Table of Contents — Email any errors to rsha256@berkeley.edu, sherryfansf@berkeley.edu

Contents

1 Chapter 0 **2**

2 Chapter 1 **5**

3 Chapter 2 **8**

4 Chapter 3 **12**

5 Chapter 4 **16**

6 Chapter 5 **17**

7 Chapter 6 **20**

8 Chapter 7 **24**

1 Chapter 0

Exercise 0.1
In each of the following situations, indicate whether $f = O(g)$, or $f = \Omega(g)$, or both (in which case $f = \Theta(g)$). We denote L'Hopital's Rule as 'LH'.

(a) $n - 100 = \Theta(n - 200)$, drop lower order terms (l.o.t.'s).

(b) $n^{1/2} = O(n^{2/3})$, $0.5 < 0.\bar{6}$. Since $\lim_{n\to\infty} \frac{n^{1/2}}{n^{2/3}} = 0 \implies f = O(g)$.

(c) $100n + \log n = \Theta(n + (\log n)^2)$. $n \gg \log^2 n$, per LH. Drop l.o.t.'s: $100n = \Theta(n)$.

(d) $n \log n = \Theta(10n \log 10n)$, g can be written as $10n(\log 10 + \log n) \in \Theta(n \log n)$.

(e) $\log 2n = \Theta(\log 3n)$, as $\log 2 + \log n$ and $\log 3 + \log n$ are both $\Theta(\log n)$.

(f) $10 \log n = \Theta(\log(n^2))$, as $\log(n^2) = 2 \log n$ and as $n \to \infty$ we drop coefficient factors.

(g) $n^{1.01} = \Omega(n \log^2 n)$ as $\lim_{n\to\infty} \frac{n^{0.01}}{\log^2 n} \to \infty$ as polynomial growth $\gg$ logarithmic growth.

(h) $\frac{n^2}{\log n} = \Omega(n(\log n)^2)$ as $\lim_{n\to\infty} \frac{n}{\log^2 n} \to \infty$ as polynomial growth $\gg$ logarithmic growth.

(i) $n^{0.1} = (n^{0.01})^{10} = \Omega((\log n)^{10})$, as polynomial growth $\gg$ logarithmic growth.

(j) $(\log n)^{\log n} = \Omega(\frac{n}{\log n})$, $\lim_{n\to\infty} \frac{\log^{\log(n)}(n)}{n/\log n} = \lim_{n\to\infty} \frac{\log^{\log(n)+1}(n)}{n} \overset{\text{LH}}{=} \lim_{n\to\infty} \frac{\frac{\mathrm{d}}{\mathrm{d}n}\left[\log^{\log(n)+1}(n)\right]}{1} \to \infty$.

(k) $\sqrt{n} = \Omega((\log n)^3)$, as $\sqrt{n} = n^{0.5}$ and polynomial growth $\gg$ logarithmic growth.

(l) $n^{1/2} = O(5^{\log_2 n})$. $5 = 2^{\log_2 5} \implies 5^{\log_2 n} = (2^{\log_2 5})^{\log_2 n} = (2^{\log_2 n})^{\log_2 5} = n^{\log_2 5} \approx n^{2.3} \gg n^{0.5}$.

(m) $n2^n = O(3^n)$ as $3^n = (1.5 \cdot 2)^n = 1.5^n \cdot 2^n$ and $n \ll 1.5^n$ asymptotically.

(n) $2^n = \Theta(2^{n+1})$ as $2^{n+1} = 2 \cdot 2^n$ and we drop coefficients as $n \to \infty$.

(o) $n! = \Omega(2^n)$, as factorial $\gg$ exponential (can be proved via induction).

(p) $(\log n)^{\log n} = O(2^{(\log_2 n)^2})$, the RHS $= n^{(\log_2 n)}$ and $\log n \ll n \implies (\log n)^{\log n} \ll n^{\log n}$.

(q) $\sum_{i=1}^{n} i^k = O(n^{k+1})$, as $\lim_{n\to\infty} \frac{\sum_{i=1}^{n} i^k}{n^{k+1}} = \lim_{n\to\infty} \frac{1 + 2^k + \cdots + (n-1)^k + n^k}{n^{k+1}} = 0$ as $n^{k+1} \gg n^k$.

Exercise 0.2
Show that, if c is a positive real number, then $g(n) = 1 + c + c^2 + \cdots + c^n$ is:

$$g(n) = 1 + c + c^2 + \cdots + c^n = \sum_{i=0}^{n} c^i = \frac{c^{n+1} - 1}{c - 1}, \ (\forall c \neq 1).$$

(a) If $c < 1$, then $\lim_{n\to\infty} g(n) = \frac{0-1}{c-1} = \frac{1}{1-c} \implies (\forall c < 1)$, $\boxed{g \in \Theta(1).}$

(b) If $c = 1$, then $g(n) = 1 + \underbrace{c^1 + c^2 + \cdots + c^n}_{n \text{ coeff.'s}} = 1 + \underbrace{1 + 1^2 + \cdots + 1^n}_{n \text{ times}} = n+1 \implies (\forall c = 1)$, $\boxed{g \in \Theta(n).}$

(c) If $c > 1$, then $\lim_{n\to\infty} g(n) = \lim_{n\to\infty} \frac{c^{n+1}}{c} = c^n \implies (\forall c > 1)$, $\boxed{g \in \Theta(c^n).}$

Exercise 0.3
The Fibonacci numbers $F_0, F_1, F_2, \ldots$, are defined by the rule:

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2}.$$

(a) **Use induction to prove that $F_n \geq 2^{0.5n}$, $(\forall n \geq 6)$**
Base Case(s): $F_6 = 8 \geq 2^{0.5 \cdot 6} = 2^{\frac{6}{2}} = 2^3 = 8$. $F_7 = 13 \geq 2^{3.5}$. $F_8 = 21 \geq 2^4$.
Inductive Hypothesis: Assume $F_k \geq 2^{0.5k}$ for some $n = k$, where $k \geq 6$.
Inductive Step: WTS: $F_k \geq 2^{0.5k} \implies F_{k+1} \geq 2^{0.5(k+1)} = 2^{0.5k+0.5}$
$F_{k+1} = F_k + F_{k-1} > 2 \cdot F_{k-1} \geq 2 \cdot 2^{0.5(k-1)} = 2^{0.5k-0.5+1} = \boxed{2^{0.5k+0.5}}$. $\square$

(b) **Find a constant $c < 1$ such that $F_n \leq 2^{cn}$, $(\forall n \geq 0)$.**
$F_n = F_{n-1} + F_{n-2} \leq 2^{(c(n-1)} + 2^{(c(n-2)} \leq 2^{(cn)} \cdot 2^{-c} + 2^{(cn)} \cdot 2^{(-2c)} \leq 2^{(cn)}(2^{-c} + (2^{-c})^2)$
$\therefore (2^{-c} + (2^{-c})^2) \leq 1$. If we look at specific case of equality with the substitution $x = 2^{-c}$, we get the following: $x^2 + x - 1 = 0$ which has roots at $x = \frac{-1 \pm \sqrt{5}}{2}$ per the quadratic formula.
Note that we take the positive root as the logarithm (used below) is only defined for inputs > 0.

$$x = 2^{-c} \implies c = -\log_2 x$$

$$\boxed{\text{This holds true } \forall c \text{ s.t. } 0.694242 \leq c < 1}.$$

(c) $F_n = \Omega(2^{cn}) \implies \exists k \in \mathbb{Z}^+ : F_n \geq k \cdot 2^{cn}$. WLOG equality holds when $c = -\log_2(\frac{1}{2}(-1 + \sqrt{5}))$

$$\implies \boxed{c \approx 0.694242}$$

Exercise 0.4

In general, to compute F_n, it suffices to raise the following 2×2 matrix, called X, to the nth power:

$$\begin{bmatrix} F_n \\ F_{n+1} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \cdot \begin{bmatrix} F_0 \\ F_1 \end{bmatrix}.$$

(a) **Show that two 2×2 matrices can be multiplied via 4 additions & 8 multiplications:**
For 2×2 matrices:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} a \cdot e + b \cdot g & a \cdot f + b \cdot h \\ c \cdot e + d \cdot g & c \cdot f + d \cdot h \end{bmatrix}$$

which clearly has four additions (denoted by $+$) and eight multiplications (denoted by $\cdot$).

