Algorithm Design and Applications

Michael T. Goodrich
Department of Information and Computer Science
University of California, Irvine

Roberto Tamassia
Department of Computer Science
Brown University

Instructor’s Solutions Manual

Chapter 1 Algorithm Analysis

Hints and Solutions

Reinforcement

R-1.1 Hint: Recall the method for graphing on a logarithmic scale.

R-1.2 Hint: Consider how it behaves on average.

Solution: The outer loop, for index j, makes n iterations. In $n/2$ of those iterations (for $j < n/2$), the next-inner loop, for index k, makes at least $n/2$ iterations. Finally, for $n/4$ of those iterations (for $k > 3n/4$), the inner-most loop, for index i, makes at least $n/4$ iterations. Thus, the MaxsubSlow algorithm uses at least $n(n/2)(n/4) = n^3/8$ steps, which is $\Omega(n^3)$.

R-1.3 Hint: Determine the place where these two functions cross.

R-1.4 Hint: Determine the place where the two functions cross.

R-1.5 Hint: Use the limit definition.

R-1.6 Hint: Note the similarity of "always" and "worst case."

R-1.7 Hint: When in doubt about two functions $f(n)$ and $g(n)$, consider $\log f(n)$ and $\log g(n)$ or $2^{f(n)}$ and $2^{g(n)}$.

Solution:

$$1/n, 2^{100}, \log\log n, \sqrt{\log n}, \log^2 n, n^{0.01}, \lceil\sqrt{n}\rceil, 3n^{0.5}, 2^{\log n}, 5n, n\log_4 n,$$

$$6n\log n, \lfloor 2n\log^2 n\rfloor, 4n^{3/2}, 4^{\log n}, n^2\log n, n^3, 2^n, 4^n, 2^{2^n}.$$

R-1.8 Hint: The numbers in the first row are quite large.

Solution: The numbers in the first row are quite large. The table below calculates it approximately in powers of 10. People might also choose to use powers of 2. Being close to the answer is enough for the big numbers (within a few factors of 10 from the answers shown).

	1 Second	1 Hour	1 Month	1 Century
$\log n$	$2^{10^6} \approx 10^{300000}$	$2^{3.6\times 10^9} \approx 10^{10^9}$	$2^{2.6\times 10^{12}} \approx 10^{0.8\times 10^{12}}$	$2^{3.1\times 10^{15}} \approx 10^{10^{15}}$
$\sqrt{n}$	$\approx 10^{12}$	$\approx 1.3 \times 10^{19}$	$\approx 6.8 \times 10^{24}$	$\approx 9.7 \times 10^{30}$
n	10^6	3.6×10^9	$\approx 2.6 \times 10^{12}$	$\approx 3.12 \times 10^{15}$
$n \log n$	$\approx 10^5$	$\approx 10^9$	$\approx 10^11$	$\approx 10^{14}$
n^2	1000	6×10^4	$\approx 1.6 \times 10^6$	$\approx 5.6 \times 10^7$
n^3	100	≈ 1500	≈ 14000	≈ 1500000
2^n	19	31	41	51
$n!$	9	12	15	17

R-1.9 Hint: We say that an algorithm is linear if its running time is proportional to its ***input*** size.

Solution: The worst case running time of find2D is $O(n^2)$. This is seen by examining the worst case where the element x is the very last item in the $n \times n$ array to be examined. In this case, find2d calls the algorithm arrayFind n times. arrayFind will then have to search all n elements for each call until the final call when x is found. Therefore, n comparisons are done for each arrayFind call. Since arrayFind is called n times, we have $n * n$ operations, or an $O(n^2)$ running time. This is not a linear time algorithm; it is quadratic. If this were a linear time algorithm, the running time would be proportional to its input size.

R-1.10 Hint: Don't forget the base case.

R-1.11 Hint: Note the structure of the loop.

Solution: The Loop1 method runs in $O(n)$ time.

R-1.12 Hint: Note the structure of the looping.

Solution: The Loop2 method runs in $O(n)$ time.

R-1.13 Hint: Note the structure of the looping.

Solution: The Loop3 method runs in $O(n^2)$ time.

R-1.14 Hint: Note the structure of the looping.

Solution: The Loop4 method runs in $O(n^2)$ time.

R-1.15 Hint: Note the structure of the looping.

Solution: The Loop5 method runs in $O(n^4)$ time.

R-1.16 Hint: Recall the definition of the big-oh notation.

R-1.17 Hint: Recall the definition of the big-oh notation.

R-1.18 Hint: Recall the definition of the big-oh notation.

R-1.19 Hint: Recall the definition of the big-oh notation.

R-1.20 Hint: Recall the definition of the big-oh notation.

Solution: By the definition of big-Oh, we need to find a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $(n+1)^5 \leq c(n^5)$ for every integer $n \geq n_0$. Since $(n+1)^5 = n^5 + 5n^4 + 10n^3 + 10n^2 + 5n + 1$, $(n+1)^5 \leq c(n^5)$ for $c = 8$ and $n \geq n_0 = 2$.

R-1.21 Hint: Recall the definition of the big-oh notation.

Solution: By the definition of big-Oh, we need to find a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $2^{n+1} \leq c(2^n)$ for $n \geq n_0$. One possible solution is choosing $c = 2$ and $n_0 = 1$, since $2^{n+1} = 2 \cdot 2^n$.

R-1.22 Hint: Recall the definition of the little-oh notation.

R-1.23 Hint: Recall the definition of the little-omega notation.

R-1.24 Hint: Recall the definition of the big-omega notation.

Solution: By the definition of big-Omega, we need to find a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $n^3 \log n \geq cn^3$ for $n \geq n_0$. Choosing $c = 1$ and $n_0 = 2$, shows $n^3 \log n \geq cn^3$ for $n \geq n_0$, since $\log n >\geq 1$ in this range.

R-1.25 Hint: Recall the definition of the big-oh notation.

R-1.26 Hint: Recall the definition of the big-oh notation.

R-1.27

R-1.28 Hint: Recall the definition of the big-oh notation.

R-1.29 Hint: Note that you can save a comparison here.

R-1.30 Hint: Recall the formula for the Chernoff bound.

R-1.31 Hint: Revisit the reason why 2 cyber-dollars were used in the original proof.

R-1.32 Hint: Use the Chernoff bound.

Creativity

C-1.1 Hint: Change the max-based formulas to if-statements and add variables that "remember" when you update the running maximum.

C-1.2 Hint: Observe the relationship between M and m and note that we can do the operations of the two loops at the same time.

C-1.3 Hint: You can essentially ignore the operations p_i were i is not a multiple of 3.

C-1.4 Hint: Consider an argument based on each bit position.

C-1.5 Hint: Notice the similarity of this equation and the sum of the numbers from 1 to n.

C-1.6 Hint: Recall the way of characterizing a geometric sum.

C-1.7 Hint: Recall how the power function is defined.

C-1.8 Hint: Recall the role of n_0 in the definition of the big-oh notation.

Solution: To say that Al's algorithm is "big-oh" of Bill's algorithm implies that Al's algorithm will run faster than Bill's for all input greater than some nonzero positive integer n_0. In this case, $n_0 = 100$.

C-1.9 Hint: Think of a function that grows and shrinks at the same time without bound.

Solution: One possible solution is $f(n) = n^2 + (1 + sin(n))$.

C-1.10 Hint: Use induction, a visual proof, or bound the sum by an integral.

Solution:

$$\sum_{i=1}^{n} i^2 < \int_0^{n+1} x^2 dx < \frac{(n+1)^3}{3} = O(n^3)$$

C-1.11 Hint: Try to bound this sum term by term with a geometric progression.

C-1.12 Hint: Use the log identity that translates $\log bx$ to a logarithm in base 2.

C-1.13 Hint: First construct a group of candidate minimums and a group of candidate maximums.

C-1.14 Hint: Note how much work is done in each iteration.

C-1.15 Hint: Consider the first induction step.

Solution: The induction assumes that the set of $n-1$ sheep without a and the set of $n-1$ sheep without b have sheep in common. Clearly this is not true with the case of 2 sheep. If a base case of 2 sheep could be shown, then the induction would be valid.

C-1.16 Hint: Look carefully at the definition of big-Oh and rewrite the induction hypothesis in terms of this definition.

C-1.17 Hint: Use a specific constant, c.

C-1.18 Hint: You need to handle the one item that is not matched.

C-1.19 Hint: Consider summing up the elements of A.

Solution: First calculate the sum $\sum_{i=1}^{n-1} = \frac{n(n-1)}{2}$. Then calculate the sum of all values in the array A. The missing element is the difference between these two numbers.

C-1.20 Hint: Try to bound from above each term in this summation.

