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

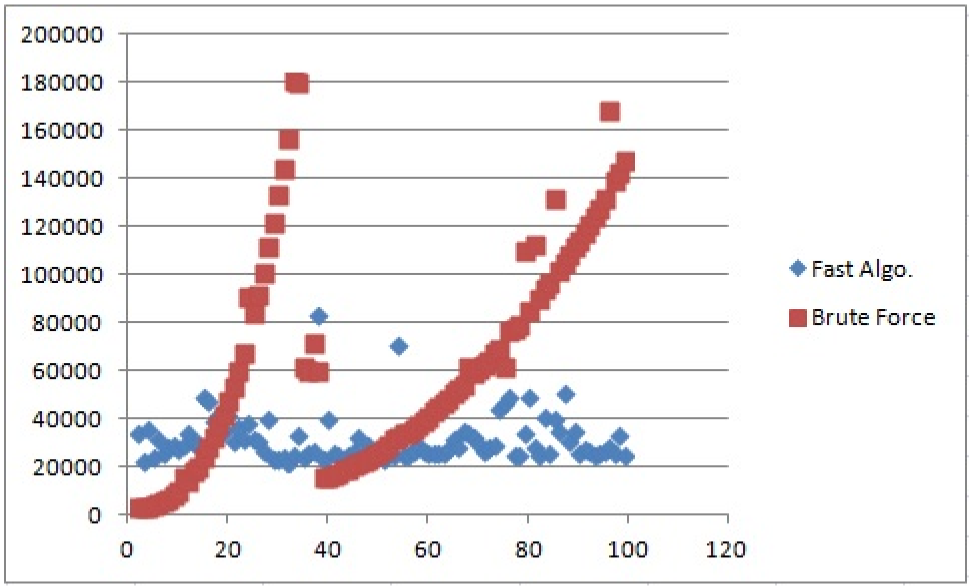
Algorithm 1 Brute Force Algorithm to Solve Maximum Subarray Problem

```
$left = 1$
$right = 1$
$max = A[1]$
$curSum = 0$
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 $(low_M, high_M, M)$
```

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

$$\begin{aligned}
P_1 &= 6 \\
P_2 &= 8 \\
P_3 &= 72 \\
P_4 &= -10 \\
P_5 &= 48 \\
P_6 &= -12 \\
P_7 &= -84
\end{aligned}$$

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

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

$$P_1 = (a + b)c = ac + bcP_2 \quad = b(c + d) = bc + bdP_3 = (a - b)d = ad + bd$$

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 $T(n) \leq cn\lg n$ where $c = \max(T(2)/2, 1)$. Then,

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

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

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

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

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

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

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

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

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 $T(n) \leq 2n\log n + 1$. First observe that this means $T(1) = 1$, so the base case is satisfied. Then we have

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

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 $T(n) \leq cn^2$.

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

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 $T(n) \leq cn^2 - 6n$

$$\begin{aligned} T(n) &= 4T(n/2+2)+n \\ &\leq 4c(n^2/4+2n+4-3n-12)+n \\ &= cn^2 - 4cn - 32c + n \end{aligned}$$

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 $T(n) \leq 2^n - 1$.

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

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.

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

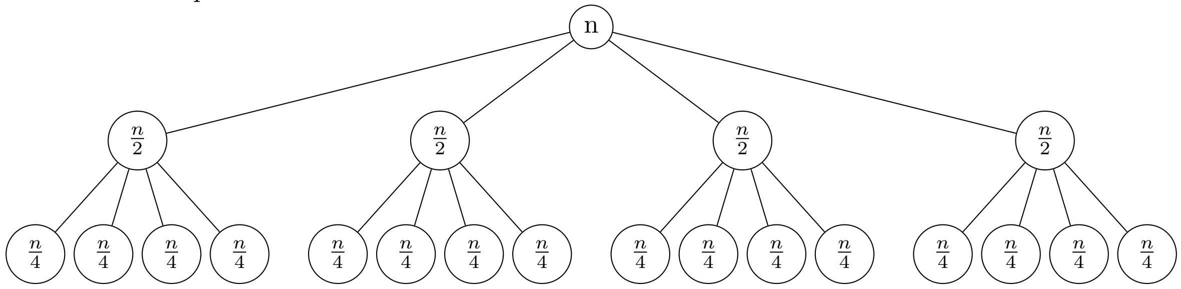
Now, to justify that this is actually a pretty tight bound, we'll show that we can't have any polynomial upper bound. That is, if we have that $T(n) \leq cn^k$ then, when we substitute into the recurrence, we get that the new coefficient for n^k can be as high as $c(1+\frac{1}{2^k})$ which is bigger 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 $cn(\log_3 n + 1) \geq cn \log_3 n = \frac{c}{log3} n \log n = \Omega(n \log n)$.

Exercise 4.4-7

Here is an example for $n = 4$.

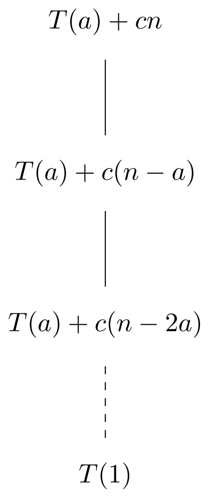

n
$\frac{n}{2}$ $\frac{n}{2}$ $\frac{n}{2}$ $\frac{n}{2}$
$\frac{n}{4}$ $\frac{n}{4}$ $\frac{n}{4}$ $\frac{n}{4}$ $\frac{n}{4}$ $\frac{n}{4}$ $\frac{n}{4}$ $\frac{n}{4}$ $\frac{n}{4}$ $\frac{n}{4}$ $\frac{n}{4}$ $\frac{n}{4}$ $\frac{n}{4}$ $\frac{n}{4}$ $\frac{n}{4}$ $\frac{n}{4}$

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

$$\begin{aligned} T(n) &= 4T(\lfloor n/2 \rfloor) + cn \\ &\leq c'n^2 + cn \end{aligned}$$

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



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 $T(n) \leq c'n\lg(n) - cn$.

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

Where we use the fact that $x\lg(x)$ is convex for the last inequality. This then completes the induction if we have $c' \geq c$ 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 $T(n) \leq n^2 \lg^2 n$.

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

provided that $n \geq 2$.

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

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

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

for $c_1 \ge k\lg b$. 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 $\exists \epsilon, d, k, \forall n \ge k$, we have $f(n) \ge dn^{\log_b a+\epsilon}$. By the regularity condition, we have that for sufficiently large n, $af(n/b) \le cf(n)$. In particular, it is true for all $n \ge bk$. 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^ik is less than n that for every $n \ge k$, $f(n) \ge dn^{\log_b a+\epsilon}$. By our definition of d, we have it is true for $i = 1$. So, suppose we have $b^{i-1}k < n \le b^ik$. Then, by regularity and the inductive hypothesis, $cf(n) \ge af(n/b) \ge ad\left(\frac{n}{b}\right)^{\log_b(a)+\epsilon}$. Solving for $f(n)$, we have

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

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

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

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 $T(k) \leq ck$ for $k < n$, then,

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

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

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

$$T(n) = T(0) + \sum_{j=1}^{n} \lg(j) = T(0) + \int_1^{n+1} \sum_{j=1}^{n+1} \chi_{(j,j+1)}(x) \lg(j) dx$$

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

$$\begin{aligned}
\mathcal{F}(z) &= \sum_{i=0}^{\infty} F_i z^i \\
&= F_0 + F_1 z + \sum_{i=2}^{\infty}(F_{i-1} + F_{i-2})z^i \\
&= z + z\sum_{i=2}^{\infty} F_{i-1}z^{i-1} + z^2\sum_{i=2}^{\infty} F_{i-2}z^{i-2} \\
&= z + z\sum_{i=1}^{\infty} F_i z^i + z^2\sum_{i=0}^{\infty} F_i z^i \\
&= z + z\mathcal{F}(z) + z^2\mathcal{F}(z).
\end{aligned}$$

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.

