertices. Then we have $H(u,v) = c(u,v) + m \leq 2m \leq c(u,w) + m + c(w,v) + m = H(u,w) + H(w,v)$. Note: it's important that the weights of edges are nonnegative.

Next we must consider how this affects the weight of an optimal tour. First, any optimal tour has exactly n edges (assuming that the input graph has exactly n vertices). Thus, the total cost added to any tour in the original setup is nm. Since the change in cost is constant for every tour, the set of optimal tours must remain the same.

To see why this doesn't contradict Theorem 35.3, let H denote the cost of a solution to the transformed problem found by a ρ-approximation algorithm and H^* denote the cost of an optimal solution to the transformed problem. Then we have $H \leq \rho H^*$. We can now transform back to the original problem, and we have a solution of weight $C = H - nm$, and the optimal solution has weight $C^* = H^* - nm$. This tells us that $C + nm \leq \rho(C^* + nm)$, so that $C \leq \rho(C^*) + (\rho - 1)nm$. Since $\rho > 1$, we don't have a constant approximation ratio, so there is no contradiction.

Exercise 35.2-3

4

From the chapter on minimum spanning trees, recall Prim's algorithm. That is, a minimum spanning tree can be found by repeatedly finding the nearest vertex to the vertices already considered, and adding it to the tree, being adjacent to the vertex among the already considered vertices that is closest to it. Note also, that we can recusively define the preorder traversal of a tree by saying that we first visit our parent, then ourselves, then our children, before returning priority to our parent. This means that by inserting vertices into the cycle in this way, we are always considering the parent of a vertex before the child. To see this, suppose we are adding vertex v as a child to vertex u. This means that in the minimum spanning tree rooted at the first vertex, v is a child of u. So, we need to consider u first before considering v, and then consider the vertex that we would of considered after v in the previous preorder traversal. This is precisely achieved by inserting v into the cycle in such a manner. Since the property of the cycle being a preorder traversal for the minimum spanning tree constructed so far is maintained at each step, it is the case at the end as well, once we have finished considering all the vertices. So, by the end of the process, we have constructed the preorder traversal of a minimum spanning tree, even though we never explicity built the spanning tree. It was shown in the section that such a Hamiltonian cycle will be a 2 approximation for the cheapest cycle under the given assumption that the weights satisfy the triangle inequality.

Exercise 35.2-4

By Problem 23-3, there exists a linear time algorithm to compute a bottleneck spanning tree of a graph. First run this linear time algorithm to find a bottleneck tree BT. Next, take full walk on the tree, skipping at most 2 consecutive intermediate nodes to get from one unrecorded vertex to the next. By Exercise 34.2-11 we can always do this, and obtain a graph with a hamiltonian cycle which we will call HB. Let B denote the cost of the most costly edge in the bottleneck spanning tree. Consider the cost of any edge that we put into HB which was not in BT. By the triangle inequality, it's cost is at most $3B$, since we traverse a single edge instead of at most 3 to get from one vertex to the next. Let hb be the cost of the most costly edge in HB. Then we have $hb \leq 3B$. Thus, the most costly edge in HB is at most $3B$. Let B^* denote the cost of the most costly edge an optimal bottleneck hamiltonian tour HB^*. Since we can obtain a spanning tree from HB^* by deleting the most costly edge in HB^*, the minimum cost of the most costly edge in any spanning tree is less than or equal to the cost of the most costly edge in any Hamiltonian cycle. In other words, $B \leq B^*$. On the other hand, we have $hb \leq 3B$, which implies $hb \leq 3B^*$, so the algorithm is 3-approximate.

Exercise 35.2-5

To show that the optimal tour never crosses itself, we will suppose that it did cross itself, and then show that we could produce a new tour that had lower cost, obtaining our contradiction because we had said that the tour we

had started with was optimal, and so, was of minimal cost. If the tour crosses itself, there must be two pairs of vertices that are both adjacent in the tour, so that the edge between the two pairs are crossing each other. Suppose that our tour is $S_1 x_1 x_2 S_2 y_1 y_2 S_3$ where S_1, S_2, S_3 are arbitrary sequences of vertices, and $\{x_1, x_2\}$ and $\{y_1, y_2\}$ are the two crossing pairs. Then, We claim that the tour given by $S_1 x_1 y_2 \text{Reverse}(S_2) y_1 x_2 S_3$ has lower cost. Since $\{x_1, x_2, y_1, y_2\}$ form a quadrilateral, and the original cost was the sum of the two diagonal lengths, and the new cost is the sums of the lengths of two opposite sides, this problem comes down to a simple geometry problem.

Now that we have it down to a geometry problem, we just exercise our grade school geometry muscles. Let P be the point that diagonals of the quadrilateral intersect. Then, we have that the vertices $\{x_1, P, y_2\}$ and $\{x_2, P, y_1\}$ form triangles. Then, we recall that the longest that one side of a triangle can be is the sum of the other two sides, and the inequality is strict if the triangles are non-degenerate, as they are in our case. This means that $||x_1 y_2|| + ||x_2 y_1|| < ||x_1 P|| + ||P y_2|| + ||x_2 P|| + ||P y_1|| = ||x_1 x_2|| + ||y_1 y_2||$. The right hand side was the original contribution to the cost from the old two edges, while the left is the new contribution. This means that this part of the cost has decreased, while the rest of the costs in the path have remained the same. This gets us that the new path that we constructed has strictly better cross, so we must not of had an optimal tour that had a crossing in it.

Exercise 35.3-1

Since all of the words have no repeated letters, the first word selected will be the one that appears earliest on among those with the most letters, this is "thread". Now, we look among the words that are left, seeing how many letters that aren't already covered that they contain. Since "lost" has four letters that have not been mentioned yet,and it is first among those that do, that is the next one we select. The next one we pick is "drain" because it is has two unmentioned letters. This only leave "shun" having any unmentioned letters, so we pick that, completing our set. So, the final set of words in our cover is $\{thread, lost, drain, shun\}$.

Exercise 35.3-2

A certificate for the set-covering problem is a list of which sets to use in the covering, and we can check in polynomial time that every element of X is in at least one of these sets, and that the number of sets doesn't exceed k. Thus, set-covering is in NP. Now we'll show it's NP-hard by reducing it from the vertex-cover problem. Suppose we are given a graph $G = (V, E)$. Let V' denote the set of vertices of G with all vertices of degree 0 removed. To each vertex $v \in V$, associate a set S_v which consists of v and all of its neighbors, and let $\mathscr{F} = \{S_v | v \in V'\}$. If $S \subset V$ is a vertex cover of G with at most k vertices, then $\{S_v | v \in S \cap V'\}$ is a set cover of V' with at most k sets. On the other hand, suppose there doesn't exist a vertex cover of G with at most k vertices. If there were

a set cover M of V' with at most k sets from $\mathscr{F}$, then we could use $\{v | S_v \in M\}$ as a k-element vertex cover of G, a contradiction. Thus, G has a k element vertex cover if and only if there is a k-element set cover of V' using elements in $\mathscr{F}$. Therefore set-covering is NP-hard, so we conclude that it is NP-complete.

Exercise 35.3-3

See the algorithm LINEAR-GREEDY-SET-COVER. Note that everything in the inner most for loop takes constant time and will only run once for each pair of letter and a set containing that letter. However, if we sum up the number of such pairs, we get $\sum_{S \in \mathcal{F}} |S|$ which is what we wanted to be linear in.

Algorithm 2 LINEAR-GREEDY-SET-COVER($\mathcal{F}$)
---
compute the sizes of every $S \in \mathcal{F}$, storing them in $S.size$.
let A be an array of length $\max_S |S|$, consisting of empty lists
for $S \in \mathcal{F}$ **do**
    add S to $A[S.size]$.
end for
let $A.max$ be the index of the largest nonempty list in A.
let L be an array of length $|\cup_{S \in \mathcal{F}} S|$ consisting of empty lists
for $S \in \mathcal{F}$ **do**
    for $\ell \in S$ **do**
        add S to $L[\ell]$
    end for
end for
let C be the set cover that we will be selecting, initially empty.
let T be the set of letters that have been covered, initially empty.
while $A.max > 0$ **do**
    Let S_0 be any element of $A[A.max]$.
    add S_0 to C
    remove S_0 from $A[A.max]$
    for $\ell \in S_0 \setminus T$ **do**
        for $S \in L[\ell]$ **do**
            Remove S from $A[A.size]$
            S.size = S.size -1
            Add S to $A[S.size]$
            if A[A.max] is empty **then**
                A.max = A.max-1
            end if
        end for
        add ℓ to T
    end for
end while

Exercise 35.3-4

Each c_x has cost at most 1, so we we can trivially replace the upper bound in inequality (35.12) by $\sum_{x \in S} c_x \le |S| \le \max\{|S| : S \in \mathscr{F}\}$. Combining inequality (35.11) and the new (35.12), we have

$$|C| \le \sum_{S \in \mathcal{C}^*} \sum_{x \in S} c_x \le \sum_{S \in \mathcal{C}^*} \max\{|S| : S \in \mathscr{F}\} = |\mathcal{C}^*| \max\{|S| : S \in \mathscr{F}\}.$$

Exercise 35.3-5

Suppose we pick the number of elements in our base set to be a multiple of four. We will describe a set of sets on a base set of size 4, and then take n disjoint copies of it. For each of the four element base sets, we will have two choices for how to select the set cover of those 4 elements. For a base set consisting of $\{1, 2, 3, 4\}$, we pick our set of sets to be $\{\{1, 2\}, \{1, 3\}, \{3, 4\}, \{2, 4\}\}$, then we could select either $\{\{1, 2\}, \{3, 4\}\}$ or $\{\{1, 3\}, \{2, 4\}\}$. This means that we then have a set cover problem with $4n$ sets, each of size two where the set cover we select is done by n independent choices, and so there are 2^n possible final set covers based on how ties are resolved.

Exercise 35.4-1

Any clause that contains both a variable and its negation is automatically satisfied, since we always have $x \lor \neg x$ is true regardless of the truth value of x. This means that if we separate out the clauses that contain no variable and its negation, we have an 8/7ths approximation on those, and we have all of the other clauses satisfied regardless of the truth assignments to variables. So, the quality of the approximation just improves when we include the clauses containing variables and their negation.

Exercise 35.4-2

As usual, we'll assume that each clause contains only at most instance of each literal, and that it doesn't contain both a literal and its negation. Assign each variable 1 with probability 1/2 and 0 with probability 1/2. Let Y_i be the random variable which takes on 1 if clause i is satisfied and 0 if it isn't. The probability that a clause fails to be satisfied is $(1/2)^k \le 1/2$ where k is the number of literals in the clause, so $k \ge 1$. Thus, $P(Y_i) = 1 - P(\overline{Y_i}) \ge 1/2$. Let Y be the total number of clauses satisfied and m be the total number of clauses. Then we have

$$E[Y] = \sum_{i=1}^{m} E[Y_i] \ge m/2.$$

Let $C*$ be the number of clauses satisfied by an optimal solution. Then $C^* \le m$ so we have $C^*/E[Y] \le m/(m/2) = 2$. Thus, this is a randomized 2-approximation algorithm.

Exercise 35.4-3

Lets consider the expected value of the weight of the cut. For each edge, the probability that that edge crosses the cut is the probability that our coin flips came up differently for the two vertices that are in the edge. This happens with probability $\frac{1}{2}$. Therefore, by linearity of expectation, we know that the expected weight of this cut is $\frac{|E|}{2}$. We know that for the best cut we can make, the weight will be bounded by the total number of edges in the graph, so we have that the weight of the best cut is at most twice the expected weight of this random cut.

Exercise 35.4-4

Let $x : V \to \mathbb{R}_{\geq 0}$ be an optimal solution for the linear-programming relaxation with condition (35.19) removed, and suppose there exists $v \in V$ such that $x(v) > 1$. Now let x' be such that $x'(u) = x(u)$ for $u \neq v$, and $x'(v) = 1$. Then (35.18) and (35.20) are still satisfied for x'. However,

$$
\begin{aligned}
\sum_{u \in V} w(u)x(u) &= \sum_{u \in V - v} w(u)x'(u) + w(v)x(v) \\
&> \sum_{u \in V - v} w(u)x'(u) + w(v)x'(v) \\
&= \sum_{u \in V} w(u)x'(u)
\end{aligned}
$$

which contradicts the assumption that x minimized the objective function. Therefore it must be the case that $x(v) \leq 1$ for all $v \in V$, so the condition (35.19) is redundant.

Exercise 35.5-1

Every subset of $\{1, 2, \ldots, i\}$ can either contain i or not contain i. If it does contain i, the sum corresponding to that subset will be in $P_{i-1} + x_i$, since the sum corresponding to the restriction of the subset to $\{1, 2, 3, \ldots\}$ will be in P_{i-1}. If we are in the other case, then when we restrict to $\{1, 2, \ldots, i - 1\}$ we aren't changing the subset at all. Similarly, if an element is in P_{i-1}, that same subset is in P_i. If an element is in $P_{i-1} + x_i$, then it is in P_i where we just make sure our subset contains i. This proves (35.23).

It is clear that L is a sorted list of elements less than T that corresponds to subsets of the empty set before executing the for loop, because it is empty. Then, since we made L_i from elements in L_{i-1} and $x_i + Li - 1$, we know that it will be a subset of those sums obtained from the first i elements. Also, since both L_{i-1} and $x_i + L_{i-1}$ were sorted, when we merged them, we get a sorted list. Lastly, we know that it all sums except those that are more than t since we have all the subset sums that are not more than t in L_{i-1}, and we cut out

anything that gets to be more than t.

Exercise 35.5-2

We'll proceed by induction on i. When $i = 1$, either $y = 0$ in which case the claim is trivially true, or $y = x_1$. In this case, $L_1 = \{0, x_1\}$. We don't lose any elements by trimming since $0(1 + \delta) < x_1 \in \mathbb{Z}_{>0}$. Since we only care about elements which are at most t, the removal in line 6 of APPROX-SUBSET-SUM doesn't affect the inequality. Since $x_1/(1 + \varepsilon/2n) \leq x_1 \leq x_1$, the claim holds when $i = 1$. Now suppose the claim holds for i, where $i \leq n - 1$. We'll show that the claim holds for $i + 1$. Let $y \in P_{i+1}$. Then by (35.23) either $y \in P_i$ or $y \in P_i + x_{i+1}$. If $y \in P_i$, then by the induction hypothesis there exists $z \in L_i$ such that $\frac{y}{(1+\varepsilon/2n)^{i+1}} \leq \frac{y}{(1+\varepsilon/2n)^i} \leq z \leq y$. If $z \in L_{i+1}$ then we're done. Otherwise, z must have been trimmed, which means there exists $z' \in L_{i+1}$ such that $z/(1 + \delta) \leq z' \leq z$. Since $\delta = \varepsilon/2n$, we have $\frac{y}{(1+\varepsilon/2n)^{i+1}} \leq z' \leq z \leq y$, so inequality (35.26) is satisfied. Lastly, we handle the case where $y \in P_i + x_{i+1}$. Write $y = p + x_{i+1}$. By the induction hypothesis, there exists $z \in L_i$ such that $\frac{p}{(1+\varepsilon/2n)^i} \leq z \leq p$. Since $z + x_{i+1} \in L_{i+1}$, before thinning, this means that

$$\frac{p + x_{i+1}}{(1 + \varepsilon/2n)^i} \leq \frac{p}{(1 + \varepsilon/2n)^i} + x_{i+1} \leq z + x_{i+1} \leq p + x_{i+1},$$

so $z + x_{i+1}$ well approximates $p + x_{i+1}$. If we don't thin this out, then we are done. Otherwise, there exists $w \in L_{i+1}$ such that $\frac{z+x_{i+1}}{1+\delta} \leq w \leq z + x_{i+1}$. Then we have

$$\frac{p + x_{i+1}}{(1 + \varepsilon/2n)^{i+1}} \leq \frac{z + x_{i+1}}{1 + \varepsilon/2n} \leq w \leq z + x_{i+1} \leq p + x_{i+1}.$$

Thus, claim holds for $i + 1$. By induction, the claim holds for all $1 \leq i \leq n$, so inequality (35.26) is true.

Exercise 35.5-3

As we have many times in the past, we'll but on our freshman calculus hats.

$$\frac{d}{dn}\left(1 + \frac{\epsilon}{2n}\right)^n =$$

$$\frac{d}{dn} e^{n \lg\left(1 + \frac{\epsilon}{2n}\right)} =$$

$$e^{n \lg\left(1 + \frac{\epsilon}{2n}\right)} \left(\left(1 + \frac{\epsilon}{2n}\right) + \frac{n\frac{-\epsilon}{2n^2}}{1 + \frac{\epsilon}{2n}}\right) =$$

$$e^{n \lg\left(1 + \frac{\epsilon}{2n}\right)} \left(\frac{\left(1 + \frac{\epsilon}{2n}\right)^2 - \frac{\epsilon}{2n}}{1 + \frac{\epsilon}{2n}}\right) =$$

$$e^{n \lg\left(1 + \frac{\epsilon}{2n}\right)} \left(\frac{1 + \frac{\epsilon}{2n} + \left(\frac{\epsilon}{2n}\right)^2}{1 + \frac{\epsilon}{2n}}\right) =$$

Since all three factors are always positive, the original expression is always positive.