But how many matrix multiplications does it take to compute X^n?

(b) **Show that $O(\log n)$ matrix multiplications suffice for computing X^n.**
Repeated Squaring Algorithm:

$$X^n = \begin{cases} (X^{\frac{n}{2}})^2 & \text{if } n \text{ is even} \\ X \cdot X^{n-1} & \text{if } n \text{ is odd} \end{cases}$$

Note that this is applied recursively with n guaranteed to be reduced by half within every 2 iterations $\implies$ Repeated Squaring Algorithm $\in O(\log n)$.

(c) **Show that all intermediate results of `fib3` are $O(n)$ bits long.**
Adding two numbers that have n bits will result in at most an $n+1$ bit number.

(d) **Prove that the running time of `fib3` is $O(M(n) \log n)$.**
There are $\log n$ multiplications, each of which is $M(n)$ $\implies$ `fib3` $\in O(M(n) \log n)$.

(e) **Can you prove that the running time of `fib3` is $O(M(n))$?**
Yes! The lengths of the numbers being multiplied get doubled with every squaring.

2 Chapter 1

Exercise 1.1
WTS: In any base $b \geq 2$, the sum of any three single-digit numbers is at most 2 digits long.
A single digit in base b is at most $b-1$, this can be seen via induction (ex. a digit in base 2 is at most 1 and a single digit in base 10 is at most 9).

Exercise 1.2
Show that any binary integer is at most 4x as long as the corresponding decimal integer. For very large numbers, what is the ratio of these two lengths, approximately?
Length of binary integer (an int expressed in bin) is $\lfloor \log_2 x \rfloor + 1$ where x : decimal value of the integer.
Length of decimal integer is $\lceil \log_{10} x \rceil$ where x is the decimal integer (integer expressed in base 10).

$$\implies \lfloor \log_2 x \rfloor + 1 \leq 4 \cdot \lceil \log_{10} x \rceil$$
$$\implies \lfloor \log_2 x \rfloor + 1 < \log_2 x + 1$$
$$\implies \log_2 x < \log_2 x + 1$$
$$\implies 4 \cdot \lceil \log_{10} x \rceil > 4 \cdot \log_{10} x$$
$$\implies \log_2 x \leq 4 \cdot \log_{10} x$$
$$\implies \log_{10} x = \frac{\log_2 x}{\log_2 10}$$
$$\implies \boxed{\log_2 x \cdot \log_2 10 \leq 4}$$

Exercise 1.5
Show that the harmonic series is $\Theta(logn)$.
Group the terms as such:

$$\left(\frac{1}{1}\right) + \left(\frac{1}{2} + \frac{1}{3}\right) + \left(\frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7}\right) + \ldots \left(\frac{1}{2^{k-1}} + \frac{1}{2^{k-1}+1} + \ldots + \frac{1}{2^k - 1} + \frac{1}{2^k}\right)$$

For an upper bound, consider the sequence

$$\left(\frac{1}{1}\right) + \left(\frac{1}{2} + \frac{1}{2}\right) + \left(\frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4}\right) + \ldots + \left(\frac{1}{2^k} + \ldots + \frac{1}{2^k}\right)$$
$$= 1 \cdot \frac{1}{1} + 2 \cdot \frac{1}{2} + 4 \cdot \frac{1}{4} + \ldots + 2^k \cdot \frac{1}{2^k}$$

If n is not a power of 2, pad with some extra terms. In this sequence $k = \lceil \log_2 n \rceil \leq \log_2 n + 1$. Thus, $\sum_{i=1}^{n} \frac{1}{i} < 1 * (\log_2 n + 1) = \Theta(\log n)$.

For an lower bound, consider the sequence

$$\left(\frac{1}{2}\right) + \left(\frac{1}{4} + \frac{1}{4}\right) + \left(\frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8}\right) + \ldots + \left(\frac{1}{2^{k-1}} + \ldots + \frac{1}{2^{k-1}}\right)$$
$$= 1 \cdot \frac{1}{2} + 2 \cdot \frac{1}{4} + 4 \cdot \frac{1}{8} + \ldots + 2^k \cdot \frac{1}{2^{k-1}}$$

Again, if n is not a power of two, truncate the extra terms. In this sequence, $k - 1 \leq \log_2 n$. Thus, $\sum_{i=1}^{n} \frac{1}{i} \geq 1 * (\frac{1}{2} * \log_2 n) = \Theta(\log n)$.

Exercise 1.6
Prove that the standard multiplication algorithm works for binary numbers.
Consider some number in binary: $b = b_0 2^0 + b_1 2^1 + ... + b_n 2^n$ and its multiplicand c.

$$b * c = cb_0 2^0 + cb_1 2^1 + ... + cb_n 2^n$$

This is exactly what the grade-school multiplication algorithm does: it multiplies c by each digit in b, and shifts if by the appropriate amount (the same as multiplying by 2^k).

Exercise 1.8
Justify the correctness of the recursive division algorithm, and show that it takes $O(n^2)$ time on n-bit inputs.
We induct on the number of bits in x. The base case is evident: a 0-bit number ($x = 0$) always has quotient and remainder 0. Suppose on a $(n-1)$-bit number, the recursive division algorithm correctly returns (q, r) such that $q * y + r = x$.
Consider an n-bit number x. By the inductive hypothesis, `divide(x/2, y)` returns q', r' such that $q'y + r' = \lfloor x/2 \rfloor$. If x is even, $x = 2q'y + 2r'$, and $r = 2r'$, $q = 2q'$. If x is odd, $x = 2q'y + 2r' + 1$, and and $r = 2r' + 1$, $q = 2q'$. This is exactly the return value of the algorithm, proving its correctness.
The only last thing we have to consider is if $r \geq y$. But in this case, $x = qy + r = y(q+1) + (r-y)$, which gives us a new remainder $r - y$ and quotient $q + 1$.

Exercise 1.11
Is $4^{1536} - 9^{4824}$ divisible by 35?
Yes. We can use CRT to show:

$$4^{1536} - 9^{4824} \equiv (-1)^{1536} - (-1)^{4824} \equiv 1 - 1 \equiv 0 \pmod 5$$
$$4^{1536} - 9^{4824} \equiv 4^{6 \cdot 256} - 2^{6 \cdot 804} \equiv 4^{6 \cdot 256} - 2^{6 \cdot 804} \overset{\text{FLT}}{\equiv} 1^{256} - 1^{804} \equiv 0 \pmod 7$$

Therefore, $\gcd(5, 7) = 1 \implies 4^{1536} - 9^{4824} \equiv 0 \pmod{35}$.