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

$$\begin{aligned}\mathcal{F}(z) &= \frac{1}{\sqrt{5}}\left(\frac{1}{1-\phi z} - \frac{1}{1-\hat{\phi} z}\right)\\ &= \frac{1}{\sqrt{5}}\left(\sum_{i=0}^{\infty}(\phi z)^i - \sum_{i=0}^{\infty}(\hat{\phi} z)^i\right)\\ &= \sum_{i=0}^{\infty}\frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i)z^i.\end{aligned}$$

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

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

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 $A[i,j] + A[i+1,j+1] \le A[i,j+1] + A[i+1,j]$. 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 $T(m,n) = O(m + n\log m)$.

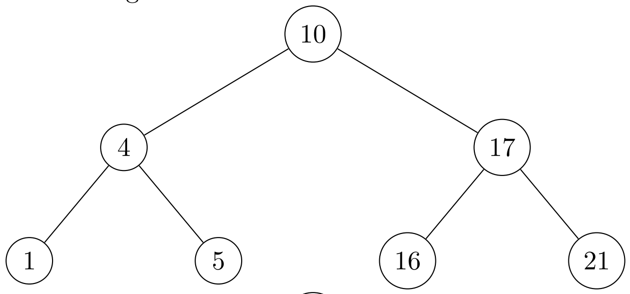
Chapter 12

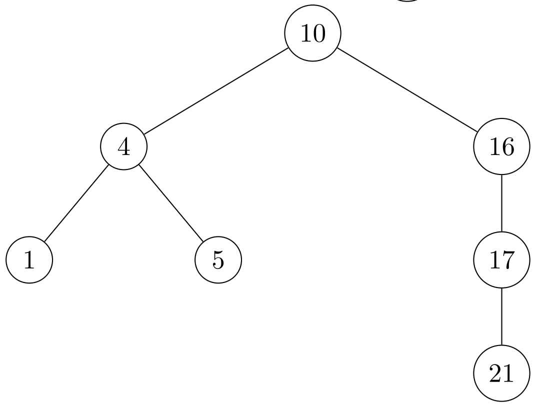
Michelle Bodnar, Andrew Lohr

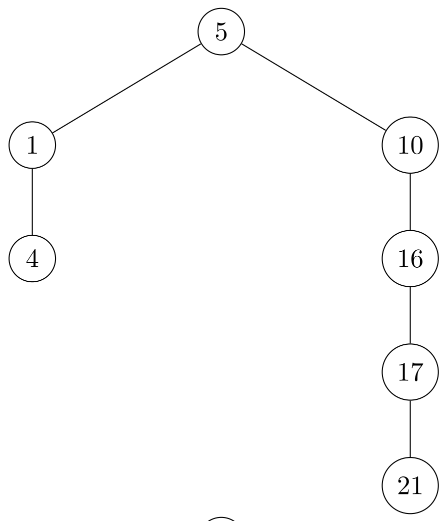
May 5, 2017

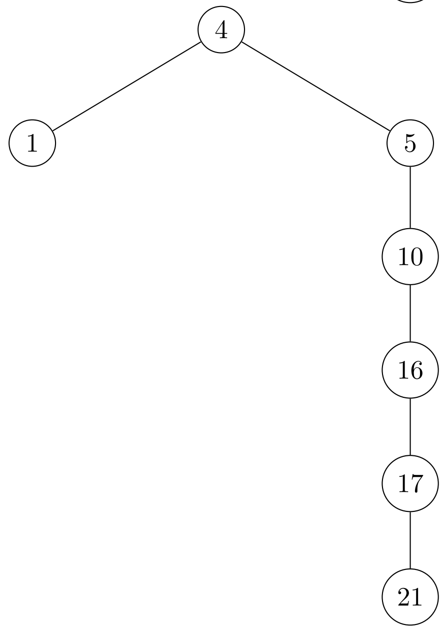
Exercise 12.1-1

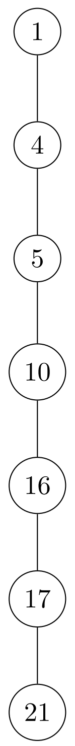
Anytime that a node has a single child, treat it as the right child, with the left child being NIL






5
1
10
4
16
17
21


4
1
5
10
16
17
21



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
```

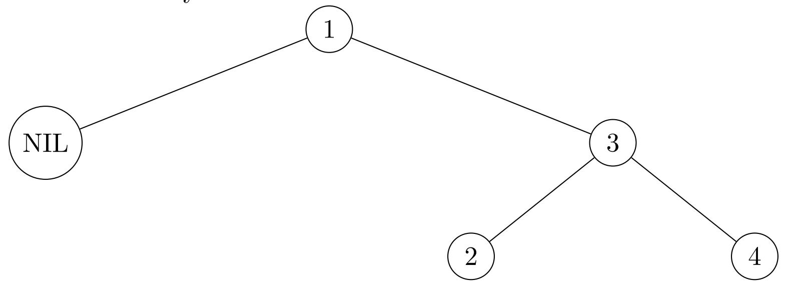
Exercise 12.2-3

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 k^{th} 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
```

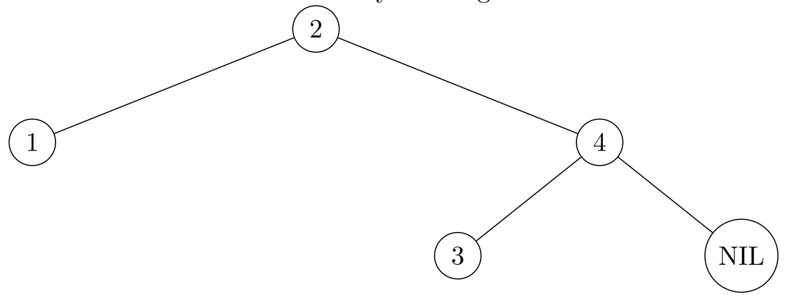
The nodes examined in the while loop of TREE-INSERT are the same as those examined in TREE-SEARCH. In lines 9 through 13 of TREE-INSERT, only one additional node is examined.

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 sizen_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 $\{0, \dots n-1\}$. 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 i^{th} 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.

$$\begin{aligned}
B(x) &= \sum_{n=0}^{\infty} b_n x^n \\
&= 1 + \sum_{n=1}^{\infty} b_n x^n \\
&= 1 + \sum_{n=1}^{\infty}\sum_{k=0}^{n-1} b_k b_{n-k-1} x^n \\
&= 1 + x\sum_{n=1}^{\infty}\sum_{k=0}^{n-1} b_k x^k b_{n-k-1} x^{n-k-1} \\
&= 1 + x\sum_{n=0}^{\infty}\sum_{k=0}^{n} b_k x^k b_{n-k} x^{n-k} \\
&= 1 + xB(x)^2.
\end{aligned}$$

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.

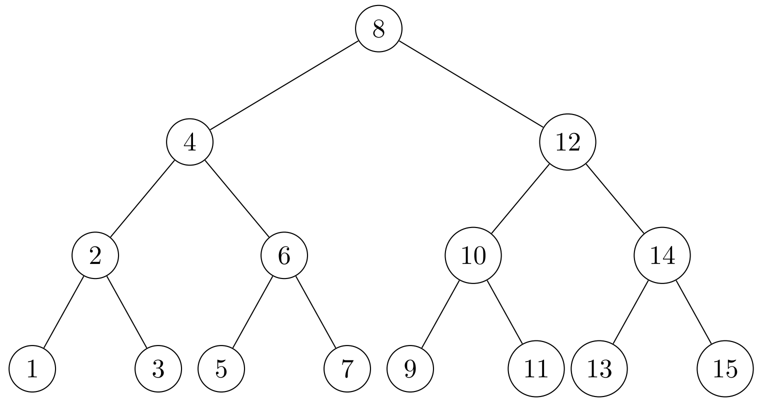
d. The asymptotic follows from applying Sirling's formula to b_n.