Exercise 35.5-4

We'll rewrite the relevant algorithms for finding the smallest value, modifying both TRIM and APPROX-SUBSET-SUM. The analysis is the same as that for finding a largest sum which doesn't exceed t, but with inequalities reversed. Also note that in TRIM, we guarantee instead that for each y removed, there exists z which remains on the list such that $y \leq z \leq y(1 + \delta)$.

Algorithm 3 TRIM(L, δ)

1: let m be the length of L
2: $L' = \langle y_m \rangle$
3: $last = y_m$
4: **for** $i = m - 1$ downto 1 **do**
5:     **if** $y_i < last/(1 + \delta)$ **then**
6:         append y_i to the end of L'
7:         $last = y_i$
8:     **end if**
9: **end for**
10: **return** L'

Algorithm 4 APPROX-SUBSET-SUM(S, t, ε)

1: $n = |S|$
2: $L_0 = \langle \sum_{i=1}^{n} x_i \rangle$
3: **for** $i = 1$ to n **do**
4:     $L_i = $ MERGE-LISTS$(L_{i-1}, L_{i-1} - x_i)$
5:     $L_i = $ TRIM$(L_i, \varepsilon/2n)$
6:     return from L_i every element that is less than t
7: **end for**
8: let z^* be the largest value in L_n
9: **return** z^*

Exercise 35.5-5

We can modify the procedure APPROX-SUBSET-SUM to actually report the subset corresponding to each sum by giving each entry in our L_i lists a piece of satellite data which is a list the elements that add up to the given number sitting in L_i. Even if we do a stupid representation of the sets of elements, we will still have that the runtime of this modified procedure only takes additional time by a factor of the size of the original set, and so will still be polynomial. Since the actual selection of the sum that is our approximate best is unchanged

by doing this, all of the work spent in the chapter to prove correctness of the approximation scheme still holds. Each time that we put an element into L_i, it could of been copied directly from L_{i-1}, in which case we copy the satellite data as well. The only other possibility is that it is x_i plus an entry in L_{i-1}, in which case, we add x_i to a copy of the satellite data from L_{i-1} to make our new set of elements to associate with the element in L_i.

Problem 35-1

a. First, we will show that it is np hard to determine if, given a set of numbers, there is a subset of them whose sum is equal to the sum of that set's compliment, this is called the partition problem. We will show this is NP hard by using it to solve the subset sum problem. Suppose that we wanted to fund a subset of $\{y_1, y_2, \ldots y_n\}$ that summed to t. Then, we will run partition on the set $\{y_1, y_2, \ldots, y_n, (\sum_i y_i) - 2t\}$. Note then, that the sum of all the elements in this set is $2\left(\sum_i y_i\right) - 2t$. So, any partition must have both sides equal to half that. If there is a subset of the original set that sums to t, we can put that on one side together with the element we add, and put all the other original elements on the other side, giving us a partition. Suppose that we had a partition, then all the elements that are on the same side as the special added element form a subset that is equal to t. This show that the partition problem is NP hard.

Let $\{x_1, x_2, \ldots, x_n\}$ be an instance of the partition problem. Then, we will construct an instance of a packing problem that can be packed into 2 bins if and only if there is a subset of $\{x_1, x_2, \ldots x_n\}$ with sum equal to that of its compliment. To do this, we will just let our values be $c_i = \frac{2x_i}{\sum_{j=1}^{n} x_j}$. Then, the total weight is exactly 2, so clearly two bins are needed. Also, it can fit in two bins if there is a partition of the original set.

b. If we could pack the elements in to fewer than $\lceil S \rceil$ bins, that means that the total capacity from our bins is $< \lfloor S \rfloor \le S$, however, we know that to total space needed to store the elements is S, so there is no way we could fit them in less than that amount of space.

c. Suppose that there was already one bin that is at most half full, then when we go to insert an element that has size at most half, it will not go into an unused bin because it would be placed in this bin that is at most half full. This means that we will never create a second bin that is at most half full, and so, since we start off with fewer that two bins that are at most half full, there will never be two bins that are at most half full (and so never two that are less than half full).

d. Since there is at most one bin that is less than half full, we know that the total amount of size that we have packed into our P bins is $> \frac{1}{2}(P-1)$. That is, we know that $S > \frac{1}{2}(P-1)$, which tells us that $2S+1 > P$. So, if we were

to have $P > \lceil 2S \rceil$, then, we have that $P \geq \lceil 2S \rceil + 1 = \lceil 2S+1 \rceil \geq 2S+1 > P$, which is a contradiction.

e. We know from part b that there is a minimum of $\lceil S \rceil$ bins required. Also, by part d, we know that the first fit heuristic causes us to sue at most $\lceil 2S \rceil$ bins. This means that the number of bins we report is at most off by a factor of $\frac{\lceil 2S \rceil}{\lceil S \rceil} \leq 2$.

f. We can implement the first fit heuristic by just keeping an array of bins, and each time just doing a linear scan to find the first bin that can accept it. So, if we let P be the number of bins, this procedure will have a runtime of $O(P^2) = O(n^2)$. However, since all we needed was that there was at most one bin that was less than half full, we could do a faster procedure. We keep the array of bins as before, but we will keep track of the first empty bin and a pointer to a bin that is less than half full. Upon insertion, if the thing we inserted is more than half, put it in the first empty bin and increment it. If it is less than half, it will fit in the bin that is less than half full, if that bin then becomes more than half full, get rid of the less than half full bin pointer. If there wasn't a nonempty bin that was less than half full, we just put the element in the first empty bin. This will run in time $O(n)$ and still satisfy everything we needed to assure we had a 2-approximation.

Problem 35-2

a. Given a clique D of m vertices in G, let D^k be the set of all k-tuples of vertices in D. Then D^k is a clique in $G^{(k)}$, so the size of a maximum clique in $G^{(k)}$ is at least that of the size of a maximum clique in G. We'll show that the size cannot exceed that.

Let m denote the size of the maximum clique in G. We proceed by induction on k. When $k = 1$, we have $V^{(1)} = V$ and $E^{(1)} = E$, so the claim is trivial. Now suppose that for $k \geq 1$, the size of the maximum clique in $G^{(k)}$ is equal to the kth power of the size of the maximum clique in G. Let u be a $(k+1)$-tuple and u' denote the restriction of u to a k-tuple consisting of its first k entries. If $\{u, v\} \in E^{(k+1)}$ then $\{u', v'\} \in E^{(k)}$. Suppose that the size of the maximum clique C in $G^{(k+1)} = n > m^{k+1}$. Let $C' = \{u' | u \in C\}$. By our induction hypothesis, the size of a maximum clique in $G^{(k)}$ is m^k, so since C' is a clique we must have $|C'| \leq m^k < n/m$. A vertex u is only removed from C' when it is a duplicate, which happens only when there is a v such that $u, v \in C$, $u \neq v$, and $u' = v'$. If there are only m choices for the last entry of a vertex in C, then the size can decrease by at most a factor of m. Since the size decreases by a factor of strictly greater than m, there must be more than m vertices which appear among the last entries of vertices in C'. Since C' is a clique, all of its vertices are connected by edges, which implies all of these vertices in G are connected by edges, implying that G contains a clique of size strictly greater than m, a contradiction.

b. Suppose there is an approximation algorithm that has a constant approximation ratio c for finding a maximum size clique. Given G, form $G^{(k)}$, where k will be chosen shortly. Perform the approximation algorithm on $G^{(k)}$. If n is the maximum size clique of $G^{(k)}$ returned by the approximation algorithm and m is the actual maximum size clique of G, then we have $m^k/n \leq c$. If there is a clique of size at least n in $G^{(k)}$, we know there is a clique of size at least $n^{1/k}$ in G, and we have $m/n^{1/k} \leq c^{1/k}$. Choosing $k > 1/\log_c(1+\varepsilon)$, this gives $m/n^{1/k} \leq 1 + \varepsilon$. Since we can form $G^{(k)}$ in time polynomial in k, we just need to verify that k is a polynomial in $1/\varepsilon$. To this end,

$$k > 1/\log_c(1+\varepsilon) = \ln c/\ln(1+\varepsilon) \geq \ln c/\varepsilon$$

where the last inequality follows from (3.17). Thus, the $(1+\varepsilon)$ approximation algorithm is polynomial in the input size, and $1/\varepsilon$, so it is a polynomial time approximation scheme.

Problem 35-3

An obvious way to generalize the greedy set cover heuristic is to not just pick the set that covers the most uncovered elements, but instead to repeatedly select the set that maximizes the value of the number of uncovered points it covers divided by its weight. This agrees with the original algorithm in the case that the weights are all 1. The main difference is that instead of each step of the algorithm assigning one unit of cost, it will be assigning w_i units of cost to add the set S_i. So, suppose that $\mathcal{C}$ is the cover we selected by this heuristic, and $\mathcal{C}^*$ is an optimal cover. So, we will have $|\sum_{S_i \in \mathcal{C}} w_i| = \sum_{x \in X} w_i c_x \leq \sum_{S_i \in \mathcal{C}^*} \sum_{x \in S_i} w_i c_x$. Then, $\sum_{x \in S_i} w_i c_x = w_i \sum_{x \in S_i} c_x \leq w_i H(|S|)$. So, $|\mathcal{C}| \leq \sum_{S_i \in \mathcal{C}^*} w_i H(|S_i|) \leq |\sum_{S_i \in \mathcal{C}^*} w_i| H(\max(|S_i|))$. This means that the weight of the selected cover is a $H(d)$ approximation for the weight of the optimal cover.

Problem 35-4

a. Let $V = \{a, b, c, d\}$ and $E = \{\{a, b\}, \{b, c\}, \{c, d\}\}$. Then $\{b, c\}$ is a maximal matching, but a maximum matching consists of $\{a, b\}$ and $\{c, d\}$.

b. The greedy algorithm considers each edge one at a time. If it can be added to the matching, add it. Otherwise discard it and never consider it again. If an edge can't be added at some time, then it can't be added at any later time, so the result is a maximal matching. We can determine whether or not to add an edge in constant time by creating an array of size $|V|$ filled with zeros, and placing a 1 in position i if the ith vertex is incident with an edge in the matching. Thus, the runtime of the greedy algorithm is $O(E)$.

c. Let M be a maximum matching. In any vertex cover C, each edge in M must have the property that at least one of its endpoints is in C. Moreover, no two edges in M have any endpoints in common, so the sice of C is at least that of M.

d. It consists of only isolated vertices and no edges. If an edge $\{u, v\}$ were contained in the subgraph of G induced by the vertices of G not in T, then this would imply that neither u nor v is incident on some edge in M, so we could add the edge $\{u, v\}$ to M and it would still be a matching. This contradicts maximality.

e. We can construct a vertex cover by taking the endpoints of every edge in M. This has size $2|M|$ because the endpoints of edges in a matching are disjoint. Moreover, if any edge failed to be incident to any vertex in the cover, this edge would be in the induced subgraph of G induced by the vertices which are not in T. By part d, no such edge exists, so it is a vertex cover.

f. The greedy algorithm of part (b) yields a maximal matching M. By part (e), there exists a vertex cover of size $2|M|$. By part (c), the size of this vertex cover exceeds the sizeof a maximum matching, so $2|M|$ is an upper bound on the size of a maximum matching. Thus, the greedy algorithm is a $2|M|/|M| = 2$ approximation algorithm.

Problem 35-5

a. Suppose that the greatest processing time is p_i. Then, any schedule must assign the job J_i to some processor. If it assigns J_i to start as soon as possible, it still won't finish until p_i units have passed. This means that since the makespan is the time that all jobs have finished, it will at or after p_i because it needs job J_i to of finished. This problem can be viewed as a restatement of (27.3), where we have an infinite number of processors and assign each job to its own processor.

b. The total amount of processing per step of time is one per processor. Since we need to accomplish $\sum_i p_i$ much work, we have to of spent at lest $\frac{1}{m}$ times that amount of time. This problem is a restatement of equation (27.2).

c. See the algorithm GREEDY-SCHEDULE

Algorithm 5 GREEDY-SCHEDULE
---
for every $i = 1, \ldots, m$, let $f_i = 0$. We will be updating this to be the latest time that we have any task running on processor i.
put all of the f_i into a min heap, H
while There is an unassigned job J_i **do**
    extract the min element of H, let it be f_j
    assign job J_i to processor M_j, to run from f_j to $f_j + p_i$.
    $f_j = f_j + p_i$
    add f_j to H
end while

Since we can perform all of the heap operations in time $O(\lg(m))$, and we need to perform one for each of our jobs, the runtime is in $O(n \lg(m))$.

d. This is Theorem 27.1 and Corollary 27.2.

Problem 35-6

a. Let $V = \{a, b, c, d\}$ and $E = \{\{a, b\}, \{b, c\}, \{c, d\}, \{b, d\}\}$ with edge weights 3, 1, 5, and 4 respectively. Then $S_G = \{\{a, b\}, \{c, d\}, \{b, d\}\} = T_G$.

b. Let $V = \{a, b, c, d\}$ and $E = \{\{a, b\}, \{b, c\}, \{c, d\}\}$. Then $S_G = \{\{a, b\}, \{c, d\}\} \neq \{\{a, b\}, \{b, c\}, \{c, d\}\} = T_G$.

c. Consider the greedy algorithm for a maximum weight spanning tree: Sort the edges from largest to smallest weight, consider them one at a time in order, and if the edge doesn't introduce a cycle, add it to the tree. Suppose that we don't select some edge $\{u, v\}$ which is of maximum weight for u. This means that the edge must introduce a cycle, so there is some edge $\{w, u\}$ which is already included in the spanning tree. However, we add edges in order of decreasing weight, so $w(w, u) > w(u, v)$ because edge weights are distinct. This contradicts the fact that $\{u, v\}$ was of maximum weight. Thus $S_G \subseteq T_G$.

d. Since the edge weights are distinct, a particular edge can be the maximum edge weight for at most two vertices, so $|S_G| \geq |V|/2$. Therefore $|T_G \backslash S_G| \leq |V|/2 \leq |S_G|$. Since every edge in $T_G \backslash S_G$ is nonmaximal, each of these has weight less than the weight of any edge in S_G. Let m be the minimum weight of an edge in S_G. Then $w(T_G \backslash S_G) \leq m|S_G|$, so $w(T_G) \leq w(S_G) + m|S_G| \leq 2w(S_G)$. Therefore $w(T_G)/2 \leq w(S_G)$.

e. Let $N(v)$ denote the neighbors of a vertex v. The following algorithm APPROX-MAX-SPANNING produces a subset of edges of a minimum spanning tree by part (c), and is a 2-approximation algorithm by part (d).