Solution:

$$\sum_{i=1} n \log_2 i < \sum_{i=1} n \log_2 n = n \log_2 n$$

C-1.21 Hint: Try to bound from below half of the terms in this summation.

Solution: For convenience assume that n is even. Then

$$\sum_{i=1}^{n} \log_2 i /geq \sum_{i=\frac{n}{2}+1}^{n} \log_2 \frac{n}{2} = \frac{n}{2} \log_2 \frac{n}{2},$$

which is $\Omega(n \log n)$.

C-1.22 Hint: Use induction to reduce the problem to that for $n/2$.

C-1.23 Hint: Consider the contribution made by one line.

C-1.24 Hint: Think first about how you can verify that a row i has the most 1's in $O(n)$ time.

Solution: Start at the upper left of the matrix. Walk across the matrix until a 0 is found. Then walk down the matrix until a 1 is found. This is repeated until the last row or column is encountered. The row with the most 1's is the last row which was walked across.

Clearly this is an $O(n)$-time algorithm since at most $2 \cdot n$ comparisons are made.

C-1.25 Hint: Take advantage of the fact that the number of rows plus the number of columns in A is $2n$.

Solution: Using the two properties of the array, the method is described as follows.

- Starting from element $A[n-1,0]$, we scan A moving only to the right and upwards.
- If the number at $A[i,j]$ is 1, then we add the number of 1s in that column $(i+1)$ to the current total of 1s
- Otherwise we move up one position until we reach another 1.

The running time is $O(n)$. In the worst case, you will visit at most $2n - 1$ places in the array.

C-1.26 Hint: Apply the multiplication formula directly.

C-1.27 Hint: Recall the multiplication algorithm taught in grade school.

Solution:

C-1.28 Hint: Be sure to handle the checking needed during a remove operation.

C-1.29 Hint: Apply the amortization analysis accounting technique using extra cyber-dollars for both insertions and removals.

C-1.30 Hint: Consider how many cyber-dollars are saved up from one expansion to the next.

Applications

A-1.1 Hint: Note that every division takes $O(n)$ time, but there are a lot of divisions.

Solution: Since r is represented with 100 bits, any candidate p that the eavesdropper might use to try to divide r uses also at most 100 bits. Thus, this very naive algorithm requires 2^{100} divisions, which would take about 2^{80} seconds, or at least 2^{55} years. Even if the eavesdropper uses the fact that a candidate p need not ever be more than 50 bits, the problem is still difficult. For in this case, 2^{50} divisions would take about 2^{30} seconds, or about 34 years.

Since each division takes time $O(n)$ and there are 2^{4n} total divisions, the asymptotic running time is $O(n \cdot 2^{4n})$.

A-1.2 Hint: Recall the methods for doing an experimental analysis.

A-1.3 Hint: Recall the methods for doing an experimental analysis.

A-1.4 Hint: Number each bottle and think about the binary expansion of each bottle's number.

Solution:

A-1.5 Hint: You can determine all the boxes with pearls in $O(\sqrt{n})$ time.

A-1.6 Hint: Try to extend the $O(\sqrt{n})$-touch solution for the previous problem.

A-1.7 Hint: Rewrite the equation as $A[i] = c - A[j]$. Now what are you looking for?

Solution: We can rewrite the equation as $A[j] = c - A[i]$. Create a Boolean array, B, indexed from 0 to $10n$, all of whose elements are initially **false**. For each element, $A[i]$, if $c - A[i] > 0$, set $B[c - A[i]]$ to **true**. Then, for each $A[j]$ in A, check if $B[A[j]]$ is **true**. If any such cell of B is **true**, then the answer is "yes." Otherwise, the answer is "no." The running time of this method is $O(n)$.

A-1.8 Hint: Reverse the array by using index pointers that start at the two ends.

A-1.9 Hint: Consider using Horner's rule, which is mentioned in an earlier exercise in this chapter.

Solution: Initialize your value $x = 0$. Go through S from beginning to end, and, for each digit d, update $x \leftarrow 10x + d$. The running time is $O(n)$.

A-1.10 Hint: Recall how we solved the maximum subarray sum problem.

A-1.11 Hint: Consider using the XOR function.

Solution: Initialize y to 0 and then XOR all the values in A with y. The result will be x.

A-1.12 Hint: Think of functions that you can compute on all the integers in A.

Solution: Compute the sum of all the integers in A and compute the sum of the squares of all the integers in A. Using the identities given in the appendix of this book, we know that, if all the numbers from 1 to n were present, then the first sum would be $n(n+1)/2$ and the second would be $n(n+1)(2n+1)/6$. So if we denote the missing numbers by i and j, and we denote the first sum by a and the second by b, then we know $a = n(n+1)/2 - i - j$ and $b = n(n+1)(2n+1)/6 - i^2 - j^2$. Solving these two equations will give us the values of i and j.

A-1.13 Hint: Try to count in terms of the losers.

Solution: Each time a game is played, one of the teams is sent home. If we start with n times, this means that there are $n - 1$ games played in total. Thus, the total time for doing this simulation is $O(n \log n)$.

A-1.14 Hint: Do a single scan.

A-1.15 Hint: Think about using a "window" that always contains k 1's.

Solution: Scan through A using two pointers, i and j, such that $A[i : j]$ always has k 1's and i is as close to j as possible. Each time you increment j, you need to move it to the next 1 and then move i to get as close to j as possible to maintain $A[i : j]$ having k 1's. Since each operation increments either i or j, we can charge $2n$ cyber-dollars to pay for all operations. Thus, the total running time is $O(n)$.

A-1.16 Hint: Consider applying the principle of induction to this problem.

Solution: Let's apply induction to this problem. Note that if there is exactly 1 cheating husband, then his wife thinks that there are 0 cheating husbands in the town. So, on the day that the mayor makes his announcement, she learns that her husband must be cheating on her, and she poisons him that very night. By induction, if i nights have passed and no husbands have been poisoned, then every wife who thinks that there are exactly i other husbands who are cheaters learns that her husband must be a cheater (for otherwise the wives of those i other husbands would have poisoned their husbands by now). So, on that night, every such wife will poison her husband. Therefore, if there are k cheaters, then they will all be poisoned on the kth night after the mayor's announcement.

A-1.17 Hint: Consider coin flips in pairs.

Solution: Perform coin flips as ordered pairs, that is, repeatedly perform two flips of this coin, where the order of the flips in each pair matters. If both flips are heads (HH) or tails (TT), then discard this trial and repeat the process with another pair of flips. If the ordered pair of flips comes up HT, however, consider this as equivalent to a "0", and if it comes up TH, consider this as equivalent to a "1." Since individual flips are independent, even for a biased coin, an outcome of HT is equal in probability to a TH, no matter how biased heads and tails are individually.

A-1.18 Hint: Adjust the probability of choosing a byte as you go.

Solution: Use a single variable to hold the chosen byte. Choose the first byte with probability 1, the second with probability $1/2$, and so on, so that you choose the ith byte with probability $1/i$. Any time you choose a byte, you use it to replace the byte you had chosen previously. It is easy to show by induction that each byte will have a probability of $1/n$ in the end of being the one chosen.

Chapter 2 Basic Data Structures

Hints and Solutions

Reinforcement

R-2.1 Hint: Think about the order of the indexing for each of the for-loops.

R-2.2 Hint: Think about the order of the indexing for each of the for-loops.

R-2.3 Hint: Review the code for insertAfter(p, e).

R-2.4 Hint: This one is a real puzzler, and it doesn't even use the operators $+$ and $\times$.

Solution: The tree expresses the formula $6/(1 - 5/7)$.

R-2.5 Solution: It is not possible for the postorder and preorder traversal of a tree with more than one node to visit the nodes in the same order. A preorder traversal will always visit the root node first, while a postorder traversal node will always visit an external node first.

It is possible for a preorder and a postorder traversal to visit the nodes in the reverse order. Consider the case of a tree with only two nodes.

R-2.6

R-2.7 Hint: Try to gain some intuition by drawing a few different binary trees such that all the external nodes have the same depth.

R-2.8

Creativity

C-2.1 Hint: Review how to implement a general list using a linked list.

C-2.2 Hint: Think about the pointers that must be updated in such implementations.

C-2.3 Hint: Consider a combined search from both ends.

C-2.4 Hint: Use one stack for enqueues and the other for dequeues. (You still need to say how and you also need to do the amortized analysis.)

