Chapter 0– Solutions

January 30, 2007

0.1. a) $n - 100 = \Theta(n - 200)$
b) $n^{1/2} = O(n^{2/3})$
c) $100n + \log n = \Theta(n + (\log n)^2)$
d) $n \log n = \Theta(10n \log 10n)$
e) $\log 2n = \Theta(\log 3n)$
f) $10 \log n = \Theta(\log(n^2))$
g) $n^{1.01} = \Omega(\log^2 n)$
h) $n^2/\log n = \Omega(n(\log n)^2)$
i) $n^{0.1} = \Omega((\log n)^{10})$
j) $(\log n)^{\log n} = \Omega(n/\log n)$
k) $\sqrt{n} = \Omega((\log n)^3)$
l) $n^{1/2} = O(5^{\log_2 n})$
m) $n2^n = O(3^n)$
n) $2^n = \Theta(2^{n+1})$
o) $n! = \Omega(2^n)$
p) $(\log n)^{\log n} = O(2^{(\log_2 n)^2})$
q) $\sum_{i=1}^{n} i^k = \Theta(n^{k+1})$

0.2. By the formula for the sum of a partial geometric series, for $c \neq 1$: $g(n) = \frac{1-c^{n+1}}{1-c} = \frac{c^{n+1}-1}{c-1}$.

a) $1 > 1 - c^{n+1} > 1 - c$. So: $\frac{1}{1-c} > g(n) > 1$.
b) For $c = 1$, $g(n) = 1 + 1 + \cdots + 1 = n + 1$.
c) $c^{n+1} > c^{n+1} - 1 > c^n$. So: $\frac{c}{1-c}c^n > g(n) > \frac{1}{1-c}c^n$.

0.3. a) Base case: $F_6 = 8 \geq 2^{6/2} = 8$.
Inductive Step: for $n \geq 6$, $F_{n+1} = F_n + F_{n-1} \geq 2^{n/2} + 2^{(n-1)/2} = 2^{(n-1)/2}(2^{1/2}+1) \geq 2^{(n-1)/2}2 \geq 2^{(n+1)/2}$.

b-c) The argument above holds as long as we have, in the inductive step, $2^{c(n-1)}(2^c + 1) \geq 2^{c(n+1)}$, i.e. as long as $2^c \leq \frac{1+\sqrt{5}}{2}$.

0.4. a) For any 2×2 matrices X and Y:

$$XY = \begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{pmatrix} \begin{pmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{pmatrix} = \begin{pmatrix} x_{11}y_{11} + x_{12}y_{21} & x_{11}y_{12} + x_{12}y_{22} \\ x_{21}y_{11} + x_{22}y_{21} & x_{21}y_{12} + x_{22}y_{22} \end{pmatrix}$$

This shows that every entry of XY is the addition of two products of the entries of the original matrices. Hence every entry can be computed in 2 multiplications and one addition. The whole matrix can be calculated in 4 multiplications and 4 additions.

b) First, consider the case where $n = 2^k$ for some positive integer k. To compute, X^{2^k}, we can recursively compute $Y = X^{2^{k-1}}$ and then square Y to have $Y^2 = X^{2^k}$ Unfolding the recursion, this can be seen as repeatedly squaring X to obtain $X^2, X^4, \cdots, X^{2^k} = X^n$. At every squaring, we are doubling the exponent of X, so that it must take $k = \log n$ matrix multiplications to produce X^n. This method can be easily generalized to numbers that are not powers of 2, using the following recursion:

$$X^n = \begin{cases} (X^{\lfloor n/2 \rfloor})^2 & \text{if } n \text{ is even} \\ X \cdot (X^{\lfloor n/2 \rfloor})^2 & \text{if } n \text{ is odd} \end{cases}$$

The algorithm still requires $O(\log n)$ matrix multiplications, of which $\log n$ are squares and at most $\log n$ are multiplications by X.

c) The entries of the matrix are the addition of the products of the entries of the original matrix. Addition leaves the number of bits close to unvaried (add at most one bit), while multiplication adds the number of bits of the operands. Hence, every time we square the matrix X, we double the number of bits of its entries. By b), we are squaring X $\log n$ times and hence all intermediate results must be of length less or equal to $2^{\log n} = O(n)$. The multiplications by X leave the sizes of the intermediate result almost unchanged (add at most one bit) and can be neglected, as well as the additions of the products of the entries within the matrix multiplication.

In the next two questions we assume that $M(n) = n^c$ for some $c \geq 1$.

d) The algorithm performs $O(\log n)$ matrix multiplications; each matrix multiplication consists of a constant number of arithmetic operations by a). Each arithmetic operation is on results of size $O(n)$ and hence takes at most time $O(M(n))$. This implies the algorithm runs in time $O(M(n) \log n)$.

e) Let $T(n)$ be the running time of the algorithm on input of size n. In the first level of the recursion, we first run the algorithm for inputs of size $n/2$, which takes time $T(n/2)$. Then, we square the results to obtain the final answer: this last operation takes time $M(n/2)$ as the results of the recursive call have bit size $O(n/2)$ by c). This implies:

$$T(n) = T(n/2) + M(n/2)$$

Expanding the recursion and applying the formula for geometric series, we obtain:

$$\begin{aligned} T(n) &= M(n/2) + M(n/4) + M(n/8) + \cdots + M(1) \leq \\ &\leq \frac{n^c}{2^c} + \frac{n^c}{4^c} + \cdots + \frac{n^c}{8^c} = \\ &= n^c \sum_{i=1}^{\infty} \frac{1}{2^{ic}} = O(n^c) = O(M(n)) \end{aligned}$$

Chapter 1–Solutions

February 2, 2007

1.1. A single digit number is at most $b-1$, therefore the sum of any three such numbers is at most $3b-3$. On the other hand, a two-digit number can be as large as b^2-1. It is enough to show that $b^2-1 \geq 3b-3$. Indeed, $b^2-1-3b-3 = (b-1)\cdot(b-2)$, which is ≥ 0 for $b \geq 2$.

1.2. For a number N we need $\lceil \log_2(N+1) \rceil$ binary digits and $\lceil \log_{10}(N+1) \rceil$ decimal digits. By the logarithm conversion formula: $\lceil \log_{10}(N+1) \rceil = \lceil \log_2(N+1) \cdot \log_2(10) \rceil \leq \lceil \log_2(N+1) \rceil \cdot \lceil \log_2(10) \rceil = 4 \cdot \lceil \log_2(N+1) \rceil$. For very large numbers, $\lceil \log_2(N+1) \cdot \log_2(10) \rceil \simeq \log_2(N+1) \cdot \log_2(10)$ so the required ratio is approximately equal to $\log_2(10) \simeq 3.8$.

1.3. The minimum depth of a d-ary tree is achieved when all nodes have precisely d children. In that case, the depth is $\log_d(n)$. For any d-ary tree, we have depth $D \leq \log_d(n) = \frac{\log(n)}{\log(d)} = \Omega(\frac{\log(n)}{\log(d)})$

1.4. We can lower bound $n!$ as

$$\underbrace{\left(\frac{n}{2}\right) \cdot \cdots \cdot \left(\frac{n}{2}\right)}_{\frac{n}{2} \text{ terms}} \leq 1 \cdot 2 \cdot 3 \cdot \cdots \cdot \left(\frac{n}{2}\right) \cdot \underbrace{\left(\frac{n}{2}+1\right) \cdot \cdots \cdot n}_{\frac{n}{2} \text{ terms}}$$

and upper bound it as

$$\underbrace{1 \cdot 2 \cdot 3 \cdot \cdots \cdot n}_{n \text{ terms}} \leq \underbrace{n \cdot \cdots \cdot n}_{n \text{ terms}}$$

Hence,

$$\begin{aligned} \left(\frac{n}{2}\right)^{\frac{n}{2}} &\leq n! \leq n^n, \\ \frac{n}{2}\log(\frac{n}{2}) &\leq \log(n!) \leq n \log n, \\ \frac{1}{2}(n \log n - n) &\leq \log(n!) \leq n \log n. \end{aligned}$$

1.5. Upper bound:

$$\begin{aligned} \sum_{i=1}^{n} \frac{1}{i} =& 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots \leq \\ \leq& 1 + \frac{1}{2} + \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \cdots + \underbrace{\frac{1}{2^k} + \cdots + \frac{1}{2^k}}_{2^k \text{ terms}} = \\ =& \underbrace{1 + 1 + \cdots + 1 + 1}_{O(\log n) \text{ terms}} = \\ =& O(\log n) \end{aligned}$$

Lower bound:

$$\begin{aligned}\sum_{i=1}^{n}\frac{1}{i} =& 1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\cdots \geq \\ \geq& 1+\frac{1}{2}+\frac{1}{4}+\frac{1}{4}+\frac{1}{8}+\frac{1}{8}+\frac{1}{8}+\frac{1}{8}\cdots+\underbrace{\frac{1}{2^k}+\cdots+\frac{1}{2^k}}_{2^{k-1}\text{ terms}} = \\ =& 1+\underbrace{\frac{1}{2}+\frac{1}{2}+\cdots+\frac{1}{2}+\frac{1}{2}}_{\Omega(\log n)\text{ terms}} = \\ =& \Omega(\log n)\end{aligned}$$

1.6. First observe that multiplication of a number N by a one-digit binary number b results in the number 0 if $b = 0$ and N if $b = 1$. Also, multiplication of a binary number by 2^k for any power of 2 results in shifting N to the left k times and adding k digits equal to 0 at the end.
Let N, M the numbers we need to multiply. Assume M is a k-digit binary number. Write $M = b_0 + 2b_1 + \cdots + 2^k b_k$. Then $N \cdot M = N \cdot (b_0 + 2b_1 + \cdots + 2^k b_k) = Nb_0 + 2Nb_1 + \cdots + 2^k Nb_k$. If we consider the above observations, this expression is the same as the one illustrated in page 24 for the grade-school multiplication.

1.7. Assume we want to multiply the n-bit number x with the m-bit number y. The algorithm must terminate after m recursive calls, because at each call y is halved (the number of digits is decreased by one). Each recursive call requires a division by 2, a check for even-odd, a multiplication by 2 (all of those take constant time) and a possible addition of x to the current result (which takes $O(n)$ time) so total $O(m \cdot n)$ time.

1.8. We will first prove the correctness of the algorithm by induction on the number x. Base case is when $x = 0$ where correctness is obvious. Assume the algorithm returns correct values of q', r' so that $\lfloor \frac{x}{2} \rfloor = q'y + r'$. If x is even then $x = 2 \cdot \lfloor \frac{x}{2} \rfloor = 2q' + 2r'$. If $2r' \geq y$ then $x = (2q' + 1)y + (2r' - y)$, otherwise $x = 2q'y + 2r'$, which is exactly what the last call of the algorithm returns. If x is odd, then $x = 2 \cdot \lfloor \frac{x}{2} \rfloor + 1 = 2q' + 2r' + 1$. Again, according to $2r' + 1 \geq y$ or not it is easy to see that the algorithm returns the correct values of q and r.
Now, considering the running time : there are n recursive calls (each time x is halved, so we have one bit less). Each call requires two multiplications by 2, a check of even or odd, a possible addition of 1 (all of those take constant time) and possibly a subtraction of y which takes $O(n)$ time. The total running time is $O(n^2)$.

1.9. $x \equiv x' \mod N$ implies that N divides $x - x'$, similarly N divides $y - y'$. Consider the difference $D_1 = x + y - x' - y' = (x - x') + (y - y')$. We easily conclude that N divides D_1 therefore $x + y \equiv x' + y' \mod N$.
Similarly, define $D_2 = xy - x'y' = xy - xy' + xy' - x'y' = x(y - y') + (x - x')y$. We can easily see that N divides D_2 so $xy \equiv x'y' \mod N$.

1.10. $a \equiv b \mod N$ means N divides $a - b$. Since M divides N, then M also divides $a - b$ which leads to the conclusion $a \equiv b \mod M$.

1.11. We first observe that $35 = 5 \cdot 7$ and by Fermat's Little Theorem and simple algebra, $a^{(5-1)\cdot(7-1)} = a^{4 \cdot 6} = a^2 4 \equiv 1 \mod 35$, for all $1 \leq a < 35$. Therefore, $4^{1536} = 4^{24 \cdot 64} \equiv 1 \mod 35$ and $9^{4824} = 9^{24 \cdot 201} \equiv 1 \mod 35$. We conclude that $4^{1536} \equiv 9^{4824} \mod 35$ so the difference is divisible by 35.

1.12. $2^{2^{2006}} = 2^{2 \cdot 2^{2005}} = 4^{2^{2005}} \equiv 1 \mod 3$.

1.13. Since 31 is a prime number, by Fermat's Little Theorem we get $a^{30} \equiv 1 \mod 31$ for all $1 \leq a < 31$. Therefore $5^{30000} \equiv 1 \mod 31$. On the other hand, $6^{123456} = 6^{123450+6} = 6^6 = 5^3 = 125 \mod 31 \equiv 1 \mod 31$. So the given difference is a multiple of 31.

1.14. We will assume that the running time of multiplying n-bit numbers is $M(n)$. According to problem 0.4 in Chapter 0, fib3 involves $O(\log n)$ arithmetic operations (multiplications). Since we are working in arithmetic $\mod p$, each intermediate result will be $\log p$ bits long, therefore the total running time of fib3 is $O(\log n \cdot M(\log p))$.

1.15. $ax \equiv bx \mod c \Rightarrow (a-b)x \equiv 0 \mod c$. If $\gcd(x, c) = 1$ then $a \equiv b \mod c$ for all a, b. Conversely, if $ax \equiv bx \Rightarrow a \equiv b \mod c$ for all a, b then assume $c = nm$. Take a, b such that $a - b \equiv n \mod c \neq 0 \mod c$. For x a multiple of m we can still get $ax \equiv bx$ without having the second assertion. Therefore, x cannot have common factors with c.

1.16. Let $b = 15$. The algorithm of repeated squaring will calculate a, a^2, a^4, a^8 and then calculate $a^{15} = a \cdot a^2 \cdot a^4 \cdot a^8$, with a total of 6 multiplications.
Now, consider the following algorithm: first compute $a^3 = a \cdot a \cdot a$, then compute $a^6 = a^3 \cdot a^3$ and finally $a^{12} = a^6 \cdot a^6$. Calculate $a^{15} = a^{12} \cdot a^3$, with a total of 5 multiplications.

1.17. The iterative algorithm multiplies each time x into the result of the previous iteration. It performs $y-1$ iterations in total. In each iteration i, the size of the intermediate result m_i is at most $m_{i-1} + n$ bits long. Therefore the total running time is $O(n^2+2n^2+3n^2+\cdots+(y-1)n^2) = O(n^2 \cdot \frac{y(y-1)}{2}) = O(n^2 \cdot y^2)$. The recursive algorithm has $O(\log y)$ iterations and during the i-th iteration, multiplies two 2^{i-1} - bit long numbers. The total running time is $O(n^2+2^2n^2+2^4n^2+\cdots+2^{\lfloor \log y \rfloor - 1}n^2) = O(n^2 \cdot \frac{(2^2)^{\lfloor \log y \rfloor}-1}{3}) = O(n^2 \cdot y^2)$. We conclude that the two algorithms have running times of the same order.

1.18. First, we compute the gcd by finding the factorization of each number: $210 = 2 \cdot 3 \cdot 5 \cdot 7$ and $588 = 2^2 \cdot 3 \cdot 7^2$ therefore $\gcd(210, 588) = 2 \cdot 3 \cdot 7 = 42$.
Next, using Euclid's algorithm : $\gcd(210, 588) = \gcd(210, 168) = \gcd(168, 42) = 42$.

1.19. We can show this by induction on n. For $n = 1$, $\gcd(F_1, F_2) = \gcd(1, 1) = 1$. Now say that the inductive hypothesis is true for all $n \leq k$. This implies that for $n = k + 1$

$$\gcd(F_{k+1}, F_{k+2}) = \gcd(F_{k+1}, F_{k+2} - F_{k+1}) = \gcd(F_{k+1}, F_k) = 1$$

Hence, the statement is true for all $n \geq 1$.

1.20. (i) $(20)^{-1} = 4 \mod 79$
(ii) $(3)^{-1} = 21 \mod 62$
(iii) $\gcd(21, 91) = 7$. Hence, no inverse exists!
(iv) $(5)^{-1} = 14 \mod 23$

1.21. A number x has an inverse modulo 11^3 if and only if $\gcd(x, 11^3) = 1$. For $\gcd(x, 11^3) \neq 1$, either $x = 0$ or shares a common factor with 11^3. But the only factors of 11^3 are $11, 121$ and 1331. Thus, a nonzero number does not have an inverse if and only if it is a multiple of 11. The number of multiples of 11 between 1 and 1330 is $1331/11 - 1 = 120$ (because we only consider numbers upto 1330). Thus, the number of numbers that have an inverse is $1331 - 120 - 1 = 1210$.

1.22. Since a has an inverse $\mod b$ it means that a, b are coprime so b also has an inverse $\mod a$.

1.23. Suppose x_1 and x_2 are two distinct inverses of $a \mod N$. Then,

$$x_1 = x_1 \cdot 1 = x_1 \cdot ax_2 = 1 \cdot x_2 = x_2 (\mod N)$$

which is a contradiction.

1.24. Since p is a prime, all elements in the given range that are not a multiple of p have inverse $\mod p^n$. Among the numbers $\{0, \cdots, p^n - 1\}$ only p^{n-1} are multiples of p. Therefore, $p^n - p^{n-1} = p^{n-1}(p-1)$ numbers have inverses.

1.25. By Fermat's Little Theorem, $2^{126} \equiv 1 \mod 127$. We can write $2^{126} = 2^{125} \cdot 2$. So 2^{125} is the inverse of $2 \mod 127$. By observation (or by extended Euclid's algorithm) we deduce that $2 \cdot 64 = 128 \equiv 1 \mod 127$, so $2^{125} \equiv 64 \mod 127$ since the inverse is unique.

1.26. Take $p = 2$, $q = 5$ so that $pq = 10$. First observe that $17^{(2-1)(5-1)} = 17^4 = 1 \mod 10$. Hence,

$$17^{17^{17}} \mod 10 = 17^{(17^{17} \mod 4)} \mod 10$$

Also, $17 = 1 \mod 4$. So, $17^{17} \mod 4 = 1^{17} \mod 4 = 1$ This gives

$$17^{17^{17}} \mod 10 = 17 \mod 10 = 7 \mod 10$$

so the least significant decimal digit is 7.

1.27. First calculate $(p-1)(q-1) = 16 \cdot 22 = 352$. We use the extended Euclid algorithm to compute the $\gcd(3, 352)$ and get the inverse d of $e \mod 352$. We easily obtain $e \cdot d \equiv 1 \mod 352 \Rightarrow d \equiv -117 \equiv 235 \mod 352$.
The encryption of the message $M = 41$ is $E(M) = M^e \mod N = 41^3 = 117 \cdot 41 = 105 \mod 391$.

1.28. We first calculate $(p-1)(q-1)$ which in our case is $6 \cdot 10 = 60$. Then we need to come up with an e which is relatively prime to 60 so that it has an inverse d. We observe that $e = 11$ has $\gcd(11, 60) = 1$ and $11 \cdot 11 = 1 \mod 60$, therefore the values $e = 11$ and $d = 11$ are appropriate. Other good pairs are $(7, 43), (13, 37), (17, 53), (19, 59), (23, 47), (29, 29), (31, 31), (41, 41))$.

1.29. (a) Here H is the same as in the example of pg.46 of the book, only with 2 coefficients instead of 4. With the same reasoning as the proof of the Property in pg.46, we assume that $X_2 \neq y_2$ and we want to determine the probability that equation $a_1(x_1 - y_1) = a_2(y_2 - x_2)$ holds. Assuming we already picked a_1, that probability is equal to $1/m$, since the only way for the equation to hold is to pick a_2 to be $(y_2 - x_2)^{-1} \cdot a_1 \cdot (x_1 - y_1) \mod m$. We can see that, since m is prime, $(y_2 - x_2)^{-1}$ is unique. Thus H is universal. We need $2 \cdot \lceil \log m \rceil$ bits.
(b)H is not universal, since according to (a) we need a unique inverse of $(y_2 - x_2) \mod m$. For this to hold m has to be prime, which is not true (unless $k = 1$). We need $2k$ bits.
(c)We calculate $P = \Pr[f(x) = f(y)]$ for $x \neq y$. We have $P = \sum_{i=1}^{m-1} \frac{1}{(m-1)^2} = \frac{1}{m-1}$. Thus H is universal. The total number of functions $f : [m] \to [m-1]$ is $(m-1)^m$ so we need $m \log(m-1)$ bits.

1.30. (a)We assume for simplicity that $n = 2^k$ for some fixed k. We perform addition as follows: first split the numbers into 2^{k-1} distinct pairs and perform addition within each pair, using lookahead circuits. Continue by splitting the result into distinct pairs each time and perform addition within each pair. We construct this way a complete binary tree with $\log n$ levels, each level using $\log m$ extra levels for addition (if n was not a power of 2 the tree would simply not be complete). Total we have $O((\log n)(\log m))$ depth.
(b) We let the i-th bit of r be the result of the addition of the i-th bits of all three numbers (i.e. $0+0+0 = 0, 0+1+0 = 1, \cdots, 1+1+0 = 0, \cdots, 1+1+1 = 1$). We also let the $i+1$ bit of s to be the carry of the addition of the i-th bit of the three numbers (i.e. if two or three of the i-th bits of the three numbers are 1, then the carry is 1). It is straightforward that $x + y + z = r + s$.
(c) In multiplication, we have to add together n copies of x appropriately shifted. We add these copies by splitting the numbers into triplets and performing the trick from (b). We repeat for k levels until there are less than 3 numbers left, i.e. when $n \cdot (\frac{2}{3})^k = 1 \Rightarrow k = \frac{\log n}{\log 3 - \log 2} = O(\log n)$.

1.31. (a) By the equation $\log N! = \log 1 + \log 2 + \cdots + \log N$ we can easily see that $N!$ is approximately $\Theta(N \cdot \log N) = \Theta(N \cdot n)$ bits long.
(b)We can compute $N!$ naively as follows:

```
factorial (N)
    f = 1
    for i = 2 to N
        f = f · i
```

Running time : we have N iterations, each one multiplying two $N \cdot n$-bit numbers (at most). By 1.30 the running time is $O(N \cdot \log(N \cdot n) = O(N \cdot n)$.

1.32. (a) Assume that N consists of n bits. We can then perform a binary search on the interval $[2^n - 1, 1]$ to find if it contains a number q such that $q^2 = N$. Every iteration takes time $O(n^2)$ to square the current element and $O(n)$ to compare the result with N. As there are $O(\log 2^n)$ iterations, the total running time is $O(n^3)$.

(b) $N = q^k \Rightarrow \log N = k \log q \Rightarrow k = \frac{\log N}{\log q} \leq \log N$ for all $q > 1$. If $q = 1$, then we must have $N = 1$.

(c)We first give an algorithm to determine if a n-bit number N is of the form q^k for some given k and $q > 1$. For this , we use the same algorithm of part (a) only instead of squaring, we will raise numbers to the k-th power and check if we obtain N. This will take $O(n)$ iterations: moreover, each powering operation takes time at most $\sum_{i=1}^{k}(in \cdot n) = O(k^2 n^2)$. Hence, one run of this algorithm takes time $O(k^2 n^3)$. To check if N is a power, we need to repeat this for all $k \leq \log N \leq n$. This yields a running time of $O(n^6)$.

1.33. The least common multiple (lcm) of any two numbers x, y can easily be seen to equal $\text{lcm}(x, y) = (x \cdot y)/\gcd(x, y)$. We therefore need $O(n^3)$ operations to compute the gcd, $O(n^2)$ operations to multiply x and y and $O(2n \cdot n) = O(n^2)$ operations to divide. Total $O(n^3)$ running time.

1.34. **Solution 1 :**
We calculate the expected (average) value of the number of times we must toss a coin before it comes up heads. Let X be the random variable corresponding to the number of tosses needed before coming up heads.

$$E[X] = \sum_{i=1}^{\infty} i \cdot P[X = i] =$$

We now calculate $P[X = i]$: For a coin to come up heads (for the first time) in exactly i tosses, we must have $i - 1$ tails followed by one head. It follows that $P[X = i] = (1 - p)^{(i-1)}p$, for $i \geq 1$. So,

$$E[X] = \sum_{i=1}^{\infty} i \cdot (1 - p)^{(i-1)}p = p \cdot \frac{d}{dp}(\sum_{i=0}^{\infty} -(1 - p)^i)$$

The above powerseries $\sum_{i=0}^{\infty} -(1 - p)^i$ converges for $0 < p < 1$ and it is equal to $-\frac{1}{1-(1-p)} = -\frac{1}{p}$. After taking the derivative, we obtain

$$E[X] = p \cdot \frac{1}{p^2} = \frac{1}{p}$$

Solution 2 :
Let X be the random variable as in solution 1. Then $E[X]$ the average number of tosses, with each possibility weighted by its probability. With probability probability p we get heads in one toss of the coin and with probability $1 - p$, we get tails and need to start again and do another $E[X]$ on average (hence we do a total of $E[X] + 1$ tosses with probability $1 - p$). Therefore $E = p \cdot 1 + (1 - p) \cdot (1 + E) = 1 + (1 - p)E$.