Algorithm 6 APPROX-MAX-SPANNING-TREE(V, E)

1: $T = \emptyset$
2: **for** $v \in V$ **do**
3:     $max = -\infty$
4:     $best = v$
5:     **for** $u \in N(v)$ **do**
6:         **if** $w(v, u) \geq max$ **then**
7:             $v = u$
8:             $max = w(v, u)$
9:         **end if**
10:     **end for**
11:     $T = T \cup \{v, u\}$
12: **end for**
13: **return** T

Problem 35-7

a. Though not stated, this conclusion requires the assumption that at least one of the values is non-negative. Since any individual item will fit, selecting that one item would be a solution that is better then the solution of taking no items. We know then that the optimal solution contains at least one item. Suppose the item of largest index that it contains is item i, then, that solution would be in P_i, since the only restriction we placed when we changed the instance to I_i was that it contained item i.

b. We clearly need to include all of item j, as that is a condition of instance I_j. Then, since we know that we will be using up all of the remaining space somehow, we want to use it up in such a way that the average value of that space is maximized. This is clearly achieved by maximizing the value density of all the items going in, since the final value density will be an average of the value densities that went into it weighted by the amount of space that was used up by items with that value density.

c. Since we are greedily selecting items based off of the amount of value per space, we only stop once we can no longer fit any more, and each time before putting some faction of the item with the next least value per space, we complete putting in the item we currently are. This means that there is at most one item fractionally.

d. First, it is trivial that $v(Q_j)/2 \geq v(P_j)/2$ since we are trying to maximize a quantity, and there are strictly fewer restrictions on what we can do in the case of the fractional packing than in the 1-0 packing. To see that $v(R_j) \geq v(Q_j)/2$, note that among the items $\{1, 2, \ldots, j\}$, we have the highest value item is v_j because of our ordering of the items. We know that the fractional solution must have all of item j, and there is some item that it may not have all of that is indexed by less than j. This part of an item is all that is thrown out when going from Q_j to R_j. Even if we threw all of that item out, it still wouldn't be as valuable as item j which we are keeping, so, we have retained more than half of the value. So, we have proved the slightly stronger statement that $v(R_j) > v(Q_j)/2 \geq v(P_j)/2$.

e. Since we knew that the optimal solution was the max of all the P_j (part a), and we know that each P_j is at most $2R_j$(part d), by selecting the max of all the the the R_j, we are obtaining a solution that is at most a factor of two less than the optimal solution.

$$\max_j R_j \leq \max_j P_j \leq \max_j 2R_j$$

Appendix A

Michelle Bodnar, Andrew Lohr

August 27, 2017

Exercise A.1-1

$$\sum_{k=1}^{n}(2k-1) = 2\sum_{k=1}^{n}k - \sum_{k=1}^{n}1 = n(n+1) - n = n^2$$

Exercise A.1-2

Using the harmonic series formula we have that

$$\sum_{k=1}^{n}\frac{1}{(2k-1)} \leq 1 + \sum_{k=1}^{n}\frac{1}{2k} = 1 + \ln(\sqrt{n}) + O(1) = \ln(\sqrt{n}) + O(1).$$

Exercise A.1-3

First, we recall equation (A.8)

$$\sum_{k=0}^{\infty}kx^k = \frac{x}{(1-x)^2}$$

for $|x| < 1$. Then, we take a derivative of each side, taking the derivative of the left hand side term by term

$$\sum_{k=0}^{\infty}k \cdot kx^{k-1} = \frac{(1-x)^2 + 2x(1-x)}{(1-x)^4} = \frac{(1-x)+2x}{(1-x)^3} = \frac{(1+x)}{(1-x)^3}$$

Lastly, since we have a x^{k-1} instead of the x^k that we'd like, we'll multiply both sides of the equation by x to get the desired equality.

$$\sum_{k=0}^{\infty}k^2x^k = \frac{x(1+x)}{(1-x)^3}$$

Exercise A.1-4

Using formula A.8 we have

$$\sum_{k=0}^{\infty} \frac{k-1}{2^k} = -1 + \frac{1}{2} \sum_{k=0}^{\infty} k \left(\frac{1}{2}\right)^k$$
$$= -1 + \frac{1}{2} \cdot \frac{1/2}{1/4}$$
$$= -1 + 1$$
$$= 0.$$

Exercise A.1-5

First, we'll start with the equation

$$\sum_{k=1}^{\infty} y^k = \frac{y}{1-y}$$

So long as $|y| < 1$. Then, we'll let $y = x^2$ to get

$$\sum_{k=1}^{\infty} (x^2)^k = \frac{x^2}{1-x^2}$$

$$\sum_{k=1}^{\infty} x x^{2k} = \frac{x^3}{1-x^2}$$

$$\sum_{k=1}^{\infty} x^{2k+1} = \frac{x^3}{1-x^2}$$

$$\sum_{k=1}^{\infty} (2k+1) x^{2k} = \frac{3x^2(1-x^2) + 2x^4}{(1-x^2)^2}$$

$$\sum_{k=1}^{\infty} (2k+1) x^{2k} = \frac{3x^2 - x^4}{(1-x^2)^2}$$

so long as $|x| < 1$.

Exercise A.1-6

Let $g_1, g_2, \ldots, g_n$ be any functions such that $g_k(i) = O(f_k(i))$. By the definition of big-oh there exist constant $c_1, c_2, \ldots, c_n$ such that $g_k(i) \le c_k f_k(i)$. Let $c = \max_{1 \le k \le n} c_k$. Then we have

$$\sum_{k=1}^{n} g_k(i) \le \sum_{k=1}^{n} c_k f_k(i) \le c \sum_{k=1}^{n} f_k(i) = O\left(\sum_{k=1}^{n} f_k(i)\right).$$

Exercise A.1-7

$$\lg\left(\prod_{k=1}^{n} 2 \cdot 4^k\right)$$

$$= \sum_{k=1}^{n} \lg(2 \cdot 4^k)$$

$$= \sum_{k=1}^{n} \lg(2) + k\lg(4)$$

$$= \left(\lg(2)\sum_{k=1}^{n} 1\right) + \left(\lg(4)\sum_{k=1}^{n} k\right)$$

$$= n + 2\frac{n(n+1)}{2}$$

$$= n(n+2)$$

This means that we need to raise 2 to this quantity to get the desired product, so out final answer is

$$2^{n(n+2)} = 2^{n^2} \cdot 4^n$$

Exercise A.1-8

We expand the product and cancel as follows:

$$\prod_{k=2}^{n} 1 - 1/k^2 = \prod_{k=2}^{n} \frac{(k-1)(k+1)}{k^2}$$

$$= \frac{1 \cdot 3}{2 \cdot 2} \cdot \frac{2 \cdot 4}{3 \cdot 3} \cdot \frac{3 \cdot 5}{4 \cdot 4} \cdots \frac{(n-1) \cdot (n+1)}{n \cdot n}$$

$$= \frac{n+1}{2n}.$$

Exercise A.2-1

Define a function $f_1 = \lceil \frac{1}{x^2} \rceil$ and $f_2 = 1 + \frac{1}{x^2}$. Note that we always have that $f_1 \le f_2$. Then we have that the desired summation is exactly equal to $\int_1^\infty f_1$ because the graph of f_1 is a bunch of rectangles of width 1 and height equal to each of the terms in the sum. By monotonicity of integrals, we have that this is $\le \int_1^\infty f_2 = 2$.

Exercise A.2-2

When $n = 2^m$ the sum becomes $n + n/2 + n/4 + \ldots + 1 = 2n - 1 = O(n)$. There always exists a power of 2 which lies between n and $2n$ for any choice of n, so let n' denote the smallest power of 2 which is greater than or equal to n. Then we have

$$\sum_{k=0}^{\lfloor \lg n \rfloor} \lceil n/2^k \rceil \leq \sum_{k=0}^{\lfloor \lg n' \rfloor} \lceil n'/2^k \rceil = 2n' - 1 \leq 4n - 1 = O(n).$$

Exercise A.2-3

Similar to the derivation of (A.10), we split up the interval $[n]$ into $\lfloor \lg(n) \rfloor - 1$ pieces, with the ith starting at $1/2^i$ and going to $1/2^{i+1}$. So, we have

$$\sum_{k=1}^{n} \frac{1}{k} \geq \sum_{i=0}^{\lg(n)-1} \sum_{j=0}^{2^i-1} \frac{1}{2^i + j}$$

$$\geq \sum_{i=0}^{\lg(n)-1} \frac{1}{2^{i+1}}$$

$$= \sum_{i=0}^{\lg(n)-1} \frac{1}{2}$$

$$= \frac{1}{2} \lg(n)$$

Which gets us that the nth harmonic number is $\Omega(\lg(n))$.

Exercise A.2-4

Since k^3 is monotonically increasing we use bound A.11. For the upper bound we have

$$\sum_{k=1}^{n} k^3 \leq \int_{1}^{n+1} x^3 dx$$

$$= \frac{x^4}{4} \Big|_{1}^{n+1}$$

$$= \frac{(n+1)^4 - 1}{4}.$$

For the lower bound we have

$$\sum_{k=1}^{n} k^3 \geq \int_{0}^{n} x^3 dx$$

$$= \frac{x^4}{4} \Big|_{0}^{n}$$

$$= \frac{n^4}{4}.$$

4

Exercise A.2-5

If we were to apply the integral approximation given in (A.12) directly to the sum, then we would be trying to evaluate the integral

$$\int_0^n \frac{dx}{x}$$

Which is an improper integral that doesn't have a finite value.

Problem A-1

a. Applying the integral approximation to this, we get that

$$\int_0^n x^r dx \le \sum_{k=1}^n k^r \le \int_1^{n+1} x^r dx$$

$$\frac{n^{r+1}}{r+1} \le \sum_{k=1}^n k^r \le \frac{(n+1)^{r+1} - 1}{r+1}$$

So, the given sum is $n^{r+1}(\frac{1}{r+1} + o(1))$.

b. We will first show that we can approximate the integral well by $g(x) = x\ln(x)^s(\frac{1}{\ln(2)^s} + f(x))$ where $f(x)$ is $O(1/\ln(x))$. The derivative of the RHS is $\ln(x)^s(\frac{1}{\ln(2)^s} + f(x)) + \ln(x)^{r-1}(\frac{1}{\ln(2)^s} + f(x)) + x\ln(x)^s f'(x)$. Since $f(x)$ is going to zero at least as fast as $\frac{1}{\ln(x)}$, we have that $f'(x) \in O(\frac{1}{x\ln(x)^2})$, this means that $\frac{d}{dx}g(x) = \lg(x)^s(1 + O(1/\ln(x)))$. So, since the integral of the derivative is the original function,

$$g(x) = \int_1^x \lg(x)^s(1 + O(1/\ln(x)))dx$$
$$= \int_1^x \lg(x)^s dx + \int_1^x O(\ln(x)^{r-1}))dx$$
$$= \int_1^x \lg(x)^s dx + O(x\ln(x)^{r-1}))$$

but this second term is insignificant enough to be absorbed into our $f(x)$ in how we defined $g(x)$. Then, since we can bound the sum above and below by the integral, just with shifted endpoints, and the derivative of the main term of the integral is small enough to fit in the remainder term, shifting it by one will not cause it's value to change asymptotically. So, we have that the sum is $n\ln(n)^s(\frac{1}{\ln(2)^s} + O(\frac{1}{\ln(n)})) = n\lg(n)^s(1 + O(\frac{1}{\ln(n)})$ which is asymptotically tight.

5

c. For this problem we show that the integral is well approximated by $g(n) = n^{r+1}/(r+1)\lg(n)^s(1 + O(\frac{1}{\ln(n)}))$. Then, we have $g'(x) = n^r\lg(n)^s(1 + O(\frac{1}{\ln(n)})) + n^r/(r+1)\lg(n)^{s-1}(\frac{1}{\ln(2)} + O(\frac{1}{\ln(n)})) + n^r\lg(n)^sO(\frac{1}{n\ln(n)^2})) = n^r\lg(n)^s + O(n^r\lg(n)^{s-1})$. So, the exact same approximation goes through as in the previous problem. So, we get that the sum is asymptotically equal to

$$\frac{n^{r+1}\lg(n)^s}{n+1}\left(1 + O\left(\frac{1}{\ln(n)}\right)\right)$$

Appendix B

Michelle Bodnar, Andrew Lohr

April 12, 2016

Exercise B.1-1
FIrst we consider

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$$
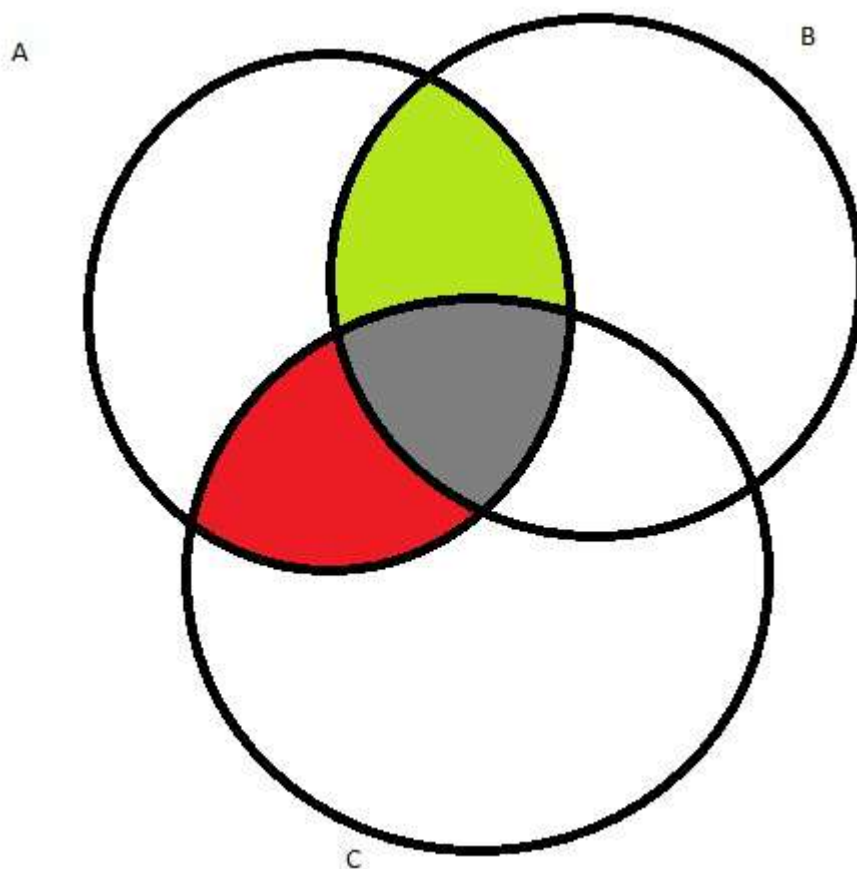
For the first picture, we can see that the shaded regions are all the regions that are in A and also in either B or C, and so are in the set described on the left hand side. Since the green and gray shaded regions are in both A and B,

1

they are in the right hand side, also, the red and gray regions are in A and C and so are also in the right hand side. There aren't any other regions that are either both in A and B or both in A and C, so the shaded regions are the right hand side of the equation as well.

Next we consider

$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$$
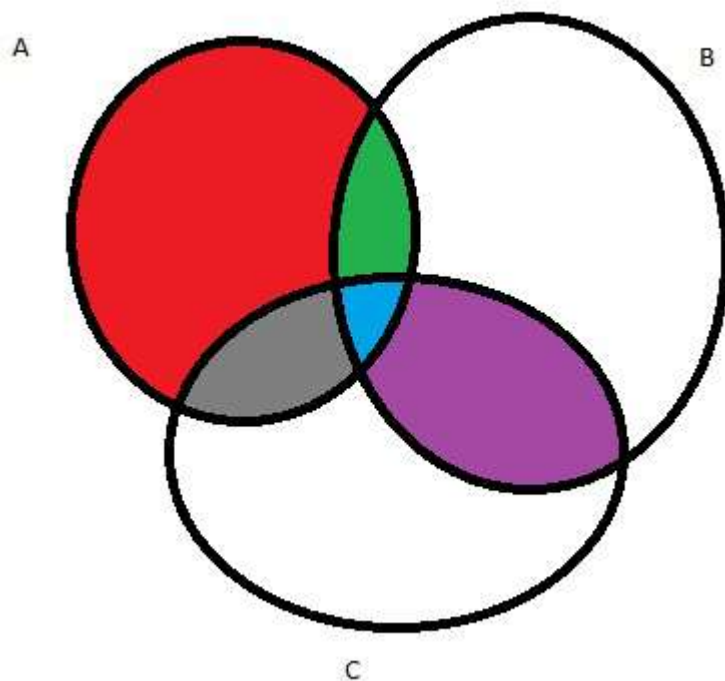
The ony regions in $B \cap C$ are blue and purple. All the other colored regions are in A. So, the shaded regions are exactly the left hand side. To see what is in the right hand side, we see what is both in $A \cup B$ and $A \cap C$. Individually both both contain all of the shaded regions plus one of the white regions. Since the white region contained differs, their intersection is all of the shaded regions.

Exercise B.1-2

We'll proceed by induction. The base case has already been taken care of for us by B.2. Suppose that the claim holds for a collection of n sets. Then we have

$$\begin{aligned}
\overline{A_1 \cap A_2 \cap \cdots \cap A_n \cap A_{n+1}} &= \overline{(A_1 \cap A_2 \cap \cdots \cap A_n) \cap A_{n+1}} \\
&= \overline{A_1 \cap A_2 \cap \cdots \cap A_n} \cup \overline{A_{n+1}} \\
&= \overline{A_1} \cup \overline{A_2} \cup \cdots \cup \overline{A_n} \cup \overline{A_{n+1}}.
\end{aligned}$$

2

An identical proof with the roles of intersection and union swapped gives the second result.

Exercise B.1-3

If we are taking the union over only two sets, then we are exactly in the case of equation (B.3).

