Chapter 1

Introduction to algorithm design

n/a

Chapter 2

Algorithm analysis

Notes

The dominance pecking order:

$$n! \gg c^n \gg n^3 \gg n^2 \gg n^{1+\epsilon} \gg n \log n \gg n \gg \sqrt{n} \gg$$
$$\log^2 n \gg \log n \gg \log n / \log \log n \gg \log \log n \gg \alpha(n) \gg 1$$

Solutions

2-10

(a) $f(n) = (n^2 - n)/2, \ g(n) = 6n$.

Is $f(n) = O(g(n))$? If so, there is c such that $f(n) \leq cg(n)$ for sufficiently large n.

$$\frac{1}{2}\left(n^2 - n\right) \leq 6n \ \rightarrow \ n^2 - n \leq 12n \ \rightarrow \ n(n-1) \leq 12n$$

Suppose there is such a c, then

$$n(n-1) \leq 12cn \ \rightarrow \ n - 1 \leq 12c$$

Clearly we can always find n such that this inequality won't hold, so $f(n) \neq O(g(n))$.

Is $g(n) = O(f(n))$? If so, there is c such that $g(n) \leq cf(n)$ for sufficiently large n.

$$6n \leq \frac{1}{2}\left(n^2 - n\right) \ \rightarrow \ 12n \leq n^2 - n = n(n-1) \ \rightarrow \ 12 \leq n - 1 \ \rightarrow \ 13 \leq n.$$

So with $c = 1$ the inequality will hold for $n_0 \geq 13$, and $g(n) = O(f(n))$.

(b) $f(n) = n + 2\sqrt{n}, \ g(n) = n^2$.

$f(n) = O(g(n)) \iff f(n) \leq cg(n)$ for sufficiently large n.

$$n + 2\sqrt{n} \leq cn^2, \ \text{with } c = 1,$$
$$n + 2\sqrt{n} \leq 2n \text{ for } n > 4,$$
$$2n \leq n^2 \text{ so } f(n) = O(g(n)).$$

$g(n) = O(f(n)) \iff g(n) \leq cf(n)$ for sufficiently large n. But this asks to find c such that $n^2 \leq c\left(n + 2\sqrt{n}\right)$; since ultimately $n^2 \gg n$, $g(n) \neq O(f(n))$.

(c) $f(n) = n \log n, \ g(n) = n\sqrt{n}$.

$$f(n) = O(g(n)) \iff n \log n \leq cn\sqrt{n}, \ \text{with } c = 1,$$
$$\rightarrow \ \log n \leq \sqrt{n/2},$$

since $\sqrt{n} \gg \log n$, $f(n) = O(g(n))$.

By the same argument, $g(n) \neq O(f(n))$.

(d) $f(n) = n + \log n$, $g(n) = \sqrt{n}$ $\rightarrow$ $n + \log n \leq c\sqrt{n}$, and since $n \gg \sqrt{n}$, any constant factor will be dominated by the linear term, so $f(n) \neq O(g(n))$. Conversely and by the same argument, $g(n) = O(f(n))$.

(e) $f(n) = 2 (\log n)^2$, $g(n) = \log n + 1$. Note that $2 (\log n)^2 = 2\log^2 n$, and $\log^2 n \gg \log n$, so $g(n) = O(f(n))$ and $f(n) \neq O(g(n))$.

(f) $f(n) = 4n \log n + n$, $g(n) = (n^2 - n)/2$. We know that $n \log n \gg n$, so we can consider just this term from $f(n)$. But ultimately the quadratic term in $g(n)$ dominates so $f(n) = O(g(n))$.

2-11

(a) $f(n) = 3n^2$, $g(n) = n^2$.

With $c = 3$, $f(n) \leq 3g(n)$ so $f(n) = O(g(n))$.

$f(n) = \Omega(n)$ $\iff$ $cg(n) \leq f(n)$ for sufficiently large n. For $c = 1$ the inequality holds, so $f(n) = \Omega(g(n))$ and $f(n) = \Theta(g(n))$.

(b) $f(n) = 2n^4 - 3n^2 + 7$, $g(n) = n^5$.

$n^5 \gg n^4$ so $f(n) = O(g(n))$ and $f(n) \neq \Omega(g(n))$.

(c) $f(n) = \log n$, $g(n) = \log n + \frac{1}{n}$.

$\lim_{n \to \infty} \frac{1}{n} = 0$, so as $n \to \infty$, $f(n) - g(n) = 0$. So no function dominates the other. Thus, $f(n) = \Theta(g(n))$.

(d) $f(n) = 2^{k \log n}$, $g(n) = n^k$.

$$f(n) = O(g(n)) \iff f(n) \leq cg(n)$$
$$\rightarrow 2^{k \log n} \leq cn^k; \text{ taking logarithms,}$$
$$\rightarrow \log\left(2^{k \log n}\right) \leq \log\left(cn^k\right) = \log c + \log n^k$$
$$\rightarrow k \log n \log 2 \leq \log c + k \log n.$$

Ignoring constant terms and multiplicative constants, we are left with $\log n \leq \log n$, so $f(n) = \Theta(g(n))$.

(e) $f(n) = 2^n$, $g(n) = 2^{2n}$.

$2^n \leq c2^{2n}$ clearly holds for $c = 1$, so $f(n) = O(g(n))$.

$c2^{2n} \leq 2^n$? Well, $2^{2n} = 2^2 \cdot 2^n = 4 \cdot 2^n$, so $4c2^n \leq 2^n$ is satisfied with $c = 1/4$. So $f(n) = \Omega(g(n))$ and finally, $f(n) = \Theta(g(n))$.

2-12 $n^3 - 3n^2 - n + 1 = \Theta(n^3)$. Note that $0 \leq 3n^2 + n \rightarrow n^3 \leq n^3 + 3n^2 + n \rightarrow n^3 - 3n^2 - n \leq n^3$. Thus $f(n) = O(n^3)$. Now $cn^3 \leq n^3 - 3n^2 - n + 1$. Consider $c = 1/2$, then

$$n^3/2 \leq n^3 - 3n^2 - n + 1$$
$$-n^3/2 \leq -3n^2 - n + 1$$
$$-n^3 \leq -6n^2 - 2n + 2$$
$$n^3 \geq 6n^2 + 2n - 2.$$

This holds for $n_0 \geq 7$, so $f(n) = \Omega(n^3)$ and finally $f(n) = \Theta(n^3)$.

2-13 $f(n) = n^2 = O\left(2^n\right)$ $\iff$ $f(n) \leq c2^n$, after some n_0. For $c = 1$,

$$n^2 \leq 2^n$$
$$\log(n^2) \leq \log(2^n)$$
$$2\log n \leq n \log 2$$
$$\log n \leq kn, \ k = \frac{\log 2}{2}$$

Since $\log n \ll n$ this inequality holds for large enough n, and $n^2 = O(2^n)$.

2-14 $\Theta(n^2) = \Theta(n^2 + 1)$? This is to say whether both classes are the same. We can show this by assuming we have $f(n) = \Theta(n^2)$, and proving that $f(n) = \Theta(n^2 + 1)$, and likewise with the other assumption.

First, $f(n) = \Theta(n^2) \to f(n) = O(n^2)$ and $f(n) = \Omega(n^2)$. Is $f(n) = O(n^2 + 1)$? This would mean $f(n) \leq c\left(n^2 + 1\right)$, for $n > n_0$. Since $f(n) = O(n^2)$ there is c_0 such that $f(n) \leq c_0 n^2$, but $c_0 n^2 \leq c_0\left(n^2 + 1\right)$ so with the same c_0 we see that $f(n) = O(n^2 + 1)$.

Now we want to show that $f(n) = \Omega(n^2 + 1)$, that is, $c\left(n^2 + 1\right) \leq f(n)$ for $n > n_0$. Since $f(n) = \Omega(n^2)$ there are c_1, n_1 such that $c_1 n^2 \leq f(n)$, for $n > n_1$. In particular this holds for $n + 1 > n_1$, so

$$c_1 n^2 < c_1(n+1)^2 \leq f(n), \quad n > n_1.$$

But note that $n^2 + 1 \leq (n+1)^2$, so for $n_0 = n_1 + 1$ above, we have that $c_1\left(n^2 + 1\right) \leq c_1(n+1)^2 \leq f(n)$ for $n > n_0$. So $f(n) = \Omega(n^2 + 1)$ and thus $f(n) = \Theta(n^2 + 1)$.

Now we need to show that $f(n) = \Theta(n^2 + 1) \to f(n) = \Theta(n^2)$.

$f(n) = \Omega(n^2) \iff cn^2 \leq f(n)$ for $n > n_0$. We know that $f(n) = \Omega(n^2 + 1)$ so there is a c_1 such that $c_1\left(n^2 + 1\right) \leq f(n)$ for $n > n_1$; since $n^2 < n^2 + 1$, by letting $c = c_1$ we see that $c_1 n^2 < c_1\left(n^2 + 1\right) \leq f(n)$, so $f(n) = \Omega(n^2)$.

$f(n) = O(n^2) \iff f(n) \leq cn^2$ for $n > n_0$. Since $f(n) = O(n^2 + 1)$ we know there are c_1 and n_1 such that $f(n) \leq c_1\left(n^2 + 1\right)$ for $n \geq n_1$; in particular, $f(n) \leq c_1\left(n_1^2 + 1\right)$. Then let $c = c_1\left(n_1^2 + 1\right)$, then $cn^2 > c_1\left(n_1^2 + 1\right) \geq f(n)$, for $n > n_1$. Thus $f(n) = O(n^2)$ and $f(n) = \Theta(n^2)$.

This shows that if a function is $\Theta(n^2)$ then it must be $\Theta(n^2 + 1)$ and viceversa—that is, both classes are the same.

2-17

a) $f(n) = n^2 + n + 1$, $g(n) = 2n^3$. We want to find $c > 0$ such that $f(n) \leq cg(n)$ for $n > 1$. $f(2) = 7$; $g(2) = 16 \to c = 1$.

b) $f(n) = n\sqrt{n}$, $g(n) = n^2$. $n\sqrt{n} < n^2 \to 2n^2 > n\sqrt{n} + n^2 \to c = 2$.

c) $f(n) = n^2 - n + 1$, $g(n) = n^2/2$. $n^2 - n + 1 < \frac{c}{2}n^2 \to$ if $n = 2$, $f(2) = 3$, and $g(2) = 2$. For $c = 2$, $n^2 - n + 1 < n^2. (f(2) = 3, 2g(2) = 4)$.

2-18 Let $f_1(n) = O(g_1(n))$, $f_2(n) = O(g_2(n))$. Show that $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$.

There are c_1, n_1 such that $f_1(n) \leq c_1 g_1(n)$ for $n > n_1$, and c_2, n_2 such that $f_2(n) \leq c_2 g_2(n)$ for $n > n_2$. Let $c = \max(c_1, c_2)$, $n_0 = \max(n_1, n_2)$. Then $f_1(n) < cg_1(n)$, and $f_2(n) < cg_2(n)$, which implies that $f_1(n) + f_2(n) < cg_1(n) + cg_2(n) = c(g_1(n) + g_2(n))$ for $n > n_0$. ∎

2-19 Let $f_1(n) = \Omega(g_1(n))$, $f_2(n) = \Omega(g_2(n))$. Then there are c_1, n_1 such that $f_1(n) \geq c_1 g_1(n)$ for $n > n_1$, and c_2, n_2 such that $f_2(n) \geq c_2 g_2(n)$ for $n > n_2$. Let $n_0 = \max(n_1, n_2)$ and let $c_0 = \min(c_1, c_2)$. Then $cg_1(n) \leq c_1 g_1(n)$ and also $cg_2(n) \leq c_2 g_2(n)$. These inequalities imply that

$$cg_1(n) \leq f_1(n), \ n > n_0,$$
$$cg_2(n) \leq f_2(n), \ n > n_0.$$

Thus $c(g_1(n) + g_2(n)) \leq f_1(n) + f_2(n), \ n > n_0$, and $f_1(n) + f_2(n) = \Omega(g_1(n) + g_2(n))$.

2-20 Let $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$. Then there are c_1, n_1, c_2, n_2 such that $f_1(n) \leq c_1 g_1(n)$ for $n > n_1$, and $f_2(n) \leq c_2 g_2(n)$ for $n > n_2$. Let $c = \max(c_1, c_2)$ and $n_0 = \max(n_1, n_2)$. Then $f_1(n) \leq cg_1(n)$ and $f_2(n) \leq cg_2(n)$ for $n > n_0$, which implies that $f_1(n) \cdot f_2(n) \leq c(g_1(n) \cdot g_2(n))$. Thus $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$.

2-21 We are to prove that $p(n) = a_k n^k + \ldots + a_0 = O(n^k)$, for $k \geq 0$ and arbitrary real coefficients a_i. This is to say that we can find a $c > 0$ and n_0 such that $p(n) \leq cn^k$, for all $n > n_0$.

We know that after some n, $a_k n^k > p(n) - a_k n^k$, that is to say, the leading order term will come to dominate. To see this, note that

$$\lim_{n \to \infty} \frac{n^m}{n^q} = 0 \iff q > m,$$

is to say that $n^q \gg n^m$.

Now take $c = \max(a_0, \ldots, a_k)$. Then clearly $cn^k + \ldots + c \geq a_k n^k + \ldots + a_0 = p(n)$. But because of the same argument as above, there is some n_0 after which $cn^k > cn^{k-1} + \ldots + c$. Then it follows that $p(n) \leq cn^k$ for $n > n_0$, which means $p(n) = O(n^k)$. ∎

2-22   Let $a, b \in \mathbb{R}$, with $b > 0$. Show that $(n + a)^b = \Theta(n^b)$.