1.35. (a) For a number $1 \leq n < p$ to be its own inverse modulo p it is necessary to have $n^2 - 1 = k \cdot p$, a multiple of p. This comes from the gcd formula : $\gcd(n, p) = 1 = n^n - k \cdot p$. Equivalently, $n^2 - 1 = (n-1)(n+1) \equiv 0 \mod p$. Solving for n we get $n = +1, p-1$. We observe that for all those values n is indeed its own inverse.

(b) Among the $p-1$ numbers, 1 and $p-1$ are their own inverses and the rest have a (different than themselves) unique inverse $\mod p$. Since inversion is a bijective function, each number a in $\{2, \cdots, p-2\}$ is the inverse of some number in the same range not equal to a. Thus,$(p-2)(p-3)\cdots 2 \equiv 1 \mod p \Rightarrow (p-1)! \equiv (p-1) \equiv -1 \mod p$.

(c) Assume, towards contradiction that N is not a prime and also $(N-1)! \equiv -1 \mod N$. Then $(N-1)! = -1 + kN \Rightarrow 1 = -(N-1)! + kN$ which implies that $\gcd((N-1)!, N) = 1$. This is false if N not a prime since there exists some prime $q < N$ that is a multiple of N which appears in the factorial product above.

(d) This rule involves calculating a factorial product which takes time exponential on the size of the input. Thus, the algorithm would not be efficient.

1.36. (a) $p \equiv 3 \mod 4$ implies that $p = 4k + 3 \Rightarrow p + 1 = 4(k+1) \Rightarrow \frac{(p+1)}{4} = k + 1$, an integer.

(b) By Fermat's Little Theorem, $a^{p-1} = 1 \mod p$, so $(a^{\frac{p-1}{2}} - 1)(a^{\frac{p-1}{2}} + 1) = 0 \mod p$. Suppose $a^{\frac{p-1}{2}} + 1 = 0 \mod p$ and x is the square root of a; then we have $x^{p-1} = -1 \mod p$, contradicting Fermat's Little Theorem. Hence, we must have $a^{\frac{p-1}{2}} - 1 = 0 \mod p$, which means $a^{\frac{p+1}{2}} = a \mod p$. This shows that $\left(a^{\frac{p+1}{4}}\right)^2 = a \mod p$. By a), $\frac{p+1}{4}$ is an integer, so that $a^{\frac{p+1}{4}}$ is well defined.

1.37. (a) Make the table and observe that each number has a different pair (i, j) of residues $\mod 3$ and $\mod 5$.

(b) Assume , towards contradiction that both i and i' with $i \neq i'$ have the same (j, k). Then $i = Ap + j = Bq + k$ and $i' = Cp + j = Dq + k$. So $i - i' = (A - C)p = (B - D)q$. Since p, q are different primes, q has to divide $(A - C)$ and p has to divide $(B - D)$. So $i - i' = 0 \mod pq$. Which means $i = i'$ since both are $< pq$.

Assume now, towards contradiction, that there is a particular pair (j, k) such that there is no integer $i < pq$ with the desired property. The total number of pairs is pq (which is the same as the number of numbers mod pq) so there must be at least two different numbers i, i' that have the same pair (j', k'), contradiction.

(c) . By (b) it is enough to show that the expression in brackets has the property $i = j \mod p$ and $i = k \mod q$.

$$\begin{aligned} i = & \quad \{j \cdot q \cdot (q^{-1} \mod p) + k \cdot p \cdot (p^{-1} \mod q)\} \mod pq \mod p \\ = & \quad \{j \cdot q \cdot (q^{-1} \mod p) + k \cdot p \cdot (p^{-1} \mod q)\} \mod p \mod pq = j \end{aligned}$$

and

$$\begin{aligned} i = & \quad \{j \cdot q \cdot (q^{-1} \mod p) + k \cdot p \cdot (p^{-1} \mod q)\} \mod pq \mod q \\ = & \quad \{j \cdot q \cdot (q^{-1} \mod p) + k \cdot p \cdot (p^{-1} \mod q)\} \mod q \mod pq = k \end{aligned}$$

(d) Part (b) is immediately extended to more than two primes. Part (c) will give the expression for i :

$$\begin{aligned} i = \{j_1 \cdot q_2 \cdot q_3 \cdot \cdots \cdot q_n((q_2 \cdot q_3 \cdot \cdots \cdot q_n)^{-1} \mod q_1) + \\ + \cdots + \\ + j_n \cdot q_1 \cdot q_2 \cdot \cdots \cdot q_{n-1}((q_1 \cdot q_2 \cdot \cdots \cdot q_{n-1})^{-1} \mod q_n) \mod p_1 p_2 \cdots p_n \end{aligned}$$

1.38. We will show (a) and (b) together. For (b) observe that is we break a decimal number N into r-tuples, say, $N_k N_{k-1} \cdots N_1$ then $N = N_1 + N_2 \cdot 10^r + \cdots + N_k \cdot (10^r)^{k-1}$. Therefore, for our divisibility criterion, we try to achieve the following :$N \mod p = N_1 + N_2 \cdot 10^r + \cdots + N_k \cdot (10^r)^{k-1} = N_1 + N_2 + \cdots + N_k$. For this equation to hold, we need $10^r = 1 \mod p$. From Fermat's Little Theorem, we know that $10^{p-1} = 1 \mod p$, so looking for the smallest such r we will get a divisor of $p-1$. For $p = 13$ we can check that the choice $r = 13 - 1 = 12$ is the smallest choice for r, while for $p = 17$ the smallest choice is $r = 17 - 1 = 16$.

1.39. From Fermat's Little Theorem, we know that $a^{p-1} = 1 \mod p$. Therefore,

$$a^{b^c} = a^{b^c \mod (p-1)} \mod p$$

we first need to calculate $b^c \mod (p-1)$. Note that this is modular exponentiation and is hence doable in polynomial time. We repeatedly square b modulo $(p-1)$ and compute $b^c \mod (p-1)$ in the same way as computing the n^{th} power of a matrix. Since we do $O(n)$ multiplications, each taking $O(n^2)$ time, the total time is $O(n^3)$. Let $b^c \mod (p-1) = d$. Then, by the equality $a^{b^c} \mod p = a^d \mod p$, all we need to do is use again the repeated squaring algorithm. So the total running time of our algorithm is $O(n^3)$.

1.40. Assume, towards contradiction that N is a prime and still x is a non-trivial square root of $1 \mod N$. Then $x^2 = 1 + kN \Rightarrow (x-1)(x+1) = kN$. Since N is a prime it must divide at least one of $(x-1)$,$(x+1)$ which leads to conclude $x \equiv 1$ or $-1 \mod N$, contradiction.

1.41. (a) Assume $a \equiv y^2 \mod N$ for $1 \leq y < N$. Then $x^2 - y^2 \equiv 0 \mod N \Rightarrow (x-y)(x+y) \equiv 0 \mod N$. This means either $x - y \equiv 0 \mod N \Rightarrow x = y$ since both $x, y < N$ or $x + y \equiv 0 \mod N \Rightarrow x = N - y$ since $x, y < N$. These are the only two possible values.
(b)We will first show that for any two numbers x, y in the range $\{1, 2, \cdots, \frac{N-1}{2}\}$ we have $x^2 \neq y^2 \mod N$. Assume towards contradiction that $x^2 \equiv y^2 \Rightarrow (x-y)(x+y) \equiv 0$. Since N is prime, $x \neq y$ this means $x + y \equiv N$. But both $x, y \leq \frac{N-1}{2}$ so $x + y \leq N - 1$, contradiction.
Therefore, each of those $\frac{N-1}{2}$ numbers defines a quadratic residue $\mod N$. Also, from (a), the rest of the $\frac{N-1}{2}$ numbers in the range $\{\frac{N-1}{2} + 1, \cdots N-1\}$ lead to the same residues. Adding 0 as a quadratic residue, we have exactly $\frac{N+1}{2}$ of them.
(c)Take, say, $N = 15$ and $a = 1$ then the equation $x^2 \equiv 1 \mod 15$ has solutions $1, -1, 4$.

1.42. We just need to calculate the inverse of $e \mod (p-1)$. But this we can do by using Extended Euclid's algorithm for the $\gcd(e, p-1) = 1$.

1.43. First pick an appropriate message m such that $m^{\frac{ed-1}{2}}$ is neither 1 nor $-1 \mod N$. Then $m^{ed-1} \equiv 1 \mod N \Rightarrow (m^{\frac{ed-1}{2}} - 1)(m^{\frac{ed-1}{2}} + 1) \equiv 0 \mod N \Rightarrow$ one of the $\gcd(N, (m^{\frac{ed-1}{2}} - 1))$ or $\gcd(N, (m^{\frac{ed-1}{2}} + 1))$ is a non-trivial factor of N. Since $N = qp$ and we have determined, say p, we can just divide $N/p = q$.

1.44. The three messages that Alice sends are M^3
$mod N_1$, $M^3 \mod N_2$, $M^3 \mod N_3$, where $M < min\{N_1, N_2, N_3\}$. By the Chinese remainder theorem there is a unique number $x < N_1 \cdot N_2 \cdot N_3$ with $x = M^3 \mod N_1$,$x = M^3 \mod N_2$,$x = M^3 \mod N_3$. We observe that $M^3 < N_1 \cdot N_2 \cdot N_3$ and also has the above residues. Therefore, we can just compute M^3 by the equations in 1.37d) and then take the cubic root i $\mathbb{Z}$ (recall we are not able to take roots in Z_N, but this is not necessary here).

1.45. a) The digital signature scheme can be used to both authenticate the identity of the sender of the message and to ensure that the message has not been altered by a third party during communication.

b) `verify`$((N, e), M^d, M)$ can be implemented by checking whether $(M^d)^e \mod N$ equals M. Then, if the signature was created by the private key d, we have $(M^d)^e \mod N = (M^e)^d \mod N = M$

mod $N = M$, by the correctness of the RSA protocol. Conversely, if an adversary was able to sign given only (N, e), the adversary would be able to exponentiate by $d \mod N$, which would allow him to decrypt, contradicting the security of the RSA protocol. Hence, if the RSA is secure, so is this scheme for digital signatures.

c) I picked $p = 137, q = 71$. Hence $N = pq = 9727$ and $\Phi(N) = (p-1)(q-1) = 9520 = 2^4 \cdot 5 \cdot 7 \cdot 17$. Then, I chose $e = 99$, which is coprime to $\Phi(n)$. d must then be the inverse of $e \mod \Phi(n)$, that is 6539, found by running Extended Euclid. The first letter of my name is L and I choose it to map to binary using the $ASCII$ code, for which $L = 76$. Then, the signature of this first number is $76^{99} \mod 9727 = 4814$. Finally, $4814^{99} \mod 9727 = 76$, as required.

d) 391 can be factored as $17 \cdot 23$. Then, $\Phi(391) = 16 \cdot 22 = 352$. Then $d = (e)^{-1} \mod 352 = 145$, by Extended Euclid. In fact, $145 * 17 = 2465 = 1 \mod 352$, as required.

1.46. a) When Eve intercepts the encrypted message $M^e \mod N$ sent by Alice to Bob, she can just ask Bob to sign it for her with his private key, to obtain $(M^e)^d \mod N = M$, by the correctness of the RSA.

b) In this case, Eve can pick k coprime to N at random and ask Bob to sign $M^e \cdot k^e \mod N$. This will yield $(Mk)^{ed} \mod N = Mk \mod N$. Then Eve can use Extended Euclid to obtain $k^{-1} \mod N$ and multiply by such inverse to find M. Notice that in this way Bob's signatures are distributed uniformly over all numbers invertible $\mod N$, as Mk is equally likely to be any of such numbers.

Chapter 2– Solutions

January 31, 2007

2.2. Consider $b^{\lceil \log_b n \rceil}$.

2.3. a)

$$\begin{aligned} T(n) &= 3T\left(\frac{n}{2}\right) + cn = \cdots = 3^k T\left(\frac{n}{2^k}\right) + cn \sum_{i=0}^{k-1} \left(\frac{3}{2}\right)^i = \\ &= 3^k T\left(\frac{n}{2^k}\right) + 2cn\left(\left(\frac{3}{2}\right)^k - 1\right) \end{aligned}$$

For $k = \log_2 n$, $T(\frac{n}{2^k}) = T(1) = d = O(1)$. Then:

$$T(n) = dn^{log_2 3} + 2cn\left(\frac{n^{log_2 3}}{n} - \right) = \Theta(n^{log_2 3})$$

as predicted by the Master theorem.

b) $T(n) = T(n-1) + c = \cdots = T(n-k) + kc$. For $k = n$, $T(n) = T(0) + nc = \Theta(n)$.

2.4. a) This is a case of the Master theorem with $a = 5, b = 2, d = 1$. As $a > b^d$, the running time is $O(n^{\log_b a}) = O(n^{\log_2 5}) = O(n^{2.33})$.

b) $T(n) = 2T(n-1) + C$, for some constant C. $T(n)$ can then be expanded to $C\sum_{i=0}^{n-1} 2^i + 2^n T(0) = O(2^n)$.

c) This is a case of the Master theorem with $a = 9, b = 3, d = 2$. As $a = b^d$, the running time is $O(n^d \log n) = O(n^2 \log n)$.

2.5. a) $T(n) = 2T(n/3) + 1 = \Theta(n^{\log_3 2})$ by the Master theorem.

b) $T(n) = 5T(n/4) + n = \Theta(n^{\log_4 5})$ by the Master theorem.

c) $T(n) = 7T(n/7) + n = \Theta(n \log_7 n)$ by the Master theorem.

d) $T(n) = 9T(n/3) + n^2 = \Theta(n^2 \log_3 n)$ by the Master theorem.

e) $T(n) = 8T(n/2) + n^3 = \Theta(n^3 \log_2 n)$ by the Master theorem.

f) $T(n) = 49T(n/25) + n^{3/2} \log n = \Theta(n^{3/2} \log n)$. Apply the same reasoning of the proof of the Master Theorem. The contribution of level i of the recursion is

$$\left(\frac{49}{25^{3/2}}\right)^i n^{3/2} \log\left(\frac{n}{25^{3/2}}\right) = \left(\frac{49}{125}\right)^i O(n^{3/2} \log n)$$

Because the corresponding geometric series is dominated by the contribution of the first level, we obtain $T(n) = O(n^{3/2} \log n)$. But, $T(n)$ is clearly $\Omega(n^{3/2} \log n)$. Hence, $T(n) = \Theta(n^{3/2} \log n)$.

g) $T(n) = T(n-1) + 2 = \Theta(n)$.

h) $T(n) = T(n-1) + n^c = \sum_{i=0}^{n} i^c + T(0) = \Theta(n^{c+1})$.

i) $T(n) = T(n-1) + c^n = \sum_{i=0}^{n} c^i + T(0) = \frac{c^{n+1}-1}{c-1} + T(0) = \Theta(c^n)$.

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

k) $T(n) = T(\sqrt{n}) + 1 = \sum_{i=0}^{k} 1 + T(b)$, where $k \in Z$ such that $n^{\frac{1}{2^k}}$ is a small constant b, i.e. the size of the base case. This implies $k = \Theta(\log \log n)$ and $T(n) = \Theta(\log \log n)$.

2.6. The corresponding polynomial is $\frac{1}{t_0}\left(1 + x + x^2 + \cdots + x^{t_0}\right)$.

2.7. For $n \neq 0$ and $\omega = e^{\frac{2\pi}{n}}$:

$$\sum_{i=0}^{n-1} \omega^i = \frac{1-\omega^n}{1-\omega} = 0$$

$$\prod_{i=0}^{n-1} \omega^i = \omega^{\sum_{i=0}^{n-1} i} = \omega^{\frac{n(n-1)}{2}}$$

The latter is 1 if n is odd and -1 if n is even.

2.8. a) The appropriate value of ω is i. We have $\mathrm{FFT}(1,0,0,0) = (1,1,1,1)$ and $\mathrm{FFT}(1/4,1/4,1/4,1/4) = (1,0,0,0)$.

b) $\mathrm{FFT}(1,0,1,-1) = (1,i,3,-i)$.

2.9. a) Use $\omega = i$. The FFT of $x+1$ is $\mathrm{FFT}(1,1,0,0) = (2, i+1, 0, i-1)$. The FFT of x^2+1 is $\mathrm{FFT}(1,0,1,0) = (2,0,2,0)$. Hence, the FFT of their product is $(4,0,0,0)$, corresponding to the polynomial $1+x+x^2+x^3$.

b) Use $\omega = i$. The FFT of $2x^2+x+1$ is $\mathrm{FFT}(1,1,2,0) = (4, -1+i, 2, -1-i)$. The FFT of $3x+2$ is $\mathrm{FFT}(2,3,0,0) = (5, 2+3i, -1, 2-3i)$. The FFT of their product is then $(20, -5-i, -2, -5+i)$. This corresponds to the polynomial $6x^3+7x^2+5x+2$.

2.10. $\left(-20, \frac{137}{3}, -\frac{125}{4}, \frac{25}{3}, -\frac{3}{4}\right)$.

2.11. For $i, j \leq \frac{n}{2}$:

$$(XY)_{ij} \quad = \quad \sum_{k=1}^{n} X_{ik} Y_{kj} = \sum_{k=1}^{n/2} A_{ik} E_{kj} + \sum_{k=1}^{n/2} B_{ik} G_{ki} = (AE+BG)_{ij}$$

A similar proof holds for the remaining sectors of the product matrix.

2.12. Each function call prints one line and calls the same function on input of half the size, so the number of printed lines is $P(n) = 2P(\frac{n}{2}) + 1$. By the Master theorem $P(n) = \Theta(n)$.

2.13. a) $B_3 = 1$, $B_5 = 2$, $B_7 = 5$. Any full binary tree must have an odd number of vertices, as it has an even number of vertices that are children of some other vertex and a single root. Hence $B_{2k} = 0$ for all k.

b) By decomposing the tree into the subtrees rooted at the children of the root:

$$B_{n+2} = \sum_{i=1}^{n+1} B_i B_{n+1-i}$$

c) We show that $B_n \geq 2^{\frac{n-3}{2}}$. Base case: $B_1 = 1 \geq 2^{-1}$ and $B_3 = 1 \geq 2^0$. Inductive step: for $n \geq 3$ odd:

$$B_{n+2} = \sum_{i=1}^{n+1} B_i B_{n+1-i} \geq \frac{n+2}{2} 2^{\frac{n-5}{2}} \geq 2^{2+\frac{n-5}{2}} = 2^{\frac{n-1}{2}}$$

.

2.14. Sort the array in time $O(n \log n)$. Then, in one linear time pass copy the elements to a new array, eliminating the duplicates.

2.15. The simplest way to implement `split` in place is the following:

```
function split(a[1, ··· , n], v)
store= 1
for i = 1 to n:
    if a[i] < v:
        swap a[i] and a[store]
        store = store +1
for i =store to n:
    if a[i] = v:
        swap a[i] and a[store]
        store = store +1
```

The first for loop passes through the array bringing the elements smaller than v to the front, so splitting the array into a subarray of elements smaller than v and one of elements larger or equal to v. The second for loop uses the same strategy on the latter subarray to split into a subarray of elements equal to v and one of elements larger than v. The body of both for loops takes constant time, so the running time is $O(n)$.

2.16. It is sufficient to show how to find n in time $O(\log n)$, as we can use binary search on the array $A[1, \cdots, n]$ to find any element x in time $O(\log n)$. To find n, query elements $A[1], A[2], \cdots, A[2^i], \cdots$, until you find the first element $A[2^k]$ such that $A[2^k] = \infty$. Then, $2^{k-1} \leq n < 2^k$. We can then do binary search on $A[2^{k-1}, \cdots, , 2^k]$ to find the last non-infinite element of A. This takes time $O(\log(2^k - 2^{k-1}) = O(\log n)$.

2.17. First examine the middle element $A[\frac{n}{2}]$. If $A[\frac{n}{2}] = \frac{n}{2}$, we are done. If $A[\frac{n}{2}] > \frac{n}{2}$, then every subsequent element will also be bigger than its index since the array values grow at least as fast as the indices. Similarly, if $A[\frac{n}{2}] < \frac{n}{2}$, then every previous element in the array will be less than its index by the same reasoning. So after the comparison, we only need to examine half of the array. We can recurse on the appropriate half of the array. If we continue this division until we get down to a single element and this element does not have the desired property, then such an element does not exist in A. We do a constant amount of work with each function call. So our recurrence relation is $T(n) = T(n/2) + O(1)$, which solves to $T(n) = O(\log n)$.

2.18. As in the $\Omega(n \log n)$ lower bound for sorting on page 52 we can look at a comparison-based algorithm for search in a sorted array as a binary tree in which a path from the root to a leaf represents a run of the algorithm: at every node a comparison takes place and, according to its result, a new comparison is performed. A leaf of the tree represents an output of the algorithm, i.e. the index of the element x that we are searching or a special value indicating the element x does not appear in the array. Now, all possible indices must appear as leaves or the algorithm will fail when x is at one of the missing indices. Hence, the tree must have at least n leaves, implying its depth must be $\Omega(\log n)$, i.e. in the worst case it must perform at least $\Omega(\log n)$ comparisons.

2.19. a) Let $T(i)$ be the time to merge arrays 1 to i. This consists of the time taken to merge arrays 1 to $i-1$ and the time taken to merge the resulting array of size $(i-1)n$ with array i, i.e. $O(in)$. Hence, for some constant c, $T(i) \leq T(i-1) + cni$. This implies $T(k) \leq T_1 + cn \sum_{i=2}^{k} i = O(nk^2)$.

b) Divide the arrays into two sets, each of $k/2$ arrays. Recursively merge the arrays within the two sets and finally merge the resulting two sorted arrays into the output array. The base case of the recursion is $k = 1$, when no merging needs to take place. The running time is given by $T(k) = 2T(k/2) + O(nk)$. By the Master theorem, $T(k) = O(nk \log k)$.

2.20. We keep $M + 1$ counters, one for each of the possible values of the array elements. We can use these counters to compute the number of elements of each value by a single $O(n)$-time pass through the

array. Then, we can obtain a sorted version of x by filling a new array with the prescribed numbers of elements of each value, looping through the values in ascending order. Notice that the $\Omega(n \log n)$ bound does not apply in this case, as this algorithm is not comparison-based.

2.21. a) Suppose $\mu > \mu_1$. If we move μ to the left by an infinitesimal amount ϵ, the distance to all $x_i < \mu$ decreases by ϵ, whilst the distance to all points $x_i > \mu$ increases by ϵ. Because $\mu > \mu_1$, there are more points to the left of μ than to the right, so that the the shift by ϵ causes the sum of distances to decrease. The same reasoning can be applied to the case $\mu < \mu_1$.

b) We use the second method:

$$\begin{aligned} \sum_i (x_i - \mu_2)^2 + n(\mu - \mu_2)^2 = \\ = \sum_i x_i^2 - 2n\mu_2^2 + n\mu_2^2 + n\mu^2 - 2n\mu\mu_2 + n\mu_2^2 = \\ = \sum_i x_i^2 - 2(\sum_i x_i)\mu + n\mu^2 = \\ = \sum_i (x_i - \mu)^2 \end{aligned}$$

Then, $\sum_i (x_i - \mu)^2$ will be minimized when the second addend of the left hand side is minimized, i.e. for $\mu = \mu_2$.

c) The maximizing x_i in $\max_i |x_i - \mu|$ will be the maximum $x_{\max}$ or the minimum $x_{\min}$ of the observations. Then μ must minimize $\max\{x_{\max} - \mu, \mu - x_{\min}\}$, so $\mu = \frac{x_{\min} + x_{\max}}{2}$. This value can be computed in $O(n)$ time by passing through the observations to identify the minimum and maximum elements and taking their average in constant time.

2.22. Suppose we are searching for the kth smallest element s_k in the union of the lists $a[1, \cdots, m]$ and $b[1, \cdots, n]$. Because we are searching for the kth smallest element, we can restrict our attention to the arrays $a[1, \cdots, k]$ and $b[1, \cdots, k]$. If $k > m$ or $k > n$, we can take all the elements with index larger than the array boundary to have infinite value. Our algorithm starts off by comparing elements $a[\lfloor k/2 \rfloor]$ and $b[\lceil k/2 \rceil]$. Suppose $a[\lfloor k/2 \rfloor] > b[\lceil k/2 \rceil]$. Then, in the union of a and b there can be at most $k - 2$ elements smaller than $b[\lceil k/2 \rceil]$, i.e. $a[1, \cdots, \lfloor k/2 \rfloor - 1]$ and $b[1, \cdots, \lceil k/2 \rceil - 1]$, and we must necessarily have $s_k > b[\lceil k/2 \rceil]$. Similarly, all elements $a[1, \cdots, \lfloor k/2 \rfloor]$ and $b[1, \cdots, \lceil k/2 \rceil]$ will be smaller than $a[\lfloor k/2 \rfloor + 1]$; but these are k elements, so we must have $s_k < a[\lfloor k/2 \rfloor + 1]$. This shows that s_k must be contained in the union of the subarrays $a[1, \cdots, \lfloor k/2 \rfloor]$ and $b[\lceil k/2 \rceil + 1, k]$. In particular, because we discarded $\lceil k/2 \rceil$ elements smaller than s_k, s_k will be the $\lfloor k/2 \rfloor$th smallest element in this union. We can then find s_k by recursing on this smaller problem. The case for $a[\lfloor k/2 \rfloor] < b[\lceil k/2 \rceil]$ is symmetric. The last case, which is also the base case of the recursion, is $a[\lfloor k/2 \rfloor] = b[\lceil k/2 \rceil]$, for which we have $s_k = a[\lfloor k/2 \rfloor] = b[\lceil k/2 \rceil]$.