Now, proceed by induction. Suppose $n > 2$, we can write $B = (A_1 \cup A_2 \cup \cdots \cup A_{n-1})$. Then, we have

$$
\begin{aligned}
|A_1 \cup A_2 \cup \cdots \cup A_n| &= |B \cup A_n| \\
&= |B| + |A_n| - |B \cap A_n| \\
&= |B| + |A_n| - |B \cap A_n| \\
&= |B| + |A_n| - |(A_1 \cup A_2 \cup \cdots A_{n-1}) \cap A_n| \\
&= |A_1 \cap A_2 \cap \cdots A_{n-1}| + |A_n| - |(A_1 \cap A_n) \cup (A_2 \cap A_n) \cup \cdots (A_{n-1} \cap A_n)| \\
&= \left(\sum_{\emptyset \neq S \subseteq \{1,2,\ldots,n-1\}} (-1)^{|S|-1} |\cap_{i \in S} A_i| \right) + |A_n| \\
&\qquad\qquad - \left(\sum_{\emptyset \neq S \subseteq \{1,\ldots,n-1\}} (-1)^{|S|-1} |A_n \cap (\cap_{i \in S} A_i)| \right) \\
&= \left(\sum_{\emptyset \neq S \subseteq \{1,2,\ldots,n-1\}} (-1)^{|S|-1} |\cap_{i \in S} A_i| + (-1)^{|S|} |A_n \cap (\cap_{i \in S} A_i)| \right) + |A_n| \\
&= \left(\sum_{\emptyset S \subseteq \{1,2,\ldots,n\}, S \neq \emptyset, S \neq \{n\}} (-1)^{|S|-1} |\cap_{i \in S} A_i| \right) + |A| \\
&= \left(\sum_{\emptyset S \subseteq \{1,2,\ldots,n\}, S \neq \emptyset} (-1)^{|S|-1} |\cap_{i \in S} A_i| \right)
\end{aligned}
$$

Where to go from line 7 to line 8, we used the fact that every subset of $\{1,\ldots,n\}$ can be counted as a subset of $\{1,\ldots,n-1\}$, and then either containing or not containing the nth element. Since the empty subset of $\{1,\ldots,n-1\}$ then corresponds both to the empty set and the singleton set $\{n\}$ under this correspondence, explaining why that value of S is dropped from the sum on line 8.

Exercise B.1-4

Let $f(k) = 2k + 1$. Then f is a bijection from $\mathbb{N}$ to the set of odd natural numbers, so they are countable.

Exercise B.1-5

For each of the elements of S, some subset of S can either contain that element or not contain that element. If the decision differs for any of the $|S|$ many elements, then you've just created a distinct set. Since we make a decision between two options $|S|$ many times, the total number of possible sets is $2^{|S|}$. It can be thought of as the number of leaves in a complete binary tree of depth $|S|$.

Exercise B.1-6

We define an n-tuple recursively as $(a_1, a_2, \ldots, a_n) = ((a_1, a_2, \ldots, a_{n-1}), a_n)$.

Exercise B.2-1

To see that it is a partial ordering, we need to show it is reflexive, antisymmetric, and transitive. To see that it is reflexive, we need to show $S \subseteq S$. That is, for every $x \in S$, $s \in S$, which is a tautology. To see that it is antisymmetric, we need that if $S_1 \neq S_2$ and $S_1 \subseteq S_2$ then $S_2 \nsubseteq S_1$. Since $S_1 \neq S_2$ there is some element that is in one of them but not in the other. Since $S_1 \neq S_2$, we know that that element must be in S_2 because if it were in S_1 it would be in S_2. Since we have an element in S_2 not in S_1, we have $S_2 \nsubseteq S_1$. Lastly, we show transitivity, suppose $S_1 \subseteq S_2 \subseteq S_3$. This means that any element that is in S_1 is in S_2. But since it is in S_2, it is in S_3. Therefore, every element in S_1 is in S_3, that is $S_2 \subseteq S_3$.

To see that it is not a total order, consider the elements $\{1, 2\}$ and $\{2, 3\}$. Neither is contained in the other, so there is no proscribed ordering between the two based off of inclusion. If it were a total ordering, we should be able to compare any two elements, which we just showed we couldn't.

Exercise B.2-2

For all positive integers a we have $a - a = 0 \cdot n$ so the relation is reflexive. If $a - b = qn$ then $b - a = (-q)n$ so the relation is symmetric. If $a - b = qn$ and $b - c = pn$ then $a - c = a - b + b - c = (q + p)n$ so the relation is transitive. Thus, equivalence modulo n is an equivalence relation. This partitions the integers into equivalence classes consisting of numbers which differ by a multiple of n.

Exercise B.2-3

a. Consider the set of vertices in some non-complete, non-empty graph where we make two vertices related if they are adjacent or the same vertex.

b. Consider the vertices in a digraph where we have aRb if there is some possibly empty path from a to b. For a concrete example, suppose we have only the set $\{0, 1\}$ and the relations $0R0$, $0R1$, and $1R1$.

c. Consider the relation that makes no two elements related. This satisfies both

the symmetric and transitive properties, since both of those require that certain elements are related to conclude that some particular other elements are related.

Exercise B.2-4

Suppose that aRb. Since R is an equivalence relation it is symmetric, so bRa. Since R is antisymmetric, aRb and bRa imply that $a = b$. Thus every equivalence class is a singleton.

Exercise B.2-5
Professor Narcissus is full of himself! The relation that makes no two elements related to each other is both symmetric and transitive, but is not reflexive.

Exercise B.3-1

a. Since f is injective, for every element in it's range, there is at most one element that maps to it. So, we proceed by induction on the number of elements in the domain of the function. If there is only a single element in the domain, since the function has to map to something, we have that $|B| \geq 1 = |A|$. Suppose that all functions that are on a domain of size n satisfy this relation. Then, look at where that $n + 1$st element gets mapped. This point will never be mapped to by any of the other elements of the domain. So, we can think of restricting the function to just the first n elements, and use the inductive assumption to get the desired relation of the two sets.

b. As a base case, assume that $|B| = 1$. Then, since the function is surjective, some element must map to that, so, $|A| \geq 1 = |B|$. Not, suppose it is true for all function with a range of size at most n. Then, just look at all the elements that map to the $n+1$st element, there is at least one by surjectivity. This gets us that $|A| \geq 1 + |A'| \geq 1 + |B'| = |B|$.

Exercise B.3-2

When the domain and codomain are $\mathbb{N}$, the function $f(x) = x + 1$ is not bijective because 0 is not in the range of f. It is bijective when the domain and codomain are $\mathbb{Z}$.

Exercise B.3-3
Define R^{-1} by $aR^{-1}b$ if and only if bRa. This clearly swaps the domain and range for the relation, and so, if R was bijective, it is the inverse function. If R was not bijective, then R^{-1} might not even be a function.

Exercise B.3-4

It is easiest to see this bijection pictorally. Imagine drawing out the elements of $\mathbb{Z} \times \mathbb{Z}$ as points in the plane. Starting from $(0,0)$, move up one unit to $(0,1)$, then right one unit to $(1,1)$, down 2 units through $(1,0)$ to $(1,-1)$, then left, and so on, continuing in a spiraling fashion, hitting each point exactly once, not skipping any as you move outwards. If point (i,j) is the kth point which is hit, then let $f(i,j) = (-1)^k \lceil k/2 \rceil$. This gives a bijection from $\mathbb{Z} \times \mathbb{Z}$ to $\mathbb{Z}$, which implies that f has an inverse g. The function g is the desired bijection.

Exercise B.4-1
Let $f(u,v)$ be equal to 1 if u shook hands with v. Then, $\sum_{v \in V} degree(v) = \sum_{v \in V} \sum_{u \in V} f(v,u)$ then, since handshaking is symmetric, we are counting both when u shakes v hand and when v shakes u's hand. So, this sum is $\sum_{e \in E} 2 = 2|E|$.

Exercise B.4-2

Suppose that $u = v_0, v_1, \ldots, v_k = v$ is a path p from u to v. If it is not simple, then it contains a cycle, so there exist i and j such that $v_i = v_j$. Let p' be the path on vertices $v_0, v_1, \ldots, v_{i-1}, v_j, \ldots, v_k$. This is a path from u to v which contains at least one fewer cycles than before. We can continue this process until the path contains no cycles. Similarly, suppose a directed graph contains a cycle $v_0, v_1, \ldots, v_k$. If it is not simple, then there exist i and $j \neq k$ such that $v_i = v_j$. Remove the vertices $v_i, v_{i+1}, \ldots, v_{j-1}$ from the cycle to obtain a cycle with at least one fewer duplicate vertices. Continuing with this process will eventually produce a cycle with no repeated vertices except the first and last, so it will be simple.

Exercise B.4-3
We proceed by induction on the number of vertices. If there is a single vertex, then the inequality trivially holds since the left hand side is the size of a set and the right hand side is zero. For the inductive case, pick one vertex in particular. We know that that vertex must have at least one edge to the rest of the vertices because the graph is connected. So, when we take the induced subgraph on the rest of the vertices, we are decreasing the number of edges by at least one. So, $|E| \geq 1 + |E'| \geq 1 + |V'| - 1 = |V| - 1$.

Exercise B.4-4

Every graph is reachable from itself by the empty path so the relation is reflexive. If v is reachable from u then there exists a path $u = v_0, v_1, \ldots, v_k = v$. Thus, $v_k, v_{k-1}, \ldots, v_0$ is a path from v to u, so u is reachable from v and the relation is symmetric. If v is reachable from u and w is reachable from v then by concatenation of paths, w is reachable from u, so the relation is transitive. Therefore the "is reachable from" relation is an equivalence relation.

Exercise B.4-5

The undirected version of the graph in B.2(a) is on the same vertex set, but has $E = \{(1,2),(2,4),(2,5),(4,1),(4,5),(6,3)\}$. That is, we threw out the antisymmetric edge, and the self edge. There is also a difference in that now the edges should be viewed as unordered unlike before.
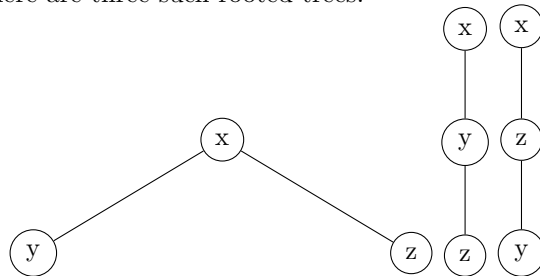
The directed version of B.2(b) looks the same, except it has arrows drawn in. One from 2 to 1, one from 1 to 5, one from 2 to 5, and one from 3 to 6.
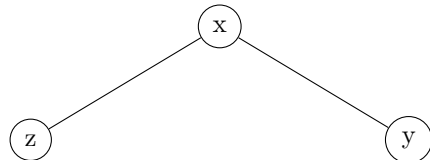
Exercise B.4-6

We create a bipartite graph as follows: Let V_1 be the set of vertices of the hypergraph and V_2 be the set of hyperedges. For each hyperedge $e = \{v_1, v_2, \ldots, v_k\} \in V_2$, draw edges (e, v_i) for $1 \leq i \leq k$.

Exercise B.5-1
There are three such rooted trees:

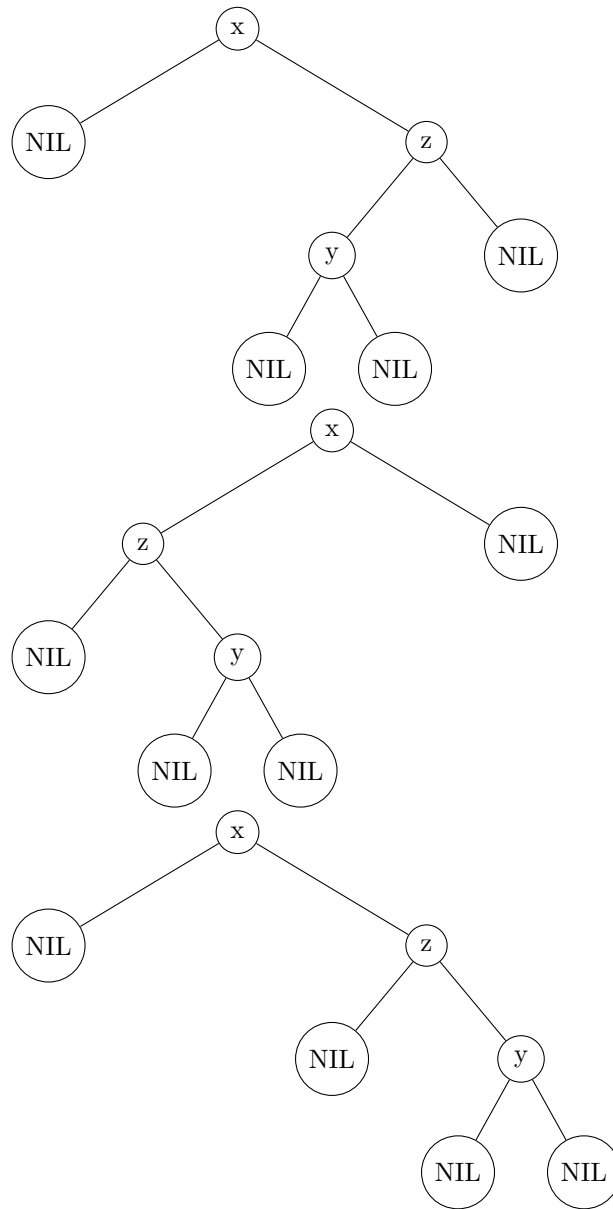The ordered trees are those listed above, in addition to the tree

The Binary trees are all ten of the following:
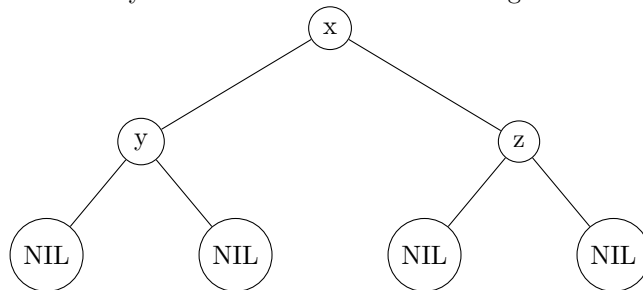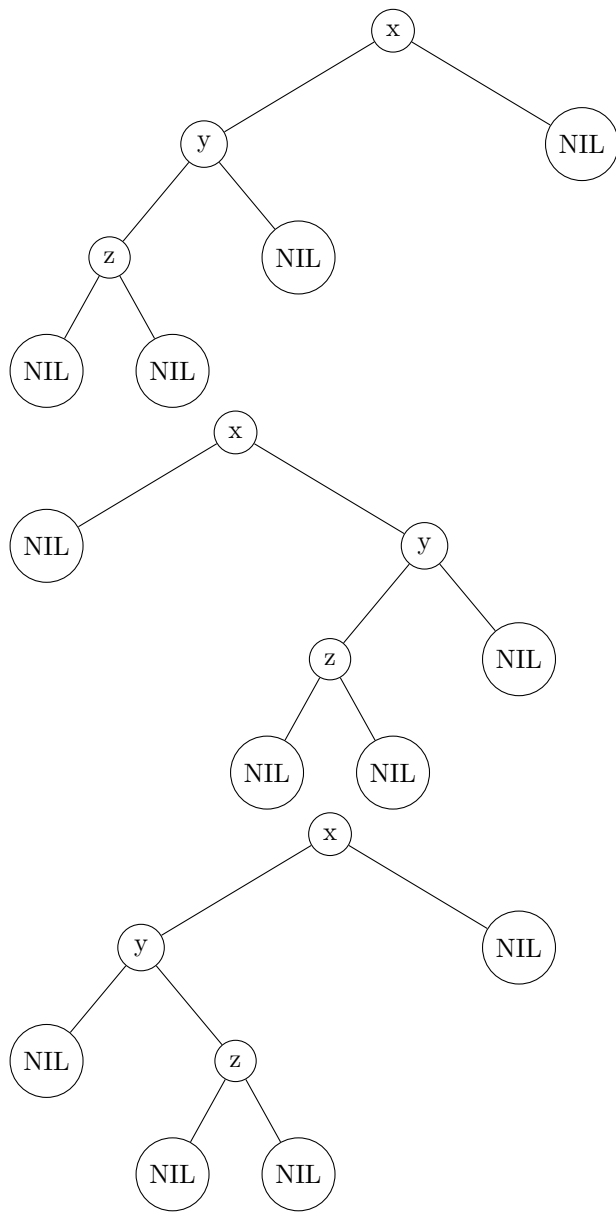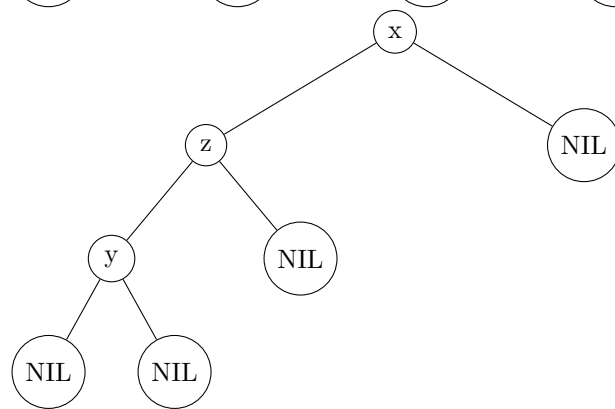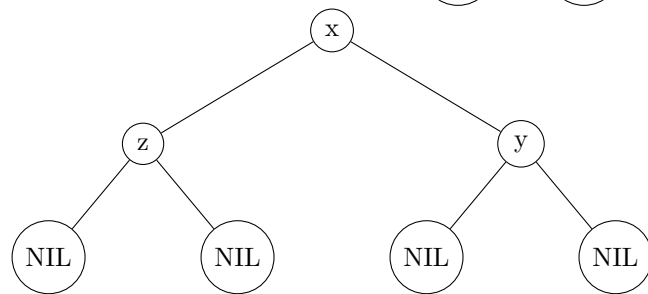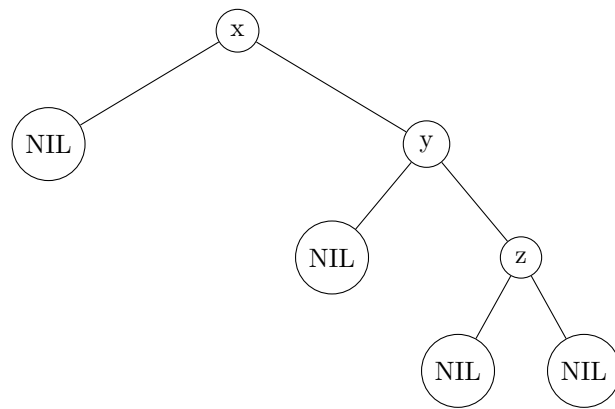
Exercise B.5-2

Suppose vertex u is not on the unique path from v_0 to v and v is not on the unique path from v_0 to u. Then there can't be an edge from v to u or from u to v, otherwise we would violate uniqueness. This implies that in the undirected version of G when we remove the arrows from the edges, there is still a unique path from v_0 to every vertex. This implies that there exists a path between every pair of vertices. Suppose the path from u to v is not unique. Then there

must be a path p from u to v which does not contain v_0. Thus, the unique path from v_0 to v differs from the path obtained by going from v_0 to u, then taking p. This is a contradiction, because the path from v_0 to v is unique. By property 2 of Theorem B.2, G is a free tree.