Solution: Name the two stacks as E and D, for we will enqueue into E and dequeue from D. To implement enqueue(e), simply call E.push(e). To implement dequeue(), simply call D.pop(), provided that D is not empty. If D is empty, iteratively pop every element from E and push it onto D, until E is empty, and then call D.pop(). For the amortized analysis, charge \$2 to each enqueue, using \$1 to do the push into E. Imagine that we store the extra cyber-dollar with the element just pushed. We use the cyber-dollar associated with an element when we move it from E to D. This is sufficient, since we never move elements back from D to E. Finally, we charge \$1 for each dequeue to pay for the push from D (to remove the element returned). The total charges for n operations is $O(n)$; hence, each operation runs in $O(1)$ amortized time.

C-2.5 Hint: Use one queue as auxiliary storage and keep track of sizes as you are using it.

Solution: To implement a stack using two queues, $Q1$ and $Q2$, we can simply enqueue elements into $Q1$ whenever a push call is made. This takes $O(1)$ time to complete. For pop calls, we can dequeue all elements of $Q1$ and enqueue them into $Q2$ except for the last element which we set aside in a temp variable. We then return the elements to $Q1$ by dequeing from $Q2$ and enqueing into $Q1$. The last element that we set aside earlier is then returned as the result of the pop. Thus, performing a pop takes $O(n)$ time.

C-2.6 Hint: Reduce the problem to that of enumerating all permutations of the numbers $\{1, 2, \ldots, n-1\}$.

C-2.7 Hint: Consider each output number one at a time.

Solution: Let i be the last number in π. Perform enough enqueue and dequeue operations to remove i from Q, and push i onto the stack, S. Then repeat this operation until Q is empty. Finally, pop all the elements from S and enqueue them into Q in this order. The result will be the numbers, $1, 2, \ldots, n$, ordered according to π. The total time to implement this algorithm is $O(n^2)$, since retrieving each number, i, can take $O(n)$ time in the worst case.

C-2.8 Hint: Think about how to extend the circular array implementation of a queue given in the previous chapter.

Solution:

Maintain a capacity variable and a elementCount variable. Also maintain the variables indexFirst and indexLast. Presuming that overflow doesn't occur, insertion at rank 0 involves inserting the element in array position indexFirst $-$ 1 if indexFrist is greater than 0. Otherwise it is inserted at capacity $- 1$. Then indexFirst is updated to reflect the array index the new element was inserted into.

Removal from rank 0 involves incrementing indexFirst mod capacity.

The array index of elemAtRank(x) can be computed as

$$x + \mathsf{indexFrist} \bmod \mathsf{capacity}.$$

C-2.9 Hint: Consider randomly shuffling the deck one card at a time.

C-2.10

C-2.11 Hint: Think about what could be the worst case number of nodes that would have to be traversed to answer each of these queries.

Solution: Pseudocode for these methods is given below.

The worst case running times for these algorithms are all $O(\log n)$ where n is the height of the tree T.

C-2.12 Hint: Derive a formula that relates the depth of a node v to the depths of nodes adjacent to v.

Solution: The idea is to perform a preorder (or postorder) traversal of the tree, where the "visit" action is to report the depth of the node that is currently visited. This can be easily done by using a counter that keeps track of the current depth.

C-2.13 Hint: Try to compute node heights and balance factors at the same time.

Solution: One way to do this is the following: in the ***external*** method, set height and balance to be zero. Then, alter the ***right*** method as follows:

```
Algorithm right():
    if $v$isInternal$(v)$ then
        if $v.leftchild.height > v.rightchild.height$ then
            $v.height = v.leftchild.height + 1$;
        else
            $v.height = v.rightchild.height + 1$;
        $v.balance = absval(v.rightchild.height - v.leftchild.height)$;
    printBalanceFactor$(v)$
```

C-2.14 Hint: Use the definition to derive a recursive algorithm.

C-2.15 Solution: Examining the Euler tree traversal, we have the following method for finding tourNext(v,α)

- If v is a leaf, then:

```
Algorithm preorderNext(Node v):
    if $v$isInternal() then
        return $v$'s left child
    else
        Node $p$ = parent of $v$
        if $v$ is left child of $p$ then
            return right child of $p$
        else
            while $v$ is not left child of $p$ do
                $v = p$
                $p = p.parent$
            return right child of $p$

Algorithm inorderNext(Node v):
    if $v$isInternal() then
        return $v$'s right child
    else
        Node $p$ = parent of $v$
        if $v$ is left child of $p$ then
            return $p$
        else
            while $v$ is not left child of $p$ do
                $v = p$
                $p = p.parent$
            return $p$

Algorithm postorderNext(Node v):
    if $v$isInternal() then
        $p =$ parent of $v$
        if $v =$ right child of $p$ then
            return $p$
        else
            $v$ = right child of $p$
            while $v$ is not external do
                $v = leftchildofv$
            return $v$
    else
        $p$ = parent of $v$
        if $v$ is left child of $p$ then
            return right child of $p$
        else
            return $p$
```

Algorithm 2.1: Methods for the solution of Exercise C-2.11.

    - ○ if α = **left**, then w is v and β = **below**,
    - ○ if α = **below**, then w is v and β = **right**,
    - ○ if α = **right**, then
        - ∗ if v is a left child, then w is v's parent and β = **below**,
        - ∗ if v is a right child, then w is v's parent and β = **right**.
- If v is internal, then:
    - ○ if α = **left**, then β = **left** and w is v's left child,
    - ○ if α = **below**, then β = **left** and w is v's right child,
    - ○ if α = **right**, then
        - ∗ if v is a left child, then β = **below** and w is v's parent,
        - ∗ if v is a right child, then β = **right** and w is v's parent.

For every node v but the root, we can find whether v is a left or right child, by asking "v=T.leftChild(T.parent(v))". The complexity is always $O(1)$.

C-2.16

C-2.17 Hint: Consider a recursive algorithm.

C-2.18 Hint: You can tell which visit action to perform at a node by taking note of where you are coming from.

Solution:

C-2.19 Hint: Use a stack.

Solution:

C-2.20 Hint: Use a queue.

Solution:

C-2.21 Hint: Modify an algorithm for computing the depth of each node so that it computes path lengths at the same time.

C-2.22 Hint: Use the fact that we can build T from a single root node via a series of n operations that expand an external node into an internal node with two leaf children.

Applications

A-2.1

A-2.2 Hint: It helps to know the relative depths of x and y.

Solution: This method runs in $O(d)$ time, where d is the depth of T.

A-2.3 Hint: Redefine the diameter in terms of node heights.

Solution: Let d_{uv} be the diameter of T. First observe that both u and v are tree leaves; if not, we can find a path of higher length, only by considering one of u's or v's children. Moreover, one of u, v has to be a leaf of highest depth. This can be proved by contradiction. Consider a tree and some leaf-to-leaf path P that does not include a leaf of highest depth. Then consider a leaf of greatest depth v; it is always possible to find a new path P' that starts at v and is longer than P. This yields the desired contradiction.

The main idea of the algorithm is to find a leaf v of highest depth and starting from this leaf to keep moving towards the tree's root. At each visited node u (including v, but excluding the tree's root), the height of u's sibling is computed and the current length L_{max} of the longest path in which v belongs is updated accordingly. When we are at T's root, L_{max} contains the diameter of T. The running time of the algorithm is linear.

We can improve on this algorithm, however. Note that the above algorithm can visit all the nodes in the tree twice, once to find the deepest node, and once to find the height subtrees. We can combine these two operations into one algorithm with a little bit of ingenuity and recursion. We observe that given a binary tree T_v rooted at a node v, if we know the diameters and heights of the two subtrees, we know the diameter of T_v. Imagine we took a path from the deepest node of the left subtree to the deepest node of the other subtree, passing through v. This path would have length $= 2 + \mathsf{height}(T_v, T_v.\mathsf{leftChild}(v)) + \mathsf{height}(T_v, T_v.\mathsf{rightChild}(v))$. Further, note that this is a longest path that runs through the root of T_v, since the height of the left and right subtrees is simply the longest path from their respective roots to any of their leafs. Therefore, the longest path in T_v is simply the maximum of longest path in the found in T_v's left and right subtrees, and the longest path running through v. Since this algorithm visits each node in the tree exactly once, and performs a constant amount of operations at each node, it follows that this algorithm runs in $O(n)$

```
**Algorithm** eulerTour(Tree $T$, Position $v$):
    $state \leftarrow start$
    **while** $state \neq done$ **do**
        **if** $state = start$ **then**
            **if** $T$.isExternal$(v)$ **then**
                left action
                below action
                right action
                $state \leftarrow= done$
            **else**
                left action
                $state \leftarrow$ on_the_left
                $v \leftarrow v.leftchild$
        **if** $state =$ on_the_left **then**
            **if** $T$.isExternal$(v)$ **then**
                left action
                below action
                right action
                $state =$ from_the_left
                $v \leftarrow v.parent$
            **else**
                left action
                $v \leftarrow v.leftchild$
        **if** $state =$ from_the_left **then**
            below action
            $state \leftarrow$ on_the_right
            $v \leftarrow v.right$
        **if** $state =$ on_the_right **then**
            **if** $T$.isExternal$(v)$ **then**
                $state =$ from_the_right
                left action
                below action
                right action
                $v \leftarrow v.parent$
            **else**
                left action
                $state \leftarrow$ on_the_left
                $v \leftarrow v.left$
        **if** $state =$ from_the_right **then**
            right action
            **if** $T$.isRoot$(v)$ **then**
                $state \leftarrow done$
            **else**
                **if** $v$ is left child of parent **then**
                    $state \leftarrow$ from_the_left
                **else**
                    $state \leftarrow$ from_the_right
                $v \leftarrow v.parent$
```