At every step we halve the number of elements we consider, so the algorithm will terminate in $\log(2k)$ recursive calls. Assuming the comparison takes constant time, the algorithm runs in time $O(\log k)$, which is $O(\log(m + n))$, as we must have $k \leq m + n$ for the kth smallest element to exist.

2.23. a) If A has a majority element v, v must also be a majority element of A_1 or A_2 or both. To find v, recursively compute the majority elements, if any, of A_1 and A_2 and check whether one of these is a majority element of A. The running time is given by $T(n) = 2T(\frac{n}{2}) + O(n) = O(n \log n)$.

b) After this procedure, there are at most $n/2$ elements left as at least one element in each pair is discarded. If these remaining elements have a majority, there exists v among them appearing at least $n/4$ times. Hence, v must have been paired up with itself in at least $n/4$ pairs during the procedure, showing that A contains at least $n/2$ copies of v. The running time of this algorithm is described by the recursion $T(n) = T(n/2) + O(n)$. Hence, $T(n) = O(n)$.

2.24. a) Modify the in place split procedure of 2.15 so that it explicitly returns the three subarrays S_L,S_R,S_v. Quicksort can then be implemented as follows:

```
function quicksort(A[1, ··· , n])
    pick k at random among 1, ··· , n
    (S_L, S_R, S_v =split(A[1, ··· , n], A[k])
    quicksort(S_L)
    quicksort(S_R)
```

b) In the worst case we always pick $A[k]$ that is the largest element of A. Then, we only decrease the problem size by 1 and the running time becomes $T(n) = T(n-1) + O(n)$, which implies $T(n) = O(n^2)$.

c) For $1 \leq i \leq n$, let p_i be the probability that $A[k]$ is the ith largest element in A and let t_i be the expected running time of the algorithm in this case. Then, the expected running time can be expressed as $T(n) = \sum_{i=1}^{n} p_i t_i$. $A[k]$ is every element of A with the same probability $\frac{1}{n}$, so $p_i = \frac{1}{n}$. Moreover, t_i is at most $O(n) + T(n-i+1) + T(i-1)$, as S_L has at most $n-i+1$ elements and S_R has at most $i-1$. Then, for some constant c:

$$T(n) \leq \frac{1}{n}\left(\sum_{i=1}^{n} O(n) + T(i-1) + T(n-i+1)\right) \leq cn + \frac{1}{n}\sum_{j=0}^{n-1} T(j) + T(n-j)$$

We use induction to show that the recurrence is satisfied by $T(n) \leq bn\log n$ for some choice of constant b. We start by rewriting $T(n)$ as:

$$T(n) \leq cn + \frac{2}{n}\sum_{j=0}^{\frac{n}{2}} T(j) + T(n-j)$$

Then, we substitute the expression from the inductive hypothesis:

$$T(n) \leq cn + \frac{2b}{n}\sum_{j=1}^{\frac{n}{2}} j\log j + (n-j)\log(n-j)$$

Next, we divide the sum as follows:

$$T(n) \leq cn + \frac{2b}{n}\sum_{j=1}^{\frac{n}{4}-1} j\log j + (n-j)\log(n-j) + \frac{2b}{n}\sum_{j=\frac{n}{4}}^{\frac{n}{2}} j\log j + (n-j)\log(n-j)$$

Finally, we can bound each term in the first sum above by $n\log n$ and each term in the second sum by $n\log(\frac{3}{4}n)$:

$$T(n) \leq cn + \frac{b}{2}n\log n + \frac{b}{2}n\log n - \frac{b}{2}n\log\left(\frac{4}{3}\right) = bn\log n + n\left(c - \frac{b}{2}\log\left(\frac{4}{3}\right)\right)$$

This is smaller than $bn\log n$ for $b > \frac{2c}{\log(\frac{4}{3})}$, completing the proof by induction.

2.25 a) The algorithm should be:

```
function pwr2bin(n)
if n = 1: return 1010_2
else:
    z =pwr2bin(n/2)
```

`return fastmultiply`(z, z)

The running time $T(n)$ can then be written as

$$T(n) = T(n/2) + O(n^a)$$

By the Master theorem, $T(n) = O(n^a)$.

b) The algorithm is the following:

```
dec2bin(x)
if n = 1: return binary[x]
else:
    split x into two decimal numbers x_L, x_R with n/2 digits each
    return {fastmultiply(dec2bin(x_L),pwr2bin(n/2)) + dec2bin(x_R)}
```

The running time $T(n)$ is expressed by the recurrence relation

$$T(n) = 2T(n/2) + O(n^a)$$

as both the `fastmultiply` and the `pwr2bin` operations take time $O(n^a)$. By the Master theorem, $T(n) = O(n^a)$, as $a = \log_2 3 > 1$.

2.26 We show how to use an algorithm for squaring integers to multiply integers asymptotically as fast. Let $S(n)$ be the time required to square a n-bit number. We can multiply n-bit integers a and b by first computing $2ab = (a+b)^2 - a^2 - b^2$ and then shifting the results $2ab$ by one bit to the right to obtain ab. This algorithm takes 3 squaring operations and 3 additions and hence has running time $3S(n) + O(n)$. But $S(n) = \Omega(n)$, as any squaring algorithm must at least read the n-bit input. This implies the running time of the multiplication algorithm above is $\Theta(S(n))$, contradicting Professor Lake's claim.

2.27 a)

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^2 = \begin{bmatrix} a^2 + bc & b(a+d) \\ c(a+d) & bc + d^2 \end{bmatrix}$$

Hence the 5 multiplications $a^2, d^2, bc, b(a+d)$ and $c(a+d)$ suffice to compute the square.

b) We do get 5 subproblems but they are *not of the same type as the original problem*. Note that we started with a squaring problem for a matrix of size $n \times n$ and three of the 5 subproblems now involve *multiplying* $n/2 \times n/2$ matrices. Hence the recurrence $T(n) = 5T(n/2) + O(n^2)$ does not make sense.

c) Given two $n \times n$ matrices X and Y, create the $2n \times 2n$ matrix A:

$$A = \begin{bmatrix} 0 & X \\ Y & 0 \end{bmatrix}$$

It now suffices to compute A^2, as its upper left block will contain XY:

$$A = \begin{bmatrix} XY & 0 \\ 0 & XY \end{bmatrix}$$

Hence, the product XY can be calculated in time $O(S(2n))$. If $S(n) = O(n^c)$, this is also $O(n^c)$.

2.28 For any column vector u of length n, let $u^{(1)}$ denote the column vector of length $n/2$ consisting of the first $n/2$ coordinates of u. Similarly, define $u^{(2)}$ to be the vector of the remaining coordinates. Note then that $(H_k v)^{(1)} = H_{k-1}v^{(1)} + H_{k-1}v^{(2)} = H_{k-1}(v^{(1)} + v^{(2)})$ and $(H_k v)^{(2)} = H_{k-1}v^{(1)} - H_{k-1}v^{(2)} = H_{k-1}(v^{(1)} - v^{(2)})$. This shows that we can find $H_k v$ by calculating $(v^{(1)} + v^{(2)})$ and $(v^{(1)} - v^{(2)})$ and recursively computing $H_{k-1}(v^{(1)} + v^{(2)})$ and $H_{k-1}(v^{(1)} - v^{(2)})$. The running time of this algorithm is given by the recursion $T(n) = 2T(\frac{n}{2}) + O(n)$, where the linear term is the time taken to perform the two sums. This has solution $T(n) = O(n \log n)$ by the Master theorem.

2.29 a) We show this by induction on n, the degree of the polynomial. For $n = 1$, the routine clearly works as the body of the loop itself evaluates p at x. For $n = k + 1$, notice that the first k iterations of the for loop evaluate the polynomial $q(x) = a_1 + a_2x + a_3x^2 + \cdots a_nx^{n-1}$ at x by the inductive hypothesis. The last iteration of the for loop evaluates then $xq(x) + a_0 = p(x)$ at point x, as required.

b) Every iteration of the for loop uses one multiplication and one addition, so the routine uses n additions and n multiplications. Consider now the polynomial $p(x) = x^n$ for $n = 2^k$. p can be evaluated at x using only $k = \log n$ multiplication simply by repeatedly squaring x. However, Horner's rule still takes n multiplications.

2.30 (a) Observe that taking $\omega = 3$ produces the following powers : $(\omega, \omega^2, \omega^3, \omega^4, \omega^5, \omega^6) = (3, 2, 6, 4, 5, 1)$. Verify that

$$\omega + \omega^2 + \omega^3 + \omega^4 + \omega^5 + \omega^6 = 1 + 2 + 3 + 4 + 5 + 6 = 21 = 0 \ (mod\ 7)$$

(b) The matrix $M_6(3)$ is the following:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 3 & 2 & 6 & 4 & 5 \\ 1 & 2 & 4 & 1 & 2 & 4 \\ 1 & 6 & 1 & 6 & 1 & 6 \\ 1 & 4 & 2 & 1 & 4 & 2 \\ 1 & 5 & 4 & 6 & 2 & 3 \end{bmatrix}$$

Multiplying with the sequence $(0, 1, 1, 1, 5, 2)$ we get the vector $(3, 6, 4, 2, 3, 3)$.

(c) The inverse matrix of $M_6(3)$ is easily seen to be the matrix

$$6 \cdot \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 5 & 4 & 6 & 2 & 3 \\ 1 & 4 & 2 & 1 & 4 & 2 \\ 1 & 6 & 1 & 6 & 1 & 6 \\ 1 & 2 & 4 & 1 & 2 & 4 \\ 1 & 3 & 2 & 6 & 4 & 5 \end{bmatrix}$$

Verify that multiplying these two matrices mod 7 equals the identity. Also multiply this matrix with vector $(3, 6, 4, 2, 3, 3)$ to get the original sequence.

(d) We first express the polynomials as vectors of dimension 6 over the integers mod 7: $(1, 1, 1, 0, 0, 0)$, and $(-1, 2, 0, 1, 0, 0) = (6, 2, 0, 1, 0, 0)$ respectively. We then apply the matrix $M_6(3)$ to both to get the transform of the two sequences. That produces $(3, 6, 0, 1, 0, 3)$ and $(2, 4, 4, 3, 1, 1)$ respectively. Then we just multiply the vectors coordinate-wise to get $(6, 3, 0, 3, 0, 3)$. This is the transform of the product of the two polynomials. Now, all we have to do is multiply by the inverse FT matrix $M_6(3)^{-1}$ to get the final polynomial in the coefficient representation : $(-1, 1, 1, 3, 1, 1)$.

2.31 a) We prove each statement of the rule separately:

* If a and b are both even, then $\gcd(a, b) = 2\gcd(a/2, b/2)$.
  Proof: $\gcd(a, b) = d$ iff $a = da'$ and $b = db'$ and $\gcd(a', b') = 1$. Since a and b are both even, $2|d$ so $a/2 = a'd/2$ and $b/2 = b'd/2$, and since $\gcd(a', b') = 1$, then $\gcd(a/2, b/2) = d/2$.
* If a is even and b is odd, then $\gcd(a, b) = \gcd(a/2, b)$.
  Proof: Since b is odd, $\gcd(a, b) = d$ is odd. If an odd $d|a$ then $d|a/2$.
* If a and b are both odd, then $\gcd(a, b) = \gcd((a - b)/2, b)$.
  Proof: On page 20 we proved that for $a \geq b$, $\gcd(a, b) = \gcd(a - b, b)$. Now since $|a - b|$ is even if both a and b are odd, we can just use part 2 to conclude that $\gcd(a, b) = \gcd((a - b)/2, b)$.

b) Consider the following algorithm:

```
function gcd(a, b)
if b == 0:
    return a
if a is even:
    if b is even:
        return 2· gcd(a/2, b/2)
    else:
        return gcd(b, a/2)
else
                    return gcd(b, (a − b)/2)
```

The algorithm is correct according to part a) of this problem. Each argument is reduced by half after at most two calls to the function, so the function is called at most $O(\log a + \log b)$ times. Each function call takes constant time, assuming the subtraction in the last line takes constant time, so the running time is $O(\log a + \log b)$.

c) More realistically, if we take the subtraction to take time $O(n)$, we have that each call takes at most time $O(n)$ and there are at most $O(\log a + \log b) = O(n)$ calls. So, the running time is $O(n^2)$, which compares favorably to the $O(n^3)$ algorithm of Chapter 1.

2.32 a) Suppose 5 or more points in L are found in a square of size $d \times d$. Divide the square into 4 smaller squares of size $\frac{d}{2} \times \frac{d}{2}$. At least one pair of points must fall within the same smaller square: these two points will then be at distance at most $\frac{d}{\sqrt{2}} < d$, which contradicts the assumption that every pair of points in L is at distance at least d.

b) The proof is by induction on the number of points. The algorithm is trivially correct for two points, so we may turn to the inductive step. Suppose we have n points and let (p_s, p_t) be the closest pair. There are three cases.
If $p_s, p_t \in L$, then $(p_s, p_t) = (p_L, q_L)$ by the inductive hypothesis and all the other pairs tested by the algorithm are at a larger distance apart, so the algorithm will correctly output (p_s, p_t). The same reasoning holds if $p_s, p_t \in R$.
If $p_s \in L$ and $p_t \in R$, the algorithm will be correct as long as it tests the distance between p_s and p_t. Because p_s and p_t are at distance smaller than d, they will belong to the strip of points with x-coordinate in $[x-d, x+d]$. Suppose that $y_s \leq y_t$. A symmetric construction applies in the other case. Consider the rectangle S with vertices $(x-d, y_s), (x-d, y_s+d), (x+d, y_s+d), (x+d, y_s)$. Notice that both p_s and p_t must be contained in S. Moreover, the intersection of S with L is a square of size $d \times d$, which, by a), can contain at most 4 points, including p_s. Similarly, the intersection of S with R can also contain at most 4 points, including p_t. Because the algorithm checks the distance between p_s and the 7 points following p_s in the y-sorted list of points in the middle strip, it will check the distance between p_s and all the points of S. In particular, it will check the distance between p_s and p_t, as required for the correctness of the algorithm.

c) When called on input of n points this algorithm first computes the median x value in $O(n)$ and then splits the list of points into those belonging to L and R, which also takes time $O(n)$. Then the algorithm can recurse on these two subproblems, each over $n/2$ points. Once these have been solved the algorithm sorts the points in the middle strip by y coordinate, which takes time $O(n \log n)$ and then computes $O(n)$ distances, each of which can be calculated in constant time. Hence the running time is given by the recursion $T(n) = 2T(\frac{n}{2}) + O(n \log n)$. This can be analyzed as in the proof of the Master theorem. The kth level of the recursion tree will contribute

$t_k = 2^k \frac{n}{2^k}(\log n - k)$. Hence, the total running time will be:

$$\sum_{k=0}^{\log n} t_k = n\log^2 n - n\sum_{k=0}^{\log n} k \leq n\log^2 n - \frac{n}{2}\log^2 n = O(n\log^2 n)$$

.

d) We can save some time by sorting the points by y-coordinate only once and making sure that the split routine is implemented as not to modify the order by y when splitting by x. Sorting takes time $O(n\log n)$, while the time required by the remaining of the algorithm is now described by the recurrence $T(n) = 2T(\frac{n}{2}) + O(n)$, which yields $T(n) = O(n\log n)$. Hence, the overall running time is $O(n\log n)$.

2.33 a) Let $M_{ij} \neq 0$. It is sufficient to bound above $\Pr[(Mv)_i = 0]$, as $\Pr[(Mv)_i = 0] \geq \Pr[Mv = 0]$. Now, $(Mv)_i = \sum_{t=1}^{n} M_{it}v_t$, so $(Mv)_i$ is 0 if and only if:

$$M_{ij}v_j = \sum_{\substack{t=1 \\ t\neq j}}^{n} M_{it}v_t$$

Consider now any fixed assignment of values to all the v_t's but v_j. If, under this assignment, the right hand side is 0, v_j has to be 0. Similarly, if the right hand side is non-zero, v_j cannot be 0. In either case, v_j can only take one of the two values $\{0, 1\}$ and $Pr[(Mv)_i = 0] \leq \frac{1}{2}$.

b) If $AB \neq C$, the difference $M = AB - C$ is a non-zero matrix and, by part a):

$$\Pr[ABv = Cv] = \Pr[Mv = 0] \leq \frac{1}{2}$$

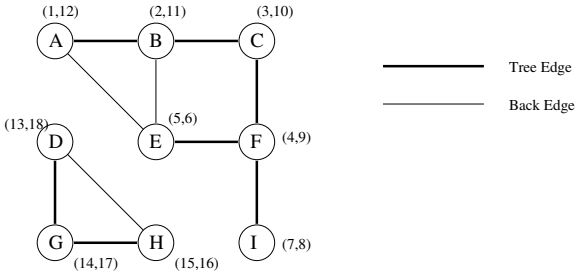
The randomized test for checking whether $AB = C$ is to compare ABv with Cv for a random v constructed as in a). To compute ABv, it is possible to use the associativity of matrix multiplication and compute first Bv and then $A(Bv)$. This algorithm performs 3 matrix-vector multiplications, each of which takes time $O(n^2)$, while the final comparison takes time $O(n)$. Hence, the total running time of the randomized algorithm is $O(n^2)$.

2.34 A linear-time algorithm requires dynamic programming. Here we give a divide-and-conquer algorithm running in polynomial time in n.

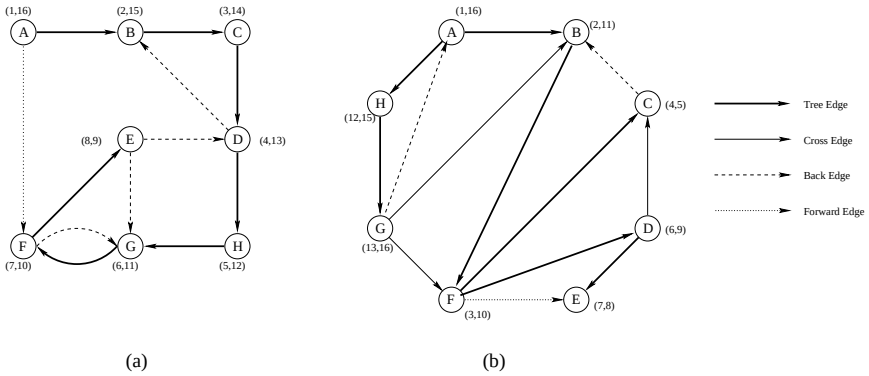
Let ϕ be the input 3SAT formula exhibiting the special local property. For $1 \leq i \leq j \leq n$, let $\phi(i, j)$ be the 3SAT formula consisting of clauses of ϕ containing only variables $x_i, x_{i+1}, \cdots, x_j$. We can then split ϕ into $\phi(1, \frac{n}{2}$, $\phi(\frac{n}{2} + 1, n)$ and the formula ψ consisting of the remaining clauses, i.e. those containing at least one variable with index $k \leq \frac{n}{2}$ and one with index $k > \frac{n}{2}$. The local property implies that ψ can only contain variables with index in the range $\{\frac{n}{2} - 9, \cdots, \frac{n}{2} + 10\}$. Hence, at most 20 variables appear in ψ. Suppose now we are given an assignment a to the variable of ψ that satisfies ψ.. To check if a can be extended to a satisfying assignment for the whole of ϕ, we can substitute the values of a for their corresponding variables within $\phi(1, \frac{n}{2}$ and $\phi(\frac{n}{2} + 1, n)$ and consider the satisfiability of these. Letting $\phi_a(1, \frac{n}{2}$ and $\phi_a(\frac{n}{2} + 1, n)$ be these new formulae, notice that ϕ will be satisfiable by an extension to a if and only if $\phi_a(1, \frac{n}{2}$ and $\phi_a(\frac{n}{2} + 1, n)$ are satisfiable, as the latter have no variables in common. Moreover, both $\phi_a(1, \frac{n}{2}$ and $\phi_a(\frac{n}{2} + 1, n)$ exhibit the same local property as ϕ, so we can recurse on them to verify their satisfiability. We can apply this procedure over all choices of a satisfying ψ to check if any assignment satisfies ϕ. The running time of this algorithm is described by the recursion $T(n) \leq 2^{20} \cdot 2T(n/2) + O(n) = O(n^2 1)$, as there are at most 2^{20} assignments a, each of which gives rise to two subproblems.

Chapter 3 – Solutions

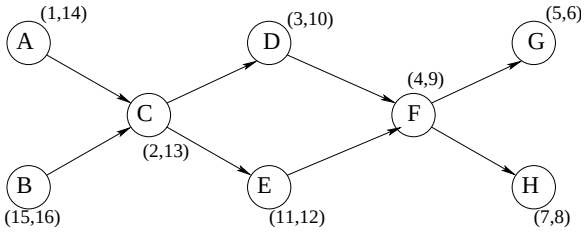
3.1 The figure below gives the `pre` and `post` numbers of the vertices in parentheses. The tree and back edges are marked as indicated.



3.2 The figure below shows `pre` and `post` numbers for the vertices in parentheses. Different edges are marked as indicated.



3.3 (a) The figure below shows the `pre` and `post` times in parentheses.



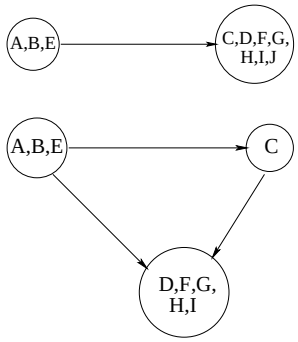
(b) The vertices A, B are sources and G, H are sinks.

(c) Since the algorithm outputs vertices in decreasing order of `post` numbers, the ordering given is B, A, C, E, D, F, H, G.

(d) Any ordering of the graph must be of the form $\{A, B\}, C, \{D, E\}, F, \{G, H\}$, where $\{A, B\}$ indicates A and B may be in any order within these two places. Hence the total number of orderings is $2^3 = 8$.

3.4 (i) The strongly connected component found first is $\{C, D, F, G, H, I, J\}$ followed by $\{A, B, E\}$. $\{C, D, F, G, H, I, J\}$ is a source SCC, while $\{A, B, E\}$ is a sink SCC. The metagraph is shown in the figure below. It is easy to see that adding 1 edge from any vertex in the sink SCC to a vertex in the source SCC makes the graph strongly connected.

(ii) The strongly connected components are found in the order $\{D, F, G, H, I\}$, $\{C\}$, $\{A, B, E\}$. $\{A, B, E\}$ is a source SCC, while $\{D, F, G, H, I\}$ is a sink. Also, in this case adding one edge from any vertex in the sink SCC to any vertex in the source SCC makes the metagraph strongly connected and hence the given graph also becomes strongly connected.



3.5 Create a new (empty) adjacency list for the reverse. We go through the list of G and if in the neighborhood of u, we find the vertex v, add u in the list of v in G^R as the *first* element in the list. Note that it would have been difficult to insert u at the end of the list, but insertion in the first position takes only $O(1)$ time.

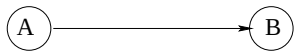
3.6 (a) Note that each edge (u, v) contributes 1 to $d(u)$ and 1 to $d(v)$. Hence, each edge contributes exactly 2 to the sum $\sum_{v \in V} d(v)$, which gives $\sum_{v \in V} d(v) = 2|E|$.

(b) Let V_o be the set of vertices with odd degree and V_e be the set of vertices with even degree. Then

$$\sum_{v \in V_o} d(v) + \sum_{v \in V_e} d(v) = 2|E| \implies \sum_{v \in V_o} d(v) = 2|E| - \sum_{v \in V_e} d(v)$$

The RHS of this equation is even and the LHS is a sum of odd numbers. A sum of odd numbers can be even, only if it is the sum of an *even number of odd numbers*. Hence, the number of vertices in V_o (equal to the number of people with odd number of handshakes) must be even.

(c) No. The following graph provides a counter-example as only one vertex (B) has odd indegree.



3.7 (a) Let us identify the sets V_1 and V_2 with the colors red and blue. We perform a DFS on the graph and color alternate levels of the DFS tree as red and blue (clearly they must have different colors). Then the graph is bipartite iff there is no monochromatic edge. This can be checked during DFS itself as such an edge must be a back-edge, since tree edges are never monochromatic by construction and DFS on undirected graphs produces only tree and back edges.

(b) The "only if" part is trivial since an odd cycle cannot be colored by two colors. To prove the "if" direction, consider the run of the above algorithm on a graph which is not bipartite. Let u and v be two vertices such that (u, v) is a monochromatic back-edge and u is an ancestor of v. The path length from u to v in the tree must be even, since they have the same color. This path, along with the back-edge, gives an odd cycle.

(c) If a graph has exactly one odd cycle, it can be colored by 3 colors. To obtain a 3-coloring, delete one edge from the odd cycle. The resulting graph has no odd cycles and can be 2-colored. We now add back the deleted edge and assign a new (third) color to one of its end points.

3.8 (a) Let $G = (V, E)$ be our (directed) graph. We will model the set of nodes as triples of numbers (a_0, a_1, a_2) where the following relationships hold: Let $S_0 = 10$, $S_1 = 7$,$S_2 = 4$ be the sizes of the corresponding containers. a_i will correspond at the actual contents of the i^{th} container. It must hold $0 \leq a_i \leq S_i$ for $i = 0, 1, 2$ and at any given node $a_0 + a_1 + a_2 = 11$ (the total amount of water we started from). An edge between two nodes (a_0, a_1, a_2) and (b_0, b_1, b_2) exists if both the following are satisfied :

- the two nodes differ in exactly two coordinates (and the third one is the same in both).