To prove that $(n+a)^b = O(n^b)$ we need to find c_0 such that $c_0 n^b \geq (n+a)^b$ for sufficiently large n. If $a = 0$, $(n+a)^b = n^b$ and with $c = 1$ the inequality holds. If $a < 0, (n + a)^b \leq n^b$, so

$$\frac{(n+a)^b}{n^b} = \left(\frac{n+a}{n}\right)^b < 1.$$

If $a > 0$ we need to see that $\left(1 + \frac{a}{n}\right)^b$ is bounded. (To see why, $(n + a)^b \leq cn^b \rightarrow (n + a/n)^b \leq c \rightarrow (1 + a/n)^b \leq c$).

Note that $\frac{a}{n}$ can get arbitrarily close to 0 for large enough n, so the entire expression tends to 1 as $n \to \infty$. Thus for, say, $c = 2$ it is possible to find n_0 large enough such that $(n + a)^b \leq 2n^b$. So $(n + a)^b = O(n^b)$.

To prove that $(n + a)^b = \Omega(n^b)$, by the same argument, $cn^b \leq (n + a)^b \rightarrow c \leq (n + a/n)^b = (1 + a/n)^b$. For $a \neq 0$ the expression $1 + a/n$ tends to 1, either "from below" or "from above"—at any rate, by taking some $c < 1$, say, $1/2$, it will be possible to find n large enough that the inequality will hold. Thus $(n + a)^b = \Omega(n^b)$, and finally $(n + a)^b = \Theta(n^b)$. ∎

2-27

(a) $f(n) = o(g(n))$ and $f(n) \neq \Theta(g(n))$. If $f(n) = o(g(n))$ then $g(n) \gg f(n)$—they are on different classes. But $f(n) \neq \Theta(g(n))$ implies that either $f(n) \neq \Omega(g(n))$ or $f(n) \neq O(g(n))$. Take $f(n) = n$, $g(n) = n^2$. Clearly $f(n) \neq \Omega(g(n))$ but $g(n) \gg f(n)$ so $f(n) = o(g(n))$.

(b) $f(n) = \Theta(g(n))$, $f(n) = o(g(n))$. If $f(n) = \Theta(g(n))$ then $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, that is, they are in the same class. So it's not possible for both $f(n) = \Omega(g(n))$ and $f(n) = o(g(n))$ to hold simultaneously.

(c) $f(n) = \Theta(g(n))$ and $f(n) \neq O(g(n))$. None, by definition.

(d) $f(n) = \Omega(g(n))$ and $f(n) \neq O(g(n))$. $f(n) = n^2$, $g(n) = n$.

2-29

(a) $f(n) = n^2 + 3n + 4$, $g(n) = 6n + 7 \rightarrow f(n) = \Omega(g(n))$.

(b) $f(n) = n\sqrt{n}$, $g(n) = n^2 - n \rightarrow f(n) = \Omega(g(n))$.

(c) $f(n) = 2^n - n^2$, $g(n) = n^4 + n^2 \rightarrow f(n) = \Omega(g(n))$.

2-30

(a) Yes—$O(n^2)$ worst case time doesn't necessarily mean it ever takes n^2 steps on any input. If the algorithm was $O(n)$ worst case, it would still be $O(n^2)$.

(b) Yes—$O(n^2)$ only talks about an upper bound.

(c) Yes. It could be that the best case time complexity of the algorithm is $O(n)$.

(d) No. Some inputs will trigger worst case behavior so will necessarily be $\Theta(n^2)$.

(e) Yes. We ignore multiplicative constants and terms of lower degree.

2-31

(a) Is $3^n = O(2^n)$? Only if there is c such that $3^n \leq c2^n$ for sufficiently large n. But if there was such c, then $(3/2)^n \leq c$, and since $3/2 > 1$, $(3/2)^n \to \infty$ as $n \to \infty$, thus $3^n \neq O(2^n)$.

(b) Is $\log 3^n = O(\log 2^n)$? Only if there is c such that $\log 3^n \leq c \log 2^n$ for sufficiently large n. Note that $\log 3^n = n \log 3$, and $c \log 2^n = cn \log 2$. So we want c such that $n \log 3 \leq cn \log 2 \rightarrow \log 3 \leq c \log 2 \rightarrow \frac{\log 3}{\log 2} \leq c$. The answer is yes, $\log 3^n = O(\log 2^n)$.

(c) Is $3^n = \Omega(2^n)$? Only if there is c such that $c2^n \leq 3^n$ for large enough n. Trivially visible for $c = 1$.

(d) Is $\log 3^n = \Omega(\log 2^n)$? Only if there exists c such that $c \log 2^n \leq \log 3^n$. Since log is monotonically increasing and $2^n \leq 3^n$, also trivially visible for $c = 1$.

2-34

(a) $\sum_1^n 3^i = \Theta(3^{n-1})$? This would imply both $\Omega(3^{n-1})$ and $O(3^{n-1})$. For the first we need c such that $c3^{n-1} \leq \sum_1^n 3^i = 3^n + \ldots + 3$. Well, for $c = 1$ this inequality holds.

Now for $O(3^{n-1})$ we need c such that $\sum_1^n 3^i \leq c3^{n-1}$. Let $c = 9$, then $c3^{n-1} = 9 \cdot 3^{n-1} = 3^{n+1} \geq \sum_1^n 3^i$ for large enough n.

Thus the answer is true.

(b) $\sum_1^n 3^i = \Theta(3^n)$? True.

(c) $\sum_1^n 3^i = \Theta(3^{n+1})$? True.

For $\Omega(3^{n+1})$: $c3^{n+1} \leq \sum_1^n 3^i = 3^n + \ldots + 3$. Let $c = 1/9$, then $c3^{n+1} = (1/9) \cdot 9 \cdot 3^{n-1} = 3^{n-1} \leq \sum_1^n 3^i$ as stated before.

For $O(3^{n+1})$: $\sum_1^n 3^i \leq c3^{n+1}$ for $c = 1$.

2-35

(a) $g(n) = 4^n$

(b) $g(n) = n \log n$

(c) $g(n) = \log^{10} n$

(d) $g(n) = n^{100}$

2-37 Each number in row $n-1$ is added exactly three times into row n. So if the value of the sum of row $n-1$ is T, then the sum of the values of row n is $3T$. This suggests a recurrence

$$T_1 = 1 = 3^0,$$
$$T_n = 3T_{n-1}, \quad n > 1.$$

The recurrence seems to hold, at least up to $n = 5$. This suggests a closed form $t(n) = 3^{n-1}$, $n \geq 1$. Base case $n = 1$, clearly $t(1) = 3^0 = 1 = T_1$. Now assume this is true up to n, then $t(n+1) = 3^{n+1-1} = 3^n = 3 \cdot 3^{n-1} = 3t(n) = 3 T_{n-1} = T_n$.

2-39 $\sum^{n+1} i = \sum^n i + (n+1) = n(n+1)/2 + n + 1$. Traverse the array adding up all the numbers and subtract the result from $\sum^{n+1} i$. The difference is the missing element.

2-40 The fragment:

```
for i=1 to n do
   for j=i to 2*i do
      output "foobar"
```

a. $T(n) = \sum_{i=1}^n \sum_{j=1}^{2i} 1$. Let's look at the inner sum. There are i elements between i and $2i$, so

b. $\sum_i^{2i} 1 = \sum_1^i 1 = i$. Then $T(n) = \sum_{i=1}^n i = n(n+1)/2$.

2-41 The fragment:

```
for i=1 to n/2 do
   for j=i to n-i do
      for k=1 to j do
         output "foobar"
```

a. $T(n) = \sum_{i=1}^{n/2} \sum_{j=i}^{n-i} \sum_{k=1}^j 1$.

b. $\sum_{k=1}^{j} 1 = j \to T(n) = \sum_{i=1}^{n/2} \sum_{j=i}^{n-1} j$.

Now consider $\sum_{j=i}^{n-i} j$. When $i = 1$, j goes from 1 to $n - 1$; for $i = 2$, j goes from 2 to $n - 2$. So for some given i, this sum has $n - i - i = n - 2i$ terms, from i to $n - i$. We can add the numbers from 1 to $n - i$, $\sum_{1}^{n-i} i = \frac{1}{2}(n - i)(n - i + 1)$, and take from that the sum from 1 to $i - 1$, $\sum_{k=1}^{i-1} k = \frac{1}{2}(i - 1)(i - 1 + 1) = \frac{i(i-1)}{2}$. Then the sum $\sum_{j=i}^{n-i} j = \sum_{k=1}^{n-i} k - \sum_{k=1}^{i-1} k = \frac{n^2 + n}{2}$.

Back to $T(n)$. We now have the outer sum $T(n) = \sum_{i=1}^{n/2} \frac{n^2 + n}{2} = (n/2)(\frac{n^2 + n}{2}) = \frac{1}{4}(n^3 + n^2)$.

2-42   Suppose we have two n-digit numbers in base b,

$$x = b^{n-1}x_{n-1} + b^{n-2}x_{n-2} + \ldots + bx_1 + b^0 x_0$$
$$y = b^{n-1}y_{n-1} + \ldots + b^0 y_0.$$

Each number is represented in base b as the concatenation of the digits x_i, as in $x =$ "$x_{n-1}x_{n-2}\ldots x_0$". Their product will be $xy = (b^{n-1}x_{n-1}+\ldots+x_0)(b^{n-1}y_{n-1}+\ldots+y_0)$ and the highest order term of this product will be $b^{n-1}x_{n-1}b^{n-1}y_{n-1} = (b^{n-1})^2 x_{n-1}y_{n-1}$. Every other term in this product will be of lower order, so the complexity will be driven by the number b^{2n-2}.

We could say that $b^{2n-2}xy$ takes $O(1)$ for xy, which is then added to itself b^{2n-2} times. Then the complexity of multiplying two numbers of n digits in base b is $O(b^{2n-2})$. Note that there are a lot of hidden multiplications in $b^{2n-2} = b \cdot b^{2n-3}$. If b is one digit then it takes $O(1)$, but for $b = 10$, well, I don't know.

2-44

(a) By definition $a^{\log_a x} = x$. So $xy = a^{\log_a x}a^{\log_a y} = a^{\log_a x + \log_a y}$. Taking logarithms, $\log_a xy = \log_a a^{\log_a x + \log_a y} = \log_a x + \log_a y$.

(b) Consider $a^{y \log_a x} = (a^{\log_a x})^y = x^y$. Then taking logarithms, $\log_a x^y = \log_a a^{y \log_a x} = y \log_a x$.

(c) Consider $\log_a x \log_b a \to b^{\log_a x \log_b a} = (b^{\log_b a})^{\log_a x} = a^{\log_a x} = x$. Taking logarithms, $\log_b x = \log_b b^{\log_a x \log_b a} = \log_a x \log_b a$. Equivalently, $\log_a x = \frac{\log_b x}{\log_b a}$.

(d) $\log_b y = w \to y = b^w$, so $y^{\log_b x} = (b^w)^{\log_b x} = (b^{\log_b x})^w = x^w = x^{\log_b y}$.

Chapter 3

Data structures

Solutions

3-5

a) Suppose the array has size 2^n. This underflow strategy has us release the top half of our allocated memory when we come down to 2^{n-1} items.

$$\square\square\square\square\square\square\square\square \quad \rightarrow \quad \square\square\square\square$$

If our program now adds an item to the dynamic array the same space needs to be allocated that was just freed, potentially moving all 2^{n-1} items. Imagine this delete-one/add-one cycle repeats, such as may happen with a stack backed by this dynamic array. Each append-one operation copies the (now) lower half to a new location, taking linear time on the number of items.

b) The problem with that underflow strategy is that both the "grow" and "shrink" events are at the same threshold—when half full, shrink by half thus making it full again, but this guarantees the next append will trigger a "grow". If the two thresholds are dissociated, we will avoid this pathological behavior. For this, only shrink the array to half size when it is a fourth full.

$$\square\square\square\square\square\square\square\square \quad \rightarrow \quad \square\square\square\square$$

We are then at the situation after a grow operation just duplicated the size of the array for us, thus amortizing the cost of grow/shrink operations.

3-6 Skiena's fridge works as a stack, so unless he takes care of unstacking every food item regularly (thus emptying the fridge), that is bad news for the first items inserted.

One improvement might be to use a queue, whatever has been in there the longest is consumed first. However that is still a naive strategy—if an item expiring tomorrow is inserted after one that will expire in a year, I still risk the last item expiring.

The answer is a *priority queue.* Prioritize the item that expires next. This ensures that items that have longer expiration dates wait the most.

3-7 The book says to keep a sentinel for the end of the list. There are three cases for deleting a node `p`: 1) delete the first node; 2) delete a node in the middle; 3) delete the last node. Suppose we have a list `l`, represented here by the first square.

1) Deleting the head: $\square\longrightarrow\bullet\longrightarrow\circ\longrightarrow\square$. Point `l` to `p->next` then free `p`: $\square \quad \circ\longrightarrow\circ\longrightarrow\square$.

2) Deleting a node in the middle of the list: $\square\longrightarrow\circ\longrightarrow\bullet\longrightarrow\circ\longrightarrow\square$. Overwrite `p` with `p->next`, then free `p->next`: $\square\longrightarrow\circ\longrightarrow\circ \quad \circ\longrightarrow\square$.

3) Deleting the last node: $\square\longrightarrow\circ\longrightarrow\bullet\longrightarrow\square$. Free `tail` and make `p` the new sentinel node: $\square\longrightarrow\circ\longrightarrow\blacksquare \quad \square$.

3-8 The winning condition is for n X or O to be placed in a row, column or diagonal. We can keep X and O counters for every row, column and diagonal. There are n rows, n columns and 2 diagonals, so we need $2n + 2$ counters to keep state, which is $O(n)$ space as requested.

Consider a move to be represented as a tuple indicating the position played and the player, for example ((1,1), X). Given a list of legal moves alternating players, execute each move by increasing the affected row and column, and diagonal counter for the player. The final move is a winning move if any of its affected counters are equal to n.

```rust
enum Player {
    X = 0,
    O = 1,
}

struct Counter {
    counter: [u32; 2],
}

type Move = (usize, usize, Player);

struct TicTacToe {
    n: usize,
    rows: Vec<Counter>,
    cols: Vec<Counter>,
    diag: [Counter; 2],
    last_move: Option<Move>,
}
```

3-9 Each sequence of digits maps to multiple potential words—we have a one-to-many relation. Conversely, a word maps to a unique numeric representation with as many digits as the words as letters.