Algorithm 2.2: Methods for the solution of Exercise C-2.18.

```
Algorithm inorder(Tree $T$):
    Stack S $\leftarrow$ new Stack()
    Node $v \leftarrow T$.root()
    push $v$
    while $S$ is not empty do
        while $v$ is internal do
            $v \leftarrow v.left$
            push $v$
        while $S$ is not empty do
            pop $v$
            visit $v$
            if $v$ is internal then
                $v \leftarrow v.right$
                push $v$
            while $v$ is internal do
                $v \leftarrow v.left$
                push $v$
```

Algorithm 2.3: Method for the solution of Exercise C-2.19.

```
Algorithm levelOrderTraversal(BinaryTree $T$):
    Queue $Q$ = new Queue()
    $Q$.enqueue($T$.root())
    while $Q$ is not empty do
        Node $v \leftarrow Q$.dequeue()
        if $T$.isInternal($v$) then
            $Q$.enqueue($v.leftchild$)
            $Q$.enqueue($v.rightchild$)
```

Algorithm 2.4: Method for the solution of Exercise C-2.20.

```
**Algorithm** LCA(Node $x$, Node $y$):
    int $x_{dpth} \leftarrow x.depth$
    int $y_{dpth} \leftarrow y.depth$
    while $x_{dpth} > y_{dpth}$ do
        $v \leftarrow v.parent$
    while $y_{dpth} > x_{dpth}$ do
        $y \leftarrow y.parent$
    while $x \neq y$ do
        $x \leftarrow x.parent$
        $y \leftarrow y.parent$
    return $x$
```

Algorithm 2.5: Method for the solution of Exercise A-2.2.

Chapter 3 Binary Search Trees

Hints and Solutions

Reinforcement

R-3.1 Hint: Be sure to keep the low, mid, and high variables straight during the algorithm simulation.

R-3.2 Hint: Review the algorithm for inserting items in a binary search tree.

R-3.3 Hint: Review the definition of a binary search tree.

R-3.4 Hint: Draw T.

Solution: There is only one binary search tree of height 4 (including the external nodes) that stores all the integers in the range from 1 to 15. This tree stores 8 at its root.

R-3.5 Hint: Review the definition of a binary search tree.

R-3.6 Hint: Review the selection algorithm given in the book, and specialize it for the case $i = 1$.

R-3.7 Hint: Review the algorithm for deletion in a binary search tree.

R-3.8 Solution: There are several solutions. One is to draw the binary search tree created by the input sequence: $9, 5, 12, 7, 13$. Now draw the tree created when you switch the 5 and the 7 in the input sequence: $9, 7, 12, 5, 13$.

R-3.9 Hint: You choose 2 out of n elements where the order is imposed later.

Solution: There are $n(n-1)/2$ such pairs.

R-3.10 Hint: Recall the worst-case example of the height of a binary search tree.

Solution: $O(n^2)$ comparisons.

R-3.11 Hint: Review the running time analysis for indexing in a binary search tree and the average height of a randomly constructed binary search tree.

R-3.12 Hint: Just describe how to do a search for those two items.

R-3.13 Hint: Review the definition of H_n.

Solution:

$$H_5 = \frac{60 + 30 + 20 + 15 + 12}{60} = \frac{147}{60}.$$

Creativity

C-3.1 Hint: Think about how to devise a probe test to determine if you are querying above or below the missing number.

C-3.2 Hint: Do a "double" binary search, and dove-tail these two searches.

C-3.3 Hint: Find the beginning and end of the run using binary search.

C-3.4 Hint: Identify the ranges of the search and then traverse the nodes in between. Show that this approach runs in the correct time.

Solution: See method findAllElements.

```
Algorithm findAllElements($k, v, c$):
  Input: The search key $k$, a node of the binary search tree $v$ and a container $c$
  Output: A set containing the found elements
   if $v$ is an external node then
       return $c$.elements()
   if $k =$ key($v$) then
       $c$.addElement($v$)
       return findAllElements($k, T$.rightChild($v$)$, c$)
   else if $k <$ key($v$) then
       return findAllElements($k, T$.leftChild($v$))
   else
       // we know $k >$ key($v$)
       return findAllElements($k, T$.rightChild($v$))
```

Algorithm 3.1: Method findAllElements.

Note that after finding k, if it occurs again, it will be in the left most internal node of the right subtree.

C-3.5 Hint: The induction hypothesis is that $H(n) \geq \lceil \log n \rceil$.

C-3.6 Hint: Identify the ranges of the search and then traverse the nodes in between. Show that this approach runs in the correct time.

C-3.7 Hint: You will need to augment T, adding a new field to each internal node and ways of maintaining this field during updates.

Solution: For each node of the tree, maintain the size of the corresponding subtree, defined as the number of internal nodes in that subtree. While performing the search operation in both the insertion and deletion, the subtree sizes can be either incremented or decremented. During the rebalancing, care must be taken to update the subtree sizes of the three nodes involved (labeled a, b, and c by the restructure algorithm).

To calculate the number of nodes in a range (k_1, k_2), search for both k_1 and k_2, and let P_1 and P_2 be the associated search paths. Call v the last node common to the two paths. Traverse path P_1 from v to k_1. For each internal node $w \neq v$ encountered, if the right child of w is in not in P_1, add one plus the size of the subtree of the child to the current sum. Similarly, traverse path P_2 from v to k_2. For each internal node $w \neq v$ encountered, if the left child of w is in not in P_2, add one plus the size of the subtree of the left to the current sum. Finally, add one to the current sum (for the key stored at node v).

C-3.8 Hint: Consider adding a small subtree of size $n^{1/2}$ to a binary search tree that is otherwise of good height.

C-3.9 Hint: Consider augmenting each key, k, by choosing a random number in the range from 1 to n for each inserted pair, and using k together with the associated random number to guide insertions.

C-3.10 Hint: Think first about how you can determine the number of 1's in any row in $O(\log n)$ time.

Solution: To count the number of 1's in A, we can do a binary search on each row of A to determine the position of the last 1 in that row. Then we can simply sum up these values to obtain the total number of 1's in A. This takes $O(\log n)$ time to find the last 1 in each row. Done for each of the n rows, then this takes $O(n \log n)$ time.

C-3.11 Hint: Treat the cases $i = 1$ and $i = n$ separately.

Solution: For the upper bound,

$$H_n = 1 + \sum_{i=2}^{n} \frac{1}{i} \leq 1 + \int_{x=1}^{n} \frac{dx}{x} = 1 + \ln n.$$

For the lower bound,

$$H_n \geq \sum_{i=1}^{n-1} \frac{1}{i} \geq \int_{x=1}^{n} \frac{dx}{x} = \ln n.$$

C-3.12 Hint: Write out the terms of the difference, $H_n - H_{n/2}$. How many terms are there?

Solution: If n is a power of 2 greater than 1, then there are $n/2$ terms in the difference, $H_n - H_{n/2}$, and each of them is at most $1/(n/2)$. Thus, the sum of these terms is at most 1. Therefore, $H_n \leq 1 + H_{n/2}$, which implies, by induction, that $H_n \leq 1 + \log n$ in this case. If n is not a power of 2, then let N be the smallest power of 2 greater than n. Then $\lceil \log n \rceil = \log N$; hence,

$$H_n \leq H_N \leq 1 + \log N = 1 + \lceil \log n \rceil.$$

Applications

A-3.1 Hint: Imagine how you would do a coordinated binary search for all the requested sizes at the same time in T so as to avoid wasting comparisons.

Solution: When a drug request comes in, specified as $\{x_1, x_2, \ldots, x_k\}$, the main idea is to coordinate k binary searches in T so as to avoid wasted comparisons. There are several ways to do this, but one way is as follows. First, do a binary search in T to locate the successor of $x_{k/2}$, which is at, say, index i. Then, recursively search in subarray of T with indices less than i, for the items $x_1, \ldots, x_{(k/2)-1}$, and recursively search in subarray of T with indices greater than i, for the items $x_{(k/2)+1}, \ldots, x_k$.