Exercise 1.12
What is $2^{2^{2006}} \pmod 3$?
We can use FLT:

$$2^{2^{2006}} \equiv 2^{4012} \boxed{\equiv 1 \pmod 3}$$

as 2 to an even power is 1 in mod 3.

Exercise 1.13
Is the difference of $5^{30,000}$ and $6^{123,456}$ a multiple of 31?
We can use FLT: $a^{p-1} \equiv 1 \pmod p$

$$\begin{aligned}
&(5^{30,000} - 6^{123,456}) \pmod{31} \\
&\equiv 5^{30 \cdot 1,000} - 6^{123,456} \pmod{31} \\
&\overset{\text{FLT}}{\equiv} 1^{1000} - 6^{123,456} \pmod{31} \\
&\equiv 0
\end{aligned}$$

So we can see the difference *is* a multiple of 31.

Exercise 1.20
Find the inverse of: $20 \bmod 79, 3 \bmod 62, 21 \bmod 91, 5 \bmod 23$**.**
We can use FLT:

$$\begin{aligned}
&(5^{30,000} - 6^{123,456}) \pmod{31} \\
\equiv\ &(5^{30,000} \pmod{31}) - (6^{123,456} \pmod{31}) \\
\equiv\ &(5^{30 \cdot 1,000} \pmod{31}) - (6^{123,456} \pmod{31}) \\
&26 \neq 0
\end{aligned}$$

So we can see the difference *is* a multiple of 31.

Exercise 1.34
Suppose a particular coin has a probability p of coming up heads. How many times must you toss it on average before it comes up heads?
$1/p$. (This is a geometric distribution).

3 Chapter 2

Exercise 2.2
Show that for any positive integers n and any base b, there must be some power of b lying in the range $[n, bn]$.
Let $x = b^{\lfloor \log_b n \rfloor}$ be the smallest power of b less than or equal to n. $bx = b^{\lfloor \log_b n \rfloor + 1} \geq b^{\log_b n} = n$. Also, since by definition $x \leq n$, $bx \leq bn$. Thus, bx is in the range $[n, bn]$.

Exercise 2.5
Solve the following recurrence relations and give a Θ bound for each of them.

(a) $T(n) = 2T(\frac{n}{3}) + 1 \implies \frac{a}{b^d} = \frac{2}{3^0} = \frac{2}{1} = 2 > 1 \implies f \in \Theta(n^{\log_b a}) = \Theta(n^{\log_3 2})$.

(b) $T(n) = 5T(\frac{n}{4}) + n \implies \frac{a}{b^d} = \frac{5}{4^1} = \frac{5}{4} > 1 \implies f \in \Theta(n^{\log_b a}) = \Theta(n^{\log_4 5})$.

(c) $T(n) = 7T(\frac{n}{7}) + n \implies \frac{a}{b^d} = \frac{7}{7^1} = 1 \implies f \in \Theta(n^d \log n) = \Theta(n^1 \log n) = \Theta(n \log n)$.

(d) $T(n) = 9T(\frac{n}{3}) + n^2 \implies \frac{a}{b^d} = \frac{9}{3^2} = 1 \implies f \in \Theta(n^d \log n) = \Theta(n^2 \log n)$.

(e) $T(n) = 8T(\frac{n}{2}) + n^3 \implies \frac{a}{b^d} = \frac{8}{2^3} = 1 \implies f \in \Theta(n^d \log n) = \Theta(n^3 \log n)$.

(f) $T(n) = 49T(\frac{n}{25}) + n^{3/2} \log n \implies \frac{a}{b^d} = \frac{49}{25^2} < 1 \implies f \in \Theta(n^{3/2} \log n)$.

(g) $T(n) = T(n-1) + 2 \implies T(n) = \sum_{i=1}^{n} 2 = 2n \in \Theta(n)$.

(h) $T(n) = T(n-1) + n^c$, where $c \geq 1$ is a constant $\implies T(n) = \sum_{i=1}^{n} i^c \in \Theta(n^{c+1})$.

(i) $T(n) = T(n-1) + c^n$, where $c > 1$ is some constant $\implies T(n) = \sum_{i=1}^{n} c^i = \frac{c(c^n - 1)}{c-1} \in \Theta(c^n)$.

(j) $T(n) = 2T(n-1) + 1 \implies T(n) = 2^n - 1$ (by induction) $\implies T(n) \in \Theta(2^n)$.

(k) $T(n) = T(\sqrt{n}) + 1 \implies T(b^{2^k}) = \underbrace{T(b^{2^{k-1}})}_{k} + 1 = \log_2(\log_b(b^{2^k})) + 1 \implies T(n) \in \Theta(\log(\log(n)))$.

Exercise 2.11
Show that block matrix multiplication is valid.
In the multiplication $XY = Z$, consider any entry z_{ij} of Z.

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

Suppose $i, j < n/2$.

$$z_{ij} = \sum_{k=1}^{n} x_{ik} y_{kj} = \sum_{k=1}^{n/2} x_{ij} y_{kj} + \sum_{k=n/2+1}^{n} x_{ij} y_{kj} = \sum_{k=1}^{n/2} a_{ij} e_{kj} + \sum_{k=n/2+1}^{n} b_{ij} g_{kj}$$

A similar proof can be done for the other 3 entries.

Exercise 2.12
How many lines, as a function of n (in $\Theta(\cdot)$ form), does the following program print? Write a recurrence and solve it. You may assume n is a power of 2.

```
function f(n)
    if n > 1:
```

```
        print_line(''still going'')
        f(n/2)
        f(n/2)
```

We can apply the Master Theorem: we have $a = 2$ recursive calls, each of which reduce the problem size by $b = 2$ and in each call, we do constant extra work so $d = 0$. This has recurrence:

$$T(n) = 2T(\lceil n/2 \rceil) + O(1)$$

This means that $\frac{a}{b^d} = \frac{2}{1} > 1 \implies f \in \Theta(n^{\log_b a})$. Therefore $f \in \Theta(n)$.

Exercise 2.13
Find the number of binary trees on 3, 5, and 7 vertices.
$B_3 = 1, B_5 = 2, B_7 = 5$.

Find a general recurrence for B_n.

$$B_n = \sum_{i=1}^{n} B_{i-1} B_{n-i}$$

Show by induction that $B_n \in \Omega(2^n)$.
We will show more specifically that $B_n \geq c2^n$.
The base cases hold if we choose some $c \leq 2^{-5}$:

$$B_3 = 1 \geq 2^{-2}$$

$$B_5 = 2 \geq 2^0$$

$$B_7 = 5 \geq 2^2$$

Assuming this holds for any $7 < k < n$, consider B_n.

$$B_n = \sum_{i=1}^{n} B_{i-1} B_{n-i}$$

$$\geq \sum_{i=1}^{n} \cdot 2^{i-1} * 2^{n-i}$$

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

$$= cn \cdot 2^{n-1}$$

Since $n \geq 8$,

$$cn \cdot 2^{n-1} \geq c \cdot 2^{n+2} \geq c2^n$$

Exercise 2.15
Show how to implement the `split` operation (which splits numbers according to the pivot) in place.

```
def split(arr, pivot):
    n = len(arr)
    i, j = 0, n - 1
    while i < j:
        if arr[i] >= pivot and arr[j] < pivot:
```

```
            arr[i], arr[j] = arr[j], arr[i]
            i, j = i + 1, j - 1
        elif arr[i] >= pivot:
            j -= 1
        elif arr[j] < pivot:
            i += 1
        else:
            i, j = i + 1, j - 1
```

Exercise 2.20
Show that any array of integers $x[1...n]$ can be sorted in $O(n+M)$ time, where

$$M = \max_i x_i - \min_i x_i.$$

Create an array `a` of size M. Use this array to count the number of each x_i you see: `a[0]` corresponds to the smallest possible integer, `a[1]` counts the the second smallest, and so on, until `a[n - 1]` which counts the largest integer. Then, simply fill a sorted array based off the counts, starting from `a[0]`. Note that this sort uses no comparisons, so it is not lower-bounded by $\Omega(n \log n)$.