If multiple queries were necessary, we might justify transforming the dictionary into a hash map of numbers to lists of words. If words are on average m characters long, and there are n of them, we would pay $O(nm)$ complexity in traversing the entire dictionary and hashing every word.

Suppose we are to use the given dictionary only to collect all words that match our sequence of digits. If our given sequence of digits is s digits long, we will have at most 4 possible characters associated with each digit, so there will be 4^s letter combinations to look up in the dictionary. So it looks like we are after all better off hashing the entire thing and querying directly with the sequence.

```rust
fn hash_char(c: char) -> char {
    match c {
        'a' | 'b' | 'c' => '2',
        ...
        'w' | 'x' | 'y' | 'z' => '9',
        _ => unreachable!(),
    }
}

fn hash_dictionary() -> HashMap<String, Vec<String>> {
    let mut result = HashMap::new();
    for (key, word) in WORDS.iter().map(|w| {
        let hash: String = w.chars().map(hash_char).collect();
        (hash, w.to_string())
    }) {
        result.entry(key).or_default().push(word);
    }
    result
}
```

3-10   The best we could possibly do is linear time—we must check that all letters match somehow. Assign subsequent prime numbers to each character in the alphabet. Hash each string by multiplying the prime numbers corresponding to each character in the string. This takes $O(n)$ time on the length of the string. If two strings are anagrams of each other, their hash will be the same number by virtue of the Fundamental Theorem of Arithmetic.

3-11   Constant time search means *random access*, so we will represent the dictionary using an array of length n. The universe of possible elements are the positive naturals up to and including n, so the absence of element k can be signaled by setting the kth element of the backing array to 0.

3-12   Maximum depth of a tree:

```
data Tree = Node Tree Tree
          | Nil

maxDepth :: Tree -> Int
maxDepth Nil = 0
maxDepth (Node l r) = 1 + max (maxDepth l) (maxDepth r)
```

3-14   Merging two binary search trees into a doubly linked list. The in order traversal of each tree provides ordered lists of their respective elements. A linear sweep, choosing the minimum of each head and advancing in the chosen list will merge the two structures. The cost of both the traversal and sweep/merge stages is $O(m + n)$, where m and n are the number of elements in each binary search tree.

Note that this is fundamentally the idea behind mergesort.

3-15   There is presumably an insertion order that guarantees height balance. Knowing all the elements and having them in order is a great advantage. Some thoughts: clearly the extremes have to be inserted among the last few elements–their levels have to be created for them to land in the right place. Also, the median element of the array has to be the root, as it gives the most "space" to its sides.

That median element determines two halves. Each half has its own median element to which the same thinking process applies. Therefore a potential algorithm for building a height balanced tree is:

```
balanced_insertion(lo, hi):
  m <- median(lo, hi)
  insert element m
  balanced_insertion(lo, m-1)
  balanced_insertion(m+1, hi)
```

With some provisions for ranges of length 1, in which case the element is a leaf and can be inserted with no further recursive calls.

Before doing the insertion, the binary search tree has to be traversed in order to build an array of all its elements. This takes $O(n)$ time.

Consider the call stack of `balanced_insertion()` with the full range of n elements. It makes two recursive calls, so the call stack is shaped like a binary tree. The height of the call tree is at most $\lg n$, because the range of elements in each recursive call is more than halved with respect to the range of its caller. Then this tree can have at most $2^{1+\lg n} = 2 \cdot 2^{\lg n} = 2n$ nodes, each representing a call to `balanced_insertion()`. Since each call does constant time work, this function's running time is $O(n)$ as well.

3-17   The definition of height balanced tree is recursive, but the recursive aspect is implicit in the way it's worded in the book. An explicit way to say it is: a binary tree is height balanced if both its children are height balanced and the difference between their heights is at most 1. A node with no children is balanced and has height 1 (or, equivalently, a "null" node is balanced and has height 0).

```
data Tree = Node Tree Tree
          | Nil

isBalanced :: Tree -> (Bool, Int)
```

```
isBalanced Nil = (True, 0)
isBalanced (Node l r) = (balanced, height)
  where (lb, lh) = isBalanced l
        (rb, rh) = isBalanced r
        height   = 1 + max lh rh
        balanced = lb && rb && abs (lh - rh) <= 1
```

Each call performs constant-time work, and there is only one call for each node in the tree. Therefore this is an $O(n)$ time algorithm.

3-18   We have a balanced tree where all of `search()`, `insert()`, `delete()`, `minimum()` and `maximum()` take $O(\log n)$ time, and we want to ensure it supports `successor()` and `predecessor()` in $O(1)$ time. Each node will have pointers to its logical predecessor and successor that will have to be maintained during update operations. This would effectively add a doubly linked list structure on top of the binary tree.

`successor()`: see pp. 85 in the book. The in order successor can be found, with a parent pointer, by considering two different cases: if the node has a right subtree, and if it hasn't.

`predecessor()`: finding it is symmetric to the successor.

- If a node is a leaf, and is the left child of its parent, traverse up until finding a node that is the right child of its parent. The parent will be the predecessor.

- If a node is a leaf and is a right child, this is the trivial case of the previous point—the parent is the predecessor.

- If a node has a left subtree, the predecessor is the maximum of that left subtree.

When deleting an item from the tree, we can get ahold of its successor and predecessor in $O(1)$ time via the pointers, delete the item, then link the pointers as done in a doubly linked list.

Note that the exercise says the tree is balanced. We can assume there is a balancing operation that takes place after each insert and delete, before the successor and predecessor pointers are updated.

3-19   We have a dictionary with $O(\log n)$ `search()`, `insert()`, `delete()`, `min()`, `max()`, `predecessor()` and `successor()`, and we want to make some changes to `insert()` and `delete()` so `min()` and `max()` will take $O(1)$ time while the update operations still take $O(\log n)$ time.

After `insert()` we can query for the new element's predecessor in $O(\log n)$ time—if there isn't one, our new element is the new minimum. Similarly with `maximum()` and `successor()`.

Before `delete()` we can query for the predecessor of the element—if there isn't one we can find the new minimum by querying for its successor (although in reality we'd know we are trying to delete the minimum because we have a $O(1)$ minimum); likewise when deleting the maximum. If before delete we do find a more extreme element, we know we are not deleting the minimum or maximum and no adjustment is necessary.

3-20   The exercise calls this structure a "set", and by its operations it does look like a set, so we'll assume the elements are unique. Furthermore, we will assume we have a balancing operation that keeps the tree height balanced in $O(\log n)$ time.

Our set structure is backed by a balanced binary search tree. The `member()` query searches for the element, taking $O(\log n)$ time. The `insert()` operation is the regular BST insert—if the key already exists, it is overwritten; this also takes $O(\log n)$ time. Finally, `delete()` has to find the k-th smallest element. Assume our BST has $O(\log n)$ `minimum()` and $O(1)$ `successor()`, as in the previous exercise. Then we need to query for the minimum element and then k calls to `successor()`, then one $O(\log n)$ delete and one $O(\log n)$ rebalance.

Unfortunately this doesn't work—consider always deleting the last element. After finding the minimum, we always do n calls to `successor()`, turning this into a $O(n)$ algorithm in the worst case. :\

3-21   This may be cheating somehow, but if the two sets are disjoint and all keys in S_1 are less than every key in S_2, we can concatenate both trees by finding the minimum element in S_2 and making the root of S_1 its left child. It would wreak havoc on the balance of the tree, but rebalancing should take at most $O(n + m)$ time, for n elements in S_1 and m elements in S_2. The traversal of S_2 to its minimum would be $O(\log m)$ and the concatenation—a simple matter of pointer manipulation—would be constant time.

3-22 We're to design a data structure that supports `insert()` and `median()` operations, both in $O(\log n)$ time. Inserting is easy enough, but the median is an element relative to all other elements in the structure; it depends on its logical position within the set of all the elements, so it's not clear that it can be determined without somehow keeping track of counts.

Suppose the structure keeps two binary search trees, the median, and counters for the elements in each tree. The elements in the left tree are less than the median, while those on the right tree are greater than the median.

When inserting, we compare the element to the median to select which side to insert it into. It's then inserted in the correct tree, taking $O(\log n)$ time—actually $O(\log m)$, with m the number of elements on that side of the median. We increase the counter on this side, and if the difference in elements between both sides is greater than 1, we know we need to shift the median.

To rebalance the structure, suppose the left side has two more elements than the right side. We can insert the median in the right tree in $O(\log n)$ time, then find and delete the maximum from the left tree, also in $O(\log n)$ time, and set it as the new median. The right counter increases by one, the left counter decreases by one, and we are balanced again.

This actually gives $O(1)$ access to the median, so it is probably wrong for the exercise.

3-23 Let p be some prefix. We wish to find all strings in a dictionary D that start with p.

Begin by querying the dictionary for p. If the prefix is not in D, insert it and make a note of it. Now query for the prefix to get a pointer x to it in D. Call `successor(D,x)` repeatedly, printing all the strings that match the prefix, stopping at the first one that does not. Finally, if the prefix was not in the dictionary to begin with, delete it from D.

Comparing strings is not like comparing integers—it takes time linear on the length of the strings. For prefix matching, since strings in D are at most l characters long, comparisons will take $O(l)$ time in the worst case. So the initial query for p does at most $\log n$ comparisons, each taking $O(l)$ time, so this takes $O(l \log n)$ time. Insertion, querying again and the potential delete at the end also take $O(l \log n)$ time. Each call to `successor()` is $O(\log n)$, and there will be m matching strings each verified by a $O(l)$ comparison, for a total of $O(ml \log n)$ time complexity.

3-24 An array is k-unique if no elements within a k-wide range are equal. For example $\langle 1, 2, 3, 2, 4 \rangle$ is not 2-unique, and $\langle 2, 0, 1, 3, 5, 0 \rangle$ is 3-unique but not 4-unique. Note that if an array is not k-unique, then it's not $(k+1)$-unique. If an array is k-unique, then it is $(k-1)$-unique.

A binary tree with k elements can be built in $O(k)$ time. So do this:

1) Insert the first k elements into the tree, while checking for duplicates. Each query is $O(\log k)$ as is each insertion, and there are k of them so this is $O(k \log k)$.

2) If no dupes were found, remove the first item in the array from the tree, leaving a tree with $k-1$ elements. This takes $O(\log k)$ time.

3) Query for the $(k+1)$th element of the array—if it is in the tree, the array is not k-unique. Otherwise, insert the element into the tree and go back to step 2), subsequently removing the lowest indexed element in the array from the tree.

4) If the last element of the array is successfully inserted, this means we didn't find any duplicates within a k element window, so the array is k-unique.

There are at most n $O(\log k)$ groups of operations as the array is traversed while querying, so this algorithm is $O(n \log k)$ in the worst case.

3-25 We can represent the bins as a binary search tree, where each node is an integer representing the available space in that bin. This will bring trouble with bins simultaneously having the same available space, but let's ignore that.

We would insert a weight by querying for the minimum in the binary tree in $O(\log n)$ time—if the weight fits, save the updated weight in a new node, delete the stale bin in $O(\log n)$ time and insert the updated node, again in $O(\log n)$ time. If the weight does not fit in the first bin returned, call `successor()` until one is found that has enough room. This search takes $O(n \log n)$ time in the worst case, as we go through all the bins. If no bin has space, create and insert a new node with the current weight inside. The minimum number of bins at the end is the number of nodes in the tree.

3-26

a) Make an n-by-n array and populate it in $O(n^2)$ time with the minimum value in $x_i, \ldots, x_j$ for i, j in $[1 .. n]$. Then given i and j, just return whatever is in the i, j position of the matrix in $O(1)$.

b) Consider an unsorted array and take the minimum, say x_i. This element partitions the array into a left side, from x_1 to x_{i-1}; the minimum itself, x_i; the right side, from x_{i+1} to x_n.

   Now make the root of a tree by pairing up (x_i, i), the minimum and its position. The left subtree's root node is the minimum in the range $[x_1 .. x_{i-1}]$ and itself splits this range in two sides (plus itself). Likewise with the right subtree and the range $[x_{i+1} .. x_n]$.

   Given a query range, we start on the root of the tree. If i is in the query range, we have our (global) minimum. Otherwise the range is entirely to the left or right of i. Traverse accordingly.

   By following this tree until we hit a minimum that is in the input interval we will find the minimum value in $O(\log n)$ steps. Since each node is its value plus its index this takes $O(n)$ space.

3-27   From doing a couple of experiments, it seems that given some k, using the oracle to test $f(S, k - x_i)$ for each $x_i \in S$ selects and eliminates the correct elements.

   Suppose a subset $T = \{x_1, x_2, x_3\}$ is a solution for set $S = \{x_1, x_2, x_3, x_4\}$ and k. That means $k = x_1 + x_2 + x_3$, and implies that there are subsets of S that add up to $k - x_1 = x_2 + x_3$, $k - x_2 = x_1 + x_3$, and $k - x_3 = x_1 + x_2$.

   If $f(S, k - x_4)$ is 0, that means no selection of elements x_j will fulfill $k - x_4 = \sum_j x_j$, equivalently that no selection of numbers that contains x_4 will add up to k.