Chapter 13

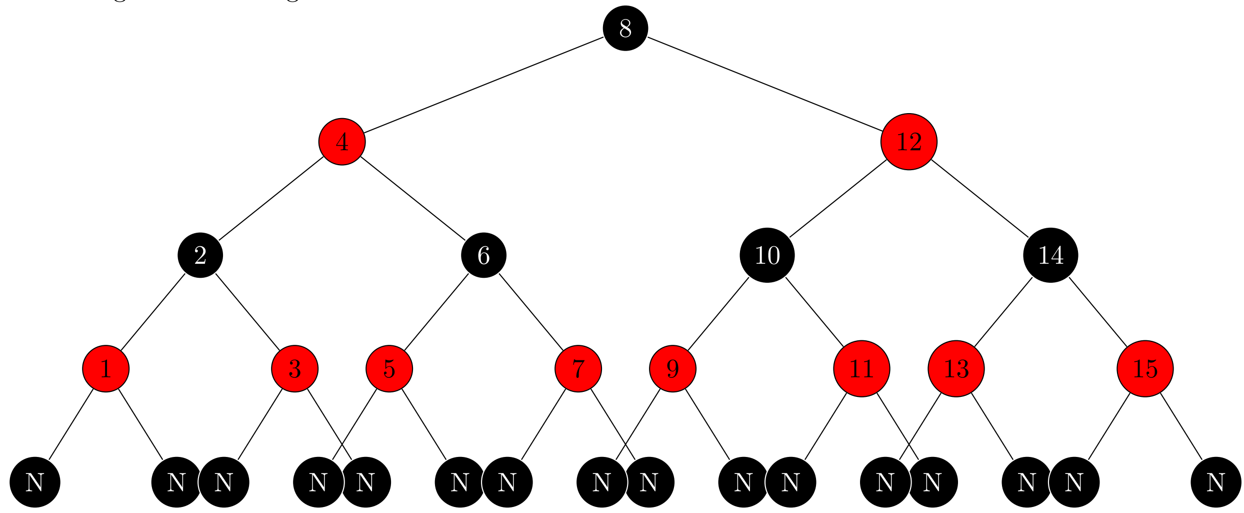
Michelle Bodnar, Andrew Lohr

April 12, 2016

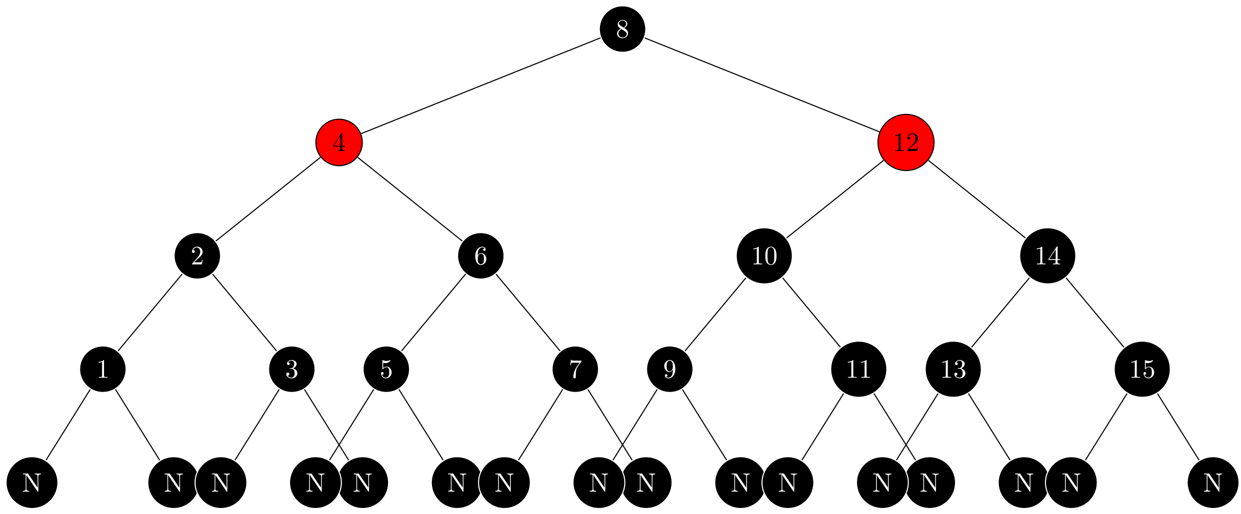
Exercise 13.1-1



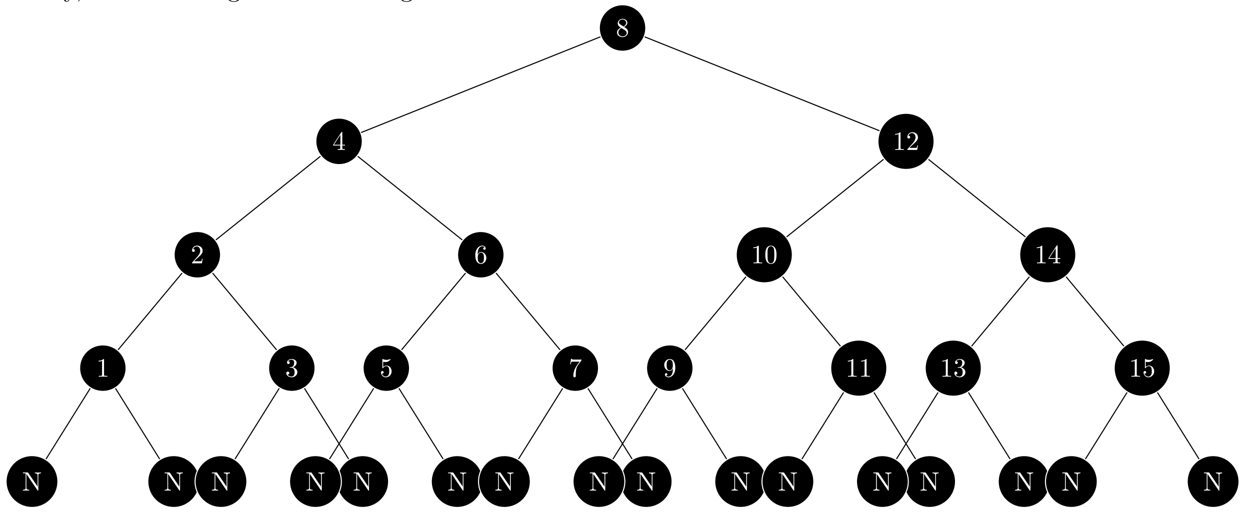
We shorten NIL to N so that it can be more easily displayed in the document. The following has black height 2.



The following has black height 3

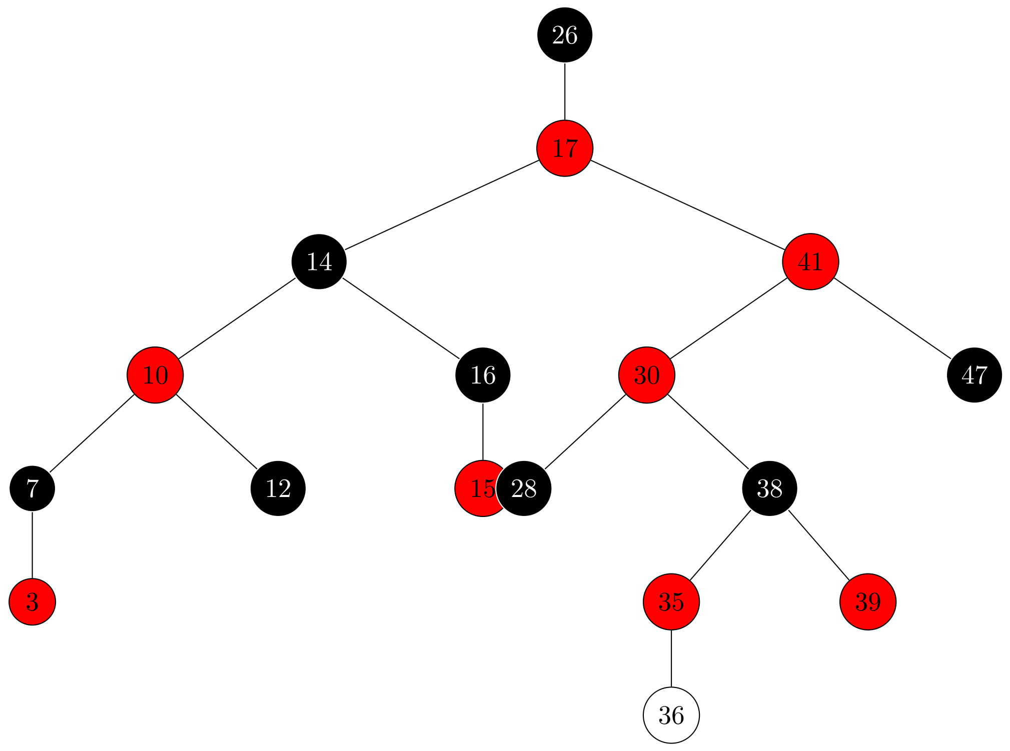


Lastly, the following has black height 4.