Exercise 2.24
Write down the code for quicksort.

```
def quicksort(arr):
    if not arr: return []
    pivot, l, e, r = arr[0], [], [], []
    for x in arr:
        if x < pivot: l.append(x)
        elif x = pivot: e.append(x)
        else: r.append(x)
    return quicksort(l) + e + quicksort(r)
```

Show that the worst-case runtime for quicksort is $\Theta(n^2)$.
Suppose the pivot is always the smallest/largest element. Then the array gets reduced by 1 each time. The total runtime is $1+2+3+...+n = \Theta(n^2)$.

Show that the expected runtime of quicksort is $O(n \log n)$.
The expected runtime satisfies the recurrence relation

$$T(n) \leq Bn + \frac{1}{n}\sum_{i=1}^{n-1}(T(i) - T(n-i))$$

$$= Bn + \frac{2}{n}\sum_{i=1}^{n-1} T(i)$$

$$nT(n) \leq Bn^2 + 2\sum_{i=1}^{n-1} T(i)$$

$$(n-1)T(n-1) \leq B(n-1)^2 + 2\sum_{i=1}^{n-2} T(i)$$

Subtract the bottom equation from the top equation.

$$nT(n) - (n-1)T(n-1) \leq 2Bn - B + 2T(n-1)$$

$$nT(n) \leq (n+1)T(n-1) + 2Bn - B$$

$$nT(n) \leq (n+1)T(n-1) + 2Bn - B$$

Solving the recurrence relation gives us that $T(n) = O(n \log n)$.

Exercise 2.29

Show that Horner's rule for polynomial evaluation gives the correct answer.

We use induction. Our loop invariant: at the end of the ith loop $z = a_n x^i + a_{n-1}x^{i-1} + ... + a_{n-i}$.
The base case is clear: the algorithm simply returns a_0 at the end of the 0th loop.
Suppose our loop invariant is true for the $(i-1)$st loop. Consider the i-th loop. By our loop invariant, we know that $z = a_n x^{i-1} + a_{n-1}x^{i-2} + ... + a_{n-i+1}$. Inside the loop, we multiply zx, then add a_i, giving us $z = a_n x^i + a_{n-1}x^{i-1} + ... + a_{n-i+1}x + a_{n-i}$, proving our invariant.
Applying our invariant to when $i = n$, we get that $z = a_n x^n + a_{n-1}x^{n-1} + ... + a_0$ at the end of the last loop, correctly evaluating our polynomial.

How may additions and multiplications does this routine use?

Horner's rule uses about O(n) additions and multiplications (one per loop). This is bad for sparse polynomials, in which most of the coefficients are 0 (for example $2x^{100} + 1$).

4 Chapter 3

Exercise 3.5
Show how to reverse a graph in linear time.

```
def reverse(G):
    reverse = int[v]
    for v in V:
        for w in G[v]:
            reverse[w] = v
    return reverse
```

Exercise 3.6
In an undirected graph, the degree $d(u)$ of a vertex u is the number of neighbors u has, or equivalently, the number of edges incident upon it. In a directed graph, we distinguish between the indegree $d_{in}(u)$, which is the number of edges into u, and the outdegree $d_{out}(u)$, the number of edges leaving u.
(a) Show that in an undirected graph, $\sum_{u \in V} d(u) = 2|E|$.
(b) Use part (a) to show that in an undirected graph, there must be an even number of vertices whose degree is odd.
(c) Does a similar statement hold for the number of vertices with odd indegree in a directed graph?.
(a) In an undirected graph, each edge is counted exactly twice in the summation.
Therefore, $\sum_{u \in V} d(u) = \sum_{(u,v) \in E} 2 = 2|E|$.
(b) Suppose there was an odd number of vertices whose degree is odd. Then the sum of degrees would be odd because we could decompose $\sum_{u \in V} d(u) = \sum_{u_1 \in V} d(u_1) + \sum_{u_2 \in V} d(u_2)$ where u_1 has even degree, and u_2 has odd degree. $\sum_{u_1 \in V} d(u_1) = \sum_{u_1 \in V} 2c_{u_1} = 2k$. $c_{u_1} = d(u_1)/2 \in \mathbb{Z}$. $\sum_{u_1 \in V} d(u_1)$ is even. $\sum_{u_2 \in V} d(u_2) = \sum_{u_2 \in V} 2c_{u_2} + 1 = 2k + \sum_{u_2 \in V} 1 = 2k' + 1$. $c_{u_2} = floor(d(u_2)/2) \in \mathbb{Z}$. $\sum_{u_2 \in V} 1$ is the count of odd vertices which we assumed is odd.
(c) No. Consider a graph with 2 vertices and one directed edge between them. The indegree of the vertices are 0 and 1. There is only 1 vertex with odd indegree.

Exercise 3.12
Either prove or give a counterexample: if $\{u, v\}$ is an edge in an undirected graph, and during depth-first search $post(u) < post(v)$, then v is an ancestor of u in the DFS tree..
Consider 2 cases. Either u is visited first or v is. If u is visited first, then while all the children of u are being processed, v will eventually have to finish before u which does not satisfy $post(u) < post(v)$. If v is visited first, then u would have to be visited and finished before v finishes. We are guaranteed that v would be an ancestor because either a descendent of v touches u or v touches u directly due to the edge.

Exercise 3.14
Show that repeatedly removing source nodes from the graph generates a topological sort in linear time.

```
def toposort(G = (V, E)):
    for v in V:
        in[v] = 0
    for (u, v) in E:
        in[v] += 1
    sources = []
    for v in V:
```

```
            if in[v] = 0:
                sources.add(v)
        while sources:
            u = sources.pop()
            for (v, u) in E:
                in[v] -= 1
                if in[v] = 0:
                    sources.add(v)
```

This algorithm runs in O(V + E) time, since each node is enqueued/dequeue at most once, and each edge is explored once.

Exercise 3.26
Show that an undirected graph has an Eulerian tour if and only if all its vertices have even degree.
If there is an Eulerian tour, then whenever you enter a vertex, you must be able to exit it with another edge. This is true for all vertices (including the start, since you exit it at the beginning and enter it at the end of the tour). Thus, the edges are paired, which means each vertex has even degree.
To prove that even degree $\implies$ Eulerian tour, we use induction on the number of edges. The base case of a single node with no edges is evident; the tour is the node itself.
Suppose on a graph with $k < n$ edges, even degree implies a Eulerian tour exists. Consider a graph with n edges. Pick a random start node, and keep following unused edges until you return to the start. (You must return to the start since the graph is connected, so there is some path $u \to v \to ... \to v$).
Remove all edges associated with this tour T. If it's a Eulerian tour, you're done. Otherwise, you'll be left with some connected components. Since any tour has even incidences along all vertices, the connected components still have even degree. By the inductive hypothesis, each of these connected components has a Eulerian tour.
Construct the Eulerian tour on n edges by following T, and taking the Eulerian tour in these subcomponents when one of them is reached. This shows that if a graph is connected and of even degree, a Eulerian tour exists.