– if i, j are the coordinates they differ in, then either $a_i = 0$ or $a_j = 0$ or $a_i = S_i$ or $a_j = S_j$.

The question that needs to be answered is wether there exists a path between the nodes $(0, 7, 4)$ and $(*, 2, *)$ or $(*, *, 2)$ where $*$ stands for any (allowed) value of the corresponding coordinate.

(b) Given the above description, it is easy to see that a DFS algorithm on that graph should be applied, starting from node $(0, 7, 4)$ with an extra line of code that halts and answers 'YES' if one of the desired nodes is reached and 'NO' if all the connected component of the starting node is exhausted an no desired vertex is reached.

(c) It is easy to see that after a few steps of the algorithm (depth 6 on the dfs tree) the node (2,7,2) is reached, so we answer 'YES'.

3.9 First, the degree of each node can be determined by counting the number of elements in its adjacency list. The array `twodegree` can then be computed by initializing `twodegree` to 0 for every vertex, and then modifying `explore` as follows to add the degree of each neighbor of a vertex u to `twodegree`$[u]$.

```
explore(G, u) {
     visited(u) = true
     previsit(u)
     for each edge (u, v) ∈ E:
         twodegree[u] = twodegree[u] + degree[v]
         if not visited(v):  explore(v)
     postvisit(u)
}
```

3.10 We use an extra variable `top` which refers to the top element in the stack. We assume that `top` is automatically updated when some element is pushed or popped. Also, we need to keep track of how many neighbors of a vertex have been already checked. For this we also need the outdegree of every vertex (which is part of the adjacency list). We assume that `neighbor`(v, i) gives the ith neighbor of the vertex v[1].

The code for the `explore` function is as follows:

```
explore(G, u) {
     visited(u) = true
     previsit(u)
     neighbors_checked(u) = 0
     push u
     while stack is not empty {
         w = top
         for i from neighbors_checked[w]+1 to outdegree[w] {
             v = neighbor(w, i)
             if not visited(v)
                 neighbors_checked[w] = i
                 push(v)
                 break
         }
         if neighbors_checked[w] = outdegree[w]
             pop(w)
             postvisit(w)
     }
}
```

3.11 Let $e = (u, v)$. The graph has a cycle containing e if and only if u and v are in the same connected component in the graph obtained by deleting e. This can be easily checked by a DFS on this graph.

[1]We have a list and cannot actually access the ith neighbor, but i changes sequentially in the loop here and hence we allow ourselves some abuse of notation.

3.12 There are two cases possible: $\texttt{pre}(u) < \texttt{pre}(v) < \texttt{post}(v) < \texttt{post}(u)$ or $\texttt{pre}(v) < \texttt{post}(v) < \texttt{pre}(u) < \texttt{post}(u)$. In the first case, u is an ancestor of v. In the second case, v was popped off the stack without looking at u. However, since there is an edge between them and we look at all neighbors of v, this cannot happen. So, the given statement is true.

3.13 (a) Consider the DFS tree of G starting at any vertex. If we remove a leaf (say v) from this tree, we still get a tree which is a connected subgraph of the graph obtained by removing v. Hence, the graph remains connected on removing v.

(b) A directed cycle. Removing any vertex from a cycle leaves a path which is not strongly connected.

(c) A graph consisting of two disjoint cycles. Each cycle is individually a strongly connected component. However, adding just one edge is not enough as it (at most) allows us to go from one component to another but not back.

3.14 The algorithm is to always pick a source, delete it from the graph and then recurse on the resulting graph. A source is just a vertex with indegree 0. Thus, we can find all the sources in the initial graph by performing a DFS and computing the indegree of all the vertices. We add all the vertices with indegree 0 to a list L.

At each step, the algorithm then removes an element from L (a source) and reduces the indegree of each of its neighbors by 1 (this corresponds to deleting it from the graph). If this changes the indegree of any vertex to 0, we add it to L. The removed element is assigned the next position in the ordering.

Computing all indegrees in the first step only requires a DFS which takes linear time. Subsequently, we only look at each edge (u, v) at most once when we are removing u from L. Hence, the total time is linear in the size of the graph.

3.15 (a) We view the intersections as vertices of a graph with the streets being directed edges, since they are one-way. Then the claim is equivalent to saying that this graph is strongly connected. This is true iff the graph has only one strongly connected component, which can be checked in linear time.

(b) The claim says that starting from the town hall, one cannot get to any other SCC in the graph. This is equivalent to saying that the SCC containing the vertex corresponding to the town hall is a sink component. This can be easily done in linear time by first finding the components, and then running another DFS from the vertex corresponding to the town hall, to check if any edges go out of the component[2].

3.16 The graph of the prerequisite relation must be a DAG, since there can be no circular dependencies between courses. The number of courses essentially corresponds to the length of the longest path in this DAG.

We first linearize this graph to obtain an ordering $c_1, c_2, \ldots, c_n$ of the courses, such that c_i is a possibly a prerequisite only for c_j with $j > i$. Any path ending at c_j can now only pass through c_i for $i < j$. If l_j is the length of the longest path ending at c_j, then

$$l_j = 1 + \max_{(i,j) \in E} \{l_i\}$$

which depends only on l_i for $i < j$. The required solution is $\max_i \{l_i\}$. This is essentially a use of dynamic programming.

Note however, that here we need all edges coming *into* c_j rather going out, which is what the adjacency list stores. This can be handled by computing the reverse of the graph (see problem 3.5 - in fact, it is even easier for a linearized DAG) or by modifying the algorithm above so that each c_i "updates" the maximum value at its neighbors, when it computes its own l_i value.

[2]In fact, it can even be done *while* decomposing the graph into SCCs by noting that the algorithm for decomposing progressively removes sinks from the graph at every stage. A component found by the algorithm is a sink if and only if there are no edges going out of the component into any component found *before* it.

3.17 (a) For the sake of contradiction, assume that there are two different vertices $u, v \in Inf(p)$ such that u and v belong to different strongly connected components, say C_1 and C_2 respectively. But since u occurs both before and after v in the trace, there must be a path from u to v and also a path from v to u. This would imply that there is a path from every vertex in C_1 to every vertex in C_2 and vice-versa, which is a contradiction.

(b) The argument in the previous part shows that any infinite trace must be a subset of a strongly connected component. It is also easy to see that any strongly connected component of size greater than 1 has an infinite trace since we can always pick two vertices in the same component and go from one to another infinitely often.

However, a graph that has all SCCs of size 1 must be a DAG and hence the problem reduces to checking if the given directed graph has a cycle. This can be done using DFS since the graph has a cycle if and only if DFS finds a back edge.

(c) Let $v \in Inf(p)$ be a good vertex visited infinitely often by the trace p. Also, there must be at least one more vertex in the component of v for the trace to be infinite. Hence, the problem reduces to checking if the graph contains a strongly connected component of size more than 1, which contains a good vertex. This can be done by decomposing the graph into its SCCs.

(d) Let p be a trace such that $Inf(p) \subseteq V_G$. Then there must exist a number N such that $v_n \in V_G$ for all $n \geq N$ (since no bad vertex is visited infinitely often). Then $Inf(p)$ must itself form a strongly connected subgraph since after time N, the trace does not take any path passing through a bad vertex. Hence, a graph contains an infinite trace which visits only good vertices infinitely often if and only if the subgraph induced by V_G contains a strongly connected component of size greater than 1 (We argued only the "only if" part above, but the other direction is trivial). This can be checked by proceeding as in part (b) with the subgraph induced by V_G.

3.18 Do a DFS on the tree starting from r and store the previsit and postvisit times for each node. Since the given graph is a tree, and we started at the root, the DFS tree is the same as the given tree. Thus, u is an ancestor of v if and only if $\texttt{pre}(u) < \texttt{pre}(v) < \texttt{post}(v) < \texttt{post}(u)$.

3.19 We modify the explore procedure so that explore called on a node returns the maximum `x` value in the corresponding subtree. The parent stores this as its `z` value, and returns the maximum of this and its own `x` value.

```
explore(G, u) {
     visited(u) = true
     z(u) = −∞
     temp = 0
     for each edge (u, v) ∈ E:
          If not visited(v):  {
              temp = explore(G, v)
              if temp > z(u):  z(u) = temp
          }
     postvisit(u)
     return max{z(u), x(u)}
}
```

3.20 We maintain the labels of all the vertices currently on the stack, in a a separate array. Since a path can have at most n vertices, the length of this array is at most n. The labels are modified using this array in the `previsit` and `postvisit` procedures.

- ```
  previsit(v) {
        current_depth = current_depth + 1
        labels[current_depth] = l(v)
  }
  ```
- ```
  postvisit(v){
        ancestor_depth = max{0, current_depth - l(v)}
  ```

```
        l(v) = labels[ancestor_depth]
        current_depth = current_depth - 1
    }
```

`ancestor_depth` identifies the level at which $p^{l(v)}$ is present in the path and stores the appropriate label in the current node. Since we add only a constant number of operations at each step of DFS, the algorithm is still linear time.

3.21 Consider the case of a strongly connected graph first. The case of a general graph can be handled by breaking it into its strongly connected components, since a cycle can only be present in a single SCC. We proceed by coloring alternate levels of the DFS tree as red and blue. We claim that the graph has an odd cycle if and only if there is an edge between two vertices of the same color (which can be checked in linear time).

If there is an odd cycle, it cannot be two colored and hence there must be a monochromatic edge. For the other direction, let u and v be two vertices having the same color and let (u, v) be an edge. Also, let w be their lowest common ancestor in the tree. Since u and v have the same color, the distances from w to u and v are either both odd or both even. This gives two paths p_1 and p_2 from w to v, one through u and one not passing through u, one of which is odd and the other is even.

Since the graph is strongly connected, there must also be a path q from v to w. Since the length of this path is either odd or even, q along with one of p_1 and p_2 will give an odd length tour (a cycle which might visit a vertex multiple times) passing through both v and w. Starting from v, we progressively break the tour into cycles whenever it intersects itself. Since the length of the tour is odd, one of these cycles must have odd length (as the sum of their lengths is the length of the tour).

3.22 Let us call a vertex from which all other vertices are reachable, a *vista* vertex. If the graph has a vista vertex, then it must have only one source SCC (since two source SCCs are not reachable from each other), which must contain the vista vertex (if it's in any other SCC, there is no path from the vista vertex to the source SCC). Moreover, in this case *every* vertex in the source SCC will be a vista vertex.

The algorithm is then simply to a DFS starting from any node and mark the vertex with the highest `post` value. This must be in a source SCC. We now again run a DFS from this vertex to check if we can reach all nodes. Since the algorithm just uses decomposition into SCCs and DFS, the running time is linear.

3.23 We start by linearizing the DAG. Any path from s to t can only pass through vertices between s and t in the linearized order and hence we can ignore the other vertices.

Let $s = v_0, v_1, \ldots, v_k = t$ be the vertices from s to t in the linearized order. For each i, we count the number of paths from s to v_i as n_i. Each path to a vertex i and an edge (i, j), gives a path the vertex j and hence

$$n_j = \sum_{(i,j)\in E} n_i$$

Since $i < j$ for all $(i, j) \in E$, this can be computed in increasing order of j. The required answer is n_k.

3.24 Start by linearizing the DAG. Since the edges can only go in the increasing direction in the linearized order, and the required path must touch all the vertices, we simply check if the DAG has an edge $(i, i+1)$ for every pair of consecutive vertices labelled i and $i + 1$ in the linearized order. Both, linearization and checking outgoing edges from every vertex, take linear time and hence the total running time is linear.

3.25 Start by linearizing the DAG. Let $v_1, \ldots v_n$ be the linearized order. Then the following algorithm finds the `cost` array in linear time.

```
find_costs() {
    for i = n to 1:
       cost[v_i] = p_{v_i}
       for all (v_i, v_j) ∈ E:
```

```
            if cost[v_j] < cost[v_i]:
               cost[v_i] = cost[v_j]
}
```

The time for linearizing a DAG is linear. For the above procedure, we visit each edge at most once and hence the time is linear. For a general graph, the `cost` value of any two nodes in the same strongly connected component will be the same since both are reachable from each other. Hence, it is sufficient to run the above algorithm on the DAG of the strongly connected components of the graph. For a node corresponding to component C, we take $p_C = \min_{u \in C}\{p_u\}$.

3.26 (a) We first prove the "only if" direction. Suppose we have an Eulerian tour for a graph G. Let u be any vertex in the graph. Suppose that we "enter" u k times during the tour. Since it is a cycle, we must also leave u exactly k times and all these edges must be distinct. Hence, the degree of u must be $2k$ which is even. Since this is true for any vertex u, the claim follows.

For the other direction we use induction on the number of vertices in the graph. First note that if $|V| = 2$, the trivially if the degree of both the vertices is even then the graph has an Eulerian tour. Let the statement be true for all graphs with $|V| = n$.

We consider a graph G on $n+1$ vertices such that all its vertices have even degrees. Let u be a vertex in this graph having neighbors $i_1, i_2, \ldots i_{2k}$. Consider a graph G' where we remove u and add edges (i_1, i_2), $(i_3, i_4), \ldots, (i_{2k-1}, i_{2k})$ to G. Since G' has n vertices and the degree of each vertex is the same as in G (and thus even), G' must have an Eulerian tour. Replace every occurence of the extra edges of the form (i_{t-1}, i_t) that we inserted, by (i_{t-1}, u) followed by (u, i_t). This gives an Eulerian tour of G.

(b) To have an Eulerian path, exactly two of the vertices in the graph must have odd degree, while all the remaining ones must have even degree.

(c) A directed graph has an Eulerian tour iff the number of incoming edges at every vertex is equal to the number of outgoing edges.

3.27 By problem 3.6, we know that the number of vertices with an odd degree in an undirected graph, must be even. Suppose the number of such vertices is $2k$. We arbitrarily pair up these vertices and add an edge between each pair so that all vertices now have even degree.

By problem 3.26, each connected component of this new graph must have an Eulerian tour. Removing the k edges we added from this set of tours, breaks it into k paths with the two ends of each path being vertices of odd degree. Furthermore, all these paths are edge-disjoint, since an Eulerian tour uses each edge exactly once. Thus, taking the two ends of each path as a pair gives the required pairing.

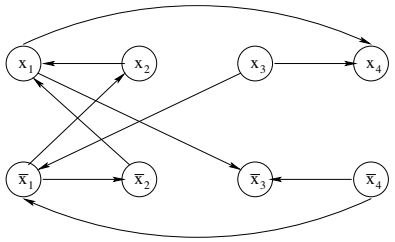
3.28 a) The formula has two satisfying assignments:

$$(x_1, x_2, x_3, x_4) = (\texttt{true}, \texttt{false}, \texttt{false}, \texttt{true})$$

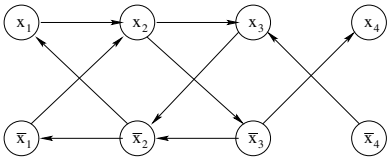
$$(x_1, x_2, x_3, x_4) = (\texttt{true}, \texttt{true}, \texttt{false}, \texttt{true})$$

b) An example is $(\bar{x}_1 \vee x_2) \wedge (\bar{x}_2 \vee x_3) \wedge (\bar{x}_3 \vee \bar{x}_2) \wedge (x_1 \vee x_2) \wedge (x_3 \vee x_4)$

c) The graphs for the given formula and the example in part b) are given below.



d) Notice that a path in the directed graph from x to y means that $x \implies y$. If x and $\bar{x}$ are in the same storngly connected componenent, then we have $x \implies \bar{x}$ and $\bar{x} \implies x$. Then there is no way to assign a value to x to satisfy both these implications.



e) We recursively find sinks in the graph of strongly connected components and assign `true` to all the *literals* in the sink component (this means that if a sink component contains the literal $\bar{x}_2$, then we assign $\bar{x}_2 = \texttt{true} \equiv x_2 = \texttt{false}$). We will then remove all the variables which have been assigned a value from the graph.

When we set the literals in a sink component to `true`, we set their negations to `false`. However, by the symmetry of the way we assigned edges, if we reverse the directions of all the edges and replace each literal x by $\bar{x}$, the graph should remain unchanged. Hence, the negations of all the variables in a sink component *must form a source component.*

Now, we claim that we did not violate any implication while doing this. Any implication of the form $l_1 \implies l_2$ is violated only when $l_1 = \texttt{true}$ and $l_2 = \texttt{false}$. All the literals in the sink are `true` so this never happens. All the literals in the source are set to `false`, so no edge going out of them can be violated. We can now recurse on the remaining graph. Since the values assigned at each stage are consistent, we end up with a satisfying assignment.

f) To perform the operations in the previous part, we only need to construct this graph from the formula, find its strongly connected components and identify the sinks - all of which can be done in linear time.

3.29 We represent all elements of the set S as a graph with an edge from x to y if $(x, y) \in R$. Since the relation is symmetric, the corresponding graph is undirected. We claim that the connected components of the graph give the required partition of S into groups $S_1, S_2, \ldots, S_k$.

Since there are no edges between different connected components, elements in different groups are definitely not related and it only remains to check that all elements in the same component are related to each other. However, this follows directly from transitivity since there is a path between any two elements in the same component.

3.30 The relation is symmetric by definition. It is also reflexive since there is a path from every vertex u to itself of length 0. To verify transitivity, if $(u, v) \in R$ and $(v, w) \in R$, then there is a path from u to v and v to w, which gives a path from u to w (this might not be a simple path, but we can remove any cycles). Similarly, we get a path from w to u.

By problem 3.29, this partitions the graph into disjoint groups such that for every two vertices u and v in the same group, there is a path from u to v and v to u. Also, this is not true for two vertices not in the same group. However, this is precisely the definition of the strongly connected components.

3.31 (a) The relation is reflexive and symmetric by definition. To verify transitivity, let $(e_1, e_2) \in R$, both being contained in a cycle C_1 and $(e_2, e_3) \in R$, contained in cycle C_2. Then, if either of C_1 and C_2 contains all three of e_1, e_2 and e_3, we are done as then $(e_1, e_3) \in R$.

If not, then neither of e_1 and e_3 can be common to both the cycles. Also, C_1 and C_2 must share some edges. Starting from e_3, we move in both directions on C_2, until we reach a vertex in C_1. This gives a (possibly small) path P in C_2, which contains e_3 and has both endpoints (say v_1 and v_2) in C_1. Now, C_1 has two simple paths between v_1 and v_2, exactly one of which contains e_1. Removing the other gives a simple cycle containing both e_1 and e_3. Hence $(e_1, e_3) \in R$.

(b) The biconnected components in the given graph are $\{AB, BN, NO, OA\}$, $\{BD\}$, $\{CD\}$, $\{DM\}$ and $\{LK, KJ, JL, KH, IJ, IF, FG, GH, HI, FH\}$. The bridges are BD, CD and DM and the separating vertices are B, D and L.

(c) We first argue that two biconnected components must share at most one vertex. For the sake of contradiction, assume that two components C_1 and C_2 share two vertices u and v. Note that there must be a path from u to v in both C_1 and C_2, since in each component, there is a simple

cycle containing one edge incident on u and one edge incident on v. The union of these two paths gives a cycle containing some edges from C_1 and some from C_2. However, this is contradiction as this would imply that an edge in C_1 is related to an edge in C_2.

We now need to prove that if two biconnected components intersect in exactly one vertex, then it must be a separating vertex. Let the common vertex be u. Let (u, v_1) and (u, v_2) be the edges corresponding to u in the two components. Then we claim that removing u disconnects v_1 and v_2. If not, then there must be a path between v_1 and v_2, which does not pass through u. However, this path, thogether with (u, v_1) and (u, v_2), gives a simple cycle containing one edge from each component which is a contradiction.

(d) The graph can in fact, be a forest. However, there cannot be a cycle as this would imply a cycle involving edges from two different biconnected components, which cannot happen since edges in different equivalence classes cannot be related.

(e) If the root has only one child, then it is effectively a leaf and removing the root still leaves the tree connected. The DFS from the first child explores every vertex reachable through a path not passing through the root. Also, since the graph is undirected, there can be no edges from the subtree of the first child to that of any other child. Hence, removing the root disconnects the tree if it has more than one children.

(f) If there is a backedge from a descendant of every child v' to an ancestor of v, each child can reach the entire tree above v the graph is still connected after removing v. If there is a child v' such that none of its descendants have a backedge to an ancestor of v, then in the graph after removing v, there is no path between an ancestor of v and v' (note that there cannot be any cross edges since the graph is undirected).

(g) While exploring each vertex u, we look at all the edges of the form (u, v) and can store at u, the lowest `pre(`v`)` value for all neighbors of u. `low(`u`)` is then given by the minimum of this value, `pre(`u`)` and the `low` values of all the children of u. Since each child can pass its `low` value to the parent when it's popped off the stack, the entire array can be computed in a single pass of DFS.

(h) A non-root node u is a separating vertex iff $\texttt{pre}(u) < \texttt{low}(v)$ for *any* child v of u. This can be checked while computing the array. Also, if u is a separating vertex and v is a child such that $\texttt{pre}(u) < \texttt{low}(v)$, then the entire subtree with v as the root must be in different biconnected components than the ancestors or other children of u. However, this subtree itself may have many biconnected components as it might have other separating vertices.

Hence, we perform a DFS pushing all the edges we see on a stack. Also, when we explore a child v of a separating vertex u such that the above condition is met, we push an extra "mark" on the stack (to mark the subtree rooted at v). When DFS returns to v, i.e. when v is popped off the stack (of vertices), we can also pop the subtree of v from the stack of edges (pop everything till the mark). If the subtree had multiple biconnected components, they would be already popped off before the DFS returned to v.

Chapter 4 – Solutions

4.1. The shortest path tree is shown in Figure 1.

	Iteration							
Node	0	1	2	3	4	5	6	7
A	0	0	0	0	0	0	0	0
B	∞	1	1	1	1	1	1	1
C	∞	∞	3	3	3	3	3	3
D	∞	∞	∞	4	4	4	4	4
E	∞	4	4	4	4	4	4	4
F	∞	8	7	7	7	7	6	6
G	∞	∞	7	5	5	5	5	5
H	∞	∞	∞	∞	8	8	6	6

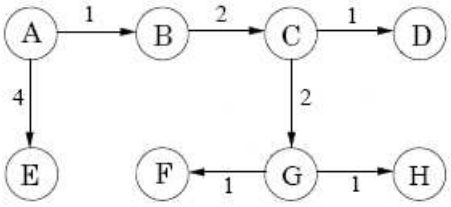


Figure 1: Shortest-path tree for 4.1.

4.2. The shortest path tree is shown in Figure 2.

	Iteration						
Node	0	1	2	3	4	5	6
S	0	0	0	0	0	0	0
A	∞	7	7	7	7	7	7
B	∞	∞	11	11	11	11	11
C	∞	6	5	5	5	5	5
D	∞	∞	8	7	7	7	7
E	∞	6	6	6	6	6	6
F	∞	5	4	4	4	4	4
G	∞	∞	∞	9	8	8	8
H	∞	∞	9	7	7	7	7
I	∞	∞	∞	∞	8	7	7

4.3. Suppose the input graph G is given as an adjacency matrix. Notice that G contains a square if and only if there are two vertices u and v that share more than one neighbor. For any u, v we can check this in time $O(|V|)$ by comparing the row of u and the row of v in the adjacency matrix of G. Because we need to repeat this process $O(|V|^2)$ to iterate over all u and v, this algorithm has running time $O(|V|^3)$. We can do better by noticing that, when comparing the rows of the adjacency matrix a_u and a_v, we are actually checking if $a_u \cdot a_v$ is greater than 1, i.e. if $[A(G)^2]_{uv} > 1$. It suffices then to

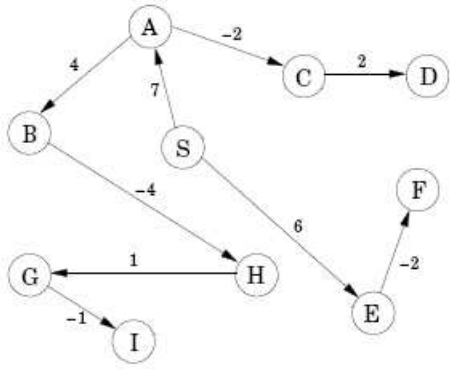


Figure 2: Shortest-path tree for 4.2.

compute $A(G)$, which we can do in time $O(|V|^{2.71}$ using our matrix multiplication algorithm and check all non-diagonal entries to see if we find one larger than 1.

4.4. The graph in Figure 3 is a counterexample: vertices are labelled with their level in the DFS tree, back edges are dashed. The shortest cycle consists of vertices $1-4-5$, but the cycle found by the algorithm is $1-2-3-4$. In general, the strategy will fail if the shortest cycle contains more than one back edge.

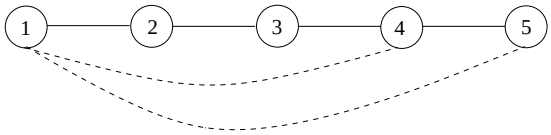


Figure 3: Counterexample for 4.4.