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 $(a_1, a_2, \dots a_s)$ and the shortest simple path $(b_1, b_2, \dots, b_t)$. 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

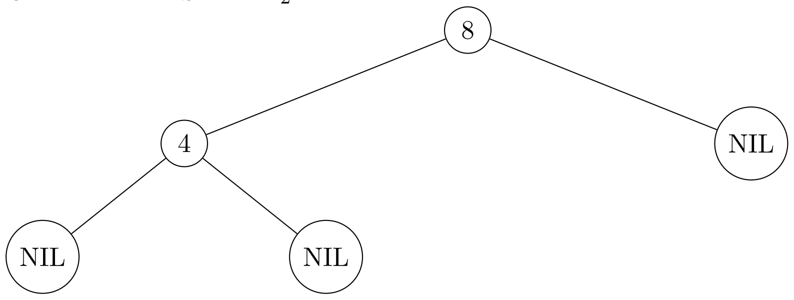
the depth of c decreases by one, the depth of b stays the same, and the depth of a increases by 1.

Exercise 13.2-4

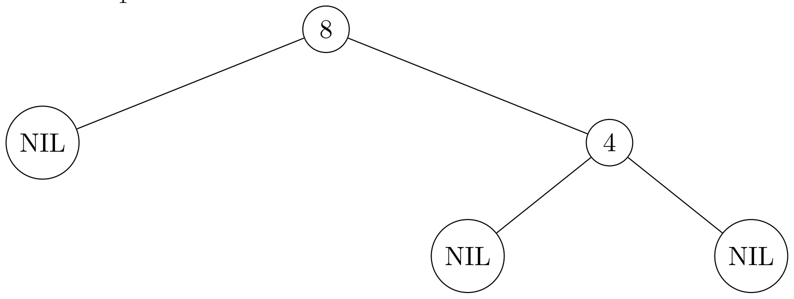
Consider transforming an arbitrary n-node BT into a right-going chain as follows: Let the root and all successive right children of the root be the elements of the chain initial chain. For any node x which is a left child of a node on the chain, a single right rotation on the parent of x will add that 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



And let T_1 be



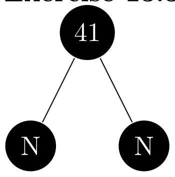
Then, there are no nodes for which its valid to call right rotate in T_1. Even though it is possible to right convert T_2 into T_1, the reverse is not possible.

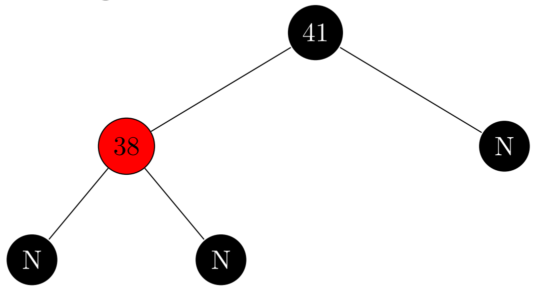
For any BST T, define the quantity $f(T)$ to be the sum over all the nodes of the number of left pointers that are used in a simple path from the root to that node. Note that the contribution from each node is $O(n)$. Since there are only n nodes, we have that $f(T)$ is $O(n^2)$. Also, when we call RIGHT-ROTATE(T,x), 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.

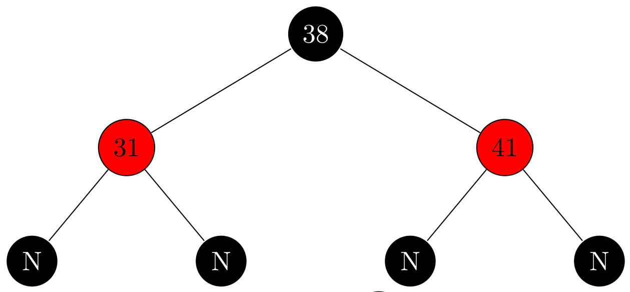
Exercise 13.3-1

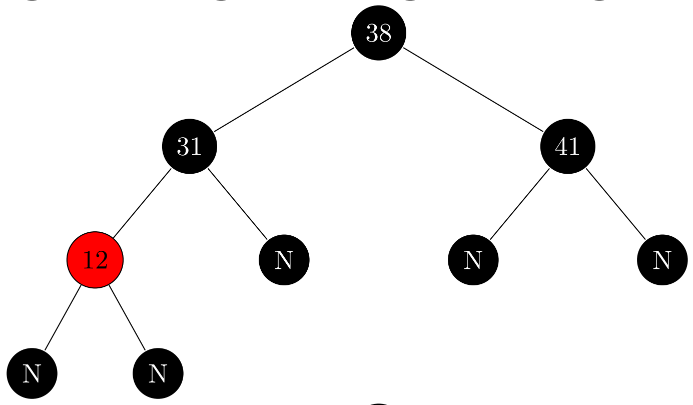
If we chose to set the color of z to black then we would be violating property 5 of being a red-black tree. Because any path from the root to a leaf under z would have one more black node than the paths to the other leaves