An Eulerian path is a path which uses each edge exactly once. Can you give a similar if-and-only-if characterization of which undirected graphs have Eulerian paths?
There is a Eulerian path iff all vertices except two have even degree.

Can you give an analog of part (a) for directed graphs?
A directed graph has a Eulerian tour iff each vertex has even indegree and outdegree.

Exercise 3.28
2SAT and Graphs

$$(x_1 \vee \bar{x_2}) \wedge (\bar{x_1} \vee \bar{x_3}) \wedge (x_1 \vee x_2) \wedge (\bar{x_3} \vee x_4) \wedge (\bar{x_1} \vee \bar{x_4})$$

Find all satisfying assignments of this 2SAT formula.
`true`, `false`, `false`, `true`
`true`, `true`, `false`, `true`

Give an instance of 2SAT with four variables and no satisfying assignment.

$$(x \vee x) \wedge (\bar{x} \vee \bar{x}) \wedge (y \vee z) \wedge (z \vee w)$$

For each clause in the 2SAT formula, convert $(a \vee b)$ into the edges $(\bar{a}, b)$ and $(\bar{b}, a)$. Show that if the graph representation has an SCC with both x and $\bar{x}$, there is no satisfying assignment.
The graph G_I described above records all implications in I, since $(a \vee b)$ is equivalent to $\bar{a} \implies b$ and $\bar{b} \implies a$. If both x and $\bar{x}$ are in the same SCC, that means there is some series of implications $x \implies ... \implies \bar{x}$ or some series of implications $\bar{x} \implies ... \implies x$. One of these is the implication true $\implies$ false, which is false.

Show the converse of the above statement: if none of G_I's SCCs contain a literal and its negation, the instance I is satisfiable.
Repeatedly pick a sink SCC of G_I. Then, assign `true` to all nodes in that sink, `false` to their negations, and remove all assigned variables.
Since G_I records the implications in I, assigning `true` to all sink variables ensures all the implications are of the form ... $\implies$ true. By the way the graph is defined, there must also be some source SCC containing negations of all the variables in the sink. Since we assign in reverse topological order, there is never an implication true $\implies$ false. Since a variable and its negation are never in the same SCC, a variable is never assigned to `true` and `false` at the same time.
This algorithm runs in linear time (since finding SCCs is linear time).

Exercise 3.30
Show that connectedness is an equivalence relation which partitions vertices into SCCs.
Reflexivity: A vertex is always connected to itself.
Symmetry: By definition, u and v are connected if there is a path u to v and a path v to u. Thus, if u is connected to v, then v must be connected to u.
Transitivity: If u is connected to v and v to w, there must be a path $u \to ... \to v \to ... \to w$. Thus, u is also connected to w. (Similarly for the reverse direction).
Equivalence relations partition a set S into disjoint groups such that any two members of a group are related, and any members not in the same group are not related. Thus, the equivalence relation of connectedness partitions V into disjoint sets, in which each vertex is connected to all other vertices in that set.

5 Chapter 4

Exercise 4.16
Find the parent/child indices of an array-based binary heap.
The parent of node j is at $\lfloor j/2 \rfloor$; the children are at $2j$ and $2j+1$.
Find the parent/child indices of an array-based d-ary heap.
The parent of node j is at $\lfloor (j-1)/d \rfloor$; its children are $dj+1, dj+2, ..., d(j+1)$.
Show that a heap can be constructed in $O(n)$ time.
Use bottom-up heapification: start from the bottom and bubble down each node that is in the incorrect position. This takes $n/2 * 1$ work on the bottom level, $n/4 * 2$ work on the second level, and so on, until $\log n$ work on the top level. Thus, the runtime is

$$\begin{aligned} n/2 + n/4 * 2 + ... + n/n * \log n &= \sum_{i=0}^{\log n} i \frac{n}{2^{i+1}} \\ &= n/2 * \sum_{i=0}^{\log n} \frac{i}{2^i} \\ &= n/2 * \sum_{i=0}^{\log n} \left(i \cdot \left(\frac{1}{2} \right)^i \right) \\ &\leq n/2 * \frac{1/2}{(1-1/2)^2} \\ &= n \in \Theta(n) \end{aligned}$$

How would you adapt `makeheap` to d-ary heaps?
Change the indices of the child/parent.

6 Chapter 5

Exercise 5.16
Consider a distribution over n outcomes with probabilities $p_1, p_2, ..., p_n$ where each outcome occurs mp_i times. Assume that each is a power of 2, and prove that Huffman encoding creates a sequence of length $\sum_{i=1}^{n} mp_i \log \frac{1}{p_i}$.

WLOG, assume that $p_1, ..., p_n$ are ordered in decreasing order. We use induction on the length of the sequence.
Base case: In a case with a single letter, it is clear that $p_1 = 1$ and

$$\sum_{i=1}^{n} mp_i \log \frac{1}{p_i}$$

$$= 1 * \log(1) = 0$$

This holds since a sequence of the exact same letter can be implicitly represented with 0 bits.
Suppose this holds for a distribution of $n-1$ letters. We now consider a distribution on n letters.
Take the lowest-frequency letter with frequency $\frac{1}{2^k}$; there must be another letter with the same frequency (in order to sum to 1). Thus, merge these two to form a "super-letter" with probability $\frac{1}{2^{k-1}}$. You now have $n-1$ letters where the frequencies are powers of two: $p_1, p_2, ..., 2p_{n-1}$; the resulting encoding has length

$$= (2mp_{n-1} \log \frac{1}{2p_{n-1}} + \sum_{i=1}^{n-2} mp_i \log \frac{1}{p_i}) + mpn - 1 + mp_n$$

Note that we must add an extra factor of $mpn - 1 + mp_n$ at the end to account for the fact that we have merged a node, and that the individual letters representing p_{n-1} and p_n will require one extra letter after "unmerging".

$$= (2mp_{n-1}(\log \frac{1}{p_{n-1}} - 1) + \sum_{i=1}^{n-2} mp_i \log \frac{1}{p_i}) + mpn - 1 + mp_n$$

$$= (2mp_{n-1} \log \frac{1}{p_{n-1}} - 2mp_{n-1} + \sum_{i=1}^{n-2} mp_i \log \frac{1}{p_i}) + mpn - 1 + mp_n$$

$$= (mp_{n-1} \log \frac{1}{p_{n-1}} + mp_{n-1} \log \frac{1}{p_{n-1}} - 2mp_{n-1} + \sum_{i=1}^{n-2} mp_i \log \frac{1}{p_i}) + mpn - 1 + mp_n$$

$$= (mp_n \log \frac{1}{p_n} + mp_{n-1} \log \frac{1}{p_{n-1}} + \sum_{i=1}^{n-2} mp_i \log \frac{1}{p_i}) - 2mp_{n-1} + mpn - 1 + mp_n$$

$$= (\sum_{i=1}^{n} mp_i \log \frac{1}{p_i}) - 2mp_{n-1} + mpn - 1 + mp_{n-1}$$

$$= \sum_{i=1}^{n} mp_i \log \frac{1}{p_i}$$

For arbitrary distributions, when is the entropy the largest? When is it the smallest?

Entropy is largest in a uniform random distribution; smallest in a deterministic one ($p_i = 1; p_j = 0, \forall j \neq i$).