   So we have gone over S once and eliminated all the numbers that could never be in a solution. But there may be more than one solution—that is, there may be proper subsets of T that add up to k. For example, for $S = \{1, 3, 8, 9, 10\}$ and $k = 18$, there are two solutions, $\{1, 8, 9\}$ and $\{8, 10\}$. In this example, the process of elimination would have left us with $T = \{1, 8, 9, 10\}$.

$$f(S - \{1\}, k - 1) = 1$$
$$f(S - \{3\}, k - 3) = f(\{1, 8, 9, 10\}, 15) = 0.$$

This says we wouldn't be able to get up to 15 to use 3 to get us to $k = 18$.

$$f(S - \{8\}, k - 8) = 1$$
$$f(S - \{9\}, k - 9) = 1$$
$$f(S - \{10\}, k - 10) = 1.$$

Out of this $O(n)$ verification we are left with $T = \{1, 8, 9, 10\}$ but $\sum_T t_i = 28 \neq k$. Now let $m = k - 1 = 17$, and let $U = T - \{1\} = \{8, 9, 10\}$. We do the same:

$$f(U - \{8\}, m - 8) = f(\{9, 10\}, 9) = 1$$
$$f(U - \{9\}, m - 9) = f(\{8, 10\}, 8) = 1$$
$$f(U - \{10\}, m - 10) = f(\{8, 9\}, 7) = 0.$$

From this, in a second $O(n)$ run, we have determined that the subset $\{1, 8, 9\}$ adds up to $k = 18$.

   Why does step two work? We know for sure a solution will include any element of T, let's say, x_i. Then the task is finding $x_j \in T$ such that $k = x_i + \sum_j x_j \rightarrow k - x_i = \sum_j x_j$. We can use our oracle to find a subset of $T - \{x_i\}$ that adds up to $k - x_i$, thus completing a solution.

3-28   We can use a *segment tree* to solve this. They support more general range queries than prefix sums, but we can treat those as range queries with $l = 0$.

   Each node in a segment tree represents a range within the array and carries the partial sum for that range, with the root node representing the total range of the array. The subtrees represent each half of the range, such that their union is equal to the range represented by the parent. The root node of each subtree holds its corresponding partial sum. Each leaf node represents a one element range; naturally, the partial sum for this is the element itself.

   To update an element, traverse to the leaf node, update it, and trace back the path modifying each node. Alternatively modify each node visited on the way to the leaf node.

For partial sum queries, traverse to the leftmost leaf and find the right end of the range, then treat it as a special case of a general range query with $l = 0$. The general range query $\sum_{[l,r]} x$ is answered by traversing the tree and using the precomputed sums of the segments. For some node corresponding to the range $[tl, tr]$ there are three cases:

1. The segment $[l, r]$ is the same as $[tl, tr]$. Just return the precomputed sum of the node.

2. The query segment is entirely contained in the domain of the left or right child. The left child covers $[tl, tm]$ and the right child covers $[tm + 1, tr]$, with $tm = (tl + tr)/2$. In this case we recurse to the correct subtree and execute the algorithm on it.

3. The final case is when the query segment intersects the domains of both children. Two recursive calls are made, each for the sum query of the *intersection* of the range query and the child's domain, and the results are combined.

The segment tree can be represented as a heap-style, packed implicit tree. We need at most $4n$ nodes for an array of size n. The root at index 1, the left child of k at $2k$, the right child at $2k + 1$. The parent is at $\lfloor k/2 \rfloor$.

Chapter 4

Sorting

Applications of Sorting: Numbers

4-1 The Grinch wants the most unbalanced game possible. This is the same as asking to maximize the difference in total skill between the two teams.

   In $O(n \log n)$, sort the player pool by skill and make each half of the result a team. Any swap between teams would bring a higher skilled player into the lower half, and a lower skilled player into the higher half, making the difference in total skill lower.

4-2

(a) Unsorted array. Find x, y that maximize $|x - y|$ in $O(n)$ time. In one sweep through the array, find the minimum and maximum values. These maximize the difference.

(b) Sorted array. Find x, y that maximize $|x - y|$ in $O(1)$ time. These are the first and last elements of the array.

(c) Unsorted array. Find x, y that minimize $|x - y|$, for $x \neq y$, in $O(n \log n)$ time. Sort the array in $O(n \log n)$ then do a linear pairwise search of consecutive values for the pair with the smallest difference in $O(n)$ time.

(d) Sorted array. Find x, y that minimize $|x - y|$, for $x \neq y$, in $O(n)$ time. Do a pairwise comparison of every element with the next in $O(n)$ and find the consecutive pair with the smallest difference.

4-3 What we want to do here is first sort the array in $O(n \log n)$ time. Then we want to pair up the biggest offender with the least problematic number, so the largest with the smallest, the second largest with the second smallest, and so on.

$$x_1 < x_2 < \ldots < x_{2n-1} < x_{2n} \quad \to \quad (x_1, x_{2n}), (x_2, x_{2n-1}), \ldots$$

   Why does this work? Suppose in my pairing I find that (x, y) are my maximum pair. By way of contradiction, I claim that there is a different partition of the $2n$ numbers whose maximum pair is less than (x, y).

   Let's say that $x < y$. The claim that a different partition does better implies that, whatever its maximum pair is, element y in this hypothetical partition has to be paired up with an element t such that $t < x$, otherwise the maximum pair could not be less than $x + y$. But the same logic applies to every element larger than y. They need to be paired up with elements smaller than x certainly, or for any $z > y$ we'd have a pair (z, x) where $z + x > x + y$ is larger than the claimed maximum.

   Now suppose there are i elements larger than y. In my original pairing, that means there are i elements smaller than x. But the claim forced me to pair y with one of those i elements smaller than x, so there are now $i - 1$ numbers available to pair with the i numbers larger than y. We see that we won't be able to find pairs for every element that don't surpass our claimed maximum pair. The contradiction shows that our original pairing was indeed optimal, and the algorithm correct.

4-4 Keep three linked lists and pointers to the end of each. Iterate over the input pairs, appending each to the list corresponding to its color. By the pointers to the ends, this can be done in $O(1)$ time. Finally concatenate all three lists, which is again constant time.

4-5 $O(n)$ is the best we can hope for, since we need to at least visit every element in the input. By using a hash table with (amortized) $O(1)$ query and insertion we can check for each value's membership, set it to 1 on first sight, then increment it on subsequent visits to construct a frequency table of all values. There are n of them, and constant work for each (query, increment, insert), so building the table is $O(n)$ time. Iterating over all the key-value pairs and selecting the maximum frequency is another $O(n)$ effort after which we have found the mode.

4-6 We can afford to sort one of the sets into a sorted array A in $O(n \log n)$ time. It's now possible to query A for membership of a number in $O(\log n)$ time by binary search. Then for each number b in the other set, let $a = x - b$, and query A for a. If $a \in A$ then we have a number $a = x - b$, which is to say $x = a + b$, and we can answer positively. If we test every number in the second set unsuccessfully, there is no pair that adds to x. In this (worst) case, we do n binary searches for a total $O(n \log n)$ time.

4-7 We can sort the array in descending order, then do a linear sweep, counting how many elements are strictly greater than their (0-based) index. Sorting is $O(n \log n)$ and the linear sweep is, well, linear, so overall this algorithm is $O(n \log n)$.

```
int h(std::vector<int>& papers) {
  auto xs = papers;
  std::sort(xs.begin(), xs.end(), std::greater<int>());
  for (int i=0; i < xs.size(); i++) {
    if (xs[i] <= i)
      return i;
  }
  return xs.size();
}
```

Starting with the values sorted in descending order, consider the array of differences between each value and its index. Then an alternative view of the algorithm above is that the h-index of an array A is the count of elements in the difference array that are greater than 0.

$$\langle 6, 5, 4, 3, 3 \rangle \to \langle 6, 4, 2, 0, -1 \rangle$$
$$\langle 5, 4, 4, 2, 1, 1, 0 \rangle \to \langle 5, 3, 2, -1, -3, -4, -6 \rangle$$
$$\langle 0, 0, 0, 0 \rangle \to \langle 0, -1, -2, -3 \rangle$$

4-8

(a) Assuming customers don't double pay, there will be at most as many checks as there are bills. Let's sort both bills and checks by the customer id. Then treat each list as a queue, and consider each bill. If the check at the front of its queue is for this bill, this customer has paid; pop the front of both queues. If the check doesn't match the bill, this bill hasn't been paid—pop the bill and put it aside in the list of unpaid bills.

Sorting the bills and checks is $O(n \log n)$, and the linear sweep is $O(n)$ for a total effort of $O(n \log n)$ time.

(b) Sort the list of books by publisher. This is $O(n \log n)$ for n books in the library. Then for each publisher use binary search to find the boundaries of the corresponding block of published books in the sorted list. (See §5.1.1 *Counting Occurrences.*)

Each binary search is $O(\lg n)$ time and there are 30 publishers, which is a constant factor so the overall effort is $O(n \log n)$ with the sorting of all the books dominating.

(c) Don't sort anything. Make an empty set, then for each card push the name into the set. The size of the set after going over all the cards is the answer.

If the set is implemented as a binary tree, each insert is $O(\log n)$ for n total cards, meaning n insertions, so $O(n \log n)$ total effort. If the set is implemented as a hash table instead, each insertion is $O(1)$ so the overall complexity is $O(n)$.

4-9   To start simple, let's consider the case for $k = 1$. We are then looking to answer if $k \in S$ in $O(\log n)$ time. This is a tall order unless we make the further assumption that S comes in the form of an already sorted array, in which case we can do a binary search. Otherwise a linear search will take $O(n)$ time and sorting, $O(n \log n)$.

For $k = 2$ we can already afford to sort the array in $O(n \log n)$. Let x be some element in S and consider $y = T - x$, then we want to see if $y \in S$. This can be done in $O(\log n)$ time since the array is already sorted and we know what we are looking for.

Thinking back to exercise 3-27, we had a similar scenario in which we were given an oracle that could answer a simplified version of this question: given T it would answer if there was a subset of *some* length that added up to it. Here we have no oracle, and we have a fixed number k of elements with which to build a solution.

As in exercise 3-27, let's start by assuming we have a solution $\{x_1, \ldots, x_k\}$, that is $T = \sum^k x_i$. This means that before we were done we found $x_k = T - \sum^{k-1}$ to complete a partial solution $\{x_1, \ldots, x_{k-1}\}$. That is, we searched $S - \{x_1, \ldots, x_{k-1}\}$ for x_k; but note that we only know to search for a specific value on the last x_i, when we know exactly what value we are missing in our solution to get to T. This is the scenario above where $k = 2$.

In selecting the $x_1, \ldots, x_{k-1}$ we don't search but try these values out. This is where the n^{k-1} factor comes from—it's the combinations of $k - 1$ elements for the first $k - 1$ positions, for each of which we can perform an intentional binary search in $O(\log n)$ for the final piece of the solution. More specifically, we do a $O(\log n)$ search for the kth value for, at most, n^{k-1} partial solutions of $k - 1$ elements.

Suppose we have a method to generate the $\binom{n}{k-1}$ combinations out of the set S. Then the algorithm would look something like this:

```
for C in combinations(S, k-1):
  if (T - sum(C)) in S-C:
    return true
return false
```

4-10

(a) If S is unsorted, we can sort it in $O(n \log n)$ time, then for each of the n elements do a binary search for the difference with x. At most we do a binary search for all n elements, each being $O(\log n)$, for a total $O(n \log n)$ effort.

(b) If the array is sorted, consider two pointers l and r, pointing to the first and last elements of the array respectively. While they haven't crossed each other, compare the sum of the pointed elements to x—if the sum is greater than x, point r to the next smaller element; if the sum is less than x, point l to the next largest element. If the two pointers cross each other, then no two elements in the array add up to x.

Let us call l, r the left and right pointers respectively. Suppose a given array and real number x have a solution at indices i, j—that is, $x = x_i + x_j$, $i < j$.

The pointers l and r only ever go in one direction, so we would fail to find a solution if $l > i$ or $r < j$ (taking some notational license).

Suppose there is a solution, yet $l = i + 1$. Then l was incremented because $x_i + x_j < x$, which contradicts the assumption that our solution was the pair (x_i, x_j). Note that $r > j$, since the array is ordered, would mean that $x_i + x_{j+1} > x$ since $x_j < x_{j+1}$. But this contradicts the fact that l was incremented to $i + 1$.

By a similar argument, we can see that r can't be less than j.

To drive the point home, consider the moment where $l = i - 1$ and $r = j + 1$. Either $x < x_{i-1} + x_{j+1}$ or $x > x_{i-1} + x_{j+1}$.

If $x < x_{i-1} + x_{j+1}$, we will increment l and have a pair (x_i, x_{j+1}) which must be greater than x, for $x = x_i + x_j$ and $x_j < x_{j+1}$. At this point the algorithm will bring r in to find pair (x_i, x_j).

If $x > x_{i-1} + x_{j+1}$, conversely, we will see r be brought in first to find pair (x_{i-1}, x_j). By the same argument, $x_{i-1} + x_j < x$, so l will be advanced to point to x_i, and again find the solution (x_i, x_j). ∎

4-11   This is cheating, but assume you have a hash table with $O(1)$ insertion and update. Traverse the list, incrementing the count of keys already in the table, or inserting them with an initial value of 1 otherwise. Finally, traverse the hash table filtering the items that fulfill the predicate.

Applications of Sorting: Intervals and Sets

4-12

(a) Sort the two arrays in $O(n \log n)$ then do a linear sweep merging from each array, while skipping duplicates. The linear sweep is, well, linear, while skipping duplicates is constant time because we will insert each unique number the first time we see it, then see it again immediately in the other set.

(b) Just do the merging part of (a).

4-13 Consider the number of people at the party at any given time as a counter. Then each a_i increments the counter while each b_i decrements it. A trivial algorithm is to gather all a_i and b_i in an array. Suppose we have a discriminant in each data point such that we can tell that a timestamp is an entrance or a departure. Sort the array in $O(n \log n)$ then do a linear sweep executing the increases and decreases on a counter while keeping track of the maximum throughout.

4-14 Sort the array of intervals by their first component in $O(n \log n)$. Then do a linear sweep, looking at the first element and its immediate successor.

If they overlap, that is if $x_{i+1} \leq y_i$, merge them and make this new larger interval the current interval. Repeat the process skipping both of the intervals just processed.