4.5 We perform a BFS on the graph starting from u, and create a variable `num_paths`(x) for the number of paths from u to x, for all vertices x. If $x_1, x_2, \ldots x_k$ are vertices at depth l in the BFS tree and x is a vertex at depth $l+1$ such that $(x_1, x), \ldots, (x_k, x) \in E$ then we want to set `num_paths`$(x) =$ `num_paths`$(x_1) + \ldots +$ `num_paths`(x_k). The easiest way to do this is to start with `num`$_p$`aths`$(x) = 0$ for all vertices $x \neq u$ and `num_paths`$(u) = 1$. We then update `num_paths`$(y) =$ `num_paths`$(y) +$ `num_paths`(x), for each edge (x, y) that goes down one level in the tree. Since, we only modify BFS to do one extra operation per edge, this takes linear time. The pseudocode is as follows

```
function count_paths(G, u, v)
    for all x ∈ V:
        dist(x) = ∞
        num_paths(x) = 0

    dist(u) = 0
    num_paths(u) = 1
    Q = [u]
    while Q is not empty:
        x = eject(Q)
        for all edges (x, y) ∈ E
            if dist(y) = dist(x) + 1:
```

```
        num_paths(y) = num_paths(y) + num_paths(x)
    if dist(y) = ∞:
        inject(Q,y)
        dist(y) = dist(x) + 1
        num_paths(y) = num_paths(x)
```

4.6. This is true as long as every node $u \in V$ is reachable from s, the node from which Dijkstra's algorithm was run. In this case, every node is connected to s through a shortest path consisting of edges of the type $\{u, \texttt{prev}(u)\}$ and the graph formed by all these edges is connected. Moreover, such graph cannot have cycles, because $\{u, \texttt{prev}(u)\}$ can be an edge only if $\texttt{prev}(u)$ was deleted off the heap before u, so that a cycle would lead to a contradiction.

4.7. Let w_e be the weight of edge e. Let $d_T(u, v)$ be the distance between u and v along the edges of T. This can be computed in linear time by either BFS or DFS. For any edge $(u, v) = e \in E - E'$, i.e. every e not belonging to the tree, check that

$$w_{(u,v)} + d_T(s, u) \geq d_T(s, v)$$

This will succeed for all $e \in E - E'$ if and only if T is a correct shortest path tree from s. We proceed to prove this.

If T is a correct shortest path tree $d_T(s, v) = d_G(s, v)$ and hence $w_{(u,v)} + d_T(s, u) \geq d_T(s, v)$ or the path through (u, v) would be shorter. If T is not a correct shortest path, consider a run of the Bellman Ford algorithm for shortest paths starting at s. Consider the first update $\texttt{update}(u, v)$ causing the distance of v to drop below $d_T(s, v)$. Let d_u be the distance label of u at the moment of that update. As this was the first update contradicting d_T, it must be the case that $d_T(s, u) \leq d_u$. Then, we have $d_T(s, u) + w_{(u,v)} \leq d_u + w_{(u,v)} < d_T(s, v)$, by construction. Hence, our algorithm will detect such edge (u, v) and correctly recognize T as an invalid shortest path tree from s.

4.8. The weighted graph in Figure 4 is a counterexample: According to the algorithm proposed by Professor

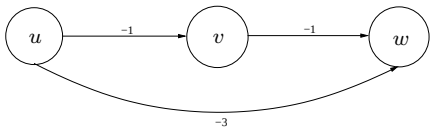


Figure 4: Counterexample for 4.8.

Lake we should add +4 to the weight of each edge. Then, the shortest path between u and w would be the edge (u, w) of weight 1. However, the shortest path in the original graph was $u - v - w$.

4.9. As stated the answer is negative, because there could be a negative cycle involving s. However, it is more interesting to ask what happens in no such cycle exists. In this case, Dijkstra's algorithm works. Consider the proof of Dijkstra's algorithm. The proof depended on the fact that if we know the shortest paths for a subset $S \subseteq V$ of vertices, and if (u, v) is an edge going out of S such that v has the minimum estimate of distance from s among the vertices in $V \setminus S$, then the shortest path to v consists of the (known) path to u and the edge (u, v). We can argue that this still holds even if the edges going out of the vertex s are allowed to negative. Let (u, v) be the edge out of S as described above. For the sake of contradiction, assume that the path claimed above is not the shortest path to v. Then there must be some other path from s to v which is shorter. Since $s \in S$ and $v \notin S$, there must be some edge (i, j) in this path such that $i \in S$ and $j \notin S$. But then, the distance from s to j along this path must be greater than that the estimate of v, since v had the minimum estimate. Also, the edges on the path between j and v must all have non-negative weights since the only negative edges

are the ones out of s. Hence, the distance along this path from s to v must be greater than the the estimate of v, which leads to a contradiction.

4.10. Perform k rounds of the `update` procedure on all edges.

4.11. Define matrix D so that D_{ij} is the length of the shortest path from vertex i to vertex j in the input graph. Row i of the matrix can be computed by a run of Dijkstra's algorithm in time $O(|V|^2)$. So we can calculate all of D in time $O(|V|^3)$. For any pair of vertices u, v we know that there is a cycle of length $D_{uv}+D_{vu}$ consisting of the two shortest paths between u and v and that this cycle is the shortest among cycles containing u and v. This shows that it suffices to compute the minimum $D_{uv} + D_{vu}$ over all pairs of vertices u, v to find the length of the shortest cycle. This last operation takes time $O(|V|^2)$, so the overall running time is $O(|V|^3)$.

4.12. Let u and v be the end vertices of e. Using Dijkstra's algorithm, compute shortest path lengths from u and v to all other vertices in $G - e$, the graph obtained removing edge e from G. Let $d_u(x)$ and $d_v(x)$ respectively denote the shortest path length from u and v to node x in this graph. For any node x, there exists then a cycle of size at most $d_u(x) + d_v(x) + 1$ containing e in G: this cycle consists of the non-overlapping parts of the shortest paths from u and v to x and of edge e. Moreover, if x belongs to the shortest cycle C containing e, $d_u(x) + d_v(x) + 1$ must be the length of C or a shorter cycle will exist. This shows that the length of C is the minimum over all x of $d_u(x) + d_v(x) + 1$, which can be calculated in time $O(|V|)$. Hence, the overall running time is $O(|V|^2)$ given by the two initial runs of Dijkstra's algorithm.

4.13. a) This can be done by performing DFS from s ignoring edges of weight larger than L.

b) This can be achieved by a simple modification of Dijkstra's algorithm. We redefine the distance from s to t to be the minimum over all paths p from s to u of the maximum length edge over all edges of p. Compare this with the original definition of distance, i.e. the minimum over all p of the sum of lengths of edges in p. This comparison suggests that by modifying the way distances are updated in Dijkstra we can produce a new version of the algorithm for the modified problem. It is sufficient to change the final loop to:

```
while H is not empty:
    u = deletemin(H)
    for all edges (u,v) ∈ E:
        if dist(v) > max(dist(u), l(u,v))
            dist(v) = max(dist(u), l(u,v))
            prev(v) = u
            decreasekey(H,v)
```

When implemented with a binary heap, this algorithm achieves the required running time.

4.14. Let P be shortest path from vertex u to v passing through v_0. Note that, between v_0 and v, P must necessarily follow the shortest path from v_0 to v. By the same reasoning, between u and v_0, P must follow the shortest path from v_0 and u in the reverse graph. Both these paths are guaranteed to exist as the graph is strongly connected. Hence, the shortest path from u to v through v_0 can be computed for all pairs u, v by performing two runs on Dijkstra's algorithm from v_0, one on the input graph G and the other on the reverse of G. The running time is dominated by looking up all the $O(|V^2|)$ pairs of distances.

4.15. This can be done by slightly modifying Dijkstra's algorithm in Figure 4.8. The array `usp[·]` is initialized to `true` in the initialization loop. The main loop is modified as follows:

```
while H is not empty:
    u = deletemin(H)
    for all edges (u,v) ∈ E:
```

```
if dist(v) > dist(u) + l(u,v):
    dist(v) = dist(u) + l(u,v)
    usp(v) = usp(u)
    decreasekey(H,v)
if dist(v) = dist(u) + l(u,v):
    usp(v) = false
```

This will run in the required time when the heap is implemented as a binary heap.

4.16. a) If the node at position j is the ith node on the kth level of the binary tree we have $j = 2^k + i - 1$. Its parent will then be the $\lceil \frac{i}{2} \rceil$th node on the $(k-1)$th level of the tree and will be found in position $2^{k-1} + \lceil \frac{i}{2} \rceil - 1 = 2^{k-1} + \lfloor \frac{i-1}{2} \rfloor = \lfloor \frac{j}{2} \rfloor$. Its children will be the $2i-1$th and $2i$th nodes on the $(k+1)$th level of the tree. Hence, they will be stored in positions $2^{k+1} + 2i - 2 = 2j$ and $2^{k+1} + 2i - 1 = 2j + 1$.

b) By the same reasoning as above, the parent of the node at position j will be at position $\lceil \frac{j-1}{d} \rceil$, while the children will be at position $dj + 1, dj, dj - 1, \cdots, dj - (d-2)$.

c) The procedure `siftdown` places element x at position i of h and rearranges the heap by letting x "'sift down"' until both its children have values greater than x. This is done by iteratively swapping x with the minimum of its children and takes at most time proportional to the height of the subtree rooted at i. The height of the whole tree is $\log n$, as there are n nodes in the heap, and the depth of node i is $\log i$, so that the subtree rooted at i has depth $\log n - \log i = \log\left(\frac{n}{i}\right)$. Because `makeheap` calls `siftdown` at all nodes in the tree, `makeheap` will take time:

$$\sum_{i=1}^{n} \log\left(\frac{n}{i}\right) = \log \frac{n^n}{n!}$$

By Stirling's formula (see page 53), $n! \geq \left(\frac{n}{e}\right)^n$, so:

$$\log\left(\frac{n^n}{n!}\right) \leq \log(e^n) = O(n)$$

d) In procedure `bubbleup`, p must be assigned to the index of the parent node of i, for which we gave the formula in b). In `minchild`, the minimum must be taken over all children of node i, the indices of which we gave in b).

4.17. a) Because every edge has length $\{0, \cdots, W\}$, all `dist` values will be in the range $\{0, 1, \cdots, W(|V|-1), \infty\}$, as any shortest path contains at most $|V| - 1$ edges. Hence, we can implement the heap on the `dist` values by mantaining an array of size $W(|V|-1)+2$ indexed by all possible values of `dist`, where each entry i is a pointer to a linked list of elements having `dist` value equal to i. With this implementation, we can perform `insert` operations in constant time, simply by appending the element at the beginning of the linked list corresponding to its value. So, `makeheap` will take time $O(|V|)$. Because during a run of Dijkstra's algorithm the minimum value on the heap is increasing, when we perform a `deletemin` operation we do not need to scan the whole array, but can start looking for the new minimum at the previous minimum value. This implies that, in scanning for the minimum element, we look at each array entry at most once, so that all the `deletemin` operations take time $O(W|V|)$. Finally, the `decreasekey` operation can be implemented by inserting a new copy of the element into the list corresponding to its new value, without removing the previous copies. This means that, when performing `deletemin` operations, we need to check whether the current minimum is a copy of an element already processed and, in that case, ignore it. But, because there at most $|E|$ `decreasekey` operations, there are at most $|E|$ copies we need to ignore and we only pay a penalty of time $O(|E|)$. Moreover, in this way, each `decreasekey` takes time $O(1)$, so that all `decreasekey` operations take time $O(|E|)$. This shows that the running time of Dijkstra's algorithm with this implementation is $O(|V|) + O(W|V|) + O(|E|) = O(W|V| + |E|)$.

b) The key observation is that there are at most $W+2$ distinct `dist` values in the heap at any one time, namely ∞ or any integer between *Min* (the current smallest value in the heap) and *Min*+W. We can now implement the heap using a binary heap with at most $W+2$ leaf nodes, where keys are possible `dist` value (including ∞), and the value associated with the key is a linked list that contains all nodes with that `dist` value. We need to make a few modifications to Dijkstra's algorithm. When we update a node's `dist` value to j, we simply append this node to the linked list at the key value j (again, we do not remove the node from the linked list corresponding to the old `dist` value). If we take out a node v during `deleteMin` that has been processed, we simply ignore the node and move on. We can do now `deletemin` and `insert` in $O(\log W)$ time, since the heap has at most $W+2$ children at any one time. We also perform at most $|V|+|E|$ `deletemin`'s and at most $|V|+|E|$ `insert`'s, hence the total running time of the algorithm is $O((|V|+|E|)\log W)$.

4.18. This is another simple variation of Dijkstra's algorithm of Figure 4.8. In the initialization loop, `best`(s) is set to 0 and all other entries of `best` are set to ∞. The main loop is modified as follows:

```
while H is not empty:
    u = deletemin(H)
    for all edges (u,v) ∈ E:
        if dist(v) > dist(u) + l(u,v):
            dist(v) = dist(u) + l(u,v)
            best(v) = best(u) + 1
            decreasekey(H,v)
        if dist(v) = dist(u) + l(u,v):
            if best(v) < best(u) + 1:
                best(v) = best(u) + 1
```

This has the same asymptotic running time as the original Dijkstra's algorithm, as the additional operations in the loop take constant time.

4.19. There are two approaches: one is a *reduction*; the other is a direct modification of Dijkstra's algorithm.

METHOD I: The idea is to use a *reduction*: on input (G,l,c,s), we construct a graph $G'=(V',E')$ where G only has edge weights (no node weights), so that the shortest path from s to t in G is essentially the same as that in G', with some minor modifications. We can then compute shortest paths in G' using Dijkstra's algorithm.

The reduction works by taking every vertex v of G and splitting it into two vertices v_i and v_o. All edges coming into v now come into v_i, while all edges going out of v now go out of v_o. Finally, we add an edge from v_i to v_o of weight $c(v)$.

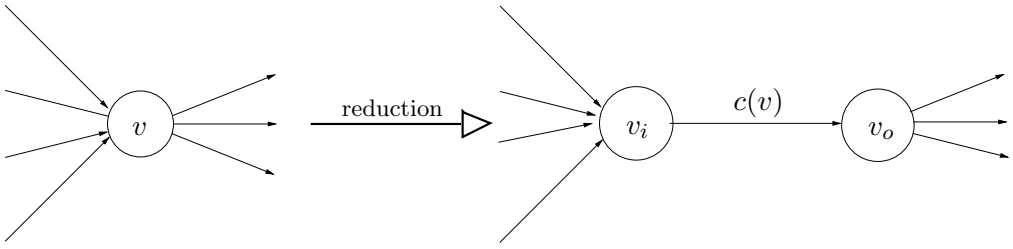


Figure 5: Reduction in 4.19.

Consider now any path in G and notice that it can be converted to an edge-weighted path of the same weight in G' by replacing the visit to vertex v with the traversal of edge (v_o,v_i). Conversely,

consider a path in G': every other edge visited is of the form (v_i, v_o) and corresponds to a vertex v of G. Replacing these edges with the corresponding vertices we obtain a path in G of the same weight as the path in G'. The time required to perform this reduction is $O(|V| + |E|)$. G' has $|V| + |E|$ edges and $2|V|$ vertices, so running Dijkstra takes time $O(|V|^2)$ and the total running time is $O(|V|^2)$.

Method II: We make the following modifications to Dijkstra's algorithm to take into account node weights:

- In the initialization phase, $\texttt{dist(s)} = \texttt{w(s)}$.
- In the update phase, we use $\texttt{dist(u)} + \texttt{l(u,v)} + \texttt{w(v)}$ instead of $\texttt{dist(u)} + \texttt{l(u,v)}$.

Analysis of correctness and running time are exactly the same as in Dijkstra's algorithm.

4.20. G is an undirected graph with edge weights l_e. If the distance between s and t decreases with the addition of $e' = (u, v)$, the new shortest path from s to t will be the concatenation of the shortest path from s to u, the edge (u, v) and the shortest path from v to t. But we can then compute the length of this path by running Dijkstra's algorithm once from s and once from t in G. With all the shortest path distances from s and t in G, we can compute in constant time the length of the shortest path from s to t going through e' for any $e' \in E'$. The shortest of these paths will give us the best edge to add and its length will tell us what improvement the addition brings, if any. The running time of this algorithm is $O(|V|^2 + |E'|)$.

4.21. a) Represent the currencies as the vertex set V of a complete directed graph G. To find the most advantageous ways to converts c_s into c_t, you need to find the path $c_{i_1}, c_{i_2}, \cdots, c_{i_k}$ maximizing the product $r_{i_1,i_2} r_{i_2,i_3} \cdots \cdot r_{i_{k-1},i_k}$. This is equivalent to minimizing the sum $\sum_{j=1}^{k-1}(-\log r_{i_j,i_{j+1}})$. Hence, it is sufficient to find a shortest path in the graph G with weights $w_{ij} = -\log r_{ij}$. Because these weights can be negative, we apply the Bellman-Ford algorithm for shortest paths to the graph , taking s as origin.

b) Just iterate the updating procedure once more after $|E||V|$ rounds. If any distance is updated, a negative cycle is guaranteed to exist, i.e. a cycle with $\sum_{j=1}^{k-1}(-\log r_{i_j,i_{j+1}}) < 0$, which implies $\prod_{j=1}^{k-1} r_{i_j,i_{j+1}} > 1$, as required.

4.22. a) Let C be the negative cycle, $E(C)$ the set of edges of C, $V(C)$ the set of vertices. Then:

$$\sum_{e \in E(C)} w_e = r \sum_{e \in E(C)} c_e - \sum_{v \in V(C)} p_v < 0$$

This shows that $\frac{\sum_{v \in V(C)} p_v}{\sum_{e \in E(C)} c_e} > r$, so that $r < r^*$.

b) The same argument as in a) yields that all cycles have ratio less than r, so that $r > r^*$.

c) We can use Bellman-Ford to detect negative cycles, so, for any r, we can check in time $O(|V||E|)$ whether r is smaller or greater than r^*. We can then perform a binary search for r^* on the interval $[0, R]$. After $\log\left(\frac{R}{\epsilon}\right)$ rounds of binary search, our lower bound r' on r^* will be at most ϵ smaller than r^*, i.e. $r' \geq r^* - \epsilon$. Consider now the weighted graph $G_{r'}$ obtained by setting the weights as above with $r = r'$. Because $r' < r^*$, the optimal cycle C^* with ratio r^* will appear as a negative cycle in $G_{r'}$. Hence, when we run Bellman-Ford on $G_{r'}$, it will detect some negative cycle C (notice C is not necessarily equal to C^*). Then, by a), the profit-cost ratio of C will be greater than r', i.e. $r(C) > r' \geq r^* - \epsilon$. This algorithm requires $\log\left(\frac{R}{\epsilon}\right) + 1$ runs on Bellman-Ford on different weighted versions of G. Its total running time is then $O(\log\left(\frac{R}{\epsilon}\right)|V||E|)$.

Chapter 5 - Solutions

5.1 (a) 19

(b) 2

(c)

Edge included	Cut
AE	$\{A, B, C, D\}$ & $\{E, F, G, H\}$
EF	$\{A, B, C, D, E\}$ & $\{F, G, H\}$
BE	$\{A, E, F, G, H\}$ & $\{B, C, D\}$
FG	$\{A, B, E\}$ & $\{C, D, F, G, H\}$
GH	$\{A, B, E, F, G\}$ & $\{C, D, H\}$
CG	$\{A, B, E, F, G, H\}$ & $\{C, D\}$
GD	$\{A, B, C, E, F, G, H\}$ & $\{D\}$

5.2 (a)

Vertex included	Edge included	Cost
A		0
B	AB	1
C	BC	3
G	CG	5
D	GD	6
F	GF	7
H	GH	8
E	AE	9

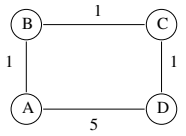
5.3 Since the graph is given to be connected, it will have an edge whose removal still leaves it connected, if and only if it is not a tree i.e. has more than $|V| - 1$ edges. We perform a DFS on the graph until we see $|V|$ edges. If we can find $|V|$ edges then the answer is "yes" else it is "no". In either case, the time taken is $O(|V|)$.

5.4 Let e_i, n_i denote the number of edges and vertices in the ith component. Since a connected graph on t vertices must have at least $t - 1$ edges,

$$|E| = \sum_{i=1}^{k} e_i \geq \sum_{i=1}^{k} (n_i - 1) = n - k$$

5.5 (a) The minimum spanning tree does not change. Since, each spanning tree contains exactly $n - 1$ edges, the cost of each tree is increased $n - 1$ and hence the minimum is unchanged.

(b) The shortest paths may change. In the following graph, the shortest path from A to D changes from $AB - BC - CD$ to AD if each edge weight is increased by 1.



5.6 Suppose the graph has two different MSTs T_1 and T_2. Let e be the lightest edge which is present in exactly one of the trees (there must be some such edge since the trees must differ in at least one edge). Without loss of generality, say $e \in T_1$. Then adding e to T_2 gives a cycle. Moreover, this cycle must contain an edge e' which is (strictly) heavier that e, since all lighter edges are also present in T_1, where e does not induce a cycle. Then adding e to T_2 and removing e' gives a (strictly) better spanning tree than T_2 which is a contradiction.

5.7 Multiply the weights of all the edges by -1. Since both Kruskal's and Prim's algorithms work for positive as well as negative weights, we can find the minimum spanning tree of the new graph. This is the same as the maximum spanning tree of the original graph.

5.8 Consider the edge $e = (s, u)$ having the minimum weight among all the edges incident on s. Since all the edge weights are positive, this is the unique shortest path from s to u and hence must be present in any shortest path tree. Also, Prim's algorithm includes this edge in construction of the MST. Since by problem 5.6, the MST must be unique (because of distict edge weights), all the shortest path trees and the MST must share at least one edge.

5.9 (a) False, consider the case where the heaviest edge is a bridge (is the only edge connecting two connected components of G.

(b) True, consider removing e from the MST and adding another edge belonging to the same cycle. Then we get e new tree with less total weight.

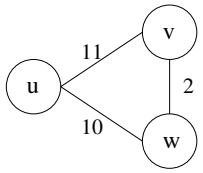
(c) True, e will belong to the MST produced by Kruskal.

(d) True, if not there exists a cycle connecting the two endpoints of e, so adding e and removing another edge of the cycle, produces a lightest tree.

(e) True, consider the cut that has u in one side and v in the other, where $e = (u, v)$.

(f) False,assume the graph consists of two adjacent 4-cycles,the one with very heavy edges of weight M and the other with very light edges of weight m. Let e be the edge in the middle with weight $m < w(e) < M$. Then the MST given by Kruskal, will pick all edges of weight m first and will not include e in the MST.

(g) False. In the following graph, the MST has edges (u, w) and (v, w) while Dijkstra's algorithm gives $(u, v), (u, w)$.



(h) False. In the previous graph the shortest path between u and v is the edge (u, v) but the only MST is $(u, w), (v, w)$

(i) True

(j) True. Suppose that the path from s to t has an edge longer than r. Then one of the edges of the r-path must be absent (otherwise there is cycle). Including this and removing the longer edge gives a better tree which is a contradiction.

5.10 Let $T \cup H = \{e_1, \ldots, e_k\}$. We use the cut property repeatedly to show that there exists an MST of H containing $T \cap H$.

Suppose for $i \geq 0$, $X = \{e_1, \ldots, e_i\}$ is contained in some MST of H. Removing the edge e_{i+1} from T divides T in two parts giving a cut $(S, G \backslash S)$ and a corresponding cut $(S_1, H \backslash S_1)$ of H with $S_1 = S \cap H$. Now, e_{i+1} must be the lightest edge in G (and hence also in H) crossing the cut , else we can include the lightest and remove e_{i+1} getting a better tree. Also, no other edges in T, and hence also in X, cross this cut. We can then apply the cut property to get that $X \cup e_{i+1}$ must be contained in some MST of H. Continuing in this manner, we get the result for $T \cap H = \{e_1, \ldots, e_k\}$.

5.11 The figure below shows the data-structure after the operations.

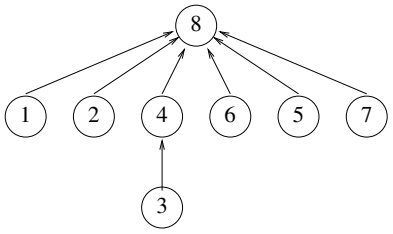
5.12 For the sake of convenience, assume that in case of a tie, we make the *higher* numbered root point to the *lower* numbered root. Let $n = 2^k$. Consider the following operations starting from the singleton sets $\{1\}, \ldots, \{2^k\}$

`union(1,2), union(3,4), ...,` `union`$(2^k - 1, 2^k)$

`union(2,4), union(6,8), ...,` `union`$(2^k - 2, 2^k)$

$\vdots$

`union`$(2^i, 2 \cdot 2^i)$`, union`$(3 \cdot 2^i, 4 \cdot 2^i)$`, ...,` `union`$(2^k - 2^i, 2^k)$



$\vdots$

`union(`$2^{k-1}, 2^k$`)` The above are $O(2^k)$ operations which make a tree in which the depth of element i is $\lfloor \log i \rfloor$ (the depth of element i increases in the first j steps, if 2^j is the largest power of 2 less that i). The cost of the above is $\sum_{i=1}^{k-1} i \cdot 2^{k-i} = O(2^k)$. We now perform a `find` for every element.
`find(1), ..., find(`2^k`)`
The cost of the find operations is $\sum_{i=1}^{2^k} \lfloor \log i \rfloor = O(k2^k) = O(n \log n)$. Hence, the above is a sequence of $O(n)$ operations taking time $O(n \log n)$.