Exercise B.5-3

As a base case, consider the binary tree that consists of a single node. This tree has no degree two nodes and one leaf, so, the number of degree two nodes is one less than the number of leaves. As an inductive step, suppose it is true for all binary trees with at most n degree two nodes. Then, let G be a binary tree with $n + 1$ internal nodes. Let v be the first node, possibly the leaf itself, that is the child of a degree two node, and can be obtained by taking the parent pointer from the leaf. Then, consider the tree obtained by removing v and all its children. Doing so removes only one leaf, and it makes the parent of v drop to being degree 2. None of the children of v were degree 2 because otherwise we would of stopped earlier when we were selecting v. Since we have decreased the number of degree two nodes and leaves both by 1, we have completed the inductive case, because if $|T'|$ is the modified tree, the number of leaves in T is one more than that in T' which means it is one more than the number of degree 2 nodes in T', which is the number of degree two nodes in T.

Since a full binary tree has each internal node with degree two, this result gets us that the number of internal nodes in a full binary tree is one more than the number of leaves.

Exercise B.5-4

A tree with 1 node has height at least 0, so the claim holds for $n = 1$. Now suppose the claim holds for n. Let T be a binary tree with $n + 1$ nodes. Select a leaf node and remove it. By our induction hypothesis, the resulting tree has height at least $\lfloor \lg n \rfloor$. If $n + 1$ is not a power of 2 then this is equal to $\lfloor \lg(n+1) \rfloor$, so we are done. Otherwise, choose a leaf of greatest depth to remove. The resulting tree has height at least $\lfloor \lg n \rfloor$. If the height in fact achieves that, then the only possible tree is the complete binary tree on n vertices. Since every internal node has two children, the only place the removed leaf could have come from is from a leaf vertex. Adding a child to any leaf vertex increases the height of the tree by 1. Since $\lfloor \lg n \rfloor + 1 \geq \lfloor \lg(n + 1) \rfloor$, the claim holds.

Exercise B.5-5

We will perform structural induction. For the empty tree that has $n = 0$, the equation is obviously satisfied. Now, let r be the root of the tree T, and let T_L and T_R be the left and right subtrees respectively. By the inductive hypothesis, we may assume that $e_L = i_L + 2n_L$ and $e_R = i_R + 2n_R$. Then, since being placed as a child of the root adds one to the depth of each of the nodes in both

T_L and T_R, we have that $i = i_L + i_R + n_L + n_R$ and

$$
\begin{aligned}
e =& e_L + |leaves(T_L)| + e_R + |leaves(T_R)| \\
=& e_L + e_R + |leaves(T)| \\
=& i_L + 2n_L + i_R + 2n_R + |leaves(T)| \\
=& i + n_L + n_R + |leaves(T)| \\
=& i + n - 1 + |leaves(T)|
\end{aligned}
$$

By problem B.5-3, since the tree is full, we know that $|leaves(T)|$ is one more than the number of internal nodes of T. So, $e = i+2n$, completing the induction.

Exercise B.5-6

We'll proceed by strong induction on the number of nodes in the tree. When $n = 1$, there is a single leaf which has depth 0, so we have $\sum_{x \in L} w(x) = 2^{-0} = 1$. Now suppose the claim holds for a tree with at most n nodes and let T be a tree on $n+1$ nodes. If the left or right subtree of T is empty then the induction hypothesis tells us that the subtree on the nonempty side satisfies the inequality with respect to depth in that tree. Since the depth in the original tree is one greater for each leaf, the claim holds. On the other hand, if T has left and right children, call the subtrees rooted at the children T_1 and T_2. By the induction hypothesis, the sums of the weights of their leaves are each less than or equal to 1. Since the depth of a node in T is one greater than its depth in T_1 or T_2, the weight of each leaf in T is halved. Thus, the sum of the weights of the leaves in each subtree is bounded by $1/2$, so the total sum of weights of leaves is bounded by 1.

Exercise B.5-7

Suppose to a contradiction that there was some binary tree with more than 2 leaves that had no subtree with a number of leaves in the desired range. Since $L > 1$, the root is not a leaf. Now, define the sequence $x_0 = root$, x_{i+1} is the larger(more leaves) of the children of x_i, until we have reached a root. Now, we consider the number of leaves in the subtree rooted at each x_i. For x_0 it is L, which is too large, and at x_h, it is 1 which is not too large. Now, we keep incrementing from x_0 to x_1 to x_2, and so on until we have some number of leaves that falls in the desired range. If it happens we are done and have contradicted the assumption that this binary tree didn't have such a subtree. Since it doesn't that means there is some step where the number of leaves jumps from more than $2L/3$ to less than $L/3$, Since both of the children of the next x_i were less than $L/3$, their sum cannot be more than $2L/3$, a contradiction.

Problem B-1

a. If the tree is unrooted, pick an arbitrary node to be the root. Assign color 0 to all nodes that are at an even height, and assign color 1 to all nodes that

are at an odd height. Since the child of any node has height one greater, there will never be two adjacent nodes that have received the same color.

b. We show the following implications in order to get equivalence of all three

$1 \Rightarrow 2$ Since the graph is bipartite, we can partition the vertices into two sets L and R so that there are no edges going between vertices in L and no edges going between vertices in R. This means that we can assign color 0 to all vertices in L and color 1 to all vertices in R without having any edges going between two vertices of the same color.

$2 \Rightarrow 3$ Suppose to a contradiction that there was a cycle of odd length, but we did have a valid two coloring. As we go along the cycle, each time we go to the next vertex, the color must change because no two adjacent vertices can have the same color. If we go around the cycle like this though, we have just flipped the color an odd number of times, and have returned back to the original color, a contradiction.

$3 \Rightarrow 2$ If G has no cycles of an odd length, then we can just perform the greedy operation to two color. That is, we pick a vertex, color it arbitrarily, then, for any vertex adjacent to a colored vertex, we color it the opposite color. If this process doesn't end up coloring everything, i.e. the graph is disconnected, we repeat it. Since the only way this process could fail is if there is an odd length cycle, it provides a two coloring, proving that the graph is two colorable.

$2 \Rightarrow 1$ Partition the vertices based on what color they received. Since there are no edges going between the vertices of the same color, there won't be any edges going between vertices that are in the same part in the partition.

c. Consider the process where we pick an arbitrary uncolored vertex and color it an arbitrary color that is not the color of any of its neighbors. Since a vertex can only have at most d neighbors, and there are $d+1$ colors to choose from, this procedure can always be carried out. Also, since we are maintaining at each step that there are no two adjacent vertices that have been colored the same, we have that the end result of all the coloring is also valid.

d. Let V' be the set of vertices whose degree is at least $\sqrt{|E|}$. The total number of edges incident to at least one of these vertices is at least $\frac{|V'|\sqrt{|E|}}{2}$ by exercise B.4-1. Since this also has to be bounded by the total number of edges in the graph, we have that $\frac{|V'|\sqrt{|E|}}{2} \leq |E|$, which gets us that $|V'| \leq 2\sqrt{|E|}$. So, we'll assign all of the vertices in V' their own distinct color. Then, so long as we color the rest of the edges with colors different from those used to color V', the edges between V' and the rest of the vertices won't be important for affecting the validity of the coloring. So, we look at the graph induced on the rest of the vertices. This graph is degree at most $\sqrt{|E|}$ because we already removed all the vertices of high degree. This means that by

13

the previous part, we can color it using at most $\sqrt{|E|} + 1$ colors. Putting it together, the total number of colors used to obtain a valid coloring is $2\sqrt{|E|} + \sqrt{|E|} + 1 \in O(\sqrt{|E|}) = O(\sqrt{|V|})$.

Problem B-2

a. Any undirected graph with at least two vertices contains at least two vertices of the same degree. Proof: Suppose every vertex has a different degree. Since the degree of a vertex is bounded between 0 and $n - 1$, the degrees of the vertices must be exactly $0, 1, \ldots, n - 1$. However, if some vertex has degree 0 then no vertex can have degree $n - 1$. Thus, some pair of vertices must have the same degree.

b. Every undirected graph on 6 vertices contains 3 vertices which are all connected to one another or 3 vertices among which there are no edges. Proof: Suppose that we have a graph which doesn't have this property. We'll show such a graph cannot exist. If vertex 1 has degree at least 3, then there can be no edges among these vertices connected to 1. Otherwise there would be a triangle. However, if there are no edges among these vertices then there are at least 3 among which there are no edges. Thus vertex 1 must have degree at most 2. Since there was nothing special about this vertex, the same argument tells us that every vertex must have degree at most 2. Now consider the graph $G' = (V, E')$ where $E' = \{(u, v) | (u, v) \notin E\}$. Observe that G has no 3 mutually connected or mutually disconnected vertices if and only if G' does, so the same argument tells us that every vertex of G' has at most degree 2, which implies that ever vertex of G has degree at least 4, a contradiction.

c. The vertex set V of any undirected graph can be partitioned into $V = V_1 \sqcup V_2$ such that at least half of the neighbors of each $v \in V_1$ are in V_2, and at least half of the neighbors of each $v \in V_2$ are in V_1. Proof: Consider an abritrary partition $V_1 \sqcup V_2$. For each edge (u, v) where $u \in V_1$ and $v \in V_2$, do the following: If u and v already have the property that more than half of their neighbors are in the opposite partition, do nothing. If both u and v fail to have this property, swap which partition u and v are in. Now suppose just one vertex fails. Without loss of generality, suppose it is u. If v has at least one more than half its neighbors in V_1, simply move u into V_1. Otherwise, swap u and v. Each time we do this for an edge (u, v), the number of edges from V_1 to V_2 strictly increases. For an edge (u, v) with u and v in the same partition, if either u or v has more than half its neighbors in its own partition, move it to the other partition. If both do, just move u. Again, this will strictly increase the number of edges between V_1 and V_2. Keep picking edges and repeating this process. Since the number of edges between V_1 and V_2 cannot increase indefinitely, this process will eventually stop, and every vertex will

have the property that at least half its neighbors are in the opposite partition.

d. If every vertex of an undirected graph has degree at least $|V|/2$ then there exists a way to draw the graph where the vertices all lie on a circle, and vertices next to one another on the circle are always connected by an edge. Proof: Let $|V| = n$. It is easy to check that the claim holds from $n = 1, 2, 3$. We'll proceed by induction on n. Suppose the claim holds for all graphs on $\leq n$ vertices and let G be a graph on $n+1$ vertices such that each vertex has degree at least $(n+1)/2$. If n is odd, consider the arrangement of the induced subgraph on n vertices. There must exist some pair of adjacent vertices both of which are connected to the $(n+1)^{st}$ vertex by an edge, so we may insert it there. If n is even, it could be the case that vertex $n+1$ is connected to every other vertex in the arrangement on n vertices. In this case, select a pair of vertices which are separated by 2 edges on the circle, both of which are connected to $n+1$. Replace the vertex v between them by $n+1$. Excluding vertex $n+1$, its two adjacent vertices (on the circle), and v, there are $n-3$ remaining vertices. Since n is even, $n-3$ is odd. Since v is connected to at least $(n-3)/2$ of these, there must exist 2 which are adjacent on the circle. We can safely place v inbetween these to obtain the desired graph.

Problem B-3

a. Suppose that $n \geq 4$, as all smaller binary trees can be easily enumerated and this fact checked for. Start at the root, let that be x_0, then, let x_{i+1} be the larger child of x_i, or it's only child if there is just one. Eventually this process will stop once it has reached a root. Let $s(v)$ be the size of the subtree rooted at v. Since we always pick the larger of the two subtrees, we have that $s(x_i) \leq 2s(x_{i+1}) + 1$. So, if we have that $s(x_{i+1}) \leq n/4$, then, we have that $s(x_i) \leq 2n/4 + 1 \leq 3n/4$. Since eventually this sequence goes to 1, which is below the range, and it starts at n, which is above, we must have that at some point it dips below, the parent of this node is the one that we need to snip off of the original tree to get the desired size subtree.

b. Take a binary tree on four vertices as follows:
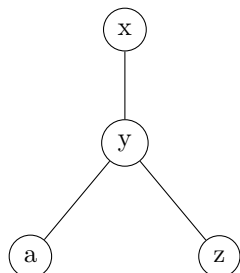
Where it is unimportant whether y is the left or right child of x. Then, any cut that is made of the three edges in the graph results in there being

one set of vertices of size 3 that are connected and one vertex that is all by its lonesome self. This means that the larger of the two sets the vertices is partitioned into has size $3 = (3/4) \cdot n$.

c. We will make a single cut to the original tree to take off a piece that is less than or equal to $\lfloor \frac{n}{2} \rfloor$. As in part a, we let x_0 be the root, and let x_{i+1} be the larger child of x_i. We show that the size of the subtree rooted at x_i, say $s(x_i)$ can old drop by at most a factor of 3. To do this, we use the fact that $s(x_i) \le 2s(x_{i+1}) + 1$. So, if $s(x_{i+1}) \le \frac{s(x_i)}{3}$, then,

$$s(x_i) \le 2s(x_{i+1}) + 1 \le \frac{2s(x_i)}{3} + 1$$
$$\frac{s(x_i)}{3} \le 1$$
$$s(x_i) \le 3$$

However, for there to even be a x_{i+1}, it must have size at least one, so, even then, we have that has dropped by at most a factor of 3.

This means that we can always select a tree that is less than a fact of three below $\lfloor \frac{n}{2} \rfloor$. So, what we do is cut off that subtree, decrease the amount we are trying to cut off by that amount, and then continue. Each cut doing this procedure must decrease the most significant digit of the ternary representation of the amount need to be cut off by 1. This means that it needs at most $2\log_3(n)$ many cuts, but this is $O(\lg(n))$, so, we are done.

Appendix C

Michelle Bodnar, Andrew Lohr

December 7, 2017

Exercise C.1-1

Since a substring is indexed by a pair of integers i, j such that $1 \leq i \leq j \leq n$. Since we want the length of the substring to be k, then, $j - i = k - 1$. Rearranging this, we can solve for j and plug it into the original set of inequalities to get that we need $1 \leq i$ and $i \leq n - k - 1$. So, i can be precicely any of $\{1..n - k + 1\}$. So, there are $n - k + 1$ substrings of length k. To get the total number of substrings, we can just add this formula over all k. Here we are assuming that a string needs to be nonempty, a point that was left undeclared in the definition given in this section.

$$\sum_{k=1}^{n} n - k + 1 = n^2 - \frac{n(n+1)}{2} + n$$
$$= \frac{n(n+1)}{2}$$
$$= \binom{n+1}{2}$$

Such a lovely formula deserves a combinatorial explaination instead of just a bunch of algebraic symbol pushing. And it turns out that we don't need to look far. Given a string of length n, and pair of distinct indices in $\{1..n+1\}$, we get a uniques subtring starting at the first index and ending the position before the second index. Put in other words, pairs of these indices are in bijection with substrings. Since there are n indices to begin with, there are $\binom{n+1}{2}$ pairs of indices picked from $\{1..n+1\}$, and so $\binom{n+1}{2}$ substrings.