Exercise 5.18
Is the entropy of a distribution larger or smaller than the expected number of bits per letter in Huffman encoding?

In an optimal Huffman encoding, where all probabilities are a power of 2, the expected bits per letter is exactly equal to the entropy (see 5.16). However, deviation from powers of 2 increases the length of the Huffman encoding; thus in general the Huffman encoding has more bits per letter than the entropy. In general, entropy is a lower bound on the compressibility of any encoding scheme.

What features of the English language besides letters and their frequencies should a compression scheme take into account?

As a language, English is not just a random collection of bits–there are syntactical and semantic rules. A good compression scheme would take into account common words and letter patterns, as well as larger grammatical rules.

Exercise 5.28
Show that a prefix-free encoding can be represented as a full binary tree.

Place letters at the leaves of the binary tree, and take the left branch to be 0 and the right branch to be 1. Follow the branching until you reach a leaf (letter).
This is a full binary tree–each node has a sibling since there is no reason to have a single extension from any leaf node; the encoding can be shortened by simply removing any "dangling" nodes. Also, only the leaves correspond to elements, since no element is a prefix of another.

Exercise 5.32
Show how to implement the greedy algorithm for Horn formula satisfiability in linear time using a graph.

Convert each literal in the formulas to a node. Add an edge from each literal on the LHS of a clause to each literal on the RHS. Call this graph G, then run the following algorithm:

```
def hornSAT(G):
    indegree = int[|V|]
    for e = (u, v) in E:
        indegree[v] += 1
    Q = {}
    for v in V:
        if indegree[v] == 0:
            Q.enqueue(v)
    while Q not empty:
        x = Q.dequeue()
        x = true
        for (x, y) in E:
            indegree[y] -= 1
            if indegree[y] == 0:
                Q.enqueue(y)
    return assignment if satisfying, else failure
```

Any implication with indegree 0 has its LHS all assigned to true, which means it must also be true.

Exercise 5.33
Show that there is an instance of the set cover problem where the optimal solution uses 2 sets, whereas the greedy uses $\log n$**.**

Consider a diagram with two rows of elements, where the sets are the top and bottom rows, along with sets of increasing size: S_1 = first two elements of row 1 and row 2, S_2 = next 2 elements of row 1 and row 2, S_3 = next 4 elements of row 1 and row 2, ... and so on. Clearly, the optimal set cover has 2 elements, the top and bottom rows.

However, the greedy algorithm will first take the set with $n/2+1$ elements, then the set with $n/4+1$, and then the set with $n/8+1$, and so on until it chooses the set with $2 = 2^0+1$ elements. $n/2+1 = 2^{\log_2 n-1}+1$, so the greedy uses a total of $\log n$ sets.

7 Chapter 6

Exercise 6.1
A contiguous subsequence of a list S is a subsequence made up of consecutive elements of S. For instance, if S is
5, 15, -30, 10, -5, 40, 10,
then 15, -30, 10 is a contiguous subsequence but 5, 15, 40 is not. Give a linear time algorithm for the following task:
Input: A list of numbers $a_1, a_2, \ldots, a_n$.
Output: The contiguous subsequence of maximum sum (a subsequence of length 0 has sum 0).
Subproblem: $f(i)$ = best sum ending at index i
Solution: $f(n)$
Base case: $f(1) = 0$ if $a_1 < 0$ else a_1
Recurrence: $f(i) = f(i-1) + a_i$ if $f(i-1) + a_i > 0$ else if $a_i > 0, a_i$ else 0
Order: increasing order of i
Time complexity: $O(n)$
Space complexity: $O(n)$ (however this could be sped up by only storing the previous value.)

Exercise 6.2
You are going on a long trip. You start on the road at mile post 0. Along the way there are n hotels, at mile posts $a1 < a2 < \cdots < an$, where each ai is measured from the starting point. The only places you are allowed to stop are at these hotels, but you can choose which of the hotels you stop at. You must stop at the final hotel (at distance an), which is your destination. You'd ideally like to travel 200 miles a day, but this may not be possible (depending on the spacing of the hotels). If you travel x miles during a day, the penalty for that day is $(200 - x)^2$. You want to plan your trip so as to minimize the total penalty—that is, the sum, over all travel days, of the daily penalties..
Subproblem: $f(i)$ = best penalty ending at index i
Solution: $f(n)$
Base case: $f(0) = 0$
Recurrence: $f(i) = \max f(i-j) + (200 - (a_i - a_{i-j}))^2$ for $0 < j <= i$
Order: increasing order of i
Time complexity: $O(n^2)$
Space complexity: $O(n)$

Exercise 6.6
Let us define a multiplication operation on three symbols a, b, c according to the following table; thus $ab = b, ba = c$, and so on. Notice that the multiplication operation defined by the table is neither associative nor commutative.

	a	b	c
a	b	b	a
b	c	b	a
c	a	c	c

Find an efficient algorithm that examines a string of these symbols, say bbbbac, and decides whether or not it is possible to parenthesize the string in such a way that the value of the resulting expression is a. For example, on input bbbbac your algorithm should return yes because $((b(bb))(ba))c = a$ Subproblem: $f(i, j, c)$ = whether it is possible to parenthesize substring [i:j] so that character c can be represented.
Solution: $f(1, n, ' c')$ where n is the length of the string.

Base case: $f(i, i, s[i]) =$ true for all i where i is the index of the string, s.
Recurrence: $f(i, j, c) = V_{k=i}^{j-1}(f(i, k, c_1)\text{and}f(k+1, k, c_2))$ where $c_1 c_2 = c$

Exercise 6.17
Given an umlimited supply of coins of denominations $x_1, x_2, ...x_n$, give a dynamic programming algorithm to decide if it is possible to make change for some quantity V.
Subproblem: $f(i, v) =$ if it is possible to make change for v using denominations $x_1, x_2, ...x_i$
Solution: $f(n, V)$
Base case: $f(0, 0) =$ true; $f(0, v) =$ false, $v \neq 0$
Recurrence: $f(i, v) = f(i-1, v)$ or $f(i, v - x_i)$
Order: increasing order of i, v
Time complexity: $O(nV)$
Space complexity: $O(V)$

Exercise 6.18
Solve the coin change problem from above where you are only allowed to use one of each denomination..
Subproblem: $f(i, v) =$ if it is possible to make change for v using denominations $x_1, x_2, ...x_i$
Solution: $f(n, V)$
Base case: $f(0, 0) =$ true; $f(0, v) =$ false, $v \neq 0$
Recurrence: $f(i, v) = f(i-1, v)$ or $f(i-1, v - x_i)$
Order: increasing order of i, v
Time complexity: $O(nV)$
Space complexity: $O(V)$

Exercise 6.19
Solve the coin change problem from above where you have an unlimited supply of each coin but can only use up to k total coins..
Subproblem: $f(v) =$ miniminum number of coins to make change for v
Solution: $f(V) \leq k$
Base case: $f(0) = 0$ Recurrence: $f(v) = \min_i f(v - x_i)$
Order: increasing order of v
Time complexity: $O(nV)$
Space complexity: $O(V)$