5.13 Huffman's algorithm assigns codewords of length 1 to D, length 2 to A and length 3 to B and C. So, one possible encoding can be 0 for D, 10 for A, 110 for B and 111 for C.

5.14 (a) $a \rightarrow 0$, $b \rightarrow 10$,$c \rightarrow 110$, $d \rightarrow 1110$, $e \rightarrow 1111$.

(b) length $= \frac{1000000}{2} \cdot 1 + \frac{1000000}{4} \cdot 2 + \frac{1000000}{8} \cdot 3 + 2 \cdot \frac{1000000}{16} \cdot 4 = 1875000$

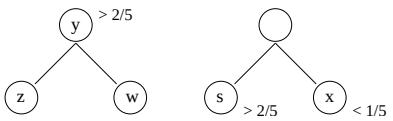
5.15 (a) $(f_a, f_b, f_c) = (2/3, 1/6, 1/6)$ gives the code $\{0, 10, 11\}$.

(b) This encoding is not possible, since the code for a (0), is a prefix of the code for c (00).

(c) This code is not optimal since $\{1, 01, 00\}$ gives a shorter encoding. Also, it does not correspond to a *full* binary tree and hence cannot be given by the Huffman algorithm.

5.16 a) Let s be the symbol with the highest frequency (probability) $p(s) > 2/5$ and suppose that it merges with some other symbol during the process of constructing the tree and hence does not correspond to a codeword of length 1. To be merged with some node, the node s and some other node x must be the two with minimum frequencies. This means there was at least one other node y (formed by merging of other nodes), with $p(y) > p(s)$ and $p(y) > p(x)$. Thus, $p(y) > 2/5$ and hence $p(x) < 1/5$.

Now, y must have been formed by merging some two nodes z and w with at least one of them having probability greater than $1/5$ (as they add up to more than $2/5$). But this is a contradiction - $p(z)$ and $p(w)$ could not have been the minimum since $p(x) < 1/5$.



b) Suppose this is not the case. Let x be a node corresponding to a single character with $p(x) < 1/3$ such that the encoding of x is of length 1. Then x must not merge with any other node till the end. Consider the stage when there are only three leaves - x, y and z left in the tree. At the last stage y, z must merge to form another node so that x still corresponds to a codeword of length 1. But, $p(x) + p(y) + p(z) = 1$ and $p(x) < 1/3$ implies $p(y) + p(z) > 2/3$. Hence, at least one of $p(y)$ or $p(z)$, say $p(z)$, must be greater than $1/3$. But then these two cannot merge since $p(x)$ and $p(y)$ would be the minimum. This leads to a contradiction.

5.17 The longest codeword can be of length $n - 1$. An encoding of n symbols with $n - 2$ of them having probabilities $1/2, 1/4, \ldots, 1/2^{n-2}$ and two of them having probability $1/2^{n-1}$ achieves this value.

5.18 ———————————

5.19 (a) It is enough to show that a symbol with frequency $p_i = 2^{-k_i}$ will be encoded with $log\frac{1}{p_i} = k_i$ bits. This is easily proved by induction : since all frequencies are a power of 1/2, the two least probable symbols will have the same probability $f = \frac{1}{2^k}$. Merging them according to the Huffman procedure into a new symbol, of probability $\frac{1}{2^{k-1}}$, will allow us to use our inductive hypothesis for the new set of $n-1$ symbols. The length of the first $n-2$ symbols is the same in the old and new tree, equal to $log\frac{1}{p_i}, i = 1, \cdots n-2$ and the length of the last two in the old tree is $1 + log\frac{1}{2^{k-1}} = k - 1 + 1 = k$.

(b) The largest possible entropy is when all the symbols have the same probability ,$1/n$. The entropy in that case is $logn$.The entropy is smallest when there exists a unique symbol with probability 1 (and all the others have probability 0). The entropy in that case is 0.

5.20 For each leaf of the tree, the edge incident on the leaf must be in the matching. Also, if we remove all these leaves, the leaves of the resulting tree have already been covered and the edges incident on them must *not* be in the matching. We can then construct the matching bottom up, including and excluding edges at alternate steps. If at any step, we need to include two edges incident on the same vertex, then no matching exists.

This can be done in linear time by maintaining the degree of every vertex and maintaining the leaves (nodes with degree 1) in a list. At each step, we update the degree of the endpoints of edges that we delete.

5.21 For a connected graph, removing the feedback arc set leaves a spanning tree. Hence, to find the minimum feedback arc set, we need to find the maximum spanning tree for every connected component of G. To do that, we run Kruskal's algorithm, negating the edge weights of the original graph. Once we have the Maximum Spanning Tree T, we output E' = the set of the edges that don't belong to T. Running time is the same as Kruskal.

5.22 (a) Consider an MST T which contains e. Removing e breaks the tree into two connected components say S and $V \backslash S$. Since all the vertices of the cycle cannot still be connected after removing e, at least one edge, say e' in the cycle must cross from S to $V \backslash S$. However, then replacing e by e' gives a tree T' such that $cost(T') \leq cost(T)$. Since T is an MST, T' is also an MST which does not contain e.

(b) If e is the heaviest edge in some cycle of G, then there is some MST T not containing e. However, then T is also an MST of $G - e$ and so we can simply search for an MST of $G - e$. At every step, the algorithm creates a new graph $(G - e)$ such that an MST of the new graph is also an MST of the old graph (G). Hence the output of the algorithm (when the new graph becomes a tree) is an MST of G.

(c) An undirected edge (u, v) is part of cycle iff u and v are in the same connected component of $G - e$. Since the components can be found by DFS (or BFS), this gives a linear time algorithm.

(d) The time for sorting is $O(|E| \log |E|)$ and checking for a cycle at every step takes $O(|E|)$ time. Finally, we remove $|E| - |V| + 1$ edges and hence the running time is $O(|E| \log |E| + (|E| - |V|) * |E|) = O(|E|^2)$.

5.23 To solve this problem, we shall use the characterization that T is an MST of G, if and only if for every cut of G, at least one least weight edge across the cut is contained in T.

The "only if" direction is easy since if the lightest edge across a cut is not in T, then we can include it and remove some edge in T that crosses the cut (there must be at least one) to get a better tree. To prove the "if" part, note that at each in Prim's algorithm, we include the lightest edge across some cut, which can be chosen from T. Since T is a possible output of Prim's algorithm, it must be an MST.

(a) Since the change only increases the cost of some other spanning trees (those including e) and the cost of T is unchanged, it is still an MST.

(b) We include e in the tree, thus creating a cycle. We then remove the heaviest edge e' in the cycle, which can be found in linear time, to get a new tree T'. It is intuitively clear that this algorithm should work, but a rigorous proof seems surprisingly tricky. To prove this, we argue that T' contains a least weight edge across every cut of G and is hence an MST.

Note that since the only changed edge is e, $T \cup \{e\}$ already includes a least weight edge across every cut. We only removed e' from this. However, any cut crossed by e', must also be crossed by at least one more edge of the cycle, which must have weight less than or equal to e'. Since this edge is still present in T', it contains a least weight edge across every cut.

(c) The tree is still an MST if the weight of an edge in the tree is reduced. Hence, no changes are required.

(d) We remove e from the tree to obtain two components, and hence a cut. We then include the lightest edge across the cut to get a new tree T'. We can now "build up" T' using the cut property to show that it is an MST.

Let $X \subseteq T'$ be a set of edges that is part of some MST, and let $e_1 \in T' \backslash X$. Then, $T' \backslash \{e_1\}$ gives a cut which is not crossed by any edge of X and across which e_1 is the lightest edge. Hence, $X \cup \{e_1\}$ is also a part of some MST. Continuing this, we can grow X to $X = T'$, which must be then an MST.

5.24 We first note that each $u \in U$ must have at least one neighbor in $V \backslash U$, else the problem has no solution. If T is the optimal tree, then $T \backslash U$ must be a spanning tree of $G \backslash U$. Moreover, it must be a minimum spanning tree since the nodes in U can be attached as leaves to *any* spanning tree. Hence, we first find an MST of $G \backslash U$ (in time $O(|E| \log |V|)$) and then for each $u \in U$, we add the lightest edge between u and $G \backslash U$ (in time $O(|E|)$).

5.25 To implement a counter of unspecified length, suppose we maintain it as a list of bits, with a pointer to the most significant bit. Each time we need to increase the length of the binary string, we can add a new element to the front of the list and update this pointer.

To increment, we set the first 0 from the right to 1 and set all the 1s to the right of it to 0. Since we start from the all 0 string, each bit is set to 1 once before it is set to 0. Hence, the total cost of k increments is at most twice the number of bits set to 1, which is $O(k)$ since each increment sets exactly one bit to 1.

Finally, note that the cost of a reset (setting all bits to 0), is at most the number of significant digits (if we maintain a pointer to the most significant 1 bit). This is at most the number of increment operations since the previous reset operation. Hence, instead of charging a reset operation, we double the cost of each increment operation which gives the cost of n increment and reset operations as $O(n)$.

5.26 We construct two graphs $G_{eq} = (V, E_{eq})$ and $G_{neq} = (V, E_{neq})$ where V $\{1, \ldots, n\}$. We have $(i, j) \in E_{eq}$ if $x_i = x_j$ is a constraint and $(i, j) \in E_{neq}$ if $x_i \neq x_j$ is a constraint. Since equality is an equivalence relation, all the variables in each connected component of G_{eq} must have the same value. The system of constraints is satisfiable if and only if there is no edge $(i, j) \in E_{neq}$ such that x_i and x_j are in the same component in G_{eq}. The decomposition in components and checking for edges can both be done in $O(m + n)$ time.

5.27 (a) $(3, 3, 1, 1)$ and $(100, 2, 1, 1)$ are both valid examples.

(b) i. Suppose the neighbors of v_1 in G are not $v_2, v_3, \ldots, v_{d_1+1}$. Therefore, there is some $i \in \{2, 3, \ldots, d_1 + 1\}$ such that $(v_1, v_i) \notin E$. In addition, since v_1 has d_1 neighbors, there must be some $j > d_1 + 1$ such that $(v_1, v_j) \in E$. In particular, $2 \leq i < j$. Next, note that the vertices v_i and v_j have d_i and $d_j - 1$ neighbors in $V - \{d_1\}$ respectively. Now, $i < j$ implies $d_i \geq d_j \geq 1$. Hence, $d_i > d_j - 1$, so there exists some vertex $u \in V - \{d_1\}$ that is a neighbor of v_i but not of v_j.

ii. Delete the edges (v_1, v_j) and (u, v_i) and add the edges (v_1, v_i) and (u, v_j).

iii. We repeatedly apply the above argument till v_1 has neighbors $v_2, \ldots, v_{d+1}$.

(c) From (b), we know that there exists a graph on n vertices with degree sequence $(d_1, \ldots, d_n)$ iff there exists a graph on $n-1$ vertices with degree sequence $(d_2 - 1, \ldots, d_{d_1+1} - 1, d_{d_1+2}, \ldots, d_n)$. The algorithm is as follows: sort (and relabel) the numbers $d_1, \ldots, d_n$ in decreasing order. Remove d_1 from the list; decrease $d_2, \ldots, d_{d_1+1}$ by 1 (output "no" if $d_1 + 1 > n$); remove any 0's in the resulting list, and recurse on the list of at most $n-1$ numbers. The base case is the empty list, where you output "yes". The running time is $O(n^2+m)$ (since you can sort numbers in a bounded range in linear time); with the appropriate data structure, you can avoid sorting at every stage, and obtain a $O(m+n)$ running time.

5.28 We construct a graph $G = (V, E)$ as follows. The vertex set consists of a vertex for each person. An edge $e = (u, v)$ between two vertices represents the relation 'person v knows person u' . The problem reduces of finding a subset V' of V such that in the induced graph each vertex has degree more than 5 and less than $|V'| - 5$. We do that iteratively : at the beginning, we look at all nodes and delete those with degree more than $|V| - 5$ and those with degree less than 5. We update our graph to be the induced graph with vertex set the remaining vertices. Then, we do the same procedure again (with the updated degrees for every node). We repeat the above until we end up with a graph G' where after running the procedure, we don't delete any vertices. The algorithm takes time $O(n)$ for every iteration of the above procedure, therefore a total of $O(n^2)$ time.

5.29 If we consider the binary tree with all strings of length k at level k and the left and right branches representing adding a 0 and 1 respectively, it contains all the binary strings and hence all the strings in the encoding (we only need to have number of levels equal to maximum length of a string). Also, since all intermediate nodes in a path from the root to a node v are prefixes of v, the strings of the encoding must be all at leaves since it is prefix-free.

To argue that the tree must be full, suppose for contradiction that a node u corresponding to a string s has only one child v corresponding to $s0$. Since a codeword is a leaf in the subtree rooted at u iff it has s, and hence also $s0$ as a prefix, replacing $s0$ by s in all these codewords gives a better encoding. However, this is a contradiction since we assumed our encoding to be minimal.

5.30 We first prove that if the number of symbols is odd then the encoding must correspond to a full ternary tree. We then make the number of symbols odd by adding a symbol with frequency 0 if needed. Once it is guaranteed that the optimal tree is full, we can simply proceed as in the binary case combining the three nodes with least frequencies at every step to form a single node. Since this reduces the number of nodes by 2, it still remains odd and we can carry on.

Suppose now that the number of symbols is odd and some non-leaf node u in the optimal tree does not have three children. Without loss of generality, we can assume that all children of u are leaves (else we can remove a leaf attached to some other node and add it as a child of u) and that u has two children (else we can delete its only child getting a better code).

Removing all children of nodes (say $u_1, \ldots, u_k$) which have two children, gives a full ternary tree. A full ternary tree can be built by always adding 3 children to some existent leaf and hence must have an odd number of leaves, since we started with just the root and the number of leaves increased by two at each step. Now, adding the children of $u_1, \ldots, u_k$ adds k more leaves (removes k and adds $2k$). Since the final number of leaves must still be odd, k must be even. We can then pair up $u_1, \ldots, u_k$ and transfer children between the pair so that half of them have 3 children, while the other half has 1 each. For each node u having only one child leaf v, we can now delete v getting a better code, which is a contradiction.

5.31 The algorithm in the chapter minimizes the sum $\sum_i f_i l_i$. To minimize $\sum_i c_i f_i l_i$, we simply treat $f_i c_i$ as the frequency of the ith word.

5.32 We simply proceed by a greedy strategy, by sorting the customers in the increasing order of service times and servicing them in this order. The running time is $O(n \log n)$. To prove the correctness, for any ordering of the customers, let $s(j)$ denote the jth customer in the ordering. Then

$$T = \sum_{i=1}^{n} \sum_{j=1}^{i-1} t_{s(j)} = \sum_{i=1}^{n} (n-i) t_{s(i)}$$

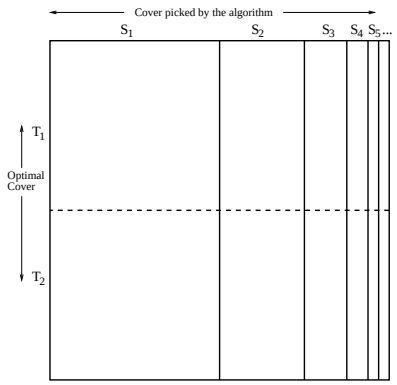
For any ordering, if $t_{s(i)} > t_{s(j)}$ for $i < j$, then swapping the positions of the two customers gives a better ordering. Since, we can generate all possible orderings by swaps, an ordering which has the property that $t_{s(1)} \leq \ldots \leq t_{s(n)}$ must be the global optimum. However, this is exactly the ordering we output.

5.33 We create a graph with a node for every variable. Also, for every implication, say $(x \wedge y \wedge z) \implies w$, we add the directed edges $(x, w), (y, w)$ and (z, w). With the edges, we also store which clause they belong to. Finally, at each variable we store a counter corresponding each clause in which appears, which is initially set to the number of variables on the LHS of the implication.

We start with assigning the value `false` to all the variables and then making some true as required by the implications. For each variable that is set to true, we decrement the counters of all its neighbors corresponding to the respective clauses. Any variable for which the counter corresponding to any clause becomes zero, is added to a list L, which contains the variable to be set to true. The list L starts with variables which start with counters zero i.e. clauses of the form (x).

5.34 Consider the base set $U = \{1, 2, \ldots, 2^k\}$ for some $k \geq 2$. Let $T_1 = \{1, 3, \ldots, 2^k - 1\}$ and $T_2 = \{2, 4, \ldots, 2^k\}$. These two sets comprise an optimal cover. We add sets $S_1, \ldots, S_{k-1}$ by defining $l_i = 2 + \sum_{j=1}^{i} 2^{k-j}$ and letting $S_i = \{l_{i-1} + 1, \ldots, l_i\}$ (take $l_0 = 0$).

Thus, S_1 contains $2^{k-1} + 2$ elements and the greedy algorithm picks this first. After the algorithm has picked i sets, each of T_1 and T_2 covers $2^{k-i-1} - 1$ new elements while S_{i+1} covers 2^{k-i-1} new elements. Hence, the algorithm picks the cover $S_1, \ldots, S_{k-1}$ containing $k - 1 = \log n - 1$ sets.



5.35 Since the algorithm in the box titled "A randomized algorithm for minimum cut" produces every minimum cut with probability at least $\frac{1}{n(n-1)}$ and the probabilities must sum to 1, the number of minimum cuts is at most $n(n-1)$.

Chapter 6– Solutions

6.1. *Subproblems:* Define an array of subproblems $D(i)$ for $0 \leq i \leq n$. $D(i)$ will be the largest sum of a (possibly empty) contiguous subsequence ending exactly at position i.

Algorithm and Recursion: The algorithm will initialize $D(0) = 0$ and update the $D(i)$'s in ascending order according to the rule:

$$D(i) = \max\{0, D(i-1) + a_i\}$$

The largest sum is then given by the maximum element $D(i)^*$ in the array D. The contiguous subsequence of maximum sum will terminate at i^*. Its beginning will be at the first index $j \leq i^*$ such that $D(j-1) = 0$, as this implies that extending the sequence before j will only decrease its sum.

Correctness: The contiguous subsequence of largest sum ending at i will either be empty or contain a_i. In the first case, the value of the sum will be 0. In the second case, it will be the sum of a_i and the best sum we can get ending at $i-1$, i.e. $D(i-1) + a_i$. Because we are looking for the largest sum, $D(i)$ will be the maximum of these two possibilities.

Running Time: The running time for this algorithm is $O(n)$, as we have n subproblems and the solution of each can be computed in constant time. Moreover, the identification of the optimal subsequence only requires a single $O(n)$ time pass through the array D.

6.2. *Subproblems:* Define subproblem $D(i)$ for $0 \leq i \leq n$ to be the minimum total penalty to get to hotel i.

Algorithm and Recursion: The algorithm will initialize $D(0) = 0$ and compute the remaining $D(i)$ in ascending order using the recursion:

$$D(i) = \min_{0 \leq j < i} \{(200 - a_j)^2 + D(j)\}$$

To recover the optimal itinerary, we can keep track of a maximizing j for each $D(i)$ and use this information to backtrack from $D(n)$.

Correctness: To solve $D(i)$ we consider all possible hotels j we can stay at on the night before reaching hotel i: for each of these possibilities, the minimum penalty to reach i is the sum of the cost of a one-day trip from j to i and the minimum penalty necessary to reach j. Because we are interested in the minimum penalty to reach i, we take the minimum of these values over all j's.

Running Time: The running time is $O(n^2)$, as we have n subproblems and each takes time $O(n)$ to solve, as we need to compute the minimum of $O(n)$ values. Moreover, backtracking only takes time $O(n)$.

6.3. *Subproblems:* Define subproblem $D(i)$ to be the maximum profit which Yuckdonald's can obtain from locations 1 to i.

Algorithm and Recursion: The algorithm will initialize $D(0) = 0$, and use the following rule to solve the other subproblems:

$$D(i) = \max\{D(i-1), p_i + D(i^*)\}$$

where i^* is the largest index j such that $m_j \leq m_i - k$, i.e. the first location preceding i and at least k miles apart from it.

Correctness: Note that, if location i is not used, the maximum profit $D(i)$ must equal $D(i-1)$. Otherwise, if location i is used, the best we can hope for is the sum of p_i and the maximum profit from the remaining locations we are still allowed to open before i, i.e. $D(i^*)$.

Running Time: This algorithm solves n subproblems; each subproblem requires finding an index i^*, which can be done in time $O(\log n)$ by binary search on the ordered list of locations, and computing a maximum of two values, which can be done in constant time. Hence, the running time is $O(n \log n)$.

6.4. a) *Subproblems:* Define an array of subproblems $S(i)$ for $0 \leq i \leq n$ where $S(i)$ is 1 if $s[1 \cdots i]$ is a sequence of valid words and is 0 otherwise.

Algorithm and Recursion: It is sufficient to initialize $S(0) = 1$ and update the values $S(i)$ in ascending order according to the recursion

$$S(i) = \max_{0 \leq j < i} \{S(j) : \texttt{dict}(s[j+1 \cdots i]) = \texttt{true}\}$$

Then, the string s can be reconstructed as a sequence of valid words if and only if $S(n) = 1$.

Correctness and Running Time: Consider $s[1 \cdots i]$. If it is a sequence of valid words, there is a last word $s[j \cdots i]$, which is valid, and such that $S(j) = 1$ and the update will cause $S(i)$ to be set to 1. Otherwise, for any valid word $S[j \cdots i]$, $S(j)$ must be 0 and $S(i)$ will also be set to 0. This runs in time $O(n^2)$ as there are n subproblems, each of which takes time $O(n)$ to be updated with the solution obtained from smaller subproblems.

b) Every time a $S(i)$ is updated to 1 keep track of the previous item $S(j)$ which caused the update of $S(i)$ because $s[j+1 \cdots i]$ was a valid word. At termination, if $S(n) = 1$, trace back the series of updates to recover the partition in words. This only adds a constant amount of work at each subproblem and a $O(n)$ time pass over the array at the end. Hence, the running time remains $O(n^2)$.

6.5. a) There are 8 possible patterns: the empty pattern, the 4 patterns which each have exactly one pebble, and the 3 patterns that have exactly two pebbles (on the first and fourth squares, the first and third squares, and the second and fourth squares).

b) Number the 8 patterns 1 through 8, and define $S \subseteq \{1, 2, \ldots, 8\} \times \{1, 2, \ldots, 8\}$ to be all (a, b) such that pattern a is compatible with pattern b. For each pattern, there are a constant number of patterns that are compatible (for example, every pattern is compatible with the empty pattern).

Sub-problems and Recursion: We consider the sub-problem $L[i, j]$, $i = 0, 1, 2, \ldots, n$ and $j \in \{1, 2, \ldots, 8\}$ to be the maximal value achievable by pebbling columns $1, 2, \ldots, i$ such that the final column has pattern j. It is easy to see that:

$$L[i+1, j] = \max_{(k,j) \in S} L[i, k]$$

The base case is $L[0, j] = 0$ for all j. In order to recover the optimal placement, we should also maintain a back-pointer: $P[i+1, j]$ is the value of k such that $(k, j) \in S$ and $L[i, k]$ is maximal.

Algorithm and Running Time: For $i = 0, 1, \ldots, n$, for $j = 1, 2, \ldots, 8$, compute $L[i, j]$ and $P[i, j]$ using the recurrence. Note that $L[i, j]$ and $P[i, j]$ can be computed using the recursion in constant time since we only need to check a constant number of possible k. The value of the optimal placement is given by $\max_j L[n, j]$.

To recover the optimal placement, let j^* be the value of j for which $L[n, j]$ is maximal. Then, column n should be pebbled using pattern j^*. Then, column $n-1$ should be pebbled using pattern $P[n, j^*]$, column $n-2$ with pattern $P[n-1, P[n, j^*]]$, and so on.

The running time of this algorithm is $O(n)$ because compute the recurrence takes $O(n)$ time and backtracking takes $O(1)$ time per column.

6.6. *Subproblems:* Let $x_1 \cdots x_n$ be the factors of the product in the order they appear. We define $C(i, j)$ to be the set of all possible values of the substring $x[i, \cdots, j]$ under all possible different parenthesizations.

Algorithm and Recursion: We can compute $C(i, i+s)$ by considering all positions j where we can break the expression into two products:

$$C(i, i+s) = \bigcup_{i \leq j < i+s} \{xy : x \in C(i, j), y \in C(j+1, i+s)\}$$

We can then recursively construct all $C(i, i+s)$ in increasing order of s.

Correctness and Running Time: The assignment to $C(i, i+s)$ is correct as it considers all possible ways of splitting $x_i \cdots x_{i+s}$ after x_j into two parenthesized expressions. The running time of the algorithm is $O(n^3)$ as there are $O(n^2)$ subproblems and each takes time $O(n)$.

6.7. *Subproblems:* Define variables $L(i, j)$ for all $1 \leq i \leq j \leq n$ so that, in the course of the algorithm, each $L(i, j)$ is assigned the length of the longest palindromic subsequence of string $x[i, \cdots, j]$.