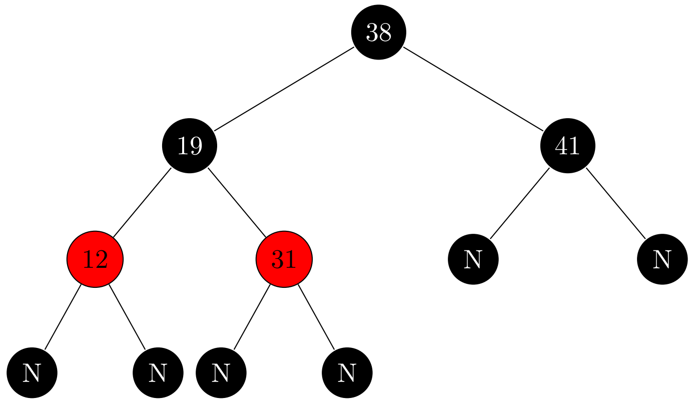
Exercise 13.3-2

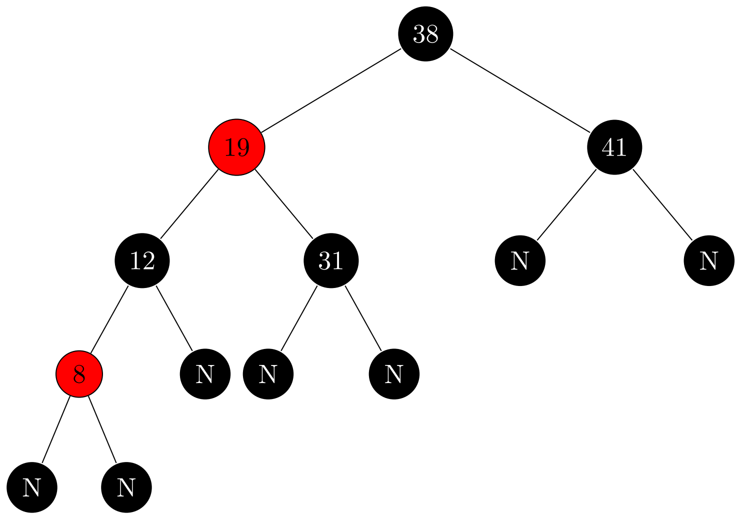





38
31
41
N
N
N
N

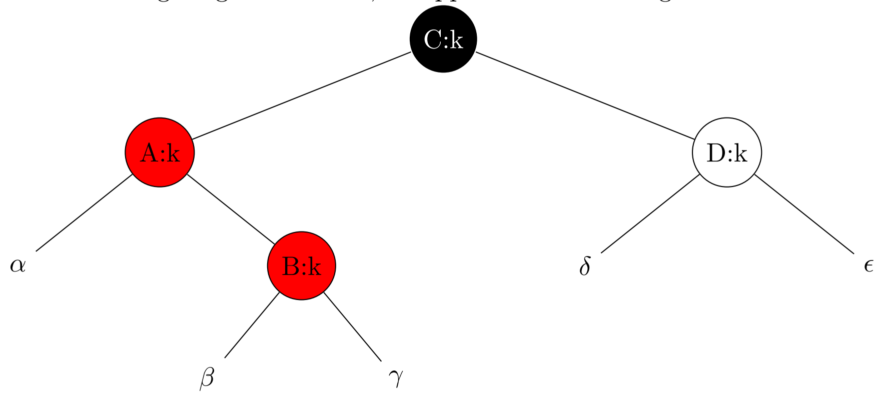

38
31
41
12
N
N
N
N
N


38
19
41
12
31
N
N
N
N
N
N



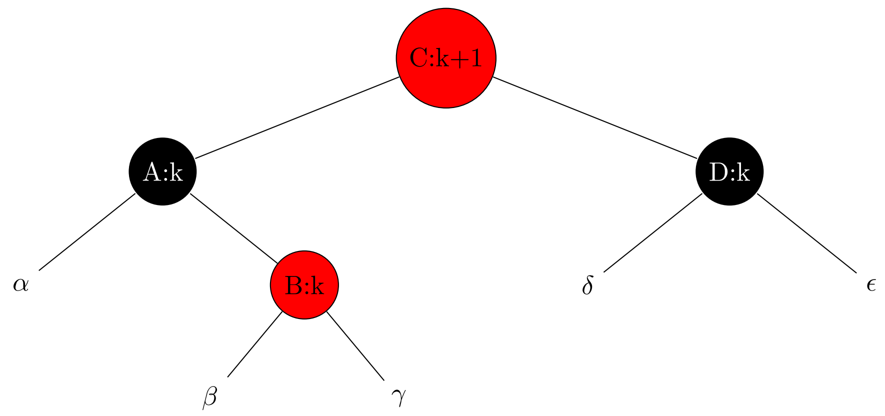
Exercise 13.3-3

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.

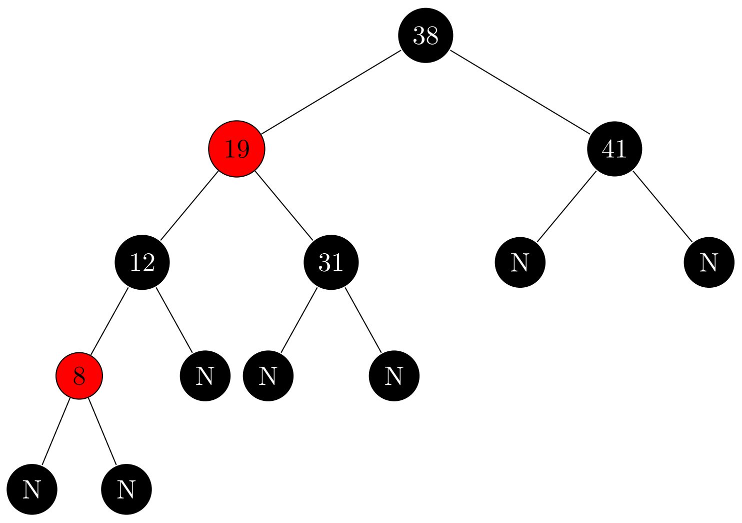
Exercise 13.4-1

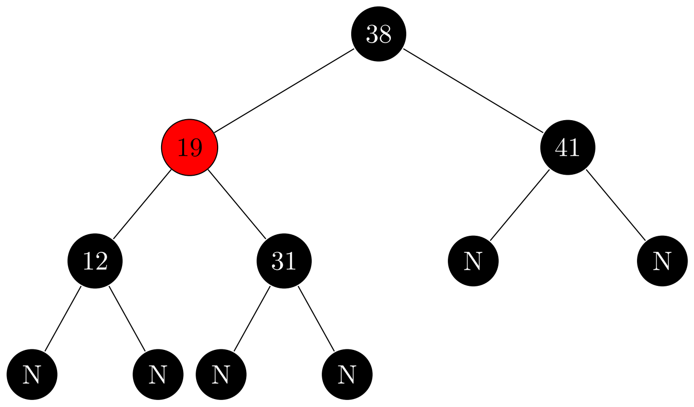
There are two ways we may of left the while loop of RB-DELETE-FIXUP. The first is that we had $x = T.root$. In this case, we set $x.color = BLACK$ on line 23. So, we must have that the root is black. The other case is that we ended the while loop because we had $x.color == RED$, but had that $x \neq T.root$. 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.

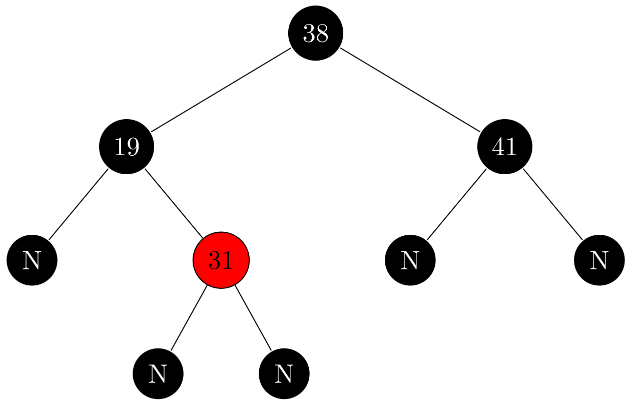
Exercise 13.4-2

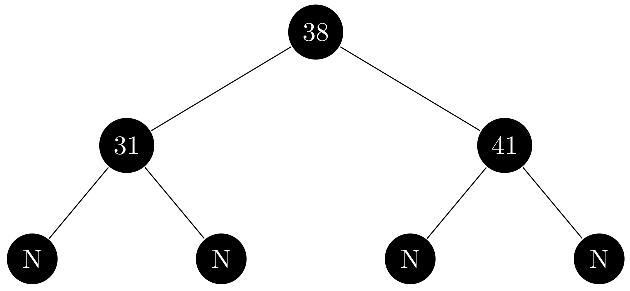
Suppose that both x and $x.p$ are red in RB-DELETE. This can only happen in the else-case of line 9. Since we are deleting from a red-black tree, the other child of $y.p$ which becomes $x's$ sibling in the call to RB-TRANSPLANT on line 14 must be black, so x is the only child of $x.p$ which is red. The while-loop condition of RB-DELETE-FIXUP(T,x) is immediately violated so we simply set $x.color = black$, restoring property 4.