If the intervals did not overlap, we finished merging a group of intersecting intervals which is disjoint from the next one. Push the interval under consideration into the list and continue ahead considering the next interval.

Finally, once we reach the end of the list, push the last interval under consideration into the list.

4-15 Consider a scenario in which two intervals intersect but don't intersect at their extremes. Then we see that in a sequential ordering of extremes, two intervals starts before one ends. The serialization of the endpoints of the example in the book would be $10\uparrow\ 15\uparrow\ 20\uparrow\ 40\downarrow\ 50\uparrow\ 60\downarrow\ 70\downarrow\ 90\downarrow$, where a $\uparrow$ means an interval opens at that point and $\downarrow$ means that an interval closes at that point.

Now consider a scenario where two endpoints meet, for example for $S = \{(10, 20), (20, 30)\}$. The serialization of endpoints of this set would be $10\uparrow\ 20\uparrow\downarrow\ 30\downarrow$, meaning that at point 20 an endpoint ends and another one begins.

There is one final case to consider, that of intervals that start and end at the same point. For example, a set $S = \{(x_0, x_1), (y_0, y_1)\}$ with $y_0 = y_1$. The serialization of endpoints of this set would be $x_0\uparrow\ y_0\updownarrow\ x_1\downarrow$ assuming that $x_1 > y_0$.

It's not hard to see that with an ordering of endpoints like this it's possible to sweep from left to right, maintaining a counter of simultaneously running intervals that changes at the points where intervals start or end. The counter increases when an interval is opened, and decreases when one is closed. This requires a little extra thought for scenarios where intervals intersect at their endpoints, as well as when 1-point intervals are involved, but the general idea is simple.

We set things up so as to process "groups of endpoints", each group being a collection of all endpoints that meet at a specific coordinate.

At the coordinate of each group, some number of intervals meet at an endpoint. These plus any other currently running intervals, represented by the running counter, make up the total number of intervals that intersect at that point. For each group, we know which endpoints are opening or closing intervals, and which are 1-point (let's call them *singular*) intervals; these are the arrows above. Therefore, when processing a group, given a counter i, counts of opening, closing and singular endpoints o, c, s respectively, and a current maximum m of intersecting intervals already seen, we have that

$$i' = i - c + o$$
$$x = c + o + s$$
$$m' = \max(m, x)$$

Where i' is the updated counter after processing this group, x is the number of intersecting intervals at this group's coordinate, and m' is the updated maximum after processing this group.

Two things to note. When updating the counter, the number s is insignificant because these are intervals that open and close at a singular point. Thus the number of running intervals before and after this point will not be affected by them. On the other hand, when calculating the number of intersecting intervals x at the current group's coordinate, the singular intervals are counted and the closing intervals are ignored, because these were already accounted for when they were opened.

Thus after processing these groups in order, we will have a final maximum carrying both a point (at some interval's endpoint) and a count of intervals that intersect at that point.

The code for this is too long to include here. See `ex04_15.hs` in the Haskell directory for a full implementation.

4-16   Sort S by the starting coordinate of each segment in $O(n \log n)$.

Start by finding the segment that covers 0 that extends furthest to the right in $O(n)$. If no segment covers 0 then there is no solution.

Call the selected segment the "current" segment. Then, while m hasn't been covered, select among the intervals that intersect the current interval the one that extends furthest to the right ($O(n)$), and make it the current segment. If no segment intersects the current segment whose right coordinate extends further to the right than that of the current segment, then there is no solution (the union of S either has a gap or can't cover all of $(0, m)$).

This process guarantees that a sequence of intersecting segments will be selected, meaning that there will be no gaps. The selection of each segment in that sequence, the one that reaches furthest to the right, is optimal: any alternative will reach at most the same number of segments to the right, or it will possibly not reach the next optimal choice of segment.

Deniz's reasoning. Let us take $m = 1$ without loss of generality.

Let $I_i = [a_i, b_i]$, $i = 1, \ldots, m$ be the collection of segments which the algorithm yields, which by construction has the property that $a_{j+1} \leq b_j < b_{j+1}$ for $j = 1, \ldots, m - 1$.

Let $J_i = [x_i, y_i]$, $i = 1, \ldots, n$ be another collection which covers $[0, 1]$ with $n < m$. We can assume the collection $\{J_i\}$ has the same property (just apply the algorithm to this collection).

Since $n < m$, we have $b_n < 1 \leq y_n$. Also, by construction, $b_1 \geq y_1$. Let S be the set of indices i such that $b_i < y_i$. Then $m \in S$ and $1 \notin S$. Let $k = \min S > 1$. Then, since $k - 1 \notin S$, we have $y_{k-1} \leq b_{k-1}$, as well as $x_k \leq y_{k-1}$ by construction and $b_k < y_k$ because $k \in S$. Thus $x_k \leq y_{k-1} \leq b_{k-1} < b_k < y_k$.

So, $J_k = [x_k, y_k]$ is a segment which intersects I_{k-1}, but again by construction, b_k is the largest right endpoint of all such intervals, implying $b_k \geq y_k$. But that means $k \notin S$, a contradiction. It follows that $m \leq n$.   □

Heaps

4-17   Building the complete heap with the regular `bubble_up()` method will require $O(n \log n)$ time, so that will not work. The key is the faster heap construction method that uses `bubble_down()` instead. Even though this operation is ostensibly $O(\log n)$, a closer look shows that $O(n \log n)$ is a generous upper bound for the cost of building a heap.

Half of the elements in the array are leaves, and can be considered well formed heaps of height 0. A fourth of the elements are heaps of height 1. According to Skiena there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h; the bottom line is that this quickly converges to linear behavior.

Thus we have a way of building a heap in $O(n)$ time. All that is left is to do k queries for the minimum, $O(1)$ each, and k delete operations, $O(\log n)$ each, for a total $O(n + k \log n)$ time.

```cpp
int left(int i) { return 2 * i + 1; }
int right(int i) { return left(i) + 1; }

int min3(vec &xs, int i) {
  int ix = i;
  int l = left(i), r = right(i);
  if (l < xs.size() and xs[l] < xs[ix]) ix = l;
  if (r < xs.size() and xs[r] < xs[ix]) ix = r;
  return ix;
}

void sift(vec &xs, int i) {
  int min = min3(xs, i);
  if (min != i) {
    std::swap(xs[i], xs[min]);
    sift(xs, min);
  }
}
```

```
void heapify(vec &xs) {
  for (int i = xs.size() / 2; i >= 0; i--)
    sift(xs, i);
}
```

4-18   To clarify, n elements across all k lists.

Put the k heads of the lists in a heap in $O(k)$ time. Suppose we build this priority queue so that, upon eliminating a minimum that belonged to list i, it pulls the new head from list i (if it's not empty yet) into the heap. Effectively, keep a k-element heap of the heads of the lists.

Repeatedly take the minimum from the heap until there are no more elements in any of the lists (in other words, when we have processed n elements). We will have produced an ordered sequence that merges all the elements in the lists.

There are n total elements, selected from the lists by the `pop()` operation of the heap which is $O(\log k)$, for a total $O(n \log k)$ time.

The powerup comes from the assistance the heap provides in quickly finding the next element in the output sequence. Without it, we would have to visit each of the k list heads to find the minimum at each step. We'd do this $O(k)$ work for each of the n elements; this is the obvious $O(kn)$ algorithm.

4-19

(a) A max heap is cheaper to construct if all we want is to query and eliminate the maximum element from a collection.

(b) Unless we wish to repeatedly delete the maximum or minimum, a heap won't help here. To delete arbitrary elements a fully sorted array does better. (That is, to locate the elements—deleting them from a sequentially allocated structure is another story).

(c) Constructing a heap is quicker than sorting an array.

(d) A max heap will not help us find the smallest element, so we have to resort to a sorted array for this.

4-20   From Knuth, vol. 3, §5.2.3 *Sorting by selection*. DEK introduces *quadratic selection*. The idea is simple: assume without loss of generality that n is a square, and partition the n elements into $\sqrt{n}$ groups of $\sqrt{n}$ elements each. Then make a new group with the minimum elements from each group. The minimum element across all n can be found in this group.

Note that the second smallest element will also be selected into the leaders group, except when it is in the same partition as the smallest element. Thus, with the minimum removed from the group of leaders, the second smallest can be found by looking at the remaining $\sqrt{n} - 1$ leaders as well as the remaining elements in the group whence the minimum element came.

Finding the minimum in a group takes $\sqrt{n} - 1$ comparisons. There are $\sqrt{n}$ groups, so there is a total initialization cost of $\sqrt{n}(\sqrt{n} - 1) = n - \sqrt{n}$ comparisons. The minimum is then determined from this group of leaders in $\sqrt{n} - 1$ comparisons.

We then look for the second smallest across $\sqrt{n} - 1$ remaining leaders and $\sqrt{n} - 1$ remaining elements in the original partition of the minimum. Thus, there are $2\sqrt{n} - 2$ elements to search, which takes $2\sqrt{n} - 3$ comparisons.

So $n - \sqrt{n} + \sqrt{n} - 1 + 2\sqrt{n} - 3 = n + 2\sqrt{n} - 4$ total comparisons to get the second smallest element. How does this fare against the $2n - 3$ comparisons of the obvious algorithm? For n a perfect square, eventually $2 < \sqrt{n}$. Therefore

$$2 < \sqrt{n} \;\to\; 2\sqrt{n} < n \;\to\; n + 2\sqrt{n} < 2n.$$

This shows that quadratic selection does fewer comparisons than the obvious algorithm.

Quicksort

4-21   Given an array of n elements we know the median would belong in position roughly $n/2$ with provisions for odd and even n as well as 0 or 1-based arrays we will conveniently disregard.

Select a pivot and partition the elements. If the pivot belongs in the median position, the pivot is the median. Otherwise recurse into the partition that contains the median position.

Partitioning in the first level looks at all n elements, but every time we recurse we ditch half the elements so the second partition looks at $n/2$ elements (in the expected case). By halving the number of elements under consideration we zero in on the median in $O(\lg n)$ time; each recursive call does $n/2^h$ partitioning work, for height $h \in [0 \mathinner{..} \lg n]$.

What really is $\sum_{h=0}^{\lg n} n/2^h$? Remember the situation with the dynamic array (§ 3.1.1, pp. 71). Here, the first level looks at all the elements, then each subsequent call looks at half the previous ones.
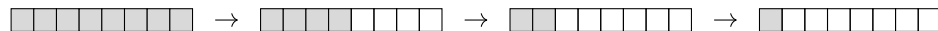
Arranging the last three "summands" together, they tend to add up to n but not quite. So we do a total of $2n$ work in partitioning (again, in the expected case) for a total $O(n)$ (expected) time.

Lomuto partition scheme

Given an arbitrary pivot, this method does a linear selection over the entire range, pushing the k elements smaller than the pivot to the first k positions of the array. As a consequence of these swaps, elements larger than the pivot are carried to positions $[k+1 \mathinner{..} r]$. The final step is to place the pivot in position $k+1$.

```
int lomuto(vec &xs, int l, int r) {
  for (int i = l; i < r; i++) {
    if (xs[i] < xs[r]) {
      std::swap(xs[i], xs[l]);
      l++;
    }
  }
  std::swap(xs[r], xs[l]);
  return l;
}
```

Sedgewick's partitioning scheme

As explained by Knuth in § 5.2.2 *Sorting by exchanging*.

Given an array A of n elements, and integers l, r the boundaries of the interval to partition (e.g. `sedgewick(A,0,n-1)` would partition the entire array). Take an arbitrary element as pivot; DEK takes the first element in the interval, l. Increase l until it points to an element that does not belong in the left partition; likewise, decrease r until it points to an element that does not belong in the right partition. If $l < r$, exchange A_l with A_r, then continue processing the elements until $l \geq r$. If l and r did cross, swap the pivot into its final position at A_r and return.

```
int sedgewick(vec &xs, int l, int r) {
  int p = l;
  l--; r++;
  while (1) {
    do { l++; } while (xs[l] < xs[p]);
    do { r--; } while (xs[p] < xs[r]);
    if (l >= r) {
      std::swap(xs[p], xs[r]);
      return r;
    }
    std::swap(xs[l], xs[r]);
  }
}
```

4-23 We have an array A of n elements of three kinds: `R`, `W` and `B`. We want them sorted like so: `R < W < B`, using only the operations `examine(A,i)` and `swap(A,i,j)`, in linear time.

The first thing that comes to mind is some algorithm inspired in the partitioning step of quicksort. Partitioning algorithms are linear, so that's a good start. We don't know how many of each kind of element there are, so we don't know exactly where to place the middle chunk. We could, however, temporarily consider `W = B`, and distribute the `R` to

the left, and all the `W`/`B` to the right. Once done, we'd know how many `R` we sorted, and therefore how much space the rest of the elements will take. A second linear pass on this subarray would sort the `W` first and the `B` second.

How would this look? Keep a pointer i initially at position 0, and a pointer j at position $n-1$. While they haven't crossed, if i points at a `R`, advance it until it doesn't; inspect j, if it points to a `R`, this element belongs on the first part of the array so `swap(A,i,j)`, increment i and decrement j. At the end of this process all `R` will be on the left side of the array. The same method can then be used on the remaining subarray to put all `W` in place. The `B` will end up accumulated on the right as expected.

4-24   The same procedure from 4-23 solves this exercise in $O(n)$ time.

4-25   If z_i and z_j are compared then one of them has been selected as a pivot and will end up in its final position at the end of the partition step. Furthermore, the pivot will not be a part of any subsequent recursive calls, so it won't be compared again with this or any other element. Therefore the answer is 1.

4-26   The minimum recursion depth happens when the median is selected as a pivot at every partition step—the work is split most evenly. In this scenario, we repeatedly halve the problem size until we get down to 1. For n elements, this recursion depth is $h = \lceil \lg n \rceil$.