Exercise C.1-2

There are $2^{(2^n)}$ n-input, 1-output boolean functions and $(2^m)^{(2^n)}$ n-input, m-output boolean functions.

Exercise C.1-3

First seat one of the professors, then there are $n-1$ choices for the person sitting immediately to his right. Then, $n-2$ choices for the person immediately to that person's right. Continue this until there is only a single person left. Multiplying together all of these choices, we get that there are $(n-1)(n-2)\cdots 1 = (n-1)!$ ways.

Exercise C.1-4

The sum of three numbers is even if and only if they are all even, or exactly two are odd and one is even. For the first case, there are $\binom{49}{3}$ ways to pick them. For the second case, there are $\binom{50}{2}\binom{49}{1}$ ways. Thus, the total number of ways to select 3 distinct numbers so that their sum is even is

$$\binom{49}{3} + \binom{50}{2}\binom{49}{1}.$$

Exercise C.1-5

We'll just do an algebraic proof, even though "nicer" combinatorial proofs do exist

$$\begin{aligned}
\binom{n}{k} &= \frac{n!}{k!(n-k)!} \\
&= \frac{n}{k}\frac{(n-1)!}{(k-1)!((n-1)-(k-1))!} \\
&= \frac{n}{k}\binom{n-1}{k-1}
\end{aligned}$$

Exercise C.1-6

We can prove this directly:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n}{n-k}\frac{(n-1)!}{k!(n-k-1)!} = \frac{n}{n-k}\binom{n-1}{k}.$$

Exercise C.1-7

Distinguish one of the objects. In the case that you do select this object, then you only need to pick $k-1$ more objects from the remaining $n-1$ items, which there are $\binom{n-1}{k-1}$ ways of doing. If you do not select the distinguished object, you still need to pick all k items, and there are only $n-1$ items left to pick from, since you know you can't pick the distinguished object. This gets us $\binom{n-1}{k}$ ways. Since these two cases are disjoint and cover all possibilities, we have that

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

Exercise C.1-8

The following shows Pascal's triangle with n increasing down columns and k increasing across rows.

```
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
```

Exercise C.1-9

There is a simple algebraic proof that can be done by induction based on the fact that $\binom{n}{2} = \frac{n^2-n}{2}$. There is a more interesting combinatorial proof as follows. Suppose that you are picking two items from a set of $n+1$ items. Then, spit into n different cases based on what the last item you pick is. Suppose that you pick position i as your last one. Notice that we must have that i is from 2 to $n + 1$. Then, for the other thing we pick, we could pick anything occurring before i, so we have $i - 1$ choices. From there, we just to a simple change of variables to get

$$\binom{n}{k} = \sum_{i=2}^{n+1} i - 1 = \sum_{i=1}^{n} i$$

Exercise C.1-10

Fix n. Then we have

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n-k+1}{k}\frac{n!}{(k-1)!(n-k+1)!} = \frac{n-k+1}{k}\binom{n}{k-1}.$$

Thus, we increase in k if and only if $\frac{n-k+1}{k} \geq 1$, which happens only when $n + 1 \geq 2k$, or $k \leq \lceil n/2 \rceil$. On the other hand, we decrease in k if and only if $\frac{n-k+1}{k} \leq 1$, so $k \geq \lfloor n/2 \rfloor$. Thus, the function is maximized precisely when k is equal to one of these.

Exercise C.1-11

$$\binom{n}{j+k} = \frac{n!}{(j+k)!(n-j-k)!}$$
$$= \frac{n!}{j!(n-j)!}\frac{j!(n-j)!}{(j+k)!(n-j-k)!}$$
$$= \binom{n}{j}\frac{j!(n-j)!}{(j+k)!(n-j-k)!}$$
$$= \binom{n}{j}\frac{(n-j)!}{(j+k)(j+k-1)\cdots(j+1)(n-j-k)!}$$
$$\leq \binom{n}{j}\frac{(n-j)!}{k!(n-j-k)!}$$
$$= \binom{n}{k}\binom{n-j}{k}$$

For a combinatorial (not algebraic) proof, we can see that if we fisrt pick j items, and then from the remaining pick k more, we have the quantity on the right. However, for each of the selections of $j+k$ items on the right, there is at least one way of picking j then picking k, giving us our inequality.

To see it is not tight, consider when $n = 2$, and $j = k = 1$. Then, we have that the left is $\binom{2}{2} = 1$. The right, however, is $\binom{2}{1}\binom{1}{1} = 2$ which is strictly greater.

Exercise C.1-12

We'll prove inequality C.6 for $k \leq n/2$ by induction on k. For $k = 0$ we have $\binom{n}{0} = 1 \leq \frac{n^n}{0^0(n-0)^{(n-0)}} = 1$. Now suppose the claim holds for k, and that $k < n/2$. Then we have

$$\binom{n}{k+1} = \frac{n-k}{k+1}\binom{n}{k}$$
$$\leq \frac{n-k}{k+1}\frac{n^n}{k^k(n-k)^{(n-k)}}$$
$$= \frac{n^n}{(k+1)k^k(n-k)^{(n-k-1)}}.$$

To show that this is bounded from above by $\frac{n^n}{(k+1)^{k+1}(n-k-1)^{(n-k-1)}}$ we need only verify that $\left(\frac{k+1}{k}\right)^k \leq \left(\frac{n-k}{n-k-1}\right)^{n-k-1}$. This follows from the fact that the left hand side, when viewed as a function of k, is increasing, and $k < n/2$ which implies that $k+1 \leq n-k$. By induction, the claim holds. Using equation $C.3$, we see that the claim extends to all $0 \leq k \leq n$ since the right hand side of the inequality is symmetric in k and $n-k$.

Exercise C.1-13

We start with $\binom{2n}{n} = \frac{(2n)!}{n!n!}$, then, applying Stirling's approximation, we get that this is

$$\frac{\sqrt{4\pi n}\left(\frac{2n}{e}\right)^{2n}\left(1+\Theta\left(\frac{1}{n}\right)\right)}{\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^{n}\left(1+\Theta\left(\frac{1}{n}\right)\right)\right)\cdot\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^{n}\left(1+\Theta\left(\frac{1}{n}\right)\right)\right)} = \frac{\sqrt{4\pi n}\left(\frac{2n}{e}\right)^{2n}\left(1+\Theta\left(\frac{1}{n}\right)\right)}{\left(2\pi n\left(\frac{n}{e}\right)^{2n}\right)\cdot\left(\left(1+\Theta\left(\frac{1}{n}\right)\right)\right)}$$

$$= \frac{\sqrt{4\pi n}}{2\pi n}2^{2n}\left(1+\Theta\left(\frac{1}{n}\right)\right)$$

$$= \frac{2^{2n}}{\sqrt{\pi n}}\left(1+\Theta\left(\frac{1}{n}\right)\right)$$

Exercise C.1-14

Differentiating the entropy function and setting it equal to 0 we have

$$H'(\lambda) = \lg(1-\lambda) - \lg(\lambda) = 0,$$

or equivalently $\lg(1-\lambda) = \lg(\lambda)$. This happens when $\lambda = 1/2$. Moreover, $H''(1/2) = \frac{-4}{\ln(2)} < 0$, so this is a local maximum. We have $H(1/2) = 1$, and since $H(0) = H(1) = 0$, this is in fact a global maximum for H.

Exercise C.1-15

$$\sum_{k=0}^{n}\binom{n}{k}k = \sum_{k=0}^{n}\frac{n!}{k!(n-k)!}k$$

$$= \sum_{k=0}^{n}\frac{(n-1)!}{(k-1)!((n-1)-(k-1))!}n$$

$$= n\sum_{k=0}^{n}\frac{(n-1)!}{(k-1)!((n-1)-(k-1))!}$$

$$= n\sum_{k=0}^{n}\binom{n-1}{k-1}$$

$$= n\sum_{k=0}^{n-1}\binom{n-1}{k}$$

$$= n2^{n-1}$$

Exercise C.2-1

The only way that Rosencrantz could of obtained more heads than Guildenstern is if he got a hears and Guildenstern got no heads, so, two tails. This

event has proscribed outcomes for all three coin flips, and so it happens with probability $\frac{1}{2^3} = \frac{1}{8}$.

Exercise C.2-2

Let $B_i = A_i \backslash (\cup_{k=1}^{i-1} A_i)$. Then $B_1, B_2, \ldots$ are disjoint and $A_1 \cup A_2 \cup \ldots = B_1 \cup B_2 \cup \ldots$. Moreover, $B_i \subseteq A_i$ for each i, so $Pr(B_i) \leq Pr(A_i)$. By third axiom of probability we have

$$Pr(A_1 \cup A_2 \cup \ldots = Pr(B_1 \cup B_2 \cup \ldots) = \sum_{i \geq 1} Pr(B_i) \leq \sum_{i \geq 1} Pr(A_i).$$

Exercise C.2-3

It is a 1/6 likelyhood. Here are two ways that you can show this fact. The first kind of messy, the second much more elegant.

This first is a direct counting argument. There are $10 \cdot 9 \cdot 8 = 720$ differerent sequences of cards picked, since we can at each step pick any card that is left. To count the number of sequences that are in increasing order, we choose the first card(call its value i), then any number larger than that for the second(say j) and any number larger than the second for the third(say k). This can be expressed by the summation:

$$\sum_{i=1}^{10} \sum_{j=i+1}^{10} \sum_{k=j+1}^{10} 1 =$$

$$\sum_{i=1}^{10} \sum_{j=i+1}^{10} 10 - j =$$

$$\sum_{i=1}^{10} 10(10-i) - \frac{10 \cdot 11}{2} + \frac{(i)(i+1)}{2} =$$

$$\sum_{i=1}^{10} 45 - 9.5i + i^2/2 =$$

$$450 - \frac{9.5 \cdot 10 \cdot 11}{2} + \frac{10 \cdot 11 \cdot 21}{2 \cdot 6} =$$

$$450 - 522.5 + 192.5 =$$

$$120$$

Where we have arrived at this count only using the formula for sum of kth powers. Then this gives us our final probability of $120/720 = 1/6$. Note also that it is unimportant to rule out the cases i=9, 10 and j=10. They are impossible cases, but if they occur then the sum will come out to be equal to zero anyways, I just figured that having it all go to 10 was more aesthetically pleasing.

The second way to show this probability is to first fix the three elements that you happened to pick and use symmetry. The set of values that are among

the cards has no bearing on whether the cards are in order. Since the numerical value on these cards is independent of the random the selection process, all possible orders for those three elements are equally likely. There are six different orderings on three elements, so the likelyhood they are ordered numerically is 1/6.

Thank you to Eric Richardson for pointing out that the originally posted solution for this exercise was misinterpreting the question.

Exercise C.2-4

We can verify this directly from the definition of conditional probability as follows:

$$Pr(A|B) + Pr(\overline{A}|B) = \frac{Pr(A \cap B)}{Pr(B)} + \frac{Pr(\overline{A} \cap B)}{Pr(B)} = \frac{Pr(B)}{Pr(B)} = 1.$$

Exercise C.2-5

We will proceed by induction on n. If $n = 2$, this is equation (C.16). Suppose that $n > 2$. Let $C = \cap i = 1^{n-1} A_i$. Then, we will use equation (C.16) followed by the inductive hypothesis to get

$$
\begin{aligned}
\Pr(A_1 \cap A_2 \cap \cdots \cap A_n) &= \Pr(C \cap A_n) \\
&= \Pr(A_n|C)\Pr(C) \\
&= \Pr(A_n|A_1 \cap A_2 \cap \cdots \cap A_{n-1})\Pr(A_1 \cap A_2 \cap \cdots \cap A_{n-1}) \\
&= \Pr(A_n|A_1 \cap A_2 \cap \cdots \cap A_{n-1}) \cdots \Pr(A_3|A_1 \cap A_2)\Pr(A_2|A_1)\Pr(A_1)
\end{aligned}
$$

Exercise C.2-6

Let $.a_1 a_2 a_3 \ldots$ be the binary representation of a/b. Flip the fair coin repeatedly, associating 1 with heads and 0 with tails, until the first time that the value of the ith flip differs from a_i. If the value is greater than a_i, output tails. If the value is less than a_i, output heads. Every number between 0 and $.99999 \ldots = 1$ has the possibility to be represented, and is created by randomly choosing its binary representation. The probability that we output tails is the probability that our number is less than a/b, which is just a/b. The expected number of flips is $\sum_{i=1}^{\infty} i2^{-i} = 2 = O(1)$.

Exercise C.2-7

Select n random colinear points. Seeing any one of them, you know nothing new about the distribution of the others. However, given two of them, you know the line that the third point would need to lie on.

Exercise C.2-8

Suppose we have a biased coin which always comes up heads, and a fair coin. We pick one at random and flip it twice. Let A be the event that the first coin is heads, B be the event that the second coin is heads, and C be the event that the fair coin is the one chosen. Then we have $Pr(A \cap B) = 5/8$ and $Pr(A) = Pr(B) = 3/4$ so $Pr(A \cap B) \neq Pr(A)Pr(B)$. Thus, A and B are not independent. However,

$$Pr(A \cap B | C) = \frac{Pr(A \cap B \cap C)}{Pr(C)} = \frac{(1/2)(1/2)(1/2)}{1/2} = 1/4$$

and

$$Pr(A|C) \cdot Pr(B|C) = \frac{Pr(A \cap C)}{Pr(C)} \frac{Pr(B \cap C)}{Pr(C)} = \frac{1/4}{1/2} \frac{1/4}{1/2} = 1/4.$$

Exercise C.2-9

This problem has many explanations. I'll give a concise one, as there are versions online that really hold your hand through the whole thing.

First, supposing that you don't switch, then the likelihood of winning will just be $\frac{1}{3}$, since the original decision was made before any additional information leaked in by way of lifting a curtain, so it must just be uniformly random.

Now, suppose that you do switch, your likelihood of winning will increase to $\frac{2}{3}$. To see this, we split into two cases. Suppose you guessed correctly initially, which happens with probability $\frac{1}{3}$, then you will fail. However, if you picked incorrectly originally, which happens with probability $\frac{2}{3}$, you will then win by switching. Therefore, you will win with probability $\frac{2}{3}$.

So, your chances of winning double by adopting the switching strategy instead of the staying strategy.

Exercise C.2-10

His chances are still $1/3$, because at least one of Y or Z would be executed, so hearing which one changes nothing about his own situation. However, the probability that Z is going free is now $2/3$. To see this, note that the probability that the free prisoner is among Y and Z is $2/3$. Since we are told that it is not Y, the $2/3$ probability must apply exclusively to Z.

Exercise C.3-1

The expectation of a single dice roll is $\frac{1}{6}(1 + 2 + 3 + 4 + 5 + 6) = \frac{21}{6} = 3.5$. So, by linearity of expectation, the expectation of their sum is 7.

We have that the probability that the maximum of the two dice results is $\leq i$ is the probability that both dice results lie in the range $1, 2, .., i$. This happens with probability $\frac{i^2}{36}$. Then, to get the probability that a particular

value of the maximum occurs, we take the difference of successive values of this cumulative density function. This gets us that the probability of a max of i is $\frac{i^2-(i-1)^2}{36} = \frac{2i-1}{36}$. So, the expected value of this max is

$$\frac{1}{36}(1+6+15+28+45+66) = \frac{161}{36} \approx 4.47$$

Exercise C.3-2

The probability that the maximum or minimum element is in a particular spot is $1/n$ since the ordering is random. Thus, the expected index of the maximum and minimum are the same, and given by:

$$E[X] = \sum_{i=1}^{n} i(1/n) = \frac{1}{n}\frac{n(n+1)}{2} = \frac{n+1}{2}.$$

Exercise C.3-3

The probability that you loose your dollar is $(5/6)^3 = \frac{125}{216}$. The probability you gain exactly one dollar is $\binom{3}{1}(1/6)(5/6)^2 = \frac{75}{216}$. The probability you gain exactly two dollars is $\binom{3}{2}(1/6)^2(5/6) = \frac{15}{216}$. The probability that you win 3 dollars is $(1/6)^3\frac{1}{216}$. putting it all together, the payoff is

$$\frac{3}{216} + \frac{30}{216} + \frac{75}{216} - \frac{125}{216} = -\frac{17}{216}$$

So, you are expected to loose $\frac{17}{216}$ of a dollar.

Exercise C.3-4

Let X and Y be nonnegative random variables. Let $Z = X+Y$. Then Z is a random variable, and since X and Y are nonnegative, we have $Z \geq \max(X,Y)$ for any outcome. Thus, $E[\max(X,Y)] \leq E[Z] = E[X] + E[Y]$.