To see that this search runs in $O(k \log(n/k))$ time, note that total time we spend is maximized when the successor items are distributed as far apart as possible, that is, at every $(n/k)$the position. In this case, the first search takes $\log n$ time, the two on either side take $2\log(n/2)$ time, the next level of 4 take $4\log(n/4)$ time, and so on, with the final ones taking $(k/2)\log(n/(k/2))$ time, which all adds up to a value that is $O(k\log(n/k))$.

A-3.2 Hint: What is the best element to store at the root of T?

Solution: Take the element at $A[n/2]$ and place it at the root, r, of T. Then recursively place the nodes in the left and right subtrees of r, from the first half of A, up to $A[(n/2)-1]$, and the last half of A, starting from $A[(n/2)+1]$. Since each recursive call in this algorithm takes $O(1)$ time, to place a single node in T, the total running time is $O(n)$.

A-3.3 Hint: Start out by sorting the points around C.

A-3.4 Hint: The main work is done by the preorder and postorder traversal of T.

A-3.5 Hint: The main idea is to determine the left and right children of each node in a top-down fashion, recursing on each side.

A-3.6 Hint: Consider augmenting each node v with a label, y_v, which is the maximum y-value for any zombie stored in the subtree of T rooted at v.

A-3.7 Hint: Consider building an augmented binary search tree on the numbers 1 to n.

Chapter 4 Balanced Binary Search Trees

Hints and Solutions

Reinforcement

R-4.1 Hint: Recall the insertion algorithm for AVL trees and remember that insertions always happen at external nodes.

R-4.2 Hint: Recall the insertion algorithm for wavl trees and remember that insertions always happen at external nodes.

R-4.3 Hint: Recall the insertion algorithm for splay trees.

R-4.4 Hint: Construct an example where you could have 2 items that could either be the root.

Solution: There are several solutions. One is to draw the AVL tree created by the input sequence: $9, 5, 12, 7, 13$ (note that no rebalances are needed). Now draw the tree created when you switch the 5 and the 7 in the input sequence: $9, 7, 12, 5, 13$.

R-4.5 Hint: Construct an example where you could have 2 items that could either be the root.

Solution: There are several solutions. One is to draw the wavl tree created by the input sequence: $9, 5, 12, 7, 13$ (and note that no rebalances are needed). Now draw the tree created when you switch the 5 and the 7 in the input sequence: $9, 7, 12, 5, 13$.

R-4.6 Hint: Recall the notation for the minimum number of nodes in an AVL tree as a function of the height.

R-4.7 Hint: Recall the notation for the minimum number of nodes in a red-black tree as a function of the black height.

R-4.8 Hint: Recall the notation for the minimum number of nodes in a wavl tree as a function of the rank, which then translates to height at the root.

R-4.9 Hint: Recall how unbalanced a splay tree is allowed to be if splay operations can occur in an order that is bad for balance.

R-4.10 Hint: Recall the definition of single and double rotation.

R-4.11 Hint: Recall the insertion algorithm for AVL trees.

R-4.12 Hint: Recall the deletion algorithm for AVL trees.

R-4.13 Hint: Recall the insertion algorithm for wavl trees.

R-4.14 Hint: Recall the removal algorithm for wavl trees.

R-4.15 Hint: Note that black height and actual height can differ by a factor of 2.

R-4.16 Hint: You need to have the tree be unbalanced enough that it would violate the internal-node property if we converted it to a wavl tree.

R-4.17 Hint: Recall the definition of a wavl tree.

R-4.18 Hint: Try working through an example.

R-4.19 Hint: Try working through an example.

R-4.20 Hint: Recall the balanced binary search tree keeps items in sorted order.

Creativity

C-4.1 Hint: Show that $O(n)$ rotations suffice to convert any binary tree into a ***left chain***, where each internal node has an external right child.

C-4.2 Hint: Note that $\varphi^2 = \varphi + 1$; hence, $\varphi^k = \varphi^{k-1} + \varphi^{k-2}$, for $k \geq 3$.

C-4.3 Hint: Recall the general case, $n_h = 1 + n_{h-1} + n_{h-2}$.

C-4.4 Hint: Follow the general framework of the proof of the theorem on the asymptotic height of an AVL tree.

C-4.5 Hint: First find the beginning and end of the set of items with key k.

Solution: See method findAllElements.

Note that after finding k, if it occurs again, it will be in the left most internal node of the right subtree.

C-4.6 Hint: Identify the ranges of the search and then traverse the nodes in between. Show that this approach runs in the correct time.

```
Algorithm findAllElements$(k, v, c)$:
  Input: The search key, $k$, a node, $v$, of the binary search tree and a container, $c$
  Output: An enumeration of the elements found
   if $v$ is an external node then
      return $c$.elements()
   if $k = \text{key}(v)$ then
      $c$.addElement$(v)$
      return findAllElements$(k, T.\text{rightChild}(v), c)$
   else if $k < \text{key}(v)$ then
      return findAllElements$(k, T.\text{leftChild}(v))$
   else
      // we know $k > \text{key}(v)$
      return findAllElements$(k, T.\text{rightChild}(v))$
```

Algorithm 4.1: Method findAllElements.

C-4.7 Hint: Think of how to cascade the imbalances up the tree.

C-4.8 Hint: Study closer the balance property of an AVL tree and the rebalance operation.

C-4.9 Hint: Study closer the balance property of an AVL tree and the rebalance operation.

C-4.10 Hint: Find the right place to "splice" one tree into the other to maintain the wavl tree property.

C-4.11

C-4.12 Hint: You can use the structure of the tree to encode the color bit.

Solution: Use the structure of the tree to encode the color bit. Since a left child must always have key less than the parent when the tree is storing distinct keys, you can use this information to encode a color bit. For instance, red nodes can have their two children swapped.

C-4.13 Hint: Search down for k and cut along this path. Now consider how to "glue" the pieces back together in the right order.

C-4.14 Hint: Consider the red and black meaning of the three possible balance factors in an AVL tree.

C-4.15 Hint: Consider storing items in a binary search tree plus a collection of binary trees for each set, with each of these augmented with each internal node storing information about its descendants.

C-4.16 Hint: You should be able to get similar bounds to standard splay trees.

C-4.17 Hint: Since you know the node x will eventually become the root, maintain a tree of nodes to the left of x and a tree of nodes to the right of x, which will eventually become the two children of x.

C-4.18 Hint: If you are having trouble with this problem, you may wish to gain some intuition about splay trees by "playing" with an interactive splay tree program.

C-4.19

C-4.20 Hint: Find the item x_j with largest j such that $\sum_{i=0}^{j-1} a_i < A/2$. Consider putting this item at the root and recursing on the two subsequences that this induces.

C-4.21 Hint: Consider using a simple tree data structure that implements the divide-and-conquer algorithm hinted above so that each recursive call runs faster than linear-time.

Applications

A-4.1 Hint: Consider how you can use a binary search tree here.

Solution: A good solution is to store the set of waiting dogs in a balanced search tree, T. When inserting a dog, d, we first do a lookup in T to find the dogs, c, and e, that are before and after d in age. Then we update d's links to point to c and e, we update c's link to an older dog to point to d, and we update e's younger link to point to d. The total time to insert a dog in this case is $O(\log n)$. The removal of a dog is the reverse of this operation and it too runs in $O(\log n)$ time assuming T is implemented with a balanced search tree.

A-4.2 Hint: Consider how you can use a binary search tree here. You may also wish to add maintain some global information, as well, for all employees.

Solution: The solution is to store employee information in a balanced binary search tree ordered by employee names. This allows for insertions and removals to take $O(\log n)$ time and for an alphabetical listing of the employees in E to run in $O(n)$ time. In addition, we should store, for each employee, x, a value, v_x, of shares that they deserve, along with a single global variable, z, for all the employees in E, of additional shares of FastCo that everyone in E deserves. That is, the number of shares that each employee v is promised at any given time is $v_x + z$. This allows the processing of a Friday promise to take place in $O(1)$ time, simply by replacing the old value of z with $z + y$. One additional important detail is that, when adding a new employee x to E, we need to initialize the value of v_x to $-z$.

A-4.3 Hint: First find the ends of the range and then find a way to efficiently enumerate the elements in between.

Solution: Search for k_1 and then k_2, each while marking the path down to that node. Observe that all the nodes between the path toward k_1 and k_2 are within the specified range. A modified DFS can be performed, using the given paths as boundaries and starting at the intersection of the two paths, to enumerate all the elements inclusively between k_1 and k_2.