Exercise 6.21
A vertex cover of a graph $G = (V, E)$ is a subset of vertices $S \subseteq V$ that includes at least one endpoint of every edge in E. Give a linear-time algorithm for the following task.
Input: An undirected tree $T = (V, E)$.
Output: The size of the smallest vertex cover of T.
Subproblem: $f(v, 1) =$ smallest vertex cover of subtree rooted at v where v is used. $f(v, 0) =$ smallest vertex cover of subtree rooted at v where v is not used
Solution: $\min(f(r, 1), f(r, 0))$ where r is the root vertex.
Base case: $f(v, 0) = 0, f(v, 1) = 1$ where v is a leaf vertex.
Recurrence: $f(v, 0) = \sum_{i=1}^{n} f(v_i, 1)$, $f(v, 1) = 1 + \sum_{i=1}^{n} f(v_i, 0)$ where v_i are children of v.
Order: bottom up (post order traversal)
Time complexity: $O(V)$ (each vertex will be part of the sum at most 4 times. Itself for both cases and as a child to the parent.)
Space complexity: $O(V)$

Exercise 6.25
Consider the following 3-PARTITION problem. Given integers $a_1, a_2, \ldots, a_n$, we want to determine whether it is possible to partition of $\{1, \ldots, n\}$ into three disjoint subsets I, J, K such that $\sum_{i \in I} a_i = \sum_{j \in J} a_j = \sum_{k \in K} a_k = \frac{1}{3} \sum_{i=1}^{n} a_i$
Devise and analyze a dynamic programming algorithm for 3-PARTITION that runs in time polynomial in n and $\sum_i a_i$
Subproblem: $f(i, s_1, s_2, s_3)$ = a boolean whether it is possible to partition into 3 subsets of size s_1, s_2, s_3 with elements with indices in $\{1, \ldots, i\}$
Solution: $f(n, \frac{1}{3} \sum_{i=1}^{n} a_i, \frac{1}{3} \sum_{i=1}^{n} a_i, \frac{1}{3} \sum_{i=1}^{n} a_i)$
Base case: $f(0, j, k, l) = 0$ for all $0 \leq j, k, l, \leq \frac{1}{3} \sum_{i=1}^{n} a_i$
Recurrence: $f(i, s_1, s_2, s_3) =$ true if $f(i-1, s_1 - a_i, s_2, s_3) \mid f(i-1, s_1, s_2 - a_i, s_3) \mid f(i-1, s_1, s_2, s_3 - a_i)$
Time complexity: $O(n(\frac{1}{3} \sum_{i=1}^{n} a_i)^3)$
Space complexity: $O((\frac{1}{3} \sum_{i=1}^{n} a_i)^3)$ (we only need to keep track of the previous ith index)

Exercise 6.26
Give a dynamic programming that takes in two strings $x[1...n], y[1...m]$ and returns the max-scoring alignment.
Subproblem: $f(i, j) =$ max alignment from $x[1...i], y[1...j]$
Solution: $f(n, m)$
Base case: $f(0, j) = 0$; $f(i, 0) = 0$
Recurrence: $f(i, j) = \max(f(i-1, j) + \delta(x[i], \_\_)), f(i, j-1) + \delta(\_\_, y[j])), f(i-1, j-1) + \delta(x[i], y[j]))$
Time complexity: $O(mn)$
Space complexity: $O(\min(m, n))$

Exercise 6.27
Solve the problem from 6.26 where the penalty for a gap of length k is $c_0 + c_1 k$.
Subproblems:
$A(i,j)$ = best score when $x[i]$ and $y[j]$ aligned
$B(i,j)$ = best score when $x[i]$ is aligned with a gap
$C(i,j)$ = best score when $y[j]$ is aligned with a gap
Solution: $\max(A(n,m), B(n,m), C(n,m))$
Base case: $A(0,0) = 0; B(i,0) = c_0 + c_1 * i; C(0,j) = c_0 + c_1 * j$
Recurrences:
$A(i,j) = \max(A(i-1,j-1) + \delta(x[i],y[j]), B(i-1,j-1) + \delta(x[i],y[j]), C(i-1,j-1) + \delta(x[i],y[j])$
$B(i,j) = \max(A(i-1,j) + c_0 + c_1, B(i-1,j) + c_1)$
$C(i,j) = \max(A(i-1,j) + c_0 + c_1, C(i-1,j) + c_1)$
Time complexity: $O(mn)$
Space complexity: $O(\min(m,n))$

Exercise
Solve the problem from 6.26 where your goal is to find a maximum-scoring substring. Subproblem: $f(i,j)$ = score of best alignment of a suffix of $x[1...i]$, $y[1...j]$
Solution: $\max_{i,j} f(i,j)$
Base case: $f(0,j) = 0$; $f(i,0) = 0$
Recurrence: $f(i,j) = \max(f(i-1,j) + \delta(x[i],\_\_)), f(i,j-1) + \delta(\_\_,y[j])), f(i-1,j-1) + \delta(x[i],y[j]))$
Time complexity: $O(mn)$
Space complexity: $O(\min(m,n))$

8 Chapter 7

Exercise 7.18

7.18. There are many common variations of the maximum flow problem. Here are four of them. (a) There are many sources and many sinks, and we wish to maximize the total flow from all sources to all sinks. (b) Each vertex also has a capacity on the maximum flow that can enter it. (c) Each edge has not only a capacity, but also a lower bound on the flow it must carry. (d) The outgoing flow from each node u is not the same as the incoming flow, but is smaller by a factor of $(1 - \epsilon_u)$, where ϵ_u is a loss coefficient associated with node u. Each of these can be solved efficiently. Show this by reducing (a) and (b) to the original max-flow problem, and reducing (c) and (d) to linear programming.

(a) Add two nodes, s', t'. For all sources, connect an infinite edge between s' and each of the sources and an infinite edge between t' and all of the sinks.

(b) Each inner vertex should be copied. The edges going into the original corresponding vertex should go into the copy instead. Then, the copy will have an outgoing edge of the vertex capacity that goes into the original vertex.

(c) In the linear program, we have the constraint that $f_{uv} \leq c_{uv}$. We should add another constraint that $f_{uv} \geq l_{uv}$ where l_{uv} is the lower bound. To switch the direction of the inequality, we can multiply be -1.

(d) Instead of having the constraint $\sum_{(w,u)\in E} f_{wu} = \sum_{(u,z)\in E} f_{uz}$, we should have $(1-\epsilon_u)\sum_{(w,u)\in E} f_{wu} = \sum_{(u,z)\in E} f_{uz}$

Exercise 7.19

Suppose someone presents you with a solution to a max-flow problem on some network. Give a linear time algorithm to determine whether the solution does indeed give a maximum flow.

For every edge in the solution, we can create the residual graph in linear time by updating all the edges in the graph. Then, attempt to augment the path by 1 unit. If there is a path from the source to the sink, then the solution is not maximal, otherwise it is maximal.

Exercise 7.22

In a particular network $G = (V, E)$ whose edges have integer capacities c_e, we have already found the maximum flow f from node s to node t. However, we now find out that one of the capacity values we used was wrong: for edge (u, v) we used c_{uv} whereas it should have been $c_{uv} - 1$. This is unfortunate because the flow f uses that particular edge at full capacity: $f_{uv} = c_{uv}$. We could redo the flow computation from scratch, but there's a faster way. Show how a new optimal flow can be computed in $O(|V| + |E|)$ time.

Now, we must find a path that went through (u, v) and then decrement it by 1. To do this, we can create a graph where the only edges available are the ones with positive flow. We can then run bfs to find a path from s to t which goes through (u, v). Decrease the flow by 1 on all these edges for the solution. However, this might not be the right result. The flow is 1 less than the solution found, however, there could be multiple min-cuts. Therefore, we can attempt to send 1 more unit of flow to arrive at the answer.