Algorithm and Recursion: The recursion will then be:

$$L(i, j) = \max\{L(i+1, j), L(i, j-1), L(i+1, j-1) + \text{equal}(x_i, x_j)\}$$

where $\text{equal}(a, b)$ is 1 if a and b are the same character and is 0 otherwise, The initialization is the following:

$$\begin{aligned} \forall i, 1 \leq i \leq n \ , &\quad L(i, i) = 0 \\ \forall i, 1 \leq i \leq n-1 \ , &\quad L(i, i+1) = \text{equal}(x_i, x_i + 1) \end{aligned}$$

Correctness and Running Time: Consider the longest palindromic subsequence s of $x[i, \cdots, j]$ and focus on the elements x_i and x_j. There are then three possible cases:

- If both x_i and x_j are in s then they must be equal and $L(i, j) = L(i+1, j-1) + \text{equal}(x_i, x_j)$
- If x_i is not a part of s, then $L(i, j) = L(i+1, j)$.
- If x_j is not a part of s, then $L(i, j) = L(i, j-1)$.

Hence, the recursion handles all possible cases correctly. The running time of this algorithm is $O(n^2)$, as there are $O(n^2)$ subproblems and each takes $O(1)$ time to evaluate according to our recursion.

6.8. *Subproblems:* For $1 \leq i \leq n$ and $1 \leq j \leq m$, define subproblem $L(i, j)$ to be the length of the longest common substring of x and y *terminating* at x_i and y_j. The recursion is:

$$L(i, j) = \begin{cases} L(i-1, j-1) + 1 & \text{if equal}(x_i, y_j) = 1 \\ 0 & \text{otherwise} \end{cases}$$

The initialization is, for all $1 \leq i \leq n$ and $1 \leq j \leq m$:

$$\begin{aligned} L(0, 0) &= 0 \\ L(i, 0) &= 0 \\ L(0, j) &= 0 \end{aligned}$$

The output of the algorithm is the maximum of $L(i, j)$ over all $1 \leq i \leq n$ and $1 \leq j \leq m$.

Correctness and Running Time: The initialization is clearly correct. Hence, it suffices to prove the correctness of the recursion. The longest common substring terminating at x_i and y_j must include x_i and y_j: hence, it will be 0 if these characters are different and $L(i-1, j-1) + 1$ if they are equal. The running time is $O(mn)$ as we have mn subproblems and each takes constant time to evaluate through the recursion.

6.9. *Subproblems and Recursion:* Let the string be indexed $x_1 \cdots x_n$. Let $0 = c_0 \leq c_1 \leq c_2 \leq \cdots \leq c_m \leq c_{m+1} = n$ be the locations of the $m+2$ cuts, which include the beginning and end of the string(i.e. c_i is the index after which the ith cut takes place). Let us the define the following subproblems for all i, j such that $0 \leq i < j \leq m+1$:

$$L(i,j) = \text{minimum cost of breaking } x[c_i + 1, \cdots, c_j] \text{ at positions } c_{i+1}, \cdots, c_{j-1}$$

Notice then that the solution to our problem will be given by the value $L(0, n)$.

Algorithm and Recursion: Initialize the dynamic program by setting $L(i, i+1) = 0$ for all i, $0 \leq i \leq n$. This is correct as the string $x[c_i + 1, \cdots, c_{i+1}]$ needs no further cutting. The recursion to evaluate the values $L(i, j)$ is then, for $j > i + 1$:

$$L(i,j) = \min_{i<k<j} \{L(i,k) + L(k,j) + (c_j - c_i)\}$$

Correctness and Running Time: The recursion is correct as it is equivalent to selecting the first cut k to make to minimize the cost of breaking up $x[c_i + 1, \cdots, c_j]$. The running time is $O(n^3)$, as we have $O(n^2)$ subproblems, each of which takes $O(n)$ time to solve.

6.10. One way to solve this problem is by dynamic programming.
Subproblems: Define $L(i, j)$ to be the probability of obtaining exactly j heads amongst the first i coin tosses.

Algorithm and Recursion: By the definition of L and the independence of the tosses, it is clear that:

$$L(i,j) = p_i L(i-1, j-1) + (1 - p_i)L(i-1, j) \quad j = 0, 1, \ldots, i$$

We can then compute all $L(i, j)$ by initializing $L(0, 0) = 1$, $L(i, j) = 0$ for $j < 0$, and proceeding incrementally (in the order $i = 1, 2, \ldots, n$, with inner loop $j = 0, 1, \ldots, i$). The final answer is given by $L(n, k)$.

Correctness and Running Time: The recursion is correct as we can get j heads in i coin tosses either by obtaining $j - 1$ heads in the first $i - 1$ coin tosses and throwing a head on the last coin, which takes place with probability $p_i L(i - 1, j - 1)$, or by having already j heads after $i - 1$ tosses and throwing a tail last, which has probability $(1 - p_i)L(i - 1, j)$. Besides, these two events are disjoint, so the sum of their probabilities equals $L(i, j)$. Finally, computing each subproblem takes constant time, so the algorithm runs in $O(n^2)$ time.

Another approach is to observe that the probability is exactly the coefficient of x^k in the polynomial $g(x) = \prod_{i=1}^{n}(p_i x + (1 - p_i))$. We can compute $g(x)$ using divide-and-conquer in $O(n \log^2 n)$ time by recursively computing $g_1(x) = \prod_{i=1}^{n/2}(p_i x + (1 - p_i))$, $g_2(x) = \prod_{i=n/2+1}^{n}(p_i x + (1 - p_i))$ and then using FFT to calculate $g(x) = g_1(x)g_2(x)$.

6.11. *Subproblems:* Let us the define the following subproblems for all i, j such that $1 \leq i \leq n$, $1 \leq j \leq m$:

$$L(i,j) = \text{length of longest common subsequence between } x[1, \cdots, i] \text{ and } y[1, \cdots, j]$$

Then, the solution to the original problem will be given by $L(n, m)$.

Algorithm and Recursion: Initialize the dynamic program by setting $L(i, 0) = 0$ and $L(0, j) = 0$ for all i, j. The recursion to compute the values $L(i, j)$ correctly is the following:

$$L(i,j) = \max\{L(i-1,j), L(i,j-1), L(i-1,j-1) + \text{equal}(x_i, y_j)\}$$

where equal(x, y) is 1 if x and y are the same character and is 0 otherwise.

Correctness and Running Time: The recursion is correct as the alignment producing the longest common subsequence for $L(i, j)$ will have in its last position either a deletion, two characters matched (either equal or different characters) or an insertion. The running time is $O(mn)$ as there are mn subproblems, each of which takes $O(1)$ time.

6.12. Let $A(i, j)$ be the minimum cost triangulation of the polygon P_{ij} spanned by vertices $i, i + 1, \ldots, j$. In the optimum triangulation of P_{ij}, the edge (i, j) is covered by some triangle, say (i, k, j). Then the optimum triangulation of P_{ij} splits it into the triangle (i, k, j) and the optimum triangulations of the polygons P_{ik} and P_{kj}. That is, for $i < j$,

$$A(i, j) = \min_{i<k<j} \{A(i, k) + A(k, j) + d(i, k) + d(k, j)\},$$

where $d(\cdot, \cdot)$ are the distances between vertices. The base cases are $A(i, i + 1) = 0$, for all i. Since each computation $A(i, j)$ can be done in $O(n)$ time and there are $O(n^2)$ pairs (i, j), the running time of the algorithm is $O(n^3)$. To recover the best triangulation, it is sufficient to keep track of a choice of k which is maximizing for each $A(i, j)$ above. Then, starting at $A(1, n)$, we can backtrack to identify the diagonals included in the triangulation.

6.13. a) Consider the sequence $(2, 9, 1, 1)$. Then, the best available card at the start has value 2, but this leads only to a score of 3, as the opponent picks the card of value 9 in the next turn. Instead, choosing the 1 at the start yields a final score of 10.

b) *Subproblems:* Define subproblems $A(i, j)$ for $i \leq j$, where $A(i, j)$ is the difference between the largest total the first player can obtain and the corresponding score of the second player when playing on sequence $s_i, s_{i+1}, \cdots, s_j$. This will be positive if and only if the first player has the larger total.

Algorithms and Recursion: We can solve all the subproblem by initializing $A(i, i) = s_i$ for all i and using the update rule for $i < j$:

$$A(i, j) = \max\{s_i - A(i + 1, j), s_j - A(i, j - 1)\}$$

We can also keep track of the optimal move at each stage by simultaneously maintaining a matrix $M(i, j)$, where $M(i, j)$ will be set to `first` if $A(i, j)$ takes the value of the first element in the maximization, and to `last` otherwise.

Correctness and Running Time: The recursion is correct, as at any stage of the game there are two possible moves for the first player: either choose the first card, in which case he will gain s_i and score $-A(i + 1, j)$ in the rest of the game, or the last card, gaining s_j and $-A(i, j - 1)$ from the remaining cards. The algorithm computes all the subproblems and the entries of M in time $O(n^2)$ as each update takes time $O(1)$. Moreover, the player can then just read off the matrix M to learn the best strategy at any point of the game.

6.14. *Subproblems:* Define XY subproblems. For $1 \leq i \leq X$ and $1 \leq j \leq Y$, let $C(i, j)$ be the best return that can be obtained from a cloth of shape $i \times j$. Define also a function `rect` as follows:

$$\texttt{rect}(i, j) = \begin{cases} \max_k c_k \text{ for all products } k \text{ with } a_k = i \text{ and } b_k = j \\ 0 \text{ if no such product exists} \end{cases}$$

Algorithm and Recursion: Then the recursion is:

$$C(i, j) = \max\{\max_{1\leq k<i} \{C(k, j) + C(i - k, j)\}, \max_{1\leq h<j} \{C(i, h) + C(i, j - h)\}, \texttt{rect}(i, j)\}$$

It remains to initialize the smallest subproblems correctly:

$$C(1, j) = \max\{0, \texttt{rect}(1, j)\}$$
$$C(i, 1) = \max\{0, \texttt{rect}(i, 1)\}$$

The final solution is then the value of $C(X, Y)$.

Correctness and Running Time: For proving correctness, notice that $C(i, j)$ trivially has the intended meaning for the base cases with $i = 1$ or $j = 1$. Inductively, $C(i, j)$ is solved correctly, as a rectangle $i \times j$ can only be cut in the $(i-1)+(j-1)$ ways considered by the recursion or be occupied completely by a product, which is accounted for by the $\texttt{rect}(i, j)$ term. The running time is $O(XY(X + Y + n))$ as there are XY subproblems and each takes $O(X + Y + n)$ to evaluate.

6.15. One way to solve this problem is by dynamic programming.
Subproblems: We define subproblem $A(i, j)$ for $1 \leq i, j \leq n$ to represent the probability that A is the first to win n games, given that after $i + j$ games A has won i.

Algorithm and Recursion: We can initialize A by setting $A(n, j) = 1$ for $j \neq n$ and $A(i, n) = 0$ for all i. The other subproblems can be solved incrementally in decreasing order of $i + j$ using the recursion:

$$A(i, j) = \frac{1}{2}\left(A(i, j + 1) + A(i + 1, j)\right)$$

Correctness and Running Time: The recursion is correct as we can compute $A(i, j)$ by conditioning on the outcome of the $(i + j + 1)$th game. Both outcomes take place with probability $\frac{1}{2}$. If A wins, then it wins the game with probability $A(i + 1, j)$. If B wins, A has a probability of winning of $A(i, j + 1)$. If we are interested in finding solutions for all subproblems, we need to solve $O(n^2)$ subproblems, each taking $O(1)$ for a total running time of $O(n^2)$. If we are only after the $A(i, j)$ subproblem, we can stop once we solved $O((n - (i + j))^2)$ subproblems.

6.16. For a subset of garage sales $S \subseteq \{g_1, \cdots, g_n\}$ and $g_j \in S$, let $C(S, j)$ be the profit of the most profitable path starting at home, visiting only garage sales in S and ending at g_j. To express $C(S, j)$ in terms of smaller subproblems, consider the laste garage sale $g_i \in S - \{g_j\}$ visited before g_j. The path profit up to g_i is $C(S - \{g_j\}, i)$, whilst the marginal profit of visiting g_j is $p_j - d_{ij}$. We must pick i as to maximize the profit, so we have:

$$C(S, j) = \max_{g_i \in S : i \neq j} \{C(S - \{g_j\}, i) + p_j - d_{ij}\}$$

We can then compute solutions by initializing $C(\{g_i\}, i)$ to be $p_i + d_{0i}$ for all $1 \leq i \leq n$ and solving the subproblems in increasing order of $|S|$. The final solution is the minimum over all S,$g_j \in S$ of $C(S^*, j^*) + d_{j0}$, where we also include the cost of returning home. Moreover, we can also keep record, for every $C(S, j)$ of which garage g_i yielded a maximum in the recursion; this g_i is the last garage visited before getting to g_j in a best path for $C(S, j)$. We can then use this information to backtrack from $C(S*, j*)$ to reconstruct the entire optimal path. This algorithm solves $O(n2^n)$ subproblems, each in $O(n)$ time, so it has total running time $O(n^2 2^n)$.

6.17. This problem reduces to Knapsack with repetitions. The total capacity is v and there is an item i of value x_i and weight x_i for each coin denomination. It is possible to make change for value v if and only if the maximum value we can fit in v is v. The running time is $O(nv)$.

6.18. Use the same reduction as in 6.17, but reduce to Knapsack without repetition. The running time is $O(nv)$.

6.19. This is similar to 6.17 and 6.18. The problem reduces to Knapsack without repetition with a capacity of v, but this time we have k items of value x_i and weight x_i for each coin denomination x_i, i.e. a total of kn items. The running time is $O(nkv)$.

6.20. Let $S(i, j)$ be the cost of cheapest tree formed by words i to j, for $1 \leq i, j \leq n$. Also, initialize $S(i, j)$ to 0 if $i > j$. Then $S(i, j)$ will be the minimum cost of the tree over all choices of word k, $i \leq k \leq j$, to

place at the root. If word k is at the root, the cost of the left subtree will be $S(i, k-1)$ and the cost of right subtree will be $S(k+1, j)$. Moreover, all words will need to pay one comparison at the root node, so the total cost of the tree will be $\sum_{t=i}^{j} p_t + S(i, k-1) + S(k+1, j)$. Hence:

$$S(i,j) = \min_{i \le k \le j} \{\sum_{t=i}^{j} p_t + S(i, k-1) + S(k+1, j)\}$$

Finally, the cost of the optimal tree will be $S(1, n)$. To reconstruct the tree, it suffices to keep track of which root k minimized the expression in the recursion for each subproblem and backtrack from $S(1, n)$.

6.21. The subproblem $V(u)$ will be defined to be the size of the minimum vertex cover for the subtree rooted at node u. We have $V(u) = 0$ if u is a leaf, as the subtree rooted at u has no edges to cover. The crucial observation is that if a vertex cover does not use a node it has to use all its neighboring nodes. Hence, for any internal node i

$$V(i) = \min \left\{ \sum_{j:(i,j)\in E} \left(1 + \sum_{k:(j,k)\in E} V(k) \right), 1 + \sum_{j:(i,j)\in E} V(j) \right\}$$

The algorithm can then solve all the subproblems in order of decreasing depth in the tree and output $V(n)$. The running time is linear in n because while calculating $V(i)$ for all i we look at most at $2 * |E| = O(n)$ edges in total.

6.22. This problem reduces to Knapsack without repetition. The knapsack will have capacity t and each number a_i will be an item of value a_i and weight a_i. A subset adding up to t will exist if and only if a total value of t can fin in the knapsack. The dynamic programming algorithm then solves the problem in time $O(n \sum_i a_i)$.

6.23. We define $P(i, b)$ to be the maximum probability that the system works correctly up to its ith stage when adding redundancy to stages 1 to i and using budget b. We can initialize $P(0, b) = 1$ for all b. To compute $P(i, b)$, we consider the number of machines m_i we want to add at stage i. If we are adding m_i machines, our success probability will be the product of $P(i-1, b - m_i c_i)$, the best we can do on the previous stages with the remaining budget, and $(1 - (1 - r_i)^{m_i})$, the probability of stage i working correctly. More formally, we can update $P(i, b)$ for $1 \le i \le n$ and $0 \le b \le B$ using the recursion:

$$P(i,b) = \max_{0 \le m_i \le \left\lfloor \frac{b}{c_i} \right\rfloor} \{P(i-1, b - m_i c_i) \times (1 - (1 - r_i)^{m_i})\}$$

We can then solve all subproblems in ascending order of i by looping over all b's before proceeding to the next i. $P(n, B)$ will be success probability associated with the best allocation of redundancy. To recover the actual number of machines added at each stage for this optimal allocation, we need to keep track for every $P(i, b)$ of which value of m_i yielded the maximum value. We can then start at $P(n, B)$ and backtrack to find the optimal allocation. We have nB subproblems, each taking time at most $O(\max_i \frac{B}{c_i})$ for a total running time of $O(nB^2 \max_i \frac{1}{c_i})$.

6.24. Assume $m = O(n)$.

a) Consider the usual matrix of subproblems $E(i, j)$. If we update the values column by column, at every point we only need the current column and the previous column to perform all calculations. Hence, if we are just interested in the final value $E(m, n)$ we may keep only two columns at every time, using space $O(n)$. Note that we are not able now to have a pointer structure to recover the optimal alignment, as we would need pointers for all subproblems, which would take space mn.

b) Together with the subproblem solution $L(i,j)$, for each $j \geq m/2$, in each of the active two columns, maintain a pointer to the index k at which the optimal path leading to (i,j) crossed the $m/2$ column. Such pointer can be easily updated at every recursion by copying the pointer of the subproblem from which the optimal solution is derived.

c) Consider the space requirement first. At any time during the running of this scheme, a single dynamic programming data structure is active, taking up space $O(n)$. All that is left to store is the values k for all the subproblems on which we recurse. These are at most m as every values corresponds to an index of x matched to one of y in the minimum edit distance alignment.. Hence, the total space required is $O(n)$. For the running time analyisis, notice that level $i+1$ of the recursion takes half the time of level i. Hence, the total running time will be bounded above by $O(mn)(1+\frac{1}{2}+\frac{1}{4}+\cdots) = 2O(mn)$.

6.25. Let $M[i,s_1,s_2]$ be 1 if there are two disjoint subsets $I,J \subseteq \{1,\ldots i\}$ such that $\sum_{j\in I} a_j = s_1$ and $\sum_{j\in J} a_j = s_2$. An instance of 3-Partition has positive solution if and only if $M[n,S/3,S/3] = 1$. We can use dynamic programming to fill out the three dimensional table specified by $M[i,s_1,s_2]$ for $1 \leq i \leq n$, $0 \leq s_1, s_2 \leq S/3$. An arbitrary entry in the table can be calculated from three previous entries:

$$M[i,s_1,s_2] = M[i-1,s_1-a_i,s_2] \vee M[i-1,s_1,s_2-a_i] \vee M[i-1,s_1,s_2].$$

In words, this definition says that if there exist two disjoint subsets $I,J \subseteq \{1,\ldots,i\}$ such that $\sum_{j\in I} a_j = s_1$ and $\sum_{j\in J} a_j = s_2$, then either $i \in I, i \notin J$ or $i \in J, i \notin I$ or $i \notin I,J$. In the case that $i \in I$, $\sum_{j\in I, j\neq i} = s_1 - a_i$ and $\sum_{j\in J} = s_2$ (since I and J are disjoint), and so $M[i-1,s_1-a_i,s_2] = 1$. The case of $i \in J$ is symmetric. In the case that $i \notin I,J$, then $I,J \subseteq \{1,\ldots,i-1\}$, and so $M[i-1,s_1,s_2] = 1$.

The base cases for the recursion are $M[i,0,0] = 1$ for all $i \geq 0$ and $M[i,s_1,s_2] = 0$ if $i < 0$, $s_1 < 0$ or $s_2 < 0$. The table size is $n \times S/3 \times S/3$. Filling in a single entry takes constant time, so the total running time for the algorithm is $O(nS^2)$. (One could also notice that the table is symmetric because $M[i,s_1,s_2] = M[i,s_2,s_1]$ but this symmetry only gives a constant factor improvement in the running time.)

6.26. This is a simple variant of the edit distance algorithm defined in class. The recursion is modified to:

$$E(i,j) = \max\{E(i-1,j) + \delta(x_i,-), E(i-1,j-1) + \delta(x_i,y_j), E(i,j-1) + \delta(-,y_j)\}$$

The initialization has also to be modified to deal properly with the new scoring for gaps. We have, for $i,j > 0$:

$$\begin{aligned} E(0,0) &= 0 \\ E(i,0) &= E(i-1,0) + \delta(x_i,-) \\ E(0,j) &= E(0,j-1) + \delta(-,y_j) \end{aligned}$$

The correctness follows by the same argument as for the edit distance algorithm. The running time is again $O(mn)$.

6.27. This is another modification of the standard edit distance algorithm, only more advanced. You should notice that there is no simple way to modify the existing recursion to incorporate the new kind of gap penalties. In particular, it would be necessary at every point to know whether the previous subproblem solutions were given by alignments with a deletion or insertion in their last position. Moreover, it might be that an optimal alignment for $E(i,j)$ is not an extension of the optimal for $E(i-1,j), E(i-1,j-1), E(i,j-1)$ as an alignment with smaller previous score but terminating with a gap might have a smaller gap penalty and beat all extensions of optimal alignments. This suggests that we should not keep a single matrix of subproblems, but 3. E will be the matrix of subproblems

where the alignments are constrained to be a substitution or a match in their last position. E_x will be the matrix of subproblems over the alignments which have a gap in the last position of string x. E_y is defined similarly for y. Given these definitions, we just need to work out the recursion correctly case by case.

$$\begin{aligned} E(i,j) &= \max\{E(i-1,j-1), E_x(i-1,j-1), E_y(i-1,j-1)\} + \delta(x_i, y_j) \\ E_x(i,j) &= \max\{E(i-1,j) - c_0 - c_1, E_x(i-1,j) - c_1, E_y(i-1,j) - c_0 - c_1\} \\ E_y(x,y) &= \min\{E(i,j-1) - c_0 - c_1, E_x(i,j-1) - c_0 - c_1, E_y(i,j-1) - c_1\} \end{aligned}$$

The output will just be the maximum of $E(m,n), E_x(m,n)$ and $E_y(m,n)$. This takes $O(mn)$ as we still have $O(mn)$ subproblems, each evaluated in constant time.

6.28. This only requires a simple modification to 6.26. $E(i,j)$ becomes the score of the best scoring substring of $x[1,\cdots,i]$ and $y[1,\cdots,j]$. The recursion is then modified to take into account the possibility that the best local alignment be empty:

$$E(i,j) = \max\{E(i-1,j) + \delta(x_i,-), E(i-1,j-1) + \delta(x_i,y_j), E(i,j-1) + \delta(-,y_j), 0\}$$

6.29. For $i \in \{1,\cdots,n\}$, let $W(i)$ be the the weight of the best subset of consistent partial matches in $x[1,\cdots,i]$. To compute $W(i)$, we consider the two following cases:

- the best subset of partial matches contains a match j with $r_j = i$,
- the best subset of partial matches does not contain such j

In the first case, $W(i)$ will be the sum of w_j and the weight of the best match on the remaining of the string, i.e. $W(l_j - 1)$. In the second case, we will just have $W(i) = W(i-1)$. This shows that the following recursion is correct:

$$W(i) = \max\{W(i-1), \max_{j:r_j=i}\{W(l_j - 1) + w_j\}\}$$

The algorithm will then proceed computing $W(i)$ in ascending order of i and will output $W(n)$ as best total weight achievable. To reconstruct the actual sequence of partial matches, it suffices to keep track, for all $W(i)$ of which j maximizes the expression in the recursion, when the second maximum is the larger. We can then follow these pointers from $W(n)$ backwards to identify the optimal alignment. The running time is $O(n+m)$, where m is the number of partial matches, as we have n subproblems and each partial match is considered once in the maximizations.

6.30. a) See Figure 1.

b) It suffices to give an algorithm that minimizes the score for length 1 sequences, i.e. single characters. We can then apply this algorithm to all positions. The final sequence assignment will have score equal to the sum of the scores of the trees for each position and, as there are all minimum, it will also be minimum score.

The dynamic programming algorithm works by solving the problem in each subtree. For each internal node u, we define $S(u)$ to be the set of the labels $s(u)$ of u over all optimal assignments to u's subtree.

Claim: Let p be an internal node and let c_1 and c_2 be its children. If $S(c_1) \cap S(c_2)$ is non-empty, then $S(p) = S(c_1) \cap S(c_2)$. Otherwise, $S(p) = S(c_1) \cup S(c_2)$.
Proof: Let $C(u)$ be the minimum cost of a labeling of the subtree rooted at u. Consider the two cases:

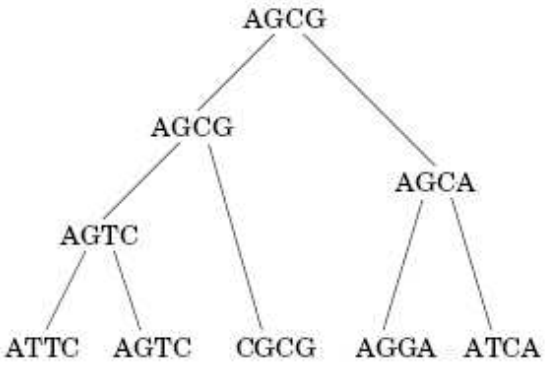


Figure 1: Parsimony tree for 6.30 a)

1) $I = S(c_1) \cap S(c_2)$ is non-empty. Then, any assignment with $s(p) \in I$ will have cost $C(c_1) + C(c_2)$, which is minimum for the subtree rooted at p. Moreover, any assignment with $s(p) \notin I$ will pay at least $C(c_1) + C(C_2) + 1$ and will not be optimal. Hence, $S(p) = I$.
2) If $I = S(c_1) \cap S(c_2)$ is empty, consider the union $U = S(c_1) \cup S(c_2)$. For any $s(p) \in U$, it is possible to construct an assignment of cost $C(c_1) + C(c_2) + 1$ by choosing a optimal assignment corresponding to $s(p)$ for one child's subtree and any assignment for the other. For $s(p) \notin U$, we pay $1 + C(c_i)$ for each child i for whose subtree we use an optimal labeling and at least the same quantity $1 + C(c_i)$ for each child i for which we do not use an optimal labeling. Hence, we pay at least $2 + C(c_1) + C(c_2)$. This shows that the optimal assignments for the subtree rooted at p are those consisting of $s(p) \in U$ and optimal assignments for the children's subtrees.