A-4.4 Hint: You need a way of searching for elements with the same Zip code.

Solution: Store the elements from S in a balanced binary search tree, T, ordered by a combination of Zip code and name, where we first go by Zip code and then break ties alphabetically. To do a search for a specific Zip code, k, we do a standard search for k and then list out all the nodes in subtrees that have k as their Zip code. The analysis is similar to that used for an in-order traversal, plus $O(\log n)$ for identifying the subtrees in T that need to be traversed.

A-4.5 Hint: Consider storing P in a balanced binary search tree T, ordered by age, where you augment the nodes of T with additional information that allows you to find the median using a binary search.

A-4.6 Hint: Consider how to use a balanced binary search tree for the drives.

Solution: Maintain the set of USB drives in a balanced binary search tree, T, ordered by their remaining storage capacity. Initially, we would simply insert a record for each drive, d, into T using d's storage capacity, which is initially 1 GB for each drive, as its search key. Next, for each image, I, we do a search in T to find the smallest drive, d, with a reserve capacity larger than the size of I. Then, we remove d from T, decrement its capacity by the size of I, and re-insert it back into T with this new capacity as its search key. Each of these operations takes $O(\log n)$ time, and repeating them for each image will implement the first fit algorithm. Thus, the entire running time of this implementation is $O(m \log n)$.

A-4.7 Hint: Consider using a binary search tree.

Solution: Store each nonzero value, v, from some $C[i, j, k]$, as a pair, $((i, j, k), v)$, in a binary search tree, B. To do a lookup, just search for (i, j, k) and if it is present, as $((i, j, k), v)$, return v; otherwise, return 0. To reduce a cell to 0, remove it from B, and to add 100 to a cell, either add 100 to its existing value in C, if it is nonzero, or cretae a new cell with value 100 at this position. Both the lookup and the search can be done in $O(\log n)$ time using a balanced search tree.

Chapter 5 Priority Queues and Heaps

Hints and Solutions

Reinforcement

R-5.1 Hint: Study the pseudocode description of this algorithm given in the book.

R-5.2 Hint: Study the pseudocode description of this algorithm given in the book.

R-5.3 Hint: Consider the mapping between nodes and array locations.

R-5.4 Solution: The iterations of selection sort are as follows:

2215	3644	103	913	2925	(5.1)
315	3644	1022	913	2925	(5.2)
39	3644	1022	1513	2925	(5.3)
39	1044	3622	1513	2925	(5.4)
39	1013	3622	1544	2925	(5.5)
39	1013	1522	3644	2925	(5.6)
39	1013	1522	3644	2925	(5.7)
39	1013	1522	2544	2936	(5.8)
39	1013	1522	2529	4436	(5.9)
39	1013	1522	2529	3644	(5.10)
					(5.11)

R-5.5 Solution:

2215	3644	103	913	2925	(5.12)
1522	3644	103	913	2925	(5.13)
1522	3644	103	913	2925	(5.14)
1015	2236	443	913	2925	(5.15)
310	1522	3644	913	2925	(5.16)
39	1015	2236	4413	2925	(5.17)
39	1013	1522	3644	2925	(5.18)
39	1013	1522	2936	4425	(5.19)
39	1013	1522	2529	3644	(5.20)
					(5.21)

R-5.6 Solution: A worst-case list for insertion sort would be one that is in descending order of keys, e.g., $44, 36, 29, 25, 22, 15, 13, 10, 9, 3$. With this list, each element will first be moved to the front and then moved back in the list incrementally, as every remaining is processed. Thus, each element will be moved n times. For n elements, this means at a total of n^2 times, which implies $\Omega(n^2)$ time overall.

R-5.7 Hint: Explicitly check the size of the array before adding a new node to the heap.

R-5.8 Solution: The largest key in a heap may be stored at any external node.

R-5.9

R-5.10 Solution: Yes, the tree T is a min-heap.

R-5.11 Solution: With a preorder traversal, a heap that produces its elements in sorted order is that which is represented by the array $(1, 2, 5, 3, 4, 6, 7)$. There does not exist a heap for which an inorder traversal produces the keys in sorted order. This is because in a heap the parent is always less than all of its children or greater than all of its children. The heap represented by $(7, 3, 6, 1, 2, 4, 5)$ is an example of one which produces its keys in sorted order during a postorder traversal.

R-5.12

R-5.13

R-5.14

R-5.15

Creativity

C-5.1 Hint: Consider both the best-case time and worst-case time and show that they are the same.

```
Algorithm BubbleSort($A$):
  Input: An array $A$ of $n$ comparable elements, indexed from 1 to $n$
  Output: An ordering of $A$ so that its elements are in nondecreasing order.
  for $i \leftarrow 1$ to $n-1$ do
      // Move the smallest element in $A[i+1\,..\,n]$ to $A[i]$ by swaps.
      for $j \leftarrow n$ down to $i+1$ do
          if $A[j-1] > A[j]$ then
              Swap $A[j-1]$ and $A[j]$
  return $A$
```

Algorithm 5.1: The bubble-sort algorithm.

C-5.2 Hint: Note that we resolve an inversion each time we move an element to the right in the array as a part of the inner loop of the insertion-sort algorithm.

C-5.3 Hint: Partition the array into a sorted part and an unsorted part. Structure the unsorted part like a heap. Think about the storage changes that need to occur after a removeMin.

C-5.4

C-5.5 Hint: Structure the insertions so that each requires lots of down-heap bubbling.

C-5.6 Hint: Consider the binary expansion of $n-1$, n, and $n+1$.

Solution: The path to the last node in the heap is given by the path represented by the binary expansion of n with the highest-order bit removed.

C-5.7 Hint: Consider the subtree of the heap containing elements less than or equal to the kth smallest element.

C-5.8 Solution: Construct a heap, which takes $O(n)$ time. Then call removeMinElement k times, which takes $O(k \log n)$ time.

C-5.9 Hint: Think about where the keys smaller than x are stored in the heap T.

Solution: Starting at the root of the tree, recursively search the left and right subtrees if the root of the subtree has a key value less than x. This algorithm takes $O(k)$ time, because there is no node in T storing a key larger than x that has a descendant storing a key less than x.

C-5.10 Hint: Think about how the LIFO protocol impacts the minimum and maximum.

C-5.11 Hint: Use the solution to the previous exercise and simulate the queue with two stacks.

Applications

A-5.1 Hint: Use two priority queues.

Solution: Keep track of the median in a variable, m. Maintain the set of values in S less than m in a maximum-oriented priority queue, L, and the set of values in S greater than m in a minimum-oriented priority queue, G. To perform a median() operation, just return the value of m. To perform an insert(x) operation, determine whether x should be added to L or G. Suppose as a result, one these groups grows to have two more values than the other, say, G is the larger one. Then add m to L, remove the minimum from G, and let it be the new m.

A-5.2 Hint: Use a heap.

A-5.3 Hint: Use a priority queue that has locator objects.

Solution: Store upgrade requests in a priority queue, Q, that is implemented with locator objects. In addition, store locators into Q for each upgrade request in an array, D. When a flyer requests an upgrade, add his or her request to Q, and store the locator for this request in D, giving the flyer the index, i, in D for this locator as the confirmation number. If a flyer requests a cancellation, using a confirmation number, i, then look up their locator, $D[i]$, and use it to remove their request from Q. To process k upgrades, perform k removeMin operations on Q. This allows all update operations to be performed in $O(\log n)$ time and for k upgrades to be processed in $O(k \log n)$ time.

A-5.4 Hint: Note that you need to be maintaining both the minimum and the maximum for both buy and sell orders.

A-5.5 Hint: Use a priority queue using floating-point magnitudes as keys.

A-5.6 Hint: Consider using one of the sorting algorithms given in this chapter.

Solution: Let us sort the input sequence of packets using insertion-sort. Because each input packet is within $O(1)$ positions of its final place in the sorted output sequence, the number of times any packet moves or causes another packet to move during the inner loop of insertion-sort is $O(1)$ in this case. Thus, the total running time of this algorithm is $O(n)$.

Another solution is to allocate an array, A of size n, indexed from 1 to n, and place each input packet with sequence number k into the cell $A[k-N+1]$. Then reading out the cells of A in order will give the output sequence in sorted order.

Chapter 6 Hash Tables

Hints and Solutions

Reinforcement

R-6.1 Hint: Review the definitions of separate chaining and load factors.

R-6.2 Hint: Review the definitions of open addressing and load factors.

R-6.3

R-6.4

R-6.5

R-6.6

R-6.7

R-6.8

R-6.9

Creativity

C-6.1

C-6.2

C-6.3

C-6.4 Hint: Consider the case when $f(k)$ and N have a common factor.