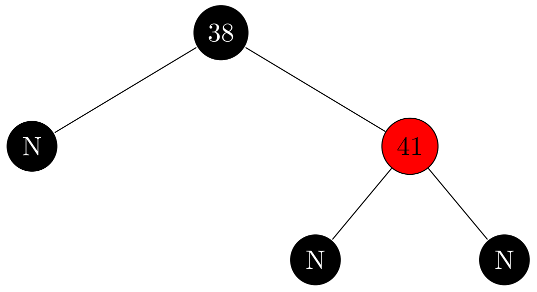
Exercise 13.4-3

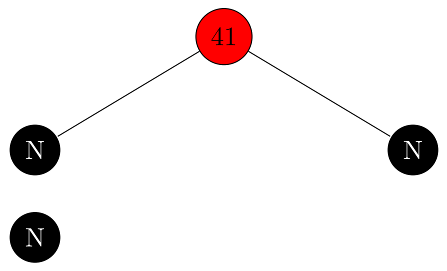



38
19
41
12
31
N
N
N
N
N
N


38
19
41
N
31
N
N
N
N


38
31
41
N
N
N
N


38
N
41
N
N



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

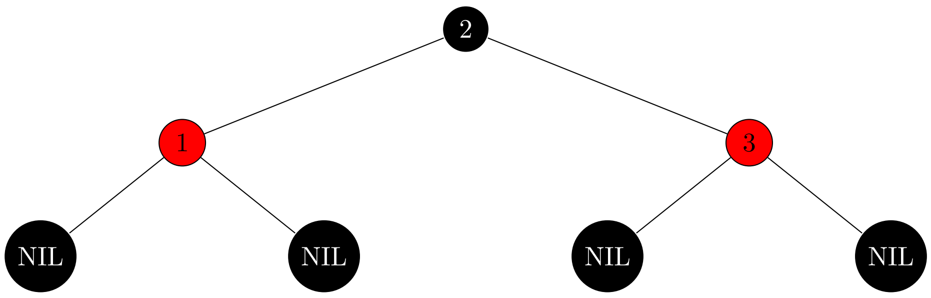
Case 4: For α and β, the count goes from 1+count(c) to 2+count(c). For γ and δ, it is 1+count(c)+count(c') both before and after. For ϵ and ζ, it is 1+ count(c) both before and after. This increase in the count for α and β is because x before indicated an extra black.

Exercise 13.4-6

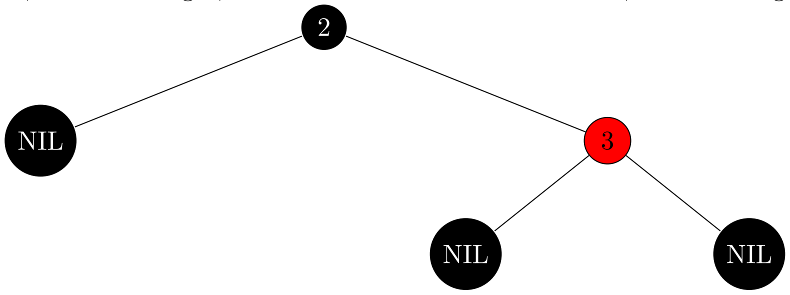
At the start of case 1 we have set w to be the sibling of x. We check on line 4 that $w.color == red$, which means that the parent of x and w cannot be red. Otherwise property 4 is violated. Thus, their concerns are unfounded.

Exercise 13.4-7

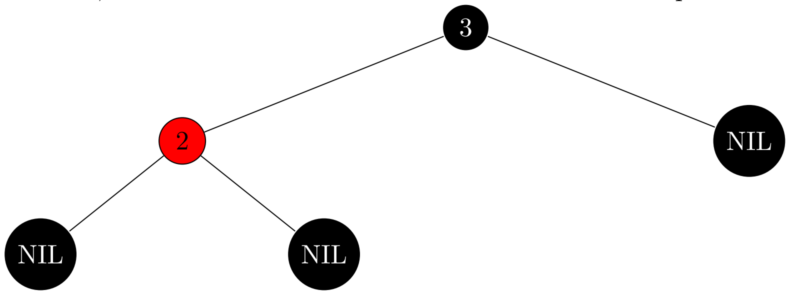
Suppose that we insert the elements $3, 2, 1$ in that order, then, the resulting tree will look like



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

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, $T(0) = 0$,

$T(1) = 1$. This is both the same recurrence and initial conditions as the Fibonacci numbers. So, recalling equation (3.25), we have that

$$T(h) = \left\lfloor \frac{\phi^h}{\sqrt{5}} + \frac{1}{2} \right\rfloor \leq n$$

Rearranging for h, we have

$$\begin{aligned} \frac{\phi^h}{\sqrt{5}} - \frac{1}{2} &\leq n \\ \phi^h &\leq \sqrt{5}\left(n + \frac{1}{2}\right) \\ h &\leq \frac{\lg(\sqrt{5}) + \lg(n + \frac{1}{2})}{\lg(\phi)} \in O(\lg(n)) \end{aligned}$$

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 $P\{X_{ik}\} = \frac{1}{(k-i+1)(k-i)}$.

h. We use part g then simplify the telescoping series:

$$\begin{aligned}
E[C] &= \sum_{j=1}^{k-1} E[X_{jk}] \\
&= \sum_{j=1}^{k-1} \frac{1}{(k-j+1)(k-j)} \\
&= \sum_{j=1}^{k-1} \frac{1}{j(j+1)} \\
&= \sum_{j=1}^{k-1} \frac{1}{j} - \frac{1}{j+1} \\
&= 1 - \frac{1}{k}.
\end{aligned}$$

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 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

$$r_n = \max\{p_n, r_1 + r_{n-1} - c, r_2 + r_{n-2} - c, \dots, r_{n-1} + r_1 - c\}$$

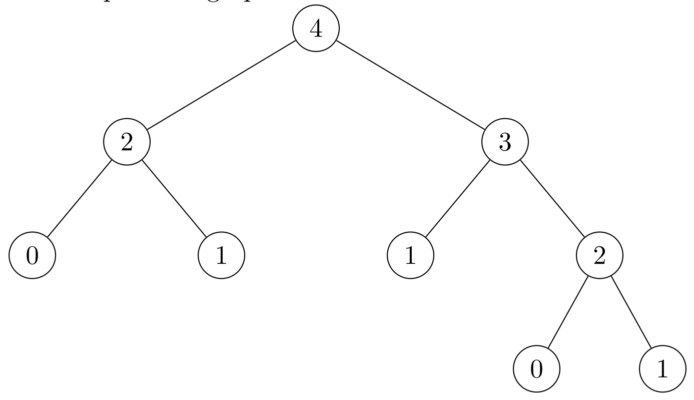
And so, to change the top down solution to this problem, we would change MEMOIZED-CUT-ROD-AUX(p,n,r) as follows. The upper bound for i on line 6 should be $n - 1$ instead of n.Also, after the for loop, but before line 8, set $q = \max\{q - c, p[i]\}$.

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 $(A_1A_2)((A_3A_4)(A_5A_6))$ 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

$$\sum_{k=0}^{n-1} 2k(n-k).$$

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.

$$\begin{aligned}\sum_{l=2}^{n}\sum_{i=l}^{n-l+1}\sum_{k=i}^{i+l-2} 2 = \sum_{l=2}^{n}\sum_{i=1}^{n-l+1}(l-1)2 &= \sum_{l=2}^{n} 2(l-1)(n-l+1)\\ &= \sum_{l=1}^{n-1} 2l(n-l)\\ &= 2n\sum_{l=1}^{n-1} l - 2\sum_{l=1}^{n-1} l^2\\ &= n^2(n-1) - \frac{(n-1)(n)(2n-1)}{3}\\ &= n^3 - n^2 - \frac{2n^3-3n^2+n}{3}\\ &= \frac{n^3-n}{3}\end{aligned}$$