We can then initialize $S(v) = \{s(v)\}$ for the leaves and proceed up the tree, updating $S(p)$ to be the intersection of $S(c_1)$ and $S(c_2)$, if this is non-empty, and their union otherwise. Once, we have reached the root and calculated $S(r)$ we can produce an actual labeling by setting $s(r)$ to any label in $S(r)$. The label for all other internal nodes u can then be calculated from the label $s(p)$ of u's parent by taking $s(u) = s(p)$ if $s(p) \in S(u)$ and $s(u)$ to be any label in $S(u)$ otherwise. For each position of the string, we have $O(n)$ subproblems $S(u)$, each of which takes time $O(|\Sigma|)$ to update. The total running time is then $O(|\Sigma|nk)$.

Chapter 8 - Solutions

8.1 Assume that $TSP(G, b)$ returns `false` if no tour of length b or less exists in G. Then the following functions solve $TSP - OPT$ using TSP.

```
TSP-OPT(G)
   S = 0
   for all u ∈ V, v ∈ V:
      S = S + dist(u, v)
   return BINARY-SEARCH-TOUR(G, 0, S)


BINARY-SEARCH-TOUR(G, l, u)
   b = (l + u)/2
   if TSP(G, b) ≠ false
      return BINARY-SEARCH-TOUR(G, l, b)
   else return BINARY-SEARCH-TOUR(G, b, u)
```

Basically, the algorithm just does a binary search over all possible lengths of the optimal tour, going from 0 to the sum of all distances. Note that binary search is necessary here and we can't just increment the value of b by 1 each time since the sum of all distances is exponential in the size of the input.

8.2 We impose an arbitrary ordering on the edges and remove the edges one by one. If the graph obtained by removing an edge from the current graph still has a Rudrata path, we remove e permanently and update the current graph. If the new graph does not have a Rudrata path, we add e back. Hence, we maintain the invariant that the graph we have always has a Rudrata path (if the given graph did). Since it is possible to remove all edges except a single Rudrata path, we will be left a Rudrata path in the end.

8.3 It's a generalization of SAT. Given a SAT formula φ with n variables, (φ, n) is an instance of STINGY SAT which has a solution if and only if the original SAT formula has a satisfying assignment.

8.4 (a) Given a clique in the graph, it is easy to verify in polynomial time that there is an edge between every pair of vertices. Hence a solution to CLIQUE-3 can be checked in polynomial time.

(b) The reduction is in the wrong direction. We must reduce CLIQUE to CLIQUE-3, if we intend to show that CLIQUE-3 is at least as hard as CLIQUE.

(c) The statement "a subset $C \subseteq V$ is a vertex cover in G is and only if the complimentary set $V - C$ is a clique in G" used in the reduction is false. C is a vertex cover if and only if $V - C$ is an *independent set* in G.

(d) The largest clique in the graph can be of size at most 4, since every vertex in a clique of size k must have degree at least $k - 1$. Thus, there is no solution for $k > 4$, and for $k \leq 4$ we can check every k-tuple of vertices, which takes $O(|V|^k) = O(|V|^4)$ time.

8.5 **3D-MATCHING to SAT**

We have a variable x_{bgp} for each given triple (b, g, p). We interpret $x_{bgp} =$ `true` as choosing the triple (b, g, p). Suppose for boy b, $(b, g_1, p_1), \ldots, (b, g_k, p_k)$ are triples involving him. Then we add the clause $(x_{bg_1p_1} \vee \ldots \vee x_{bg_kp_k})$ so that at least one of the triples is chosen. Similarly for the triples involving each girl or pet. Also, for each pair of triples involving a common boy, girl or pet, say (b_1, g, p_1) and (b_2, g, p_2), we add a clause of the form $(\bar{x}_{b_1gp_1} \vee \bar{x}_{b_1gp_1})$ so that at most one of the triples is chosen.

The total number of triples, and hence the number of variables, is at most n^3. The first type of clauses involve at most n^2 variables and we add $3n$ such clauses, one for each boy, girl or pet. The second type of clauses involve triples sharing one common element. There are $3n$ ways to choose the common element and at most n^4 to choose the rest, giving at most $3n^5$ clauses. Hence, the size of the new formula is polynomial in the size of the input.

If there is a matching, then it must involve n triples $(b_1, g_1, p_1), \ldots, (b_n, g_n, p_n)$. Setting the variables $x_{b_1g_1p_1}, \ldots, x_{b_ng_np_n}$ to `true` and the rest to `false` gives a satisfying assignment to the above formula. Similarly, choosing only the triples corresponding to the `true` variables in any satisfying assignment must correspond to a matching for the reasons mentioned above. Hence, the formula is satisfiable if

and only if the given instance has a 3D matching.

RUDRATA CYCLE to SAT

We introduce variables x_{ij} for $1 \leq i, j \leq n$ meaning that the ith vertex is at the jth position in the Rudrata cycle. Each vertex must appear at some position in the cycle. Thus, for every vertex i, we add the clause $x_{i1} \vee x_{i2} \vee \ldots \vee x_{i_n}$. This adds n clauses with n variables each.

Also, if the ith vertex appears at the jth position, then the vertex at $(j+1)$th position must be a neighbor of i. In other words, if u, v are *not* neighbors, then either u appears at the jth position, or v appears at the $(j+1)$th position, but not both. Thus for every $(u, v) \notin E$ and for all $1 \leq j \leq n$, add the clause $(\bar{x}_{uj} \vee (\bar{x}_{v(j+1)})$. This adds at most $O(n^2) \times n = O(n^3)$ clauses with 2 variables each.

Using the "meanings" of the clauses given above, it is easy to see that every satisfying assignment gives a Rudrata cycle and vice-versa.

8.6 a) Let the number of variables be n and the number of clauses be m. Note that each variable x can satisfy at most 1 clause (because it appears as x and $\bar{x}$ at most once). We construct a bipartite graph, with the variables on the left side and the clauses on the right. Connect each variable to the clauses it appears in. For the formula to be satisfiable, each clause must pick one (at least) variable it is connected to with each variable being picked by *at most* one of the clauses it is connected to. If we connect all the variables to a source s, all clauses to a sink t and fix the capacity of all edges as 1, this is equivalent to asking if we can send a flow of m units from s to t. Since the flow problem can be solved in polynomial time, so can the satisfiability problem.

b) Consider the reduction from 3SAT to INDEPENDENT SET. If each literal is allowed to appear at most twice, we claim that the graph we create in this reduction has degree at most 4 for each vertex. Let $(x_1 \vee x_2 \vee x_3)$ be a clause in the original formula. Then, in the graph, the vertex corresponding to x_1 in this clause, has one edge each to x_2 and x_3. Also, it has edges to all occurrences of $\bar{x}_1$. But, since $\bar{x}_1$ is allowed to appear at most twice, this can add at most two edges. Similarly, each vertex in the graph has at most 4 edges incident to it.

However, we know that the special case of 3SAT, with each literal occurring at most twice is NP-complete. The above argument shows that this special case reduces to the special case of INDEPENDENT SET, with each vertex having degree at most 4. Hence, this special case of INDEPENDENT SET must also be NP-complete.

8.7 Consider a bipartite graph with clauses on the left and variables on the right. Consider any subset S of clauses (left vertices). This has exactly $3|S|$ edges going out of it, since each clause has exactly 3 literals. Since each variable on the right has at most 3 edges coming into it, S must be connected to at least $|S|$ variables on the right. Hence, this graph has a matching using Hall's theorem proved in problem 7.30.

This means we can match every clause with a unique variable which appears in that clause. For every clause, we set the matched variable appropriately to make the clause true, and set the other variables arbitrarily. This gives a satisfying assignment. Since a matching can be found in polynomial time (using flow), we can construct the assignment in polynomial time.

8.8 We start from 3SAT and reduce it to an instance of EXACT 4SAT. We can assume that 3SAT formula has no clauses with one variable, since those variables can directly be assigned. Let $C_2 = (l_1 \vee l_2)$ and $C_3 = (l_3 \vee l_4 \vee l_5)$ be two clauses having 2 and 3 literals respectively. We can write them as the following equivalent groups of clauses with exactly 4 literals - we need to add one new variable for C_3 and two new ones for C_2.

$$C_3' \equiv (x \vee l_3 \vee l_4 \vee l_5) \wedge (\bar{x} \vee l_3 \vee l_4 \vee l_5)$$

$$C_2' \equiv (y \vee z \vee l_1 \vee l_2) \wedge (\bar{y} \vee z \vee l_1 \vee l_2) \wedge (y \vee \bar{z} \vee l_1 \vee l_2) \wedge (\bar{y} \vee \bar{z} \vee l_1 \vee l_2)$$

Any assignment satisfying C_3' must assign either `true` or `false` to x, making one of the clauses true - in which case the other clause becomes exactly C_3. A similar argument holds for C_2'.

8.9 This is a generalization of VERTEX-COVER. Given a graph G, consider each edge $e = (u, v)$ as a set containing the elements u and v. Then, finding a hitting set of size at most b in this particular family of sets is the same as finding a vertex cover of size at most b for the given graph.

8.10 a) We can view this as a generalization of the CLIQUE problem. Given an input (G, k) for CLIQUE, let H be a graph consisting of k vertices with every pair connected by an edge (i.e. a clique of size k). Then G contains a clique of size k if and only if H is a subgraph of G.

b) This is a generalization of RUDRATA-PATH. Given a graph G with n vertices, let $g = n - 1$. Then $(G, n-1)$ is an instance of LONGEST PATH. However, a simple path of length $n-1$ must contain n vertices and hence must be a Rudrata path. Also, any Rudrata path is of length $n-1$. Hence for any graph with n vertices, LONGEST-PATH$(G, n-1)$ is precisely the asking for the Rudrata path.

c) Given a formula ϕ with m clauses, setting $g = m$ gives SAT as a special case of MAX-SAT.

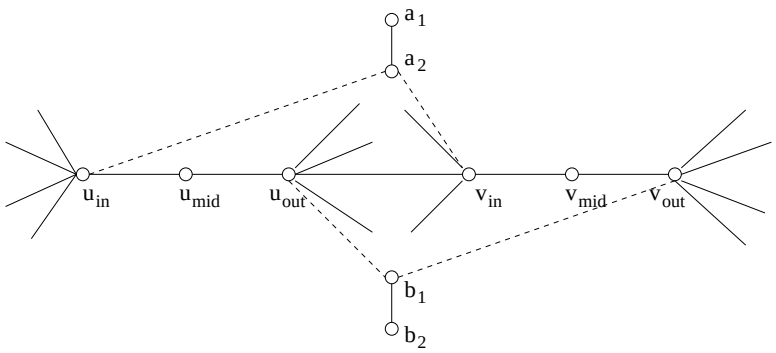
d) This also a generalization of CLIQUE. Given an instance (G, k) let $a = k$, $b = k(k-1)/2$. Any subgraph of G with k vertices and containing $k(k-1)/2$ edges, must have an edge between every possible pair out of these k vertices and hence must be a clique. Similarly, a clique on k vertices must contain $k(k-1)/2$ edges.

e) This is a generalization of INDEPENDENT SET. Given (G, k), an instance for independent set, let $a = k$, $b = 0$. Then (G, a, b) is an instance for SPARSE SUBGRAPH. Also, any subgraph of G with k vertices and 0 edges must be an independent set of size k and vice-versa.

f) Generalizes VERTEX COVER. Consider each vertex as a set of the edges incident upon it. Then a SET COVER of this family of sets corresponds exactly to picking vertices (sets) such that at least one vertex corresponding to each edge is picked (i.e. at least one set containing every element is picked).

g) Generalizes Rudrata cycle. Given a graph $G = (V, E)$, take $b = n$ and $d_{ij} = 1\ \forall (i, j) \in E$ and $d_{ij} = 2$ otherwise. Also, take $r_{ij} = 2$ for every (i, j).
Then this is an instance of the RELIABLE NETWORK problem which corresponds to picking few edges such that the sum of the weights of all the edges is n and between every points i and j, there must be 2 vertex-disjoint paths i.e. they should be part of a cycle. Also, since every pair must have two paths between them, the graph must be connected. Finally, because of the weights, the graph can have at most n edges. Hence, it must have exactly n edges and exactly one cycle (since a connected graph with n edges can have only 1 cycle), which contains all the vertices. This is exactly the Rudrata cycle. It is also easy to see that every Rudrata cycle satisfies the properties for the above RELIABLE NETWORK instance.

8.11 (a) Given a graph $G = (V, E)$ as an instance of the DIRECTED RUDRATA PATH problem, we construct an instance of undirected RUDRATA PATH which consists of three copies of $|V|$, denoted by vertex sets V_{in}, V_{mid} and V_{out}. Let the new graph be $G' = (V_{top} \cup V_{mid} \cup V_{bot}, E')$.
We have $\{u_{out}, v_{in}\} \in E'$ (an undirected edge) for every $(u, v) \in E$. Also, $\{v_{in}, v_{mid}\}, \{v_{mid}, v_{out}\} \in E'$ for all $v \in V$. Hence, the only way to include v_{mid} in a path is to pass from v_{top} to v_{bot} or vice-versa.



Finally, to ensure that paths start in V_{in} and end at V_{out}, we add four vertices a_1, a_2 and b_1, b_2 and edges $\{a_1, a_2\}, \{b_1, b_2\}$ and $\{a_2, v_{in}\}, \{v_{out}, b_2\} \forall v \in V$. Since a_1 and b_1 have degree 1, they must be the end points of any Rudrata path.

A Rudrata path starting from a_1 must visit a_2 and then some vertex in V_{in}. Similarly, a path ending at b_1 must come through V_{out}. To cover V_{mid}, the path must be of the form $a_1, a_2, v_{in}^{(1)}, v_{mid}^{(1)}, v_{out}^{(1)}, \ldots, v_{in}^{(1)}, v_{mid}^{(1)}, v_{out}^{(1)}, b_2, b_1$. Then $v^{(1)}, ldots, v^{(n)}$ gives a directed Rudrata path. Similarly, an undirected Rudrata path in G' can be constructed from a directed Rudrata path in G.

(b) We use the same construction of G' as in the previous part. It is also a valid instance of the undirected Rudrata (s,t)-path problem with $s = a_1$ and $t = b_1$. Since all paths we obtained above were always of this form, the same argument still works.

8.12 a) To show that k-SPANNING TREE is a search problem, we need to show that it is possible to verify a solution in polynomial time. Given a spanning tree T for a graph G we need to verify that it is indeed a tree, that each edge in T is present in G and that all vertices of G are present in T and have degree k. All this can be done by a single DFS on T, comparing each edge with the corresponding edge in G, which takes polynomial time in total.

b) Consider the 2-SPANNING TREE problem. We are required to find a tree with each vertex having degree at most 2. However, such a tree must be a path, since creating a branch at any vertex makes the degree of that vertex as 3. Also, if the tree spans the graph it must be a an undirected Rudrata path. Hence, this problem is same as the undirected Rudrata path problem which we showed to be NP-complete in Problem 8.11.

We now reduce k-SPANNING TREE for $k > 2$, to the 2-SPANNING TREE problem. Given a graph $G = (V, E)$, at each vertex $v \in V$, we add $k-2$ "buffer vertices" $v_1, \ldots, v_{k-2}$ connected only to v. We add a fresh set of buffer vertices for each original vertex in the graph. Call this new graph G'. It is easy to see that a k-spanning tree of G' must contain all the buffer vertices as leaves, since they all have degree 1. Removing, these gives a 2-spanning tree of G. Similarly, adding $k-2$ "buffer leaves" to each vertex in any 2-spanning tree of G, will give a k-spanning tree of G'. Hence, the k-SPANNING TREE problem is NP-Complete for every $k \geq 2$.

8.13 (a) This can be solved in polynomial time. Delete all the vertices in the set L from the given graph and find a spanning tree of the remaining graph. Now, for each vertex $l \in L$, add it to any of its neighbor present in the tree. It is clear that such a tree, if it is possible to construct one, must have all the vertices in L as leaves. If the graph becomes unconnected after removing L, or some vertex in L has no neighbors in $G \backslash L$, then no spanning tree exists having all vertices in L as leaves.

(b) This generalizes the (undirected) (s,t)-RUDRATA PATH problem. Given a graph G and two vertices s and t, we set $L = \{s, t\}$. We now claim that the tree must be a path between s and t. It cannot branch out anywhere, because each branch must end at a leaf and there are no other leaves available. Also, since it is a spanning tree, the path must include all the vertices of the graph and hence must be Rudrata path. Similarly, every Rudrata path is a tree of the type required above.

(c) This is also a generalization of (undirected) (s,t)-RUDRATA PATH. We use the same reduction as in the previous part. Note that a tree must have at least two leaves. Hence for $L = \{s, t\}$, the set of leaves must be exactly equal to L.

(d) Again, setting $k = 2$, gives exactly the (undirected) Rudrata path problem, since a spanning tree with at most two leaves must be a path containing all the vertices. Also, it will have exactly two leaves (since that's the minimum a tree can have).

(e) The solution to this problem uses the reduction from problem 8.20. We first note that in the reduction from VERTEX COVER to DOMINATING SET, if we add edges (u, v) for all vertices $u, v \in V$ of the given graph, the dominating set we obtain is necessarily connected. Hence the reduction also gives the hardness of the problem of finding a *connected* dominating set of size less than or equal to k. However, a graph has a connected dominating set of size at most k if and

only if it has a spanning tree with $|V| - k$ or more leaves (since the graph obtained by removing the leaves is exactly a connected dominating set). Hence, the given problem is NP-hard.

(f) Same as the part d), setting $k = 2$ gives the Rudrata path problem.

8.14 We can reduce CLIQUE to the given problem. Given an instance (G, k) of CLIQUE with n vertices, we create G', which is G together with another n vertices, but without any extra edges. The extra vertices trivially provide an independent set of size $k \leq n$ for any k. Hence, G has a clique of size k if and only if G' has a clique as well as an independent set of size k, since the extra vertices have no edges between them.

8.15 This is a generalization of CLIQUE. Given (G, k) as an instance of CLIQUE with n vertices, take $G_1 = G$, $b = k$ and G_2 as a clique of size n. Then (G_1, G_2, b) has a solution if and only if G has a clique of size at least k.

8.16 This is a generalization of INDEPENDENT SET. Let D be the adjacency matrix of an undirected graph (i.e. penalty is 1 if two vertices are neighbors and 0 otherwise) and let $p = 0$. Then the maximum number of ingredients (vertices) that can be chosen is exactly the size of the maximum independent set. Hence, EXPERIMENTAL CUISINE is NP-Complete which means there must be a polynomial time reduction from 3SAT to EXPERIMENTAL CUISINE.

8.17 Since the problem Π is in NP, there must be an algorithm which verifies that a given solution is correct in polynomial time, say bounded by a polynomial $q(n)$. Since the algorithm reads the given solution, the size of the solution can be at most $q(n)$ bits. Hence, we run the algorithm on *all* $2^{q(n)}$ binary strings of length $q(n)$. If the given instance has any solution, then one of these inputs must be a solution. The running time is $O(q(n)2^{q(n)}) = O(2^n 2^{q(n)}) = O(2^{q(n)+n})$. Taking $p(n) = n + q(n)$, we are done.

8.18 Since FACTORING is in NP (we can check a factorization in polynomial time), $\mathbf{P} = \mathbf{NP}$ would mean the factors of a number can be *found* in polynomial time. Since in RSA, we know (N, e) as the public key, we can factor N to find p and q and in polynomial time. We can then compute $d = e^{-1} \mod (p-1)(q-1)$ using Euclid's algorithm. If X is the encrypted message, then $X^e \mod N$ gives the original message.

8.19 We give a reduction from CLIQUE to KITE. Given an instance (G, k) of CLIQUE, we add a tail of k new vertices to every vertex of G to obtain a new graph G'. Since the added tails are just paths and cannot contribute to a clique, G' has a kite with $2k$ nodes if and only if G has a clique of size k.

8.20 We reduce VERTEX-COVER to DOMINATING-SET. Given a graph $G = (V, E)$ and a number k as an instance of VERTEX-COVER, we convert it to an instance of DOMINATING-SET as follows. For each edge $e = (u, v)$ in the graph G, we add a vertex a_{uv} and the edges (u, a_{uv}) and (v, a_{uv}). Thus we create a "triangle" on each edge of G. Call this new graph $G' = (V', E')$.

We now claim that a G' has a dominating set of size at most k if and only if G has a vertex cover of size at most k. It is easy to see that vertex cover for G is also a dominating set for G' and hence one direction is trivial.

For the other direction, consider a dominating set $D \subseteq V'$ for G'. For each triangle (u, v, a_{uv}) (corresponding to edge (u, v)), at least one of the three vertices must be in D, since the only neighbors of a_{uv} are u and v. Since we can exchange a_{uv} with u or v, still maintaining a dominating set, we can assume that none of the added vertices (a_{uv}s) is in D. Since D must then contain at least one endpoint for every edge, it is also a vertex cover.

8.21 (a) Constructing a digraph as in the hint, a Rudrata path exactly corresponds to a sequence of k-mers such that the last $k-1$ characters of each element match the first $k-1$ characters of the next element. This is just the reconstructed sequence.

(b) Let S be the *set* (not multiset) of all the strings formed by the first $k-1$ characters and all strings formed by the last $k-1$ characters of all the given k-mers. Construct a directed graph with $V = S$, with an edge $(u, v) \in E$ if there exists a k-mer having u as the $k-1$ characters and v as the last $k-1$ characters. It is easy to see that each k-mer corresponds to exactly one edge. An Euler path in this graph gives a sequence of elements as the Rudrata path above.

8.22 (a) Given a graph and a feedback arc set, it is easy to check that the size of the set is at most b and that the graph produced by removing the edges in the set is acyclic (through DFS). Hence, the problem is in NP.

(b) We claim that if $U \subseteq V$ is a vertex cover for G, then $F = \{(w_i, w_i') | v_i \in U\}$ is a feedback arc set (of the same size) for G'.

To prove this, first note that G' is a bipartite graph with (say) $w_1, \ldots w_n$ on the left and $w_1', \ldots, w_n'$ on the right. Now, any cycle must involve some edge, say (w_i', w_j) from right to left. However, the only incoming edge into w_i' is (w_i, w_i') and the only outgoing edge from w_j is (w_j, w_j') and hence both of these must be in the cycle. Also, since (w_i', w_j) is an edge, $(v_i, v_j) \in E$ and either v_i or v_j must be in U which means $F \cap \{(w_i, w_i'), (w_j, w_j')\} \neq \emptyset$. Hence, removing the edges in F breaks this cycle.

(c) We replace all the edges of the form (w_i', w_j) in the given feedback arc set, say F, by edges of the form (w_k, w_k'). By the argument in the previous part, any cycle containing (w_i', w_j) must also contain (w_i, w_i') and (w_j, w_j') and hence $F \backslash \{(w_i', w_j)\} \cup \{(w_i, w_i')\}$ is also a feedback arc set. Continuing in this manner, we never increase the size of the set (but might decrease it we include the same edge, say (w_i, w_i') for two removed edges (w_i', w_j) and (w_i', w_k)), we get a feedback arc set F' such that $|F'| \leq b$.

We now claim that $U = \{v_i | (w_i, w_i') \in F'\}$ must be a vertex cover for G. If not, then there must be an edge (v_j, v_k) such that $v_j, v_k \notin U$ and hence $(w_j, w_j'), (w_k, w_k') \notin F'$. But then $w_j \to w_j' \to w_k \to w_k' \to w_j$ would be a cycle in G' even after removing F', which is a contradiction.

8.23 For a given 3SAT formula with n variables and m clauses, we create a graph with $m + n$ source sink pairs - one for each variable and one for each clause. For each clause c of the form $(l_1 \vee l_2 \vee l_3)$, we create 6 new vertices $l_1, l_2, l_3, \bar{l}_1, \bar{l}_2$ and $\bar{l}_3$.

We add the edges $(s_c, l_i), (l_i, t_c)$ for $i = 1, 2, 3$ so that the only paths connecting this source-sink pair are the ones that pass through at least one of the literals (not its complement) in the clause. Think of this as making the literal "true". Also, for all variables x, we connect the occurrences of x in all the clauses in a path and connect its endpoints to s_x and t_x. We also do this for all the occurrences of $\bar{x}$. A path from s_x to t_x must contain either all the occurrences of x or all the occurrences of $\bar{x}$.

Given node disjoint paths, we now construct a satisfying assignment. If for variable x, the solution has the path containing all occurrences of x, we assign x `= false` (and x `= true` otherwise). Thus, for a path from s_x to t_x we set all the literals in the path to `false`. Since each clause c must have a path from s_c to t_c containing a literal appearing in the clause, which has *not* been set to `false`, the clause must be satisfied. By the same argument, we can also construct a set of paths from a satisfying assignment.