Exercise 7.25

Write the max flow problem as a linear program.

We represent the flow along a s-t path as a variable x_p.

$$\max \sum_p x_p$$

$$x_p \geq 0; \forall e$$

$$\sum_{p:e\in p} x_p \leq c_e; \forall e$$

Find the dual of the LP max flow problem.

$$\min \sum_e c_e y_e$$

$$\sum_{e\in p} y_e \geq 1$$

$$y_e \geq 0$$

Interpret the dual LP.
The dual LP can be seen as finding a min cut on the graph. An edge is included in the cut iff $y_e = 1$. We minimize over the sum of included edges, and ensure that across any s-t path, at least one edge on this path is chosen.

Exercise 7.28
Show that finding a shortest s-t path is equivalent to finding a flow of size 1 minimizing $\sum_e l_e f_e$ with no capacity constraints.
We can interpret the flow found as follows: if an edge has 1 unit of flow, include it in the shortest path; otherwise do not. Clearly, this minimizes the sum of edges along any s-t path, and finds the shortest path.

Write the shortest path problem as an LP.

$$\min \sum_e l_e f_e$$

$$\sum_{e=(u,v)} f_e - \sum_{e=(v,w)} f_e \geq 0, \forall v \neq s, t$$

$$\sum_{e=(v,t)} f_e \geq 1$$

Find the dual of this LP.

$$\max x_t$$

$$x_v - x_u \leq f_{e=(u,v)}$$

$$x_v \geq 0$$

Exercise 7.31
Show how to find the fattest s-t path using a variant of Dijkstra's algorithm.

```
widest = int[v]
queue = []
while queue:
    v = queue.dequeue()
    for w in G[v]:
        if widest[w] < min(cap(v, w), widest[v]):
            widest[w] = min(cap(v, w), widest[v])
```

Show that the maximum flow in G is a sum of individual flows along at most $|E|$ paths from s to t.
Take the edge $e = (u, v)$ with minimum flow f_e. If it is part of a cycle, remove f_e units of flow from each edge in the cycle. If it is part of a path, remove f_e units of flow from each edge in the path. This maintains conservation of flow: the net flow into u is decreased by f_e and so it the net outflow; similarly for v. Remove this edge from the graph and repeat–after at most $|E|$ iterations, all edges will be gone and the graph will be decomposed into the paths/cycles found during this process.

Show that increasing the flow along the fattest path allows Ford-Fulkerson to terminate in at most $O(|E| \log F)$ iterations.
Suppose the max flow has size $|F|$. Let F_k be the amount of flow left to reach the max flow, in the kth iteration. Since the max flow can be decomposed into at most $|E|$ paths, the fattest path must have flow greater than or equal to $|F|/|E|$. Pushing flow along this path results in $F_{k+1} = (1 - 1/|E|)F_k$. Using induction, we can see that F_k reaches 0 when $k = E \log F$, meaning we terminate after that many iterations.

Exercise 8.9
In the HITTING SET problem, we are given a family of sets $\{S_1, S_2, \ldots, S_n\}$ and a budget b, and we wish to find a set H of size $\leq b$ which intersects every S_i, if such an H exists. In other words, we want $H \cap S_i \neq \emptyset$ for all i.
Show that HITTING SET is NP-complete.
Hitting Set can varified in polynomial time as taking the intersection will take at most $O(n^2)$.
Finding a vertex cover actually reduces to the hitting set. Suppose we are given some graph $G = (V, E)$. Each set given would correspond to the two vertices around each edge. Additionally, the budget of the vertex cover g would be the budget of the hitting set, b.

If there is a solution to that vertex cover, then the hitting set would be the set of vertices that make up the cover. Because the vertex cover was solved, each set would need to have at least one of its elements in the hitting set otherwise, the edge corresponding to that set would not be covered.

If there is a solution to the hitting set problem, then that solution will be the vertex cover. every edge would have one of its neighboring vertices in the hitting set.

Exercise 8.14
Prove that the following problem in NP-complete: given an undirected graph $G = (V, E)$ and an integer k, return a clique of size k as well as an independent set of size k, provided both exist.

This problem is verifiable in polynomial time because we could check that each vertex in the clique has connections all the other vertices in $O(n^2)$ time. We can also check that the independent set is independent by looking at each pair of vertex and making sure they are not connected in $O(n^2)$ time. There are a couple options that can be reduced to this, but we will show that finding a clique of size g in a graph can be reduced to this problem. To transform the graph $G = (V, E)$ for the clique problem into the graph for this problem, we will make a copy of the graph. make a copy of the vertices of the graph without any edges between them then union that with the vertices of the original graph. This will be our new graph G‘.
Suppose that we have found a solution to our problem. Then this will also output a clique which will solve finding the clique of size k. The independent set of size k will be guaranteed because the added vertices can always be part of that independent set. Every vertex will be part of the original graph because none of the added vertices can be part of the clique since they are not connected to anything else.

Suppose that we have found the clique for the original graph. Then this will be the same clique for our problem because the new graph contains all the vertices and edges of our original graph. The independent set of size k will be in the new graph because of the added vertices.

Exercise 8.17
Show that for any problem Π in NP, there is an algorithm which solves Π in time $O\left(2^{p(n)}\right)$, where n is the size of the input instance and $p(n)$ is a polynomial (which may depend on Π).

Any problem can in NP can be reduced to an instance of SAT. However there must be $p(n)$ variables and clauses in this version of SAT otherwise the SAT might not be verifiable in polynomial time. It would take polynomial time to check every instance, and $O\left(2^{p(n)}\right)$ instances of assignments. Therefore, we would have time $O(n^k 2^{n^l})$ for some $k, l \geq 0$. However, $n^k 2^{n^l} \in O(2^{n^{l+k}}) \in O(2^{p(n)})$

Exercise 8.18
Show that if P = NP then the RSA cryptosystem (Section 1.4.2) can be broken in polynomial time.

In the cryptosystem, we are given $N, e, y = x^e mod N$ and want to figure out x. One way of doing so is by factoring N which we can finish in polynomial time is **P = NP**. We would now need to invert e with respect to mod $(p-1)(q-1)$. We want to find numbers a, b such that $ae + b(p-1)(q-1) = 1$. We can use the euclidean algorithm to do so.

Exercise 8.20
In an undirected graph $G = (V, E)$, we say $D \subseteq V$ is a dominating set if every $v \in V$ is either in D or adjacent to at least one member of D. In the DOMINATING SET problem, the input is a graph and a budget b, and the aim is to find a dominating set in the graph of size at most b, if one exists. Prove that this problem is NP-complete.

We can actually solve vertex cover with DOMINATING SET. To transform the vertex cover, we could add a new vertex to correspoond to each edge uv. Then we would connect that vertex to both u and v.
Suppose we found the dominating set for this. If a vertex is chosen to be in a uv vertex, we may mark either vertex u or v for the vertex cover, causing the vertex cover to have at most b chosen vertices. Every edge is covered. Suppose an edge uv is not covered. Then in the dominating set neither vertex u, v, or the added vertex would be chosen. However, the added vertex would not be adjacent to or in the dominating set, a contradiction.
Now, suppose we have a solution for the vertex cover, we could just choose the corresponding vertices for the dominating set. Then, the added vertex has to be adjacent since either one of the 2 vertices it is next to has to be marked. Additionally, if there are any original vertices not part of or the dominating set, then the adjacent vertices in the original graph must be in the dominating set. Otherwise, the edge between the two original vertices not in the dominating set will not be covered.