The maximum recursion depth happens, as expected, when the worst possible pivot is selected at every turn. There are two of these: the minimum and the maximum elements in the input. In this scenario we split the work most unevenly—since the pivot is frozen in place and it is one of the extremes, all the remaining work goes to a single recursive call! Since every recursion level advances by freezing only one element, we end up with a recursion depth $h = n$. ∎

4-27   Suppose we have a permutation p of the integers $[1 .. n]$ and an operation `reverse(i,j)` that reverts elements $p_i, \ldots, p_j$. We want to sort the permutation in increasing order using only the `reverse()` operation.

In particular the first thing we are asked to do is show that $O(n)$ reversals are enough to sort p. But first we can explore this operation a bit and see what it can do for us. One readily apparent fact is that it is possible to implement `swap(i,j)` in terms of `reverse(i,j)`. To swap elements i and j, call `reverse(i,j)` then call `reverse(i+1,j-1)`.

An example. Let $p = [a, b, c, d, e, f, g]$ and swap elements b and f.

```
                [a,b,c,d,e,f,g]
reverse(1,5) -> [a,f,e,d,c,b,g]
reverse(2,4) -> [a,f,c,d,e,b,g]
```

Note that `swap()` makes a constant number of calls to `reverse()`. In other words, `swap()` costs whatever `reverse()` costs.

Skiena didn't mention operations `inspect(i)` or `compare(i,j)`, which he did in 4-23, but we'd be hard pressed to sort any permutation without looking at the elements or being able to assert if a range is in order. Therefore we will assume these operations are legal.

Now, on to the task at hand.

(a) With `swap()` at our disposal we may implement a selection sort: find the smallest item and swap it with the element at index 0. Find the second smallest and swap it with the element at index 1, and so on. It should be clear that $O(n)$ swaps are done, for each element—once found—is placed in its final position and not moved again. Each swap takes a constant number of `reverse()` operations (at most two), for the desired $O(n)$ reversals. ∎

Mergesort

4-28   The recursion tree in the canonical "split at the half" version of merge sort has height $\lg n$, namely how many times n elements can be split until we reach 1 element per part. If we were to split three ways we'd end up with a tree of height $\log_3 n$—but the merging is still linear, so we are left again with a $\Theta(n \log n)$ algorithm. The insight by Skiena: a change of base does not affect the class of a logarithmic function.

One interesting consideration is how the merge operation is affected by this change. Whereas before we merged from two subarrays, picking each element at the cost of one comparison, merging three ways requires two comparisons. The process is still linear but it takes twice as much work now.

4-29   Each time a new array is merged we do a linear pass over all the partial work already done. This partial work does not remain constant however—it is n elements larger every time we merge a new array, so that on the last merge we are making a pass over an array of $(k-1)n$ elements.

So the first pass merges $2n$ elements, the next $3n$ and so on:

$$\sum_{i=1}^{k} in = n \sum_{i=1}^{k} i = n\frac{k(k+1)}{2} = O(nk^2).$$

4-30   We have k arrays, n elements each, sorted. The proposed algorithm divides them into $k/2$ pairs of arrays. Then it merges each pair to get $k/2$ arrays of length $2n$. These arrays are in turn paired up to form $k/4$ pairs, which are then merged, and so on and so forth, until there are two arrays of length $(k-1)n$ merged into a single array of length kn.

First note that we can pair up arrays $\lg k$ times until we end up with a single one. What is the cost of each merge? Well, the first looks at $2n$ elements $k/2$ times. The second pairing/merging looks at $4n$ elements $k/4$ times.

$$2n\frac{k}{2} + 4n\frac{k}{4} + \cdots + (k-1)n\frac{k}{2^{k-1}} = \sum_{i=1}^{\lg k} 2^i n\frac{k}{2^i} = \sum_{i=1}^{\lg k} kn = O(kn \lg k).$$

Other sorting algorithms

4-31   To make sure merge sort is stable, the items on the partition with the lowest index must be selected first during the merge process. These items appear first in the original array, and must therefore be placed first in the sorted array to maintain the relative order of equal keys. (Realistically speaking, you would have to go out of your way to make an unstable merge sort.)

4-32   Let's consider the example permutation of $[1..6]$ from the book, and the relation between each consecutive element: $[3 > 1 < 4 > 2 < 6 > 5]$. These would be correct insofar as they are alternating, but it is not because it starts with an element that is larger than the next.

To correct this situation in $O(n)$ time it's worth observing what constitutes an incorrect placement. For this we have to consider triples of consecutive numbers: a relation is wrong (meaning the related elements are in an incorrect position) if two consecutive relations are the same. If $a < b$ is correct, then $a < b < c$ means b and c are placed incorrectly. Likewise for $a > b > c$.

Now, if $a < b < c$ then naturally $a < c$. Therefore if $a < b$ is correct, then so is $a < c$, and we may swap b and c without affecting the correct relation between a and its consecutive element. This is the key to the algorithm.

The wiggle sort algorithm will look at each pair of elements and consider their ordering, taking into account what the ordering was between the last pair visited. To begin let $l \leftarrow (>)$. Look at elements x_0 and x_1 at the front of the array; if $x_0 > x_1$ then we have two consecutive pairs under the same relation (in this case, the previous pair is x_0 and a hypothetical element. We want the relation between the first two elements to be $<$, therefore we start with l set to $(>)$.) Otherwise if $x_0 < x_1$ then these elements are in the right order and we may move on. Set $l \leftarrow (<)$ and look at elements x_1 and x_2. Once again, if the relation between x_1 and x_2 is the same as l, we are in front of a pair that repeats the relation of the last visited pair, and we must swap them. Repeat this process until visiting the last pair.

Here is an example run with the example from the book. The pair under observation is signaled by the presence of the relation. The value of l while considering each pair is on the right; whenever it is the same as the relation between the elements under consideration, the elements are swapped.

$$\begin{aligned}
[3 > 1, 4, 2, 6, 5] &\rightarrow [1, 3, 4, 2, 6, 5] \qquad (>) \\
[1, 3 < 4, 2, 6, 5] &\rightarrow [1, 4, 3, 2, 6, 5] \qquad (<) \\
[1, 4, 3 > 2, 6, 5] &\rightarrow [1, 4, 2, 3, 6, 5] \qquad (>) \\
[1, 4, 2, 3 < 6, 5] &\rightarrow [1, 4, 2, 6, 3, 5] \qquad (<) \\
[1, 4, 2, 6, 3 < 5] &\rightarrow [1, 4, 2, 6, 3, 5] \qquad (>)
\end{aligned}$$

It's clear this algorithm takes $O(n)$ time as each index is visited at most twice and swaps are constant time. It takes $O(1)$ space to keep track of the last relation (1 bit?). An argument for its correctness goes somewhat as follows: as we saw before, if $a < b < c$ and $a < b$ is correct, then $a < c$ is also correct, and what's more important $a < c > b$ is correct.

Thus we can correct each pair without breaking previously correct relations. In particular, if the last pair is incorrect, then it can be safely swapped without disturbing the rest of the array.

Wiggle sort is very easy to implement recursively (in which case we must look the other way with respect to the $O(n)$ space usage of the call stack or a potential accumulated list):

```
wiggle :: [Int] -> [Int]
wiggle xs = go GT xs
  where
    go :: Ordering -> [Int] -> [Int]
    go _  (a:[])               = [a]
    go LT (a:b:xs) | a < b     = b : (go GT (a:xs))
                   | otherwise = a : (go GT (b:xs))
    go GT (a:b:xs) | a > b     = b : (go LT (a:xs))
                   | otherwise = a : (go LT (b:xs))
```

4-33   We have n positive integers in the range $[1 .. k]$, and want to show they can be sorted in $O(n \log k)$ time.

Traverse the array and build a binary tree of lists of each of the k keys. The tree will have at most k elements, meaning each insertion is an $O(\log k)$ operation in the worst case; there are n of those insertions, each of which can be done in $O(1)$ time, for a total effort of $O(n \log k)$ to build the tree. An in-order traversal of the tree that outputs every element in each list will visit all n elements across the k nodes. In the worst case, each node will have a single element and cause n `successor()` calls, each of which is $O(\log k)$. Therefore the traversal is also $O(n \log k)$.

4-34   We have a sequence of n integers such that there are $O(\log n)$ *distinct* integers among them, and seek an $O(n \log \log n)$ algorithm to sort them.

The principle here may be somewhat similar to that of 4-33—instead of k distinct elements and a total effort of $O(n \log k)$ we have $O(\log n)$ distinct elements, so it stands to reason that the same process may be used to sort the sequence in $O(n \log \log n)$ time.

The binary search tree would have $O(\log n)$ nodes—therefore it would have $O(n \log \log n)$ levels, making the cost of individual `insert()`, `minimum()` and `successor()` operations $O(\log \log n)$. For n total elements in the sequence, building the tree and traversing it in order would be $O(n \log \log n)$ total time.

4-35   Out of n elements the first $n - \sqrt{n}$ are already sorted, meaning the last $\sqrt{n}$ elements are potentially not sorted. We are therefore excused in being lavish and extravagant and sorting those last $\sqrt{n}$ elements using a quadratic algorithm. With $\sqrt{n}$ elements that will be $O(\sqrt{n}^2) = O(n)$ total time. "Substantially" better is in the eye of the beholder, but that is certainly better than $O(n \log n)$. If we were short on computing time, we may still use an optimal sorting algorithm for a total effort of $O(\sqrt{n} \log \sqrt{n})$. Note that $\sqrt{n} \log \sqrt{n} \le \sqrt{n}^2 = n$, so we may call this an even more substantially better approach.

We now have two separate partitions: the elements that were initially sorted, and the $\sqrt{n}$ elements just sorted. There is no guarantee that these are all larger than the first group, so it's necessary to do the `merge()` step of merge sort between the two partitions. This takes $O(n)$ time for an overall $O(n)$ time algorithm.

4-36   We could use bucket sort for this, but would need an extra twist due to the n^2 elements in the key space. If we were to just count how many of each key are in A directly in an array of length n^2 we would then have to pay the $O(n^2)$ price to traverse it to output all n elements. Since we are willing to trade space for time, we could still use this much space (and then some) but do better than that. (Unfortunately this approach will not work, but let's entertain the idea for fun.)

Let B and C be integer arrays of length n^2 and $\log \log n$ respectively. Start by traversing A. For each $a \in A$, look at B_a—if it is 0, or `null`, this is the first time we see a; push a pair $(a, 1)$ to the end of C, and record its index in B_a. We now have a way to index into C to increase this count every time we see another a.

Once we finish traversing A and counting each element we can discard B and concentrate on C, which can be sorted on the first element of each pair. Since C has length $\log \log n$ it can be sorted in $O((\log \log n) \log(\log \log n))$ time. A traversal over C, now sorted, to output as many copies of each key as were counted in the first stage, completes the task in $O(n)$ time, as there are n elements to output. ∎

What is faulty about this algorithm? We first stated that it would be no good to count directly on B for we would then have to traverse it in $O(n^2)$ time. The issue is in verifying if we have already seen an element a when looking up the index to its counter: in order to be able to distinguish a new element from one previously seen we must zero out B. There is no way around it; this takes $O(n^2)$ time, so we are back where we started. Therefore it seems our best bet is once again to build a binary search tree that holds a count along with each key. The tree will have at most $\log \log n$ elements and $\log \log \log n$ levels, so building it and traversing it will be $O(n \log \log \log n)$ time—not too bad, but still more than the $O(n)$ time it will take to output all the original n elements.

4-37

(a) As in exercise 4-23, the limited key space is very strong. In this case, partitioning sorts all elements in linear time using $n - 1$ comparisons. It is optimal insofar as every element needs to be visited and compared to the pivot to know if it has to be moved.

Lower bounds

4-39 $n \log \sqrt{n} = \frac{n}{2} \log n = O(n \log n)$.

4-40 If such a priority queue existed we would be able to sort n elements in $O(n)$ time by populating the priority queue with n insertions and getting them in order with n calls to `extract()`, in clear violation of the lower bound for sorting.

Searching

4-41 At ten thousand names, sorted, a binary search will do $\lg 10000 = 4$ hops in the worst case to land on an individual name. If we put the 40% good customers in their own array that's 4000 names, so 60% of the searches will resolve in $\lg 4000 \approx 3.6$ hops, while now the other 40% of not-so-good customers must first pay those 3.6 searches and then a further $\lg 6000 \approx 3.7$ hops in the worst case to land on a name in the second level array. The benefit for good customers is negligible, while the cost for the other 60% of customers almost doubles.

4-42 We can generate a sorted array with the cubes of successive integers up to $\lceil \sqrt[3]{n} \rceil$ and no further, because any subsequent cube cannot be a part of a sum of two integers that add up to n. This gives us a cost of $O(\sqrt[3]{n})$ for generating the cubes to search.

We can then apply the method from 4-9 and use binary search to find out, for each of those cubes, if another one exists such that the two add up to n. After a second search, with provisions to prevent double counting due to commutativity, if two pairs are found, it's easy to determine their cube roots based on their indices in the array. In the worst case there will be two entire passes over the array, performing a binary search for each element. The total effort then will be $O(\sqrt[3]{n} \log \sqrt[3]{n})$.

Now we seek an efficient way to generate all Ramanujan numbers between 1 and n. This algorithm can produce a list of 4-tuples (a, b, c, d) such that $a^3 + b^3 = c^3 + d^3 = i$ for each $i \in [1 .. n]$, unique up to commutativity.

By the same argument as before, the integers a, b, c, d are bounded by $\lceil \sqrt[3]{n} \rceil$, meaning we can select from the same ordered array as in the first part of the problem. A brute force approach would inspect all $\sqrt[3]{n}^2$ pairs (a^3, b^3) and try to find a second pair that adds up to $a^3 + b^3$. Using the same method as before to search for that second pair will perform $\sqrt[3]{n}$ binary searches at most, so the total complexity will be

$$\sqrt[3]{n}^2 \sqrt[3]{n} \log \sqrt[3]{n} = n^{2/3} n^{1/3} \log \sqrt[3]{n} = O(n \log \sqrt[3]{n}).$$

As it is, this algorithm does at least double the amount of work necessary—more if there is more than one way of writing some i as the sum of two cubes. It keeps no record of already verified Ramanujan numbers, so when the second pair found is visited later another search is done.