Exercise C.3-5

the value of $f(X)$ is only a function of X, and, so it may only leak information about the state of X. Similarly $g(Y)$ can only depend on and leak information about the state of Y. Since the states of X and Y don't affect each other, knowing the value of functions of the two random varaibles also will not affect each other.

Exercise C.3-6

We can verify directly from the definition of expectation:

$$E[X] = \sum_{i=0}^{\infty} i \cdot Pr(X = i)$$

$$\geq \sum_{i=t}^{\infty} i \cdot Pr(X = i)$$

$$\geq t \sum_{i=t}^{\infty} Pr(X = i)$$

$$= t \cdot Pr(X \geq t).$$

Dividing both sides by t gives the result.

Exercise C.3-7

Let $S' \subseteq S$ be the states of the sample space for whcih X' takes a value that is $\geq t$. For any of these states, we will also have that X takes a value that is at least t. This gets us that the set of states for which X is at least t is a superset of S'. Then, by monotonicity of probability, we have the desired inequality on the probabilities.

Exercise C.3-8

The expectation of the square of a random variable is larger. To see this, note that by (C.27) we have $E[X^2] - E^2[X] = E[(X - E[X])^2] \geq 0$ since the square of a random variable is a nonnegative random variable, so its expectation must be nonnegative.

Exercise C.3-9

Since X only takes the values 0 and 1, we will always have that $X^2 = X$. Then, the probability that X takes the value of 1 is equal to the expected value of X. So,

$$Var[X] = E[X^2] - (E[X])^2$$
$$= E[X] - (E[X])^2$$
$$= E[X](1 - E[X])$$
$$= E[X](E[1] - E[X])$$
$$= E[X](E[1 - X])$$

Exercise C.3-10

Proceeding from (C.27) we have

$$Var[aX] = E[(aX)^2] - E^2[aX]$$
$$= E[a^2X^2] - a^2E^2[X]$$
$$= a^2E[X^2] - a^2E^2[X]$$
$$= a^2(E[X^2] - E^2[X])$$
$$= a^2Var[X].$$

Exercise C.4-1

Axiom 2 states that the probability of the entire space should be one. This is saying that if we add up the probabilities of each i occurring over all i, we should get one. This is easy to see because

$$\sum_{i=1}^{\infty}(1-p)^{i-1}p = \frac{p}{1-(1-p)} = \frac{p}{p} = 1$$

Exercise C.4-2

Let X be the number of times we must flip 6 coins before we obtain 3 heads and 3 tails. The probability that we obtain 3 heads and 3 tails is $\binom{6}{3}(1/2)^6 = 5/16$, so the probability that we don't is $11/16$. Moreover, X has a geometric distribution, so by (C.32) we have

$$E[X] = 1/p = 16/5.$$

Exercise C.4-3

To go from the first line to the second, we use the fact that picking k items is identical to selecting the $n-k$ items you aren't picking.

$$b(k; n, p) = \binom{n}{k}p^k(1-p)^{n-k}$$
$$= \binom{n}{n-k}p^k(1-p)^{n-k}$$
$$= b(n-k; n, 1-p)$$

Exercise C.4-4

Using Stirling's approximation we have $b(k; n, p) \approx \frac{\sqrt{n}n^n}{\sqrt{2\pi}\sqrt{k(n-k)}k^k(n-k)^{n-k}}p^kq^{n-k}$.
The binomial distribution is maximized at its expectation. Plugging in $k = np$ gives

$$b(np; n, p) \approx \frac{\sqrt{n}n^np^{np}(1-p)^{n-np}}{\sqrt{np(n-np)}(np)^{np}(n-np)^{n-np}} = \frac{1}{\sqrt{2\pi npq}}.$$

Exercise C.4-5

The probability that there will be no successes is equal to

$$\lim_{n\to\infty} b(0; n, 1/n) = \lim_{n\to\infty} \binom{n}{0} \frac{1}{n}^0 (1 - \frac{1}{n})^{n-0}$$

$$= \lim_{n\to\infty} (1 - \frac{1}{n})^n$$

$$= \lim_{n\to\infty} ((1 + \frac{1}{-n})^{-n})^{-1}$$

$$= e^{-1} = \frac{1}{e}$$

Similarly, the probability of one success is equal to

$$\lim_{n\to\infty} b(1; n, 1/n) = \lim_{n\to\infty} \binom{n}{1} \frac{1}{n}^1 (1 - \frac{1}{n})^{n-1}$$

$$= \lim_{n\to\infty} \frac{n}{n(1 - \frac{1}{n})}(1 - \frac{1}{n})^n$$

$$= \left(\lim_{n\to\infty} \frac{n}{n(1 - \frac{1}{n})} \right) \cdot \left(\lim_{n\to\infty} (1 - \frac{1}{n})^n \right)$$

$$= 1 \cdot \frac{1}{e} = \frac{1}{e}$$

Exercise C.4-6

There are $2n$ total coin flips amongst the two professors. They get the same number of heads if Professor Guildenstern flips k heads and Professor Rosencrantz flips $n - k$ tails. If we imagine flipping a head as a success for Professor Rosencrantz and flipping a tail as a success for Professor Guildenstern, then the professors get the same number of heads if and only if the total number of successes achieved by the professors is n. There are $\binom{2n}{n}$ ways to select which coins will be a successes for their flipper. Since the outcomes are equally likely, and there are $2^{2n} = 4^n$ possible flip sequences, the probability is $\binom{2n}{n}/4^n$.

To verify the identity, we can imagine counting successes in two ways. The right hand side counts via our earlier method. For the left hand side, we can imagine first choosing k successes for Professor Guildenstern, then choosing the remaining $n - k$ successes for Professor Rosencrantz. We sum over all k to get the total number of possibilities. Since the two sides of the equation count the same thing they must be equal.

Exercise C.4-7
We apply (C.7) with $\lambda = \frac{k}{n}$, then,

$$b(k; n, 1/2) = \binom{n}{k}(1/2)^k(1 - 1/2)^{n-k}$$
$$= \binom{n}{k}2^{-n}$$
$$\leq 2^{nH(k/n)2^{-n}}$$
$$= 2^{nH(k/n)-n}$$

Exercise C.4-8

Let $a_1, a_2, \ldots, a_n$ be uniformly chosen from $[0, 1]$. Let X_i be the indicator random variable that $a_i \leq p_i$ and Y_i be the indicator random variable that $a_i \leq p$. Let $X' = \sum_{i=1}^{n} X_i$ and $Y' = \sum_{i=1}^{n} Y_i$. Then for any event we have $X' \leq Y'$, which implies $P(X' < k) \geq P(Y' < k)$. Let Y be the number of successes in n trials if each trial is successful with probability p. Then X has the same distribution as X' and Y has the same distribution of Y', so we conclude that $P(X < k) \geq P(Y < k)$.

Exercise C.4-9

Consider having the underlying state space be n copies of the unit interval. Then, a particular state will be a sequence of n numbers in $[0, 1]$. Then, X' will be the number of times the ith coordinate is less than or equal to p_i. Similarly, X is the number of times that the ith coordinate is less than or equal to p_i'. Since $p_i \leq p_i'$, we have that for any particular state, $X \leq X'$. Then we can directly apply the results of C.3-7 to get the desired result.

Exercise C.5-1

Which one is less likely depends on the value of n.

The probability that you obtain no head when flipping n times is just 2^{-n}, straight from the definition of the binomial distribution.

The probability P that we will have fewer than n heads from $4n$ flips is exactly

$$P = \Pr\{X < n\} = \sum_{k=0}^{k=n-1} b(k, 4n, 1/2) = 2^{-4n} \sum_{k=0}^{k=n-1} \binom{4n}{k}$$

In particular, since all the terms in the sum are positive, the probability is at least the value of the largest term, so

$$P \geq \binom{4n}{n-1}2^{-4n}$$

13

So, if we can just show that the quantity $\binom{4n}{n-1}2^{-3n} > 1$, then we have that P is greater than the probability that we have no head when flipping n times.

This is in fact not true in general, but we will show it is true for $n \geq 20$. We can then show by explicit computation what happens for $n = 1\ldots 19$. As in inductive step, we can confirm that $\binom{80}{19}2^{-60} \approx 1.005 > 1$.

Now, suppose that we have $\binom{4n}{n-1}2^{-3n} > 1$.

$$
\begin{aligned}
\binom{4n+4}{n}2^{-3n-3} &= \frac{(4n+4)!}{8(n)!(3n+4)!}2^{-3n} \\
&= \frac{(4n+4)(4n+3)(4n+2)(4n+1)}{8n(3n+4)(3n+3)(3n+2)}\frac{(4n)!}{(n-1)!((3n+1)!}2^{-3n} \\
&= \frac{(4n+4)(4n+3)(4n+2)(4n+1)}{8n(3n+4)(3n+3)(3n+2)}\binom{4n}{n-1}2^{-3n} \\
&> \frac{(4n+4)(4n+3)(4n+2)(4n+1)}{8n(3n+4)(3n+3)(3n+2)} \\
&= \frac{32}{27}\frac{(n+1)(n+3/4)(n+1/2)(n+1/4)}{8n(n+4/3)(n+1)(n+2/3)}
\end{aligned}
$$

Using elementary calculus, we can see that the limit of this rational expression in n is equal to $\frac{32}{27}$, so, there is some n so that for all n past there, it will be greater than 1, so, we will have $\binom{4n+4}{n}2^{-3n-3} > 1$.

More concretely, we can show that the rational function is an increasing function on n for $n > 0$ by taking it's derivative:

$$
\frac{-24 - 108n - 93n^2 + 116n^3 + 144n^4}{3n^2(2+3n)^2(4+3n)^2}
$$

and noticing it is positive for $n \geq 1$ because the numerator has real roots at $\approx -.368$ and $\approx .937$ and end behavor of $+\infty$ as $n \to \infty$. Then, since $n = 3$ causes the original rational expression to be $\frac{35}{33}$, we have that it is greater than 1 for all $n \geq 3$. Since we are only considering $n \geq 20$ here, we definitly have that the expression is > 1.

Lastly, we turn our attention to the finitely many values of n that we left out before. Since we only need to consider $n = 1\ldots 19$, we can just perform the explicit computation for P as $\sum_{k=0}^{k=n-1} b(k, 4n, 1/2)$ at each value of n and compare it to 2^{-n}.

n	P	2^{-n}
1	1/16	1/2
2	9/256	1/4
3	79/4096	1/8
4	697/65536	1/16
5	1549/262144	1/32
6	55455/16777216	1/64
7	249589/134217728	1/128
8	4514873/4294967296	1/256
9	10249879/17179869184	1/512
10	93396401/274877906944	1/1024
11	3414035527/17592186044416	1/2048
12	31278197839/281474976710656	1/4096
13	71797952431/1125899906842624	1/8192
14	1321035185597/36028797018963968	1/16384
15	6086999773681/288230376151711744	1/32768
16	224723513577529/18446744073709551616	1/65536
17	519244699305955/73786976294838206464	1/131072
18	4805026093205835/118059162071741130344	1/262144
19	4451531435563063/18889465931478580854784	1/524288

We can stop the table here since our inductive proof that P was larger was starting at a base case of $n = 20$. So, by looking at the numerical values in the table, we can see that if $n < 18$, then it is more unlikely that you have fewer than n heads out of $4n$ flips and if $n \geq 18$, then it more unlikely to have no heads after flipping n times.

Exercise C.5-2

For Corollary C.6 we have $b(i; n, p)/b(i - 1; n, p) \leq \frac{(n-k)p}{kq}$. Let $x = \frac{(n-k)p}{kq}$. Note that $x < 1$, so the infinite series below converges. Then we have

$$Pr(X > k) = \sum_{i=k+1}^{n} b(i; n, p)$$

$$\leq \sum_{i=k+1}^{n} x^{i-k} b(k; n, p)$$

$$= b(k; n, p) \sum_{i=1}^{n-k} x^i$$

$$\leq b(k; n, p) \sum_{i=1}^{\infty} x^i$$

$$= b(k; n, p) \frac{x}{1-x}$$

$$= b(k; n, p) \frac{(n-k)p}{k-np}.$$

For Corollary C.7, we use Corollary C.6 and the fact that $x < 1$ as follows:

$$\frac{Pr(X > k)}{Pr(X > k-1)} = \frac{Pr(X > k)}{Pr(X > k) + Pr(X = k-1)} \leq \frac{xb(k; n, p)}{xb(k; n, p) + b(k; n, p)} < \frac{1}{2}.$$

Exercise C.5-3

Divide both sides of the expression through by $(a+1)^n$. Since $a > 0$, this quantity is > 1, so the inequalities remain the way they are. This means that it is equivalent to show that

$$\sum_{i=0}^{k-1} \binom{n}{i} a^i (a+1)^{-n} < \frac{kb(k; n, a/(a+1))}{na - k(a+1)}$$

So, we do some algebra, and apply Theorem C.4.

$$\sum_{i=0}^{k-1} \binom{n}{i} a^i (a+1)^{-n} = \sum_{i=0}^{k-1} \binom{n}{i} \left(\frac{a}{a+1}\right)^i \left(\frac{1}{a+1}\right)^{n-i}$$

$$= \sum_{i=0}^{k-1} \binom{n}{i} \left(\frac{a}{a+1}\right)^i \left(1 - \frac{a}{a+1}\right)^{n-i}$$

$$= \sum_{i=0}^{k-1} b(i; n, \frac{a}{a+1})$$

$$< \frac{k\left(1 - \frac{a}{a+1}\right)}{n\frac{a}{a+1} - k} b(k; n, \frac{a}{a+1})$$

$$= \frac{k\left(\frac{1}{a+1}\right)}{n\frac{a}{a+1} - k} b(k; n, \frac{a}{a+1})$$

$$= \frac{k}{na - k(a+1)} b(k; n, \frac{a}{a+1})$$

completing the proof.

Exercise C.5-4

Using Lemma C.1 and Corollary C.4 we have

$$\sum_{i=0}^{k-1} p^i q^{n-i} \le \sum_{i=0}^{k-1} \binom{n}{i} p^i q^{n-i}$$

$$= \sum_{i=0}^{k-1} b(i; n, p)$$

$$\le \frac{kq}{np - k} b(k; n, p)$$

$$\le \frac{kq}{np - k} \left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k}.$$

Exercise C.5-5

(note that this problem had extensive bugs prior to the third printing of the third edition, see errata.)

Since the values of p_i and q_i appear nowhere in this expression, we can swap the two. This has the effect of changing the mean to $n - \mu$. This means that the condition $r > \mu$ becomes $r > n - \mu$ which we have. Then, the right hand side becomes $\Pr\{(n - X)(b - \mu) \ge r\} = \Pr\{\mu - X \ge r\}$. Then, just replacing

17

the μ occurring in the left, we obtain that

$$\Pr\{\mu - X\} \leq \left(\frac{(n - \mu)e}{r}\right)^r$$

Similarly, we can apply corollary C.9 to this distribution counting the number of failures (swapping p and q). This has the result of making

$$\Pr\{np - X\} = \Pr\{n - X - n(1 - p)\} = \Pr\{X - nq \geq r\} \leq \left(\frac{nqe}{r}\right)^r$$

again, the restriction on r changes exactly into what we are given.

Exercise C.5-6

As in the proof of Theorem C.8, we'll bound $E[e^{\alpha(X-\mu)}]$ and substitute a suitable value for α. First we'll prove (with a fair bit of work) that if $q = 1 - p$ then $f(\alpha) = e^{\alpha^2/2} - pe^{\alpha q} - qe^{-\alpha p} \geq 0$ for $\alpha \geq 0$. First observe that $f(0) = 0$. Next we'll show $f'(\alpha) > 0$ for $\alpha > 0$. To do this, we'll show that $f'(0) = 0$ and $f''(\alpha) > 0$. We have

$$f'(\alpha) = \alpha e^{\alpha^2/2} - pqe^{\alpha q} + pqe^{-\alpha p}$$

so $f'(0) = 0$. Moreover

$$f''(\alpha) = \alpha^2 e^{\alpha^2/2} + e^{\alpha^2/2} - pq(qe^{\alpha q} + pe^{-\alpha p}).$$

Since $\alpha^2 e^{\alpha^2/2} > 0$ it will suffice to show that $e^{\alpha^2/2} \geq pq(qe^{\alpha q} + pe^{-\alpha p})$. Indeed, we have

$$pq(qe^{\alpha q} + pe^{-\alpha p}) \leq (1/4)(qe^{\alpha q} + pe^{-\alpha p} \leq (1/4)e^{-\alpha p}(e^{\alpha} + 1) \leq (1/4)(e^{\alpha} + 1)$$

so we need to show $4e^{\alpha^2/2} \geq e^{\alpha} + 1$. Since $e^{\alpha^2/2} > 1$, it is enough to show $3e^{\alpha^2/2} \geq e^{\alpha}$. Taking logs on both sides, we need $\alpha^2/2 - \alpha + \ln(3) \geq 0$. By taking a derivative we see that this function is minimized when $\alpha = 1$, where it attains the value $\ln(3) - 1/2 > 0$. Thus, the original inequality holds. Now can proceed with the rest of the proof. As in the proof of Theorem C.8, $E[e^{\alpha(X-\mu)}] = \prod_{i=1}^{n} E[e^{\alpha(X_i - p_i)}]$. Using the inequality we just proved we have

$$E[e^{\alpha(X_i - p_i)}] = p_i e^{\alpha q_i} + q_i e^{-\alpha p_i} \leq e^{\alpha^2/2}.$$

Thus, $E[e^{\alpha(X-\mu)}] \leq \prod_{i=1}^{n} e^{\alpha^2/2} = e^{n\alpha^2/2}$. By this and (C.43) and (C.44) we have

$$Pr(X - \mu \geq r) \leq E[e^{\alpha(X-\mu)}]e^{-\alpha r} \leq e^{n\alpha^2/2 - \alpha r}.$$

Finally, taking $\alpha = r/n$ gives the desired result.