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 $\infty$.
 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 $C[L]$
```

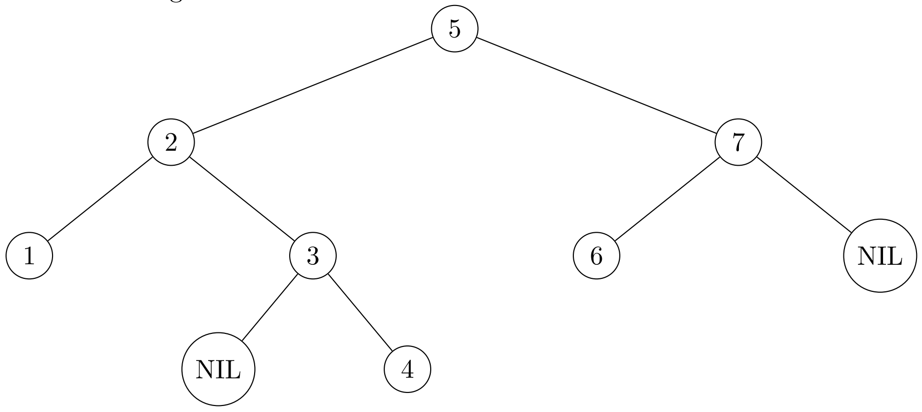
Exercise 15.5-1

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 $l_{k+1}, \dots, l_n$. 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 $l_1, l_2, \dots, l_n$. 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 $\infty$
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 $\nu_0$
end if
for edges $(\nu_0, \nu_1) \in V$ for some $\nu_1$ 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 $\nu_0, res$
        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 $\nu_0$
end if
let $sols.seq = NO - SUCH - PATH$, and $sols.prob = 0$
for edges $(\nu_0, \nu_1) \in V$ for some $\nu_1$ do
    if $\sigma(\nu_0, \nu_1) = \sigma_1$ then
        $res = PROB - VITERBI(G, (\sigma_2, \dots, \sigma_k), \nu_1)$
        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 m^{th} 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 $\leq |S_{ij}| \in O(n)$. 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

$$\begin{aligned} 0000000 &\to a \\ 0000001 &\to b \\ 000001 &\to c \\ 00001 &\to d \\ 0001 &\to e \\ 001 &\to f \\ 01 &\to g \\ 1 &\to h \end{aligned}$$

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 codewords. 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

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 $(x_0, x_1, \dots, x_k)$ 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 $\infty$
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

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 matriod) 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)

```
Initialize an array $D$ of size $n$.
for $i = 1$ to $n$ do
    $a_i.time = a_i.deadline$
    if $D[a_i.deadline] \neq NIL$ then
        $y =$ FIND-SET($D[a_i.deadline]$)
        $a_i.time = y.low - 1$
    end if
    $x =$ MAKE-SET($a_i$)
    $D[a_i.time] = x$
    $x.low = x.high = a_i.time$
    if $D[a_i.time - 1] \neq NIL$ then
        UNION($D[a_i.time - 1], D[a_i.time]$)
    end if
    if $D[a_i.time + 1] \neq NIL$ then
        UNION($D[a_i.time], D[a_i.time + 1]$)
    end if
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,

$$c[i, S] = \min_{x \in \{S\}} (c[i + 1, \{r_i\} \cup (S - \{x\})] + (1 - \chi_{\{r_i\}}(x)))$$

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.

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 of21. 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 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 $size_{i-1}/3 = num_{i-1} = num_i + 1$.

$$\begin{aligned}
\hat{c}_i &= c_i + \phi_i - \Phi_{i-1} \\
&= num_i + 1 + |2 \cdot num_i - size_i| - |2 \cdot num_{i-1} - size_{i-1}| \\
&\leq num_i + 1 + \left(\frac{2}{3}size_{i-1} - 2 \cdot num_i\right) - (size_{i-1} - 2 \cdot num_i - 2) \\
&= num_i + 1 + (2 \cdot num_i + 2 - 2 \cdot num_i) - (3 \cdot num_i + 3 - 2 \cdot num_i) \\
&= 2
\end{aligned}$$

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 amoritized 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 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.

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 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 19

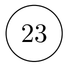
Michelle Bodnar, Andrew Lohr

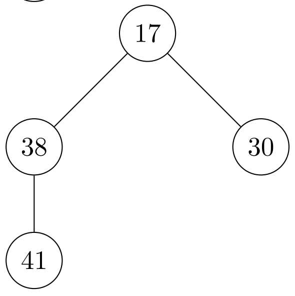
April 12, 2016

Exercise 19.2-1

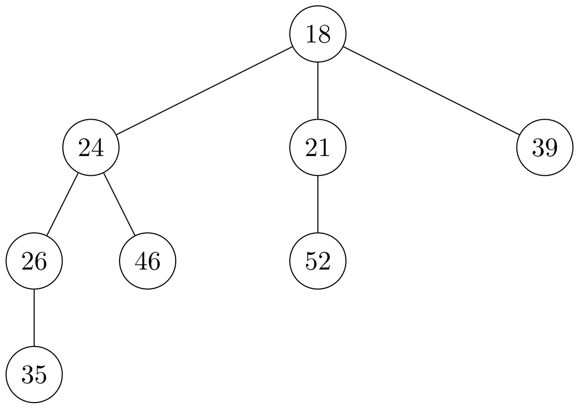
First, we take the subtrees rooted at 24, 17, and 23 and add them to the root list. Then, we set H.min to 18. Then, we run consolidate. First this has its degree 2 set to the subtree rooted at 18. Then the degree 1 is the subtree rooted at 38. Then, we get a repeated subtree of degree 2 when we consider 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 i^{th} 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:

$$\begin{aligned}
f_k &= k + 1 \geq 1 = \alpha^0 \\
f_{k+1} &= k + 3 \geq \alpha^1 \\
&\vdots \\
f_{k+k} &= k + \frac{(k+1)(k+2)}{2} = k + k + 1 + \frac{k(k+1)}{2} \geq 2k + 1 + \alpha^{k-1} \geq \alpha^k.
\end{aligned}$$

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

$$f_{m+k} = f_{m-1-k} + f_m = k + \sum_{i=0}^{m-1} f_i + f_m = k + \sum_{i=0}^{m} f_i.$$

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:

$$\begin{aligned} size(x) &\geq s_m \\ &\geq k + \sum_{i=k}^{m} s_{i-k} \\ &\geq k + \sum_{i=k}^{m} f_i \\ &\geq f_{m+k} \\ &\geq \alpha^m. \end{aligned}$$

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 $k, k-1, \dots, 0$, 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 k^{th} 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 $x, B_0, B_1, \dots, B_{m-1}$ 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 $B_0, B_1, \dots, B_{m-1}$ 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 $B_{i_1}, B_{i_2}, \dots, B_{i_k}$ and $B_{j_1}, B_{j_2}, \dots, B_{j_m}$ 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 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 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 $\Theta(n)$.

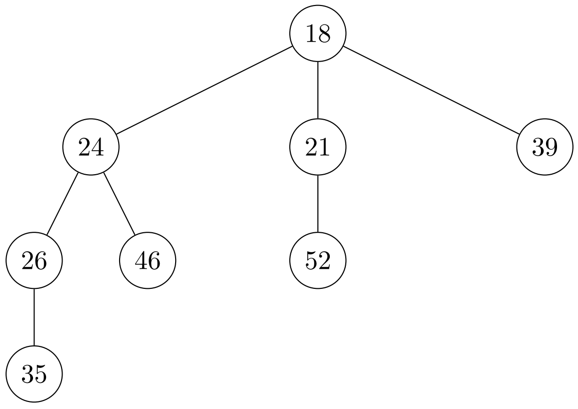
Exercise 21.2-5

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-SET$x_i$
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 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 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 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 $(NIL, \infty)$
$T = \{r\}$
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 $1, \dots, |V|$, 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

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 $O(E^2)$.