Implementation challenges

4-43 We have an $n \times n$ array such that the elements are strictly increasing along a row, and strictly decreasing along a column. Skiena gives the hint that with these constraints there can't be two zeros in the same row or column. (Further than that, there can't be any repeating number.)

Given that each row is ordered, it's possible to find a zero in it in $O(\lg n)$ time with a binary search. Suppose a zero is found in position (i, j)—it logically deletes the entire row i and column j from the search space, bomberman style. It's clear that this splits the array in up to four disjoint subarrays, each of which can be recursively searched.

In reality, if we were searching row by row and did not find zeros in, say, the first two rows, those would be out of the search space. Therefore we need only ever partition a subarray vertically in two pieces. Put another way, we are always searching in the topmost row of the search space, and splitting the array in groups of consecutive columns spanning down to the nth row.

The total number of zeros can be counted like so: start on row 0, with the search space being the horizontal range $[0 \mathinner{.\,.} n-1]$, the full row. Do a binary search on the current row. If no zero is found, move on to the next row and repeat. If a zero is found, say, at column j, then assuming we are on row i and the current search space is the range $[s \mathinner{.\,.} t]$, make two recursive calls to search the subarrays $A = \{(y, x) \mid s \le x < j,\ i < y < n\}$ and $B = \{(y, x) \mid j < x < t,\ i < y < n\}$ using the same process. Then the total number of zeros in the array is 1 plus the total number of zeros in A and B. ▌

In the worst case, there are no zeros in the array and the algorithm performs n unsuccessful binary searches; therefore the running time is $O(n \lg n)$.
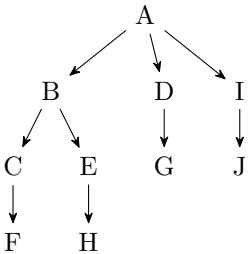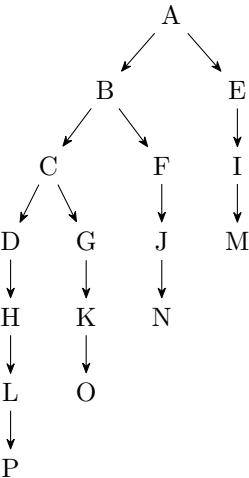
Chapter 7

# Graph traversal

Solutions

Simulating graph algorithms

7-1 With BFS, the first graph's edges are processed in the following order: $A \to B \to D \to I \to C \to E \to G \to J \to F \to H$. The traversal tree makes it easy to see that BFS processes each level in sequence

The second graph is processed in the following order: $A \to B \to E \to C \to F \to I \to D \to G \to J \to M \to H \to K \to N \to L \to O \to P$. And this is the corresponding traversal tree:

In the representation of this graph as it is printed, the BFS path looks like this:

$$A \longrightarrow B \qquad C \qquad D$$
$$E \qquad F \qquad G \qquad H$$
$$I \qquad J \qquad K \qquad L$$
$$M \qquad N \qquad O \longrightarrow P$$
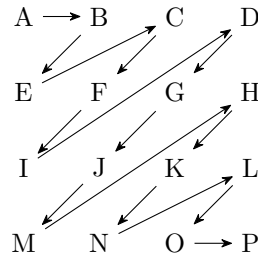
With DFS the first graph is traversed in the following order: $F \to I \to J \to H \to G \to D \to E \to C \to B \to A$.

And the second one: $M \to N \to O \to P \to L \to K \to J \to I \to E \to F \to G \to H \to D \to C \to B \to A$.

In the printed representation, the DFS path makes it evident that the processing order is the reverse of the discovery order:

$$A \longleftarrow B \longleftarrow C \longleftarrow D$$
$$E \longrightarrow F \longrightarrow G \longrightarrow H$$
$$I \longleftarrow J \longleftarrow K \longleftarrow L$$
$$M \longrightarrow N \longrightarrow O \longrightarrow P$$
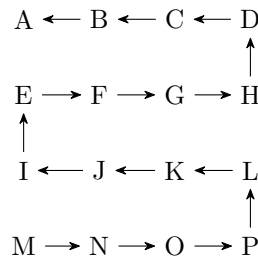
**7-2** $H \to A \to B \to D \to E \to G \to I \to J \to C \to F$.

Traversal

7-3 Suppose there is more than one path between vertices v and u. Then at some point before or at vertex u there must have been more than one incident edge, for otherwise the path would be unique. This means that some vertex must have more than one parent, in contradiction of the definition of a tree. ▌

7-4 Consider the BFS tree of an undirected graph. There can't be any forward edges because the descendant would have been discovered and processed as a child of the ancestor, making it a tree edge. There also can't be a back edge for the same reason: the edge would have been traversed—in the opposite direction—while processing v.
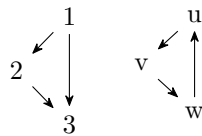
There can obviously be tree edges, and there can also be cross edges. Consider the following two examples.
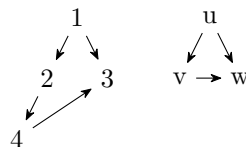
In the left example, node 3 is already processed when the cross edge $4 \to 3$ is visited. In the right example, node w is discovered but not yet processed.

The four cases of tree, forward, back and cross edges exhaustively categorize all possible edges that can be seen during traversals. ▌

7-5   Skiena says in § 19.7 *Vertex coloring* (pp. 604) that finding the chromatic number of a graph is NP-complete—a general exact solution can be found using backtracking. We can expect that the restriction on the degree of the vertices is what will save us here.
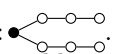
First let's get something out of the way. Here is a graph that fulfills the conditions and is not bipartite: ⬦. It is clear that this graph is tripartite. More generally, no vertex can have three neighbors that may force the use of a fourth color, so a graph with this property can be at most tripartite. ▌

An algorithm to color a graph of this kind could go as follows. Let us identify colors as increasing natural numbers. Color the initial vertex '1'. Do a BFS. Whenever a new node is visited, one of its edges must be the parent, therefore we will need a different color. If this new vertex has degree 1, it can be colored to the lowest available different color because it has no other neighbors. Otherwise it has a second neighbor, and either both neighbors share the same color, in which case we take the lowest different color, or each neighbor is of a different color, in which case we take the third available color. This algorithm is $O(n + m)$, the cost of doing the BFS.

7-6   An $O(n + m)$ algorithm is to do a DFS, which partitions the edges in tree and back edges. Back edges link to ancestor nodes and provide an alternative path to them. Any one of these edges will do.

More precisely, we seek an edge that is not a bridge. A *bridge* is a tree edge (u, v) where no back edge connects from v or a descendant to u or an ancestor. By that definition, an edge (u, v) that is not a tree edge, or such that there is an edge from v or a descendant to u or an ancestor is a safe node to delete. Any edge found by DFS that is not a tree edge is a back edge and fits that description.

7-7

a) Consider a star graph where v is the center node: ⟨star⟩. In processing v during a BFS, $n - 1$ nodes will be discovered before marking v processed and stepping out of it.

b) A linked list where v is the leftmost node: ●─○─○─○. DFS will walk to the last node of the list, marking all n nodes discovered before marking the last node processed and walking back the stack.

c) Consider a linked list with $2n + 1$ nodes, and make v the center node: ⟨list⟩. This vertex has two neighbors, each the first of an n node sublist. By running DFS on v, one of the two sublists will be traversed first to the end, and marked processed on the way back to v. At this point, n vertices are marked processed while another n are not yet discovered, namely those on the second sublist.

7-8   Here is an example tree and its pre and in-order traversals:

Pre-order: $A \to B \to D \to E \to C \to F \to G$
In-order: $D \to B \to E \to A \to F \to C \to G$.
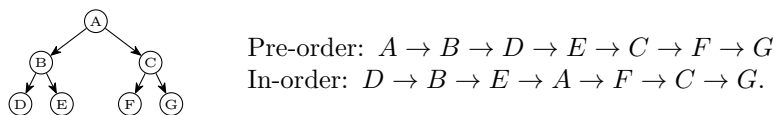
The first element in the pre-order indicates the root of the tree, A. By searching for A in the in-order traversal we can determine the elements of the left and right subtrees, namely $\{D, B, E\}$ and $\{F, G, C\}$; but each of these subtrees has to be reconstructed as well.

From the in-order traversal we see how many elements are in the left subtree: these come immediately after the root in the pre-order. Recursively, we identify B as the root of this subtree, and find that it partitions elements $\{D, B, E\}$ such that D and E are its left and right children respectively. The same method applied to the elements to the right of root A (in the in-order), namely $\{F, G, C\}$, does the same for the right subtree of the root.

But how does this method fare against incomplete binary trees?

Pre-order: $A \to B \to D \to C \to E \to F$
In-order: $B \to D \to A \to E \to C \to F$.
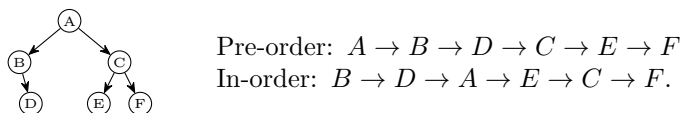
Out of a final group of 3 (two leaves and their parent), the parent is always in the middle in the in-order, so it shows up first for the case of missing left child, and last for the case of missing right child (and alone for no children). In the example above B appears first as it is the root of a subtree with no left child.

Here is a (slightly) more precise definition of the procedure. To reconstruct a binary tree, take the first element in the pre-order—this is the root. Find the element in the in-order, counting elements to the left of the root. This count is the number of elements that belong in the left subtree. Recurse to reconstruct the left subtree, with subsequences of both orders corresponding only to the elements identified as being in the left subtree. Finally, use the remaining elements—namely, those appearing after the root in the in-order—to reconstruct the right subtree. ▌

For the case where we are given pre and post-order traversals it's not possible to devise such a method. As a counterexample, here are two different trees that produce the same pair of traversals:

Pre-order: $A \rightarrow B \rightarrow C \rightarrow D$
Post-order: $C \rightarrow D \rightarrow B \rightarrow A$.

7-9   To convert from an adjacency matrix to an adjacency lists representation we have to check all n^2 positions of the matrix, so the best we can do is $O(n^2)$.

```
for i in [0 .. n-1]
  for j in [0 .. n-1]
    if M[i,j] == 1
      insert_edge(G, i, j)
```

Next we have an adjacency list representation and seek to convert it to an *incidence matrix* representation, which has n rows, one for each vertex, and m columns, one for each edge. In the incidence matrix, element $M[i, j]$ is 1 if vertex i is part of edge j.

It's worth illustrating this with an example. In this incidence matrix, the first row represents vertex A, the second vertex B and the third vertex C.

$$\begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{pmatrix}$$

It should be clear that just zeroing out the memory for an $n \times m$ matrix will take $O(nm)$ time; since walking the vertices and edges in the adjacency list representation will be $O(n + m)$, the total time complexity of this algorithm is $O(nm)$.

The data type used in the book for the adjacency list representation does not have an identifier for edges. In other words, while traversing the edges adjacent to a vertex we have no way of knowing which column in the incidence matrix corresponds to each one.

Suppose we have a counter that indicates the next column to populate. If we can afford the memory space, we can use a secondary matrix M' of size $n \times n$ that is not only an adjacency matrix but also tells us what column of the incidence matrix represents the edge that connects to vertices.

Let us run this idea for the example above with three vertices and two edges. Initially our column pointer is 1, the first edge. Visit A and traverse its first edge: $A \rightarrow B$. We check M' and see that $M'[A, B] = 0$. This means we have a new edge, therefore we set $M'[A, B] = M'[B, A] = 1$, the current value of the column pointer. In the incidence matrix, mark $M[A, 1] = 1$, then increment the column pointer to 2. Now $M = \left(\begin{smallmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \end{smallmatrix}\right)$ and $M' = \left(\begin{smallmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{smallmatrix}\right)$.

As A has no more edges, we now visit vertex B. The first edge in B is $B \rightarrow A$. We check $M'[B, A]$ and see it indicates this edge is represented by column 1 in the incidence matrix, therefore mark $M[B, 1] = 1$. The next edge in B is $B \rightarrow C$; $M'[B, C]$ is not set, so this is a new edge. Mark $M[B, 2] = 1$ and $M'[B, C] = M'[C, B] = 2$. Increment the column pointer to 3. Now $M = \left(\begin{smallmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 0 \end{smallmatrix}\right)$ and $M' = \left(\begin{smallmatrix} 0 & 1 & 0 \\ 1 & 0 & 2 \\ 0 & 2 & 0 \end{smallmatrix}\right)$.

Finally we visit C and see that its only edge, $C \rightarrow B$ is already known and represented in column 2, since $M'[C, B] = 2$. So we mark $M[C, 2] = 1$, and since this is the last edge of the last vertex, we are done with $M = \left(\begin{smallmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{smallmatrix}\right)$. ▌

This algorithm visits each vertex and each edge only once, so it takes $O(n + m)$ time. Because of the auxiliary matrix, it uses $O(\max(n^2, nm))$ space.

Finally, we must convert from an incident matrix to an adjacency list representation. The approach is clearer—go column by column, inserting each edge into the adjacency list of the two affected vertices. Scanning each column takes $O(n)$ time, and there are m columns, for a total $O(mn)$ time.

7-10 Evaluation of an expression tree has a natural recursive implementation. Let us assume the tree nodes are represented as a sum type, where the two variants are either a binary operation along with two subtrees, or a terminal node with a single number. To evaluate an expression tree, evaluate the subtrees and apply the operation to the two results; to evaluate a terminal node, just return its value. ▮

Each node is visited once, with the results of more nested computations bubbling up to the final binary operation. The running time is therefore $O(n)$.

7-11 To evaluate an expression with shared subexpressions represented as a DAG we can run a DFS with an auxiliary array to store the computed values for each node. By hooking into `process_vertex_late()` the DFS ensures that the values of all dependencies are already computed by the time an internal node is processed.