C-6.5 Hint: You are allowed to use additional memory or to mark the cells of the two tables.

C-6.6 Hint: Use a hash table with linked lists attached.

C-6.7 Hint: Think of how you might generalize the polynomial hash function.

Applications

A-6.1 Hint: Consider how to use a hash table for this solution.

Solution: A good solution is to store the set of currently admitted patients in a hash table. This allows both admissions and discharges to run in expected $O(1)$ time, which should cut down long lines of patients at discharge time.

A-6.2 Hint: Stick to files that are in the same language so as to keep the statistics consistent.

A-6.3 Hint: Consider how to use a hash table for this solution.

A-6.4 Hint: Consider using a cuckoo hash table. What should be the keys and values, though?

Solution: Create a cuckoo hash table, T, that uses (i,j) pairs as keys and counts, c, as values, where a count of c indicates that the pair of (i,j) has occurred c times at half-time in a Bears-Anteaters game. Initially, T is empty. For the processing task, scan through the list of half-time scores. For each (i,j), do a lookup in T using this key. If you find an entry, c, then remove this entry for (i,j) and replace it with $c+1$. If you did not find an entry for (i,j), on the other hand, insert a new entry with $c = 1$. At half-time, you can then perform the query task by doing a lookup for the current score, (i,j), in T, to see the number of times (i,j) has occurred before (and if you don't find (i,j) in T, this is the first time there has been a score of i-versus-j between these two teams). Then the processing phase will take linear expected time and the query task will take worst-case constant time.

A-6.5 Hint: Consider using the polynomial hash function and figure out the difference between the hash value already computed for a string of m words and the hash value for adding one more word on the right and dropping the first.

Solution: Use a hash table of size at least $2N$, where N is the total length of the documents in D. Also, use a polynomial hash function, f, to map a sequence of m words $W = (w_1, w_2, \ldots, w_m)$ to hash values, based on calls to h. The important observation is that if you have the hash value, $f(W)$, for a sequence, $W = (w_1, w_2, \ldots, w_m)$, of m words, then we can compute the hash value, $f(W')$, for the next sequence, $W' = (w_2, w_3, \ldots, w_{m+1})$, which adds one more word on the right and drops the one on the left, using the formula $f(W') = f(W)a - w_1 a^{m-1} + w_{m+1}$, where a is the constant used in the polynomial hash function f. This allows us to process the document d in $O(n+m)$ expected time instead of $O(nm)$ time, assuming there are not too many collisions (which each take $O(m)$ time to see if it is an act of plagiarism).

A-6.6

A-6.7 Hint: Use a hash table.

A-6.8 Hint: Use a hash table and shoot for $O(n + m)$ time.

A-6.9 Hint: Use a hash table as a way to keep track of word counts.

A-6.10 Hint: Build a hash table for D and then think about how to use it to process w. Also, note that the number of letters in the English alphabet is a constant.

Chapter 7 Union-Find Structures

Hints and Solutions

Reinforcement

R-7.1 Hint: Draw a picture.

R-7.2 Hint: Note that the find method is called again in the arguments to the call to the union operation.

Solution: The makeSet method is called n times, the union method is called $n - c$ times, and the find method is called $2m + 2(n - c)$ times.

R-7.3 Hint: The maze consists of exactly one connected component.

Solution: The maze construction algorithm removes $n - 1$ edges, so in this case it removed 899 edges.

R-7.4 Hint: Consider how this assumption changes the performance of the operations in the sequence.

R-7.5 Hint: The set is represented as a linked list of its elements.

Solution: Simply traverse the linked list of the elements for a set by following the pointer from the head node.

R-7.6 Hint: Start with 0 and work your way up.

R-7.7 Hint: Start with 0 and work your way up.

R-7.8 Hint: Start with 0 and work your way up.

Creativity

C-7.1 Hint: Think about the amortized time for growing an array.

C-7.2 Hint: Consider how you might modify the linked list so that removals take constant time.

Solution: Change the representation of the linked lists so that they are now doubly linked lists. This will allow for removals to be performed in $O(1)$ time.

C-7.3 Hint: Generalize the linked-list approach.

C-7.4 Hint: Sort first.

Solution: First we sort the objects of A. Then we can walk through the sorted sequence and remove all duplicates. This takes $O(n \log n)$ time to sort and n time to remove the duplicates. Overall, therefore, this is an $O(n \log n)$-time method.

C-7.5 Hint: Go through the details of the proof given in the book and note that pretty much every step can still be used in this case.

C-7.6 Hint: Note that the amortization argument gets really easy.

C-7.7 Hint: Consider how you might combine the simple argument used for the tree-based implementation with the amortization argument used for the list-based method.

Solution: First, observe that there can be at most n union operations and each takes $O(1)$ time. So the total time for unions is $O(n)$, which is $O(m)$. Let us use an amortization scheme for any find operation, where we charge 1 cyberdollar to the find operation for the last pointer hop, which goes from a node to the root (hence, we don't do a path compression for this node). For any node where we go from the node to a non-root parent in doing a find, let us charge this node 1 cyberdollar. But observe that if we make such a charge, then, because of path compression, the node now points to a new parent node whose size is at least double that of its previous parent. Thus, the total number of such charges that any element can receive is $\log n$. Therefore, since there are n elements, the total running time is $O(n \log n + m)$, which is $O(m)$ in this case.

C-7.8 Hint: Consider how bad a tree might look if you did the unions in a bad order.

Applications

A-7.1 Hint: You will probably need to create a super-top cell and a super-bottom cell and connect them to the top and bottom cells, respectively.

A-7.2 Hint: Group sequences of insertions into sets, which are separated by removeMin() operations. When a removeMin() is answered and deleted from σ, we would then need to union the sets of insertions on either side of this operation.

A-7.3 Hint: Keep track of the sets of contiguous regions that you have seen so far using a union-find structure.

A-7.4 Hint: The entire left and right boundaries are possible places where a connection could be made.

A-7.5 Hint: Think about how you might augment connected components to keep track of the number of gold cells they contain.

Chapter 8 Merge-Sort and Quick-Sort

Hints and Solutions

Reinforcement

R-8.1 Solution: For each element in a sequence of size n there is exactly one exterior node in the merge-sort tree associated with it. Since the merge-sort tree is binary with exactly n exterior nodes, we know it has height $\lceil \log n \rceil$.

R-8.2 Hint: Study how the nodes are moved in the figure illustrating the merge operation for two sorted linked lists.

R-8.3

R-8.4 Hint: Consider how the pivots work on a sorted sequence.

Solution: $O(n \log n)$ time. This is because the pivot splits a sorted sequence in half each time, since it is the median.

R-8.5 Hint: Review the worst-case performance example for standard quick-sort.

Solution: The sequence should have the property that the selected pivot is the largest element in the subsequence. This is to say that if G contains no elements, the worst case $\Theta(n^2)$ bound is obtained.

R-8.6

R-8.7 Hint: Notice where the partitioning stops.

Creativity

C-8.1 Hint: Avoid the use of recursion, by doing things bottom-up by levels, and use the auxiliary array as a "buffer" so as to swap S and T with each level.

C-8.2 Hint: Sort first.

Solution: First we sort the objects of A. Then we can walk through the sorted sequence and remove all duplicates. This takes $O(n \log n)$ time to sort and n time to remove the duplicates. Overall, therefore, this is an $O(n \log n)$-time method.

C-8.3 Solution: Merge sequences A and B into a new sequence C (i.e., call merge(A, B, C)). Do a linear scan through the sequence C removing all duplicate elements (i.e., if the next element is equal to the current element, remove it).

C-8.4 Hint: Use the ***Chernoff bound*** that states that if we flip a coin k times, then the probability that we get fewer than $k/16$ heads is less than $2^{-k/8}$.

C-8.5 Solution: For the red and blue elements, we can order them by doing the following. Start with a marker at the beginning of the array and one at the end of the array. While the first marker is at a blue element, continue incrementing its index. Likewise, when the second marker is at a red element, continue decrementing its index. When the first marker has reached a red element and the second a blue element, swap the elements. Continue moving the markers and swapping until they meet. At this point, the sequence is ordered. With three colors in the sequence, we can order it by doing the above algorithm twice. In the first run, we will move one color to the front, swapping back elements of the other two colors. Then we can start at the end of the first run and swap the elements of the other two colors in exactly the same way as before. Only this time the first marker will begin where it stopped at the end of the first run.

C-8.6 Hint: Sort A and B first.

C-8.7 Hint: Sort first.

Solution: Sort the elements of S, which takes $O(n \log n)$ time. Then, step through the sequence looking for two consecutive elements that are equal, which takes an additional $O(n)$ time. Overall, this method takes $O(n \log n)$ time.