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 ordera,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

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)

```
INITIALIZE-SINGLE-SOURCE($G, x$)
$S = \emptyset$
$Q = G.V$
while $Q \neq \emptyset$ do
    $u$ = EXTRACT-MIN($Q$)
    $S = S \cup \{u\}$
    for each vertex $v \in G.Adj[u]$ do
        if $v.d < u.d \cdot r(u, v)$ then
            $v.d = u.d \cdot r(u, v)$
            $v.\pi = u$
        end if
    end for
end while
while $y \neq x$ do
    Print $y$
    $y = y.\pi$
end while
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 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), $\delta(s, v_j) \le \delta(s, v_i) + w(v_i, v_j)$, 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 $x_i - x_j \le b_k$ and $x_j - x_i \le -b_k$, 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 $Ax \le b$ is always satisfied. We also have that $x_i = \delta(v_0, v_i) \le w(v_0, v_i) = 0$ so $x_i \le 0$ 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 $y_i \le \delta(v_0, v_i) = x_i$. 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 $y_{i_2} - y_{i_1} \le w(v_{i_1}, v_{i_2}), \ldots, y_{i_k} - y_{i_{k-1}} \le w(v_{i_{k-1}}, v_{i_k})$. Summing these up we have

$$y_i \le y_i - y_1 \le \sum_{m=2}^{k} w(v_{i_m}, v_{i_{m-1}}) = \delta(v_0, v_i) = x_i.$$

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 $x_i \leq b_k$ if and only if $y_i - x \leq b_k$ and $x_i \geq b_k$ if and only if $y_i - x \geq b_k$. Finally, $x_i - x_j \leq b$ if and only if $y_i - y_j \leq b$. 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 $x_i \leq b_k$, create edge (x, y_i) with weight b_k. For every constraint of the form $x_i \geq b_k$, 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 $x_1, x_2, \dots, x_n$ be the vertices on the cycle. Then we have $x_{i-1}.d + w(x_{i-1}, x) = x_{i-2}.d + w(x_{i-2}, x_{i-1}) + w(x_{i-1}, w_i) = \dots = x_i.d + \sum_{j=1}^{n} w(x_j) < w.d$ 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, $(v_1, \dots, v_{|V|})$ is a topological ordering of the vertices. Similarly, for G_b, $(v_{|V|}, \dots, v_1)$ 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 $v_{k_1}, v_{k_2}, \dots, v_{k_m}$. 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 $x = (x_1, \dots, x_d)$ nests with box $y = (y_1, \dots, y_d)$ and box y nests with box $z = (z_1, \dots, z_d)$. Then there exist permutations π and σ such that $x_{\pi(1)} < y_1, \dots, x_{\pi(d)} < y_d$ and $y_{\sigma(1)} < z_1, \dots, y_{\sigma(d)} < z_d$. This implies $x_{\pi(\sigma(1))} < z_1, \dots, x_{\pi(\sigma(d))} < z_d$, 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:

$$
\begin{aligned}
&R[i_1, i_2] \cdot R[i_2, i_3] \cdot \cdots \cdot R[i_k, i_1] > 1 \\
&\ln(R[i_1, i_2]) + \ln(R[i_2, i_3]) + \cdots + \ln(R[i_k, i_1]) > 0 \\
&-\ln(R[i_1, i_2]) - \ln(R[i_2, i_3]) - \cdots - \ln(R[i_k, i_1]) < 0
\end{aligned}
$$

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

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 i^{th} 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 $s = v_0, v_1, \dots, v_n = v$ 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

$$\begin{aligned}\delta_i(s, v) &= \sum_{m=1}^{n} w_i(v_{m-1}, v_m) \\ &\le \sum_{m=1}^{n} [2w_{i-1}(u, v) + 1] \\ &\le 2\sum_{m=1}^{n} w_{i-1}(u, v) + n \\ &\le 2\delta_{i-1}(s, v) + |V| - 1.\end{aligned}$$

On the other hand, we also have

$$\begin{aligned}\delta_i(s, v) &= \sum_{m=1}^{n} w_i(v_{m-1}, v_m) \\ &\ge \sum_{m=1}^{n} 2w_{i-1}(v_{m-1}, v_m) \\ &\ge 2\delta_{i-1}(s, v)\end{aligned}$$

d. Note that every quantity in the definition of $\hat{w}_i$ is an integer, so $\hat{w}_i$ is clearly an integer. Since $w_i(u, v) \ge 2w_{i-1}(u, v)$, it will suffice to show that $w_{i-1}(u, v) + \delta_{i-1}(s, u) \ge \delta_{i-1}(s, v)$ to prove nonnegativity. This follows immediately from the triangle inequality.

e. First note that $s = v_0, v_1, \dots, v_n = v$ 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 have

$$\begin{aligned}\hat{\delta}_i(s,v) &= \sum_{m=1}^{n} w_i(v_{m-1},v_m) + 2\delta_{i-1}(s,v_{m-1}) - 2\delta_{i-1}(s,v_m)\\ &= \sum_{m=1}^{n} w_i(v_{m-1},v_m) - 2\delta_{i-1}(s,v_n)\\ &= \delta_i(s,v) - 2\delta_{i-1}(s,v)\end{aligned}$$

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 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 $\delta(s,u) \le \delta(s,v) - x$. Or, rearranging, we have $\delta(s,u) + x \le \delta(s,v)$. Since we have inequalities both ways, we must have equality.

d. To see this, we find a vertex v and natural number $k \le n-1$ 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

$$\min_{v\in V} \max_{0\le k\le n-1} \frac{\delta_n(s,v) - \delta_k(s,v)}{n-k} \ge 0$$

The previous part says that there is some v on each minimum weight cycle so that

$$\max_{0\le k\le n-1} \frac{\delta_n(s,v) - \delta_k(s,v)}{n-k} = 0$$

which means that

$$\min_{v\in V} \max_{0\le k\le n-1} \frac{\delta_n(s,v) - \delta_k(s,v)}{n-k} \le 0$$

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 $\min_{v\in V} \max_{0\le k\le n-1} \frac{\delta_n(s,v)-\delta_k(s,v)}{n-k} = 0$. 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 $k \in \{1,\dots,n\}$ 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 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 $0 < x_1y_2 - x_2y_1$, 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 $\overline{p_ip_j}$ 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 $\overline{p_0p_3}$ and $\overline{p_1p_2}$ 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 $\overline{p_0p_1}$. 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 $\overline{p_ip_{i+1}}$ 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 $\triangle p_ip_{i+1}p_{i+2}$, 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 $\triangle p_ip_{i+1}p_{i+2}$. 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 $\overline{b_sa_s}$ is to the right of $\overline{b_sb_f}$, 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

$$x = \frac{b_s.y - m_2b_s.x - a_s.y + m_1a_s.x}{m_1 - m_2}.$$

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

$$b_s.y - m_2b_s.x - a_s.y + m_1a_s.x < (m_1 - m_2)x_0$$

which is equivalent to this gross mess, which fortunately requires only addition, subtraction, multiplication, and comparison, so it is numerically stable:

$$(a_f.x - a_s.x)(b_f.x - b_s.x)(b_s.y - a_s.y) - (a_f.x - a_s.x)(b_f.y - b_s.y)b_s.x + (b_fx - b_s.x)(a_f.y - a_s.y)a_s.x$$
$$< (b_f.x - b_s.x)(a_f.y - a_s.y)x_0 - (a_f.x - a_s.x)(b_f.y - b_s.y)x_0.$$

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 $\overline{p_0p_1}, \overline{p_1p_2}, \ldots, \overline{p_{n-1}p_0}$ of the boundary are between consecutive segments $\overline{p_ip_{i+1}}$ 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 suppsed 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 $\text{CH}(Q_{i-1})$. 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 i^{th} iteration, S contains $\text{CH}(Q_{i-1})$. 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 $\text{CH}(Q_j)$ in counterclockwise order from bottom to top. Thus, when we push p_i, S contains exactly the vertices of $\text{CH}(Q_j \cup \{p_i\})$.

We now show that this is the same set of points as $\text{CH}(Q_i)$. 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 $\text{CH}(Q_i - \{p_t\}) = \text{CH}(Q_i)$. Applying this equality repeatedly for each point removed from S in the while loop of lines 8-9, we have $\text{CH}(Q_j \cup \{p_i\}) = \text{CH}(Q_i)$.

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 $\{q_0, q_1, \dots, q_m\}$, 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 $P = \{p_0, p_1, \dots p_m\}$ 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 $p_i, \dots p_j$ 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 i^{th} step of the algorithm we'll compute $C_i = \text{CH}(\{p_1, \dots, p_i\})$ 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($p_1, p_2, \dots, p_n$)

```
if $n \leq 3$ then
    return $(p_1, \dots, p_n)$
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 $q+1, q+2, \dots, q'-1$ 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 section, 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

$$P = \{(r_i, 0)\} \cup \{(0, \pm i) | i = 1, 2, \dots \ell\} \cup \{(-i, 0) | i = 1, 2, \dots \ell\}$$

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 $n \leq k > 0$, 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

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 $T(n) \in O(n)$.