Note that this requires traversing the adjacency list of each node twice—once during the graph traversal, then again when processing each internal node in order to collect all the intermediate results computed "backwards" in the DAG. The second pass over the edges does not result in traversing the graph again, as every result is already stored in an auxiliary array and can be accessed in $O(1)$. Concretely, shared subexpressions need not be traversed twice; their results are cached in the auxiliary array and can be accessed directly by the time another part of the expression requires their value.

The DFS takes $O(n + m)$ time, then the m edges are visited again while unwinding the call stack to collect the computed values of every subexpression referred to by internal nodes, for a total $O(n + m)$ running time.

Applications

7-13 The chutes and ladders graph can be viewed as a directed graph where there is an edge from each cell to the next one in number, which are replaced by edges to arbitrary cells in ones that represent chute mouths or ladder bases.

We need to determine a shortest path from the starting cell to the final one, but one thing to consider is we only have one six sided die, so some paths may require more than one throw. We know BFS will give us a shortest path from start to end, but a condition of the game is that one must *land* on a ladder or chute to take the shortcut. Therefore these vertices in the path have to be precisely targeted with die throws.

Suppose we have a shortest path, courtesy of BFS. To minimize the number of throws we always want to advance as much as possible towards the next shortcut. A shortcut cell can be identified by the fact that the following cell is not its immediate successor. For example, if we had a path $[1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 23 \rightarrow 24]$ we can see that 8 is the base of a ladder; whereas a path like $[5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 3 \rightarrow 4]$ tells us that 10 is the mouth of a chute.

A linear scan over the shortest path would let us count the number of throws necessary to traverse it: take as many steps as possible (up to six) towards the next shortcut or the final cell. The BFS is $O(n + m)$ while the linear scan is $O(n)$ since in the worst case all nodes will be part of the shortest path.

7-14 We don't need to find a shortest path—any path will do, as long as it is safe. Let us consider the poles as being the vertices of a complete graph. A modified DFS, in which only the neighbors within a safe distance are visited, will eventually make its way to the target pole if a safe path exists. Since the graph is complete, there is a quadratic number of edges with respect to the number of vertices, therefore this is an $O(n^2)$ algorithm.

7-15 This is a coloring problem, each table being a color. The input is easily interpreted as a graph, each guest being a vertex, and the list of guests with whom they are on bad terms being its adjacency list. The algorithm, based on BFS, traverses the components in this graph assigning the first guest in each component to, say, table 1, and then each visited neighbor to the complement of the table of the parent. If any guest is already assigned to the same table as one of its neighbor vertices, the graph is not bipartite and there is no suitable seating arrangement.

Algorithm design

7-16 Skiena forgot to finish the problem statement, but let's assume he meant for us to construct the square of some given graph.

In the square of a directed graph there is an edge between any two vertices that are at distance 2 in the original graph (that is, they are separated by two edges). The intuitive way to think about the construction of the square of a graph is that, for every vertex v, we want to list all of its neighbors' neighbors, and draw the direct connections in a new graph with the same set of vertices.

$$G = \qquad\qquad G^2 =$$
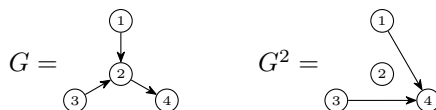
With an adjacency matrix representation, start by initializing and zeroing out a new adjacency matrix of dimension $n \times n$. Then for each vertex v in the original graph, traverse the row or column of its neighbors u. For each vertex w that is a neighbor of u, add an edge in the new adjacency matrix between v and w. Looking at all the neighbors of each vertex u is $O(n^2)$, and this is done n times, one for each vertex v, making this algorithm $O(n^3)$.

With an adjacency list representation, the algorithm proceeds by traversing the adjacency list of each vertex v. For every edge (v, u), the adjacency list of u is traversed, adding edges from v to each neighbor w of u in the new graph. The traversal of the neighbors of u takes $O(n)$ time, and there are m edges in the graph, for a total $O(nm)$ time algorithm.

7-17 Let $G = (V, E)$ be a graph. We seek to find minimum size vertex covers—subsets of V such that every edge in the original graph is still incident to some vertex.

First, let G be a tree. Intuitively, and visually, we can safely remove every leaf and be left with a vertex cover. Now we must select the parents of those discarded leaves into the cover. We can now consider the tree that results from ignoring the discarded leaves and their selected parents, and apply the same process until we have trimmed the entire tree.
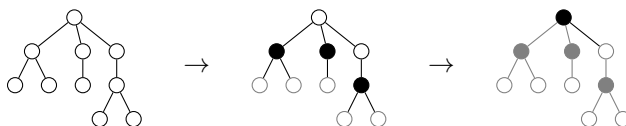
Now let G be a tree such that the weight of each vertex is equal to its degree. We seek a minimum weight vertex cover of G. This is trickier. Here is an example, with vertices labeled with their weight.
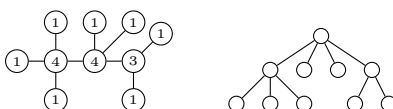
Consider any leaf vertex v. It must be of degree 1. Its only neighbor (its "parent") can be of degree larger or equal to v. It can only be of degree 1 when it is the root of the tree and has a single child, otherwise it has a parent and therefore is of degree higher than v. Thus it is always convenient to select the leaves of the tree to cover their edges. By the same logic as before, after selecting the leaves into the cover and discarding their parents, ignore them all and consider the resulting tree to proceed recursively.
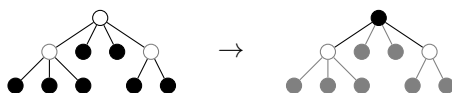
Note that after selecting the two leaf vertices connected to the root we can't discard their parent because it has other child nodes that have been discarded, therefore the edges connecting them to the root vertex still have to be covered. In other words, we can only discard a parent node if all of its children have been selected.

Finally, let G be a tree with arbitrary weights. Again we seek an algorithm to find a minimum weight vertex cover.

7-18 The DFS tree of a (undirected) graph can have tree edges or back edges. Consider any leaf. If it is of degree 1 then it can be safely deleted because its discoverer will cover its one edge; otherwise it has back edges to already visited vertices, and it can also be safely deleted because those vertices will cover said back edges. ▍

7-19 Now we turn to the concept of an *independent set* of an undirected graph $G = (V, E)$. This is a set of vertices U such that no edge in E is incident to two vertices in U; equivalently, a set U such that every edge in E is incident to *at most* one vertex in U. Behind this equivalent definition rests the fact that an independent set of vertices allows for edges not adjacent to any vertices.

An intuitive way to understand the construction of an independent set is that, for any edge in E, only one of its endpoints can be selected into U. For trivial cases, consider that the empty set and any unit set—if the graph is simple or has no edges from and to the same vertex—form independent sets.

First, we are concerned with selecting an independent set of maximum size. It is clear that selecting a vertex precludes any of its neighbors from being selected. In the case of trees, selecting a vertex prevents its parent and children from being selected, but has no bearing on the selection of its siblings. Since a single node can have one or more children, it always pays to select the leaves at the cost of discarding their parents. This is the inverse logic to that of 7-17—select leaves, discard parents, recurse.

Now we seek to maximize the weight, where each vertex in the tree has weight equal to its degree. Now in choosing between all the leaves and their parent, in all but the case of the root the parent will be of degree one more than the sum of all the leaves; therefore it always pays to discard the leaves and select their parent. This selection in turn discards the parent's parent; ignore all three generations and recurse. (Note that cases like the tree below can result in uncovered edges.)
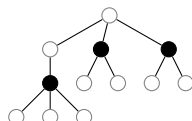
Finally we seek to maximize the total weight of an independent set of a tree with arbitrary weights. We established that a choice has to be made between a vertex and its children, and the traversal that lets us aggregate the total weight of the immediate children and compare it with the weight of the parent is the post-order traversal.

Consider a tree of depth 2. Each leaf can only report its own weight, and mark itself as selected, for the choice is not in conflict with any subtrees and it maximizes the total local weight. Then the parent may make a choice as to select itself or its children.

Now consider a higher level node, v. When processing such a node there are three possible scenarios: 1) all children are discarded, 2) all children are selected, 3) some children are discarded and some are selected. Scenario 1) is the easiest: each subtree is maximized already, so we can safely select v if it will increase the total weight, namely if its weight is positive. Scenario 2) is not as simple. It requires a decision about whether it is convenient to select v at the cost of flipping the children. Finally, scenario 3) is sort of an intermediary state—if v is selected, the selected children have to be flipped; if v is discarded, then no changes are necessary to any children.

Let each node carry its weight, state (selected or discarded) and the sum of the total selected and discarded weight across its subtrees, each including the node's own weight. Traverse the tree post-order, and suppose we are processing some non-leaf node v. Because of the post-order traversal, every subtree has already been processed and maximized.

If all children are discarded, select v provided it adds to the total weight.

If all children are selected, compute the potential weight as the sum of the discarded weight over all subtrees plus the current node weight; if it is larger than the sum of all the selected weight across children, select v and recursively flip its subtrees. Otherwise discard v.

Finally, if some children are selected and some are discarded, perform the same comparison as above with the selected children, and determine if they need to be flipped so v can be selected.

After processing each node, compute and store the aggregate weight of each kind. The maximum weight will be stored in the root node of the tree. ∎

This is far from a proof. Particularly the notion of "flipping" a subtree may be worth exploring more. When a child is selected, flipping it does not necessarily imply that its children also have to be flipped—it is acceptable for a discarded node to have discarded children. Therefore a more sophisticated method of reprocessing a subtree may be required to guarantee the maximum possible weight under the constraint that the root is to be selected or discarded.

7-20 An *independent vertex cover* is a set of vertices that both forms a vertex cover, and is an independent set. From the definitions above, this is a set of vertices such that every edge in G is covered by exactly one vertex. Finding such a set is equivalent to 2-coloring—if the graph is bipartite we will find that discarding one of the groups leaves an independent vertex cover. The basic BFS method to color the graph can both find such a coloring and test for bipartiteness.

7-21 What does a triangle in a graph look like? One obvious property is that a triangle's vertices must have degree at least 2.

The obvious brute force algorithm takes each vertex v and, for every other vertex u, if $(u, v) \in E$, it looks over every other vertex w to see if $(v, w) \in E$. There are $|V|$ total vertices in G, and three nested passes over V, so this is a $O(|V|^3)$ algorithm. Building the adjacency matrix is $O(|V|^2)$ for zeroing out the memory and $O(|V| + |E|)$ for traversing the adjacency lists.

Better time complexity can be achieved by considering the BFS tree of a graph. A triangle will be detected as soon as a cross-edge to a sibling is explored. This is an edge between two nodes with the same parent.

NB. However, the BFS is $O(|V| + |E|)$ so this is not the desired $O(|V| \cdot |E|)$ algorithm Skiena had in mind. I can't think of what that algorithm is, and this is the reason for the shoveling sign.

7-22 We have k movies and a set of customers, each of whom wants to see two movies during the weekend. We must find a screening schedule that lets everyone enjoy the movies they want.

This feels somewhat like a two-coloring problem, but it's not readily apparent what the correct modeling is. Since multiple movies can be screened at once, one trivial solution is to screen every movie every day; whatever pair of movies (M_i, M_j) a customer wants to see will necessarily be screened, and therefore everyone will be satisfied.

But what about a schedule where every movie is screened only once? We can assume consumers don't care in which order they watch their movies. If every movie must be screened once, then for any pair of movies a customer wants to watch, those two movies can't be screened on the same day. Consider then the set of movies as vertices, and for every consumer, draw an edge between the two movies they want to watch. Then 2-color the graph, one color representing each day of the weekend. This can be done with BFS; if the movie graph is not bipartite, there is no schedule that works. ▍

7-23 One important detail about this exercise—that I overlooked at first—is that it defines, and asks to determine, the diameter of a *tree*, not a general graph. This should hopefully make the task simpler.

Let's consider a couple of examples to drive the thinking.
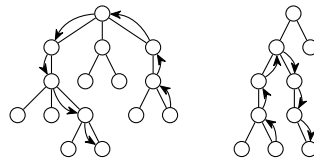
The arrows mark the longest path between nodes in each tree. The left tree has diameter 7, and the right tree has diameter 6. The right tree makes it clear that the longest path between nodes isn't always through the root. In fact, the longest path will be between two subtrees such that the sum of their depth is maximized throughout all subtrees of a common parent.

In the first example, the root has subtrees of depth 4, 2 and 3. The maximum depth sum is 7, and no other node has two subtrees whose depth adds up to a larger number. Therefore, the deepest leaves on those two subtrees provide a path whose edge count is the diameter of the tree.

In the second example the root has subtrees of depth 4 and 1, but its left child has subtrees of depth 3 and 3; their sum is 6 and this represents the maximum possible distance between two leaves in the tree.

To show the correctness of this reasoning, first note that the diameter of a tree must be given by a path between two leaves; if it were not, and one or both of the endpoints were not leaves, we could extend the length of the path with edges to their leaves.

Now suppose that we determine the diameter of the tree as the sum of the depth of the two deepest subtrees of some node, as we mentioned above, and furthermore suppose that this is incorrect, and there is some larger path between two nodes in the tree. These two nodes must share an ancestor, and the length of the path between them must be the sum of the paths between each of them and this common ancestor. But this is exactly the measure by which we had found our maximal length path, so this new sum can't be larger. ▍

Now, on to the method by which we determine the diameter of a tree. Traverse the tree post-order. When finished processing a node, take the maximum depth across children plus one, as well as the sum of the two maximum depths across children, and return them to the parent node. Each child will have reported the sum of the two deepest subtrees, so each node can choose to send that value instead of it can't do better with its own subtrees. The maximum depth across all children serves to add up the total depth of the subtree, by which a node's parent knows which two subtrees to add up as the longest path. This maximum sum is the diameter of each subtree—at the end of the traversal, when processing the root node, this will be the diameter of the tree.