Exercise C.5-7

We put on our freshman calculus hats. With them securely affixed, we take a derivative of the expression with respect to α. This gets us $(\mu e^\alpha - r)\exp(\mu e^\alpha - \alpha r)$. Since a minima of the original expression will occur when the derivative is zero, and the end behavior of the expression is increasing without bound, we set this derivative equal to zero. The second factor is an exponential and so can never be zero. So, we are trying to solve $\mu e^\alpha - r = 0$. This means that $\alpha = \ln(r/\mu)$. at the minimum.

Problem C-1

a. For each of the n balls, we can make one of b different decisions about where to place it. Since this number of possible decisions is independent of the previous choices, the total number of possibilities is just b^n.

b. First, we pretend that we can distingush the sticks. This means that we want to arrange a total of n balls and $b - 1$ sticks, that is, $n + b - 1$ total things. There are exactly $(b + n - 1)!$ ways of doing this. Then, we realize that a bunch of these are the same, in particular, no matter how we permute the $b-1$ sticks in some arrangement, we end up with the same answer. Therefore, we can arrange n distinguishable balls and $b - 1$ indistinguishable sticks in $\frac{(n+b-1)!}{(b-1)!}$ many ways.

Once we have such an arrangement, we can relate it back to the original statement, where we imagine the sticks as being the dividing lines between bins, and the ordered balls between them being the ordered balls in each bin.

c. Taking the result from before, we notice that any of the n permutations of the balls will result in the same configuration, so, we must divide our count from the previous part by $n!$. This means that we are left with $\frac{(n+b-1)!}{n!(b-1)!} = \binom{n+b-1}{n}$.

d. Since each bin can either contain a bin or not, we are selecting a set of bins to contain balls in the part. The number of non-empty bins must be equal to the number of balls, so we are selecting n bins. That is, we are selecting a subset of size n of the bins from the set of all bins, which there are n of. This is then exactly the combinatorial definition of $\binom{b}{n}$ defined in terms of selecting subsets.

e. Since no bin may be left empty, this means that we need to put one ball into each bin. This means that we have $n - b$ balls left, and, since each bin now contains at least one, we can put them into the bins with no further restriction. This means we are in the case of part c. The only difference is that the number of balls that we have to distribute it $n-b$ not n. This means that the answer is

$$\binom{(n-b)+b-1}{n-b} = \binom{n-1}{n-b} = \binom{n-1}{(n-1)-(n-b)} = \binom{n-1}{b-1}$$

19

Appendix D

Michelle Bodnar, Andrew Lohr

April 12, 2016

Exercise D.1-1

Let $C = A + B$ and $D = A - B$. Then, since A and B are symmetric, we know that $a_{ij} = a_{ji}$ and $b_{ij} = b_{ji}$. We consider these two matrices C and D now.

$$c_{ij} = a_{ij} + b_{ij}$$
$$= a_{ji} + b_{ji}$$
$$= c_{ji}$$

$$d_{ij} = a_{ij} - b_{ij}$$
$$= a_{ji} - b_{ji}$$
$$= d_{ji}$$

Exercise D.1-2

From the definitions of transpose and matrix multiplication we have

$$(AB)_{ij}^T = (AB)_{ji}$$
$$= \sum_{k=1}^{n} a_{jk} b_{ki}$$
$$= \sum_{k=1}^{n} b_{ki} a_{jk}$$
$$= (B^T A^T)_{ij}.$$

Therefore $(AB)^T = B^T A^T$. This implies $(A^T A)^T = A^T (A^T)^T = A^T A$, so $A^T A$ is symmetric.

Exercise D.1-3

Suppose that A and B are lower triangular, and let $C = AB$. Being lower triangular, we know that for $i < j$, $a_{ij} = b_{ij} = 0$. To see that C is lower triangular,

$$
\begin{aligned}
c_{ij} &= \sum_{k=1}^{n} a_{ik} b_{kj} \\
&= \sum_{k=1}^{j-1} a_{ik} b_{kj} + \sum_{k=j}^{n} a_{ik} b_{kj} \\
&= \sum_{k=1}^{j-1} a_{ik} 0 + \sum_{k=j}^{n} 0 b_{kj} \\
&= \sum_{k=1}^{j-1} 0 + \sum_{k=j}^{n} 0 \\
&= 0 + 0 = 0
\end{aligned}
$$

Exercise D.1-4

Suppose row i of P has a 1 in column j. Then row i of PA is row j of A, so PA permutes the rows. On the other hand, column j of AP is column i of A, so AP permutes the columns. We can view the product of two permutation matrices as one permutation matrix permuting the rows of another. This preserves the property that there is only a single 1 in each row and column, so the product is also a permutation matrix.

Exercise D.2-1

$$
\begin{aligned}
I &= I \\
AC &= AB \\
B(AC) &= B(AB) \\
(BA)C &= (BA)B \\
IC &= IB \\
C &= B
\end{aligned}
$$

Exercise D.2-2

Let L be a lower triangular matrix. We'll prove by induction on the size of the matrix that the determinant is the product of its diagonal entries. For $n = 1$, the determinant is just equal to the matrix entry, which is the product of the only diagonal element. Now suppose the claim holds for n, and let L be $(n+1) \times (n+1)$. Let L' be the $n \times n$ submatrix obtained from L by deleting

the first row and column. Then we have $\det(L) = L_{11} \det(L')$, since $L_{1j} = 0$ for all $j \neq 1$. By the induction hypothesis, $\det(L')$ is the product of the diagonal entries of L', which are all the diagonal entries of L except L_{11}. The claim follows since we multiply this by L_{11}.

We will prove that the inverse of a lower triangular matrix is lower triangular by induction on the size of the matrix. For $n = 1$ every matrix is lower triangular, so the claim holds. Let L be $(n+1) \times (n+1)$ and let L' be the submatrix obtained from L by deleting the first row and column. By our induction hypothesis, L' has an inverse which is lower triangular, call it L'^{-1}. We will construct a lower triangular inverse for L:

$$L^{-1} = \begin{bmatrix} 1/l_{11} & 0 & \cdots & 0 \\ a_1 & & & \\ \vdots & & L'^{-1} & \\ a_n & & & \end{bmatrix}$$

where we define a_i recursively by

$$a_1 = -L_{21}/(L_{11}L'_{11}) \text{ and } a_i = -\left(L_{(i+1),1}/L_{11} + \sum_{k=1}^{i-1} L'_{ik} a_k \right) / L'_{ii}.$$

It is straightforward to verify that this in fact gives an inverse, and it is well-defined because L is nonsingular, so $L_{ii} \neq 0$.

Exercise D.2-3
We will show that it is invertible by showing that P^T is its inverse. Suppose that the only nonzero entry of row i is a one in column $\sigma(i)$. This means that the only nonzero entry in column i of P^T is in row $\sigma(i)$. If we let $C = PP^T$, then, for every i and j, we have that $c_{ij} = \sum_{k=1}^n p_{ik} p_{jk} = p_{i\sigma(i)} p_{j\sigma(i)}$. Since σ is a bijection, we have that if $i \neq j$, then $\sigma(i) \neq \sigma(j)$. This means that $c_{ij} = 0$. However, if $i = j$, then $c_{ij} = 1$. That is, their product is the identity matrix. This means that P^T is the inverse of P.

Since a permutation matrix is defined as having exactly one one in each row and column, we know that this is true of P. However, the set of rows of P^T is the set of columns of P, and the set of columns of P^T are the set of rows of P. Since all of those have exactly one one, we have that P^T is a permutation matrix.

Exercise D.2-4

Assume first that $j \neq i$. Let C_{ij} be the matrix with a 1 in position (i, j), and zeros elsewhere, and $C = I + C_{ij}$. Then $A' = CA$, so $A'^{-1} = A^{-1}C^{-1}$. It is easy to check that $C^{-1} = I - C_{ij}$. Moreover, right multiplication by C^{-1} amounts to subtracting column i from column j, so the claim follows. The claim fails to hold when $i = j$. To see this, consider the case where $A = B = I$. Then A' is invertible, but if we subtract column i from column j of B we get a matrix with a column of zeros, which is singular, so it cannot possibly be the inverse

of a matrix.

Exercise D.2-5

To see that the inverse of a matrix that has only real entries must be real. Imagine instead that that matrix was a matrix over $\mathbb{R}$. Then, it will have a inverse in this ring of matrices so long as it is non-singular. This inverse will also be an inverse of the matrix when viewed as a matrix of $\mathbb{C}$. Since inverses are distinct, this must be its inverse, and is all real entries because it was originally computed as a matrix that was in the set of real entry matrices.

To see the converse implication, just swap the roles, computing the inverse of the inverse. This will be the original matrix.

Exercise D.2-6

We have $(A^{-1})^T = (A^T)^{-1} = A^{-1}$ so A^{-1} is symmetric, and $(BAB^T)^T = (B^T)^T A^T B^T = BAB^T$.

Exercise D.2-5

We will consider the contrapositive. That is, we will show that a matrix has rank less than n if and only if there exists a null vector.

Suppose that v is a null vector, and let $a_1, a_2, \ldots a_n$ be the columns of A. Since multiplying a matrix by a vector produces a linear combination of its columns whoose coefficients are determined by the vector v that we are multiplying by. Since v evaluates to zero, we have just found a linear combination of the set of columns of A that sums to zero. That is, the columns of A are linearly dependent, so the column rank is less than n, since the set of all vectors is not linearly independent, the largest independent set must be a proper subset.

Now, suppose that the column rank is less than n. This means that there is some set of column vectors which are linearly dependent. Let v_i be the coefficient given to the ith column of A in this linear combination that sums to zero. Then, the v_i values combine into a null vector.

Exercise D.2-8

Let A be $m \times p$ and B be $p \times n$. Recall that $\mathrm{rank}(AB)$ is equal to the minimum r such that there exist F and G of dimensions $m \times r$ and $r \times n$ such that $FG = AB$. Let A' and A'' be matrices of minimum r' such that $A'A'' = A$ and the dimensions are $m \times r'$ and $r' \times p$. Let B' and B'' be the corresponding matrices for B, which minimize r''. If $r' \le r''$ we have $A'(A''B'B'') = AB$, so $r \le r'$ since r was minimal. If $r'' \le r'$ we have $(A'A''B')B'' = AB$, so $r \le r''$, since r was minimal. Either way, $\mathrm{rank}(AB) \le \min(\mathrm{rank}(A), \mathrm{rank}(B))$.

The product of a nonsingular matrix and another matrix preserves the rank of the other matrix. Since the rank of a nonsingular $n \times n$ matrix is n and the rank of an $m \times n$ or $n \times m$ matrix is bounded above by $\min(m, n)$, the rank of

4

AB is bounded above by the minimum of n and the rank of the other matrix, which is the minimum of the rank of each of the matrices.

Problem D-1

We'll proceed by induction on n. If $n = 1$, then, the matrix is just the 1×1 matrix with it's only entry 1. This clearly has a determinant of 1. Also, since there is no way to pick distinct i and j in the product on the right hand side, the product is just one as well.

Now, suppose it is true for $n \times n$ Vandermonde matrices, we will show it for $(n + 1) \times (n + 1)$ Vandermonde matrices. Now, suppose that we, as the hint suggests, starting at the second from rightmost column and working left, add $(-x_0)$ times that column to the one to its right. Since the determinant is unaffected by this sort of operation,we won't be changing the determinant. For every entry in the top row except the leftmost one, we have made it zero, since it becomes $x_0^{j-1} - x_0^{j-2} \cdot x_0 = 0$. This means that we can do a expansion by minors along the top row to compute the determinant. The only nonzero term in this is the top left column. This means that the determinant is the same as the determinant of that one minor. Everything in the ith row of that minor has a factor of $(x_i - x_0)$. This means that if we take all those factors out, we have $\prod_{i=1}^{n-1}(x_i - x_0)$, which are all the factors in the formula that include x_0. Once we take that factors out, we are left with a $n \times n$ Vandermonde matrix on the variables $x_1, \ldots, x_{n-1}$. So, we know that it will contain all the other factors we need to get the right hand side, completing the induction.

Problem D-2

a. Without loss of generality we may assume that the first r columns of A are linearly independent. Then for x_1 and $x_2 \in S_n$ such that x_1 and x_2 are not identical in the first r entries and have 0's in the remaining entries we have that $Ax_1 \neq Ax_2$. This is because the first r entries of each are a linear combination of the first r rows of A, and since they are independent there can't be two different linear combinations of them which are equal. Since there are at least 2^r non-equivalent vectors $x \in S_n$, we must have $|R(A)| \geq 2^r$. On the other hand, x is a vector which doesn't have 0's in the coordinates greater than r. Then $Ax = \sum_{x_i} a_i$ where a_i is the ith column of A. Since each of the last $n - r$ columns of A is in fact a linear combination of the first r columns of A, this can be rewritten as a linear combination of the first r columns of A. Since we have already counted all of these, $|R(A)| = 2^r$. If A doesn't have full rank then the range can't include all 2^n elements of S_n, so A can't possibly define a permutation.

b. Let $y \in R(A)$ and $x_r, x_{r+1}, \ldots, x_{n-1}$ be arbitrary. Set $z = \sum_{i=r}^{n-1} a_i x_i$. Since the first i columns of A span the range of A and z is in the range of A, $y - z$ is in the range of A and there exist $x_0, x_1, \ldots, x_{r-1}$ such that $\sum_{i=0}^{r-1} a_i x_i = y - z$. Then we have $Ax = y - z + z = y$. Since the last $n - r$ entries of x were

arbitrary, $|P(A, y)| \geq 2^{n-r}$. On the other hand, there are 2^r elements in $R(A)$, each with at least 2^{n-r} preimages, which means there are at least $2^r \cdot 2^{n-r} = 2^n$ preimages in total. Since $|S_n| = 2^n$, there must be exactly 2^{n-r} preimages for each element of the range.

c. First observe that $|B(S', m)|$ is just the number of blocks which contain an element of the range of S. Since the first m rows of A only affect the first m positions of Ax, they can affect the value of Ax by at most $2^m - 1$, which won't change the block. Thus, we need only consider the last $n - m$ rows. Without loss of generality, we may assume that S consists of the first block of S_n, so that only the first m columns of A are relevant. Suppose that the lower left $(n - m) \times m$ submatrix of A has rank r. Then the range of Ax consists of vectors of the form $[*, \cdots, *, x_0, x_1, \cdots x_{r-1}, 0, \cdots, 0]^T$, where there are m $*$'s. There are 2^{m+r} such vectors, spanning 2^r blocks. Thus, $|B(S', m)| = 2^r$. Since it is only the choice of $x_0, \ldots, x_{r-1}$ which determines the block we're in, and we can pick every possible combination, every block must be hit the same number of times. Thus, the number of numbers in S which map to a particular block is $2^m / 2^r = 2^{m-r}$.

d. The number of linear permutations is bounded above by the number of pairs (A, c) where A is an $n \times n$ matrix with entries in $GF(2)$ and c is an n-bit vector. There are 2^{n^2+n} of these. On the other hand, there are $(2^n)!$ permutations of S_n. For $n \geq 3$, $2^{n^2+n} \leq (2^n)!$.

e. Let $n = 3$ and consider the permutation $\pi(0) = 0$, $\pi(1) = 1$, $\pi(2) = 2$, $\pi(3) = 3$, $\pi(4) = 5$, $\pi(5) = 4$, $\pi(6) = 6$ and $\pi(7) = 7$. Since $A \cdot 0 + c = 0$ we must have c be the zero vector. In order for $\pi(1)$ to equal 1, the first column of A must be $[1\ 0\ 0]^T$. To have $\pi(2) = 2$, the second column of A must be $[0\ 1\ 0]^T$. To have $\pi(3) = 3$, the third column of A must be $[0\ 0\ 1]^T$. This completely determines A as the identity matrix, making it impossible for $\pi(4) = 5$, so the permutation is not achievable by any linear permutation.