C-8.8 Hint: Study how things change when an element is equal to the pivot.

C-8.9 Hint: Try to modify the merge-sort algorithm to solve this problem.

C-8.10 Hint: Consider a sequence that is opposite of a sequence with no inversions.

Solution: Any reverse-ordered sequence (in descending order) will serve as an example.

C-8.11 Hint: Consider the graph of the equation $x = a + b$ for a fixed value of x.

C-8.12 Hint: Consider why this problem might be in this chapter.

Solution: Sort the numbers by nondecreasing values. Next we can scan the sequence to keep track, for each run of numbers that are all the same, how long that sequence is. In addition, we can store the length and x_i-value for the longest sequence we have seen so far. When we complete the scan of the sequence, this x_i value is the mode. The running time for this method is dominated by the time to sort the sequence, which can be done in $O(n \log n)$ time in the worst-case by using the merge-sort algorithm.

C-8.13 Hint: Think about what would occur if you repeatedly asked for the same comparison until you get a valid answer.

Solution: One possible way to get Rustbucket to sort n elements is to use randomized quick-sort, but modify the comparisons so that each comparison is repeated until Rustbucket returns a response of 1 (for **true**) or -1 (for **false**), instead of 0. The expected running time of this algorithm can be shown to be $O(n \log n)$ by a simple modification to the analysis given in the book. The main modification is that the analysis in the book assumes that the worst-case running time to do a partition is $O(n)$, whereas this modified algorithm implemeneted with Rustbucket the expected time to do a partition is $O(n)$, since each comparison now has an expected number of 2 times to be repeated until it gets a result that is not 0. The total running time is still a sum of the running time for each level in a recursion tree of expected depth $O(\log n)$, but now the running time for each level has an expected value of $O(n)$ instead of having this value in the worst case. Nevertheless, by the linearity of expectation, we can sum up all these expected running times to get a total expected running time for sorting on Rustbucket that is $O(n \log n)$.

Another solution is to use merge-sort, but repeat each comparison until it is successful. In this case, merging to lists will run in $O(n)$ time, since the expected number of comparisons before getting one to work is 2. Thus, the expected running time of this version of merge-sort is characterized by the equation, $T(n) = 2T(n/2) + n$, which is $O(n \log n)$.

Applications

A-8.1 Hint: Think of generalizing merge-sort so that it works with a k-ary recursion tree.

A-8.2 Hint: Sort first, choosing the appropriate key for comparisons.

Solution: First sort the sequence S by the candidate's ID. Then walk through the sorted sequence, storing the current max count and the count of the current candidate ID as you go. When you move on to a new ID, check it against the current max and replace the max if necessary.

A-8.3 Hint: Think of a data structure that can be used for sorting in a way that only stores k elements when there are only k distinct keys.

Solution: In this case we can store candidate ID's in a balanced search tree, such as an AVL tree or red-black tree, where in addition to each ID we store in this tree the number of votes that ID has received. Initially, all such counts are 0. Then, we traverse the sequence of votes, incrementing the count for the appropriate ID with each vote. Since this data structure stored k elements, each such search and update takes $O(\log k)$ time. Thus, the total time is $O(n \log k)$.

A-8.4 Hint: Try to design an efficient divide-and-conquer algorithm.

Solution: This problem can be solved using a divide-and-conquer approach. First, we choose a random bolt and partition the remaining nuts around it. Then we take the nut that matches the chosen bolt and partition the remaining bolts around it. We can continue doing this until all the nuts and bolts are matched up. In essence, we are doing the randomized quicksort algorithm. Thus, we have an average running time of $O(n \log n)$.

A-8.5 Hint: Review the merging algorithm for the merge-sort algorithm.

Solution: The algorithm is similar to the merge algorithm for the merge-sort algorithms. The merging is for the document identifiers, ordered by their ranking scores. Any time a document ID is found in the two lists, it is copied to the output list. If an ID is found only in one list, it is discarded.

A-8.6 Hint: Notice how the merge algorithm has a similar kind of running time to the standard merge-sort algorithm.

Solution: The OddEvenMerge algorithm has the same running-time analysis as merge-sort, in that its running time can be characterized by the recurrence equation, $t(n) = 2t(n/2) + bn$, for some constant b. Thus, the OddEven-Merge algorithm runs in $O(n \log n)$ time. Plugging this into the merge step of the merge-sort algorithm, then results in a recurrence for the running time that is $T(n) = 2T(n/2) + cn \log n$, for some constant c. This implies that the running time for odd-even merge-sort is $O(n \log^2 n)$.

A-8.7 Hint: Prove the probability statement by induction, and analyze the running time in a similar way as was done for merge-sort.

Solution: We can prove that every card has an equal probability of being the top card by induction. If we have just a single card in the deck, this is clearly true. Otherwise, suppose inductively that after doing a recursive-riffle, each card in the first half is equally likely to be the top card in its deck and each card in the second half is equally likely to be the top card in its deck. Since we then choose randomly between these two cards in the first step of a riffle-shuffle, every original card is equally likely to be the top card after a random-riffle algorithm is applied to the deck. For the running time, note that it can be characterized using the recurrence equation, $T(n) = 2T(n/2) + bn$, for some constant b, which implies that the running time is $O(n \log n)$.

A-8.8 Hint: You may wish to use an auxiliary data structure in addition to the array, A, and the input list of names.

Solution: Create an auxiliary array, B, of size m that is indexed by the standard ordering of the letters in the alphabet, e.g., using their ASCII or Unicode representations. For any letter, x, $B[x]$ should store the rank of x in the new alphabetic ordering. In order to construct B, we need to sort pairs, (x, i), where x is a letter from the alphabet and i is its rank in the new alphabetic ordering, and the ordering we use to sort the pairs is the standard alphabetic ordering for their first coordinate (which is a letter). This can be done in $O(m \log m)$ time, e.g., by merge-sort. Then we sort the names, e.g., also by merge-sort. Given the description of merge-sort from the book, the only detail that needs to be spelled out is how we perform comparisons. For this we compare two characters, x and y, at similar places in two different names, by comparing $B[x]$ and $B[y]$, to see which rank number is lower. Since names are assumed to be constant length, this implies that we can compare two names in constant time according to this alternative order in $O(1)$ time. Thus, we can sort the collection of candidate names in $O(n \log n)$ time. Therefore, the total time needed for this algorithm is $O(m \log m + n \log n)$.

A-8.9 Hint: Consider a proof by contradiction, where you first assume there is an arrangement that is not sorted and produces an optimal error term, e_n, and then show that this leads to a contradiction.

Solution: Suppose there is a better arrangement for the x_i's that is not in sorted nondecreasing order that results in a minimum error term, e_n. Since this ordering is not sorted, there is a pair of numbers, x_j and x_k, such that $j < k$ and $x_j > x_k$. Notice that the contribution to e_n for these two terms is ϵ times $(n - j - 1)x_j$ and $(n - k - 1)x_k$. Since $x_j > x_k$, and $j < k$, we have that $x_j - x_k > 0$ and $k - j > 0$. This implies that

$$\begin{aligned}(n-j+1)x_j + (n-k+1)x_k - [(n-k+1)x_j + (n-j+1)x_k] &= \\ (k-j)x_j + (j-k)x_k &= \\ (k-j)\cdot(x_j - x_k) &> 0.\end{aligned}$$

Thus, we could swap x_j and x_k in the arrangement and reduce what was supposed to be the optimal value for e_n. Therefore, giving the numbers in sorted nondecreasing order is best. Hence, by using the merge-sort algorithm for sorting the n floating-point numbers, we can arrange them in an order that minimizes the error term, e_n, in $O(n \log n)$ time.

Chapter 9 Fast Sorting and Selection

Hints and Solutions

Reinforcement

R-9.1 Hint: Review each of these comparison-based algorithms.

Solution: The bubble-sort and merge-sort algorithms are stable.

R-9.2 Hint: Review the case for $d = 2$ and generalize this approach.

Solution: Preform the stable bucket sort on m, l, and then k. In general, perform the stable bucket sort on k_d down to k_1.

R-9.3 Hint: Recall that an algorithm is in-place if it uses a small amount of memory in addition to the input itself.

Solution: No. Bucket-sort does not use a constant amount of additional storage. It uses $O(n + N)$ space.

R-9.4 Hint: Review the in-place quick-sort algorithm.

R-9.5 Hint: Review the quick-select algorithm.

R-9.6 Hint: Derive a new recurrence equation for this alternative algorithm.

R-9.7 Hint: Study the pseudocode for the algorithm.

Solution: The weighted median algorithm returns the median element in this case.

Creativity

C-9.1 Hint: Change the way elements are compared with each other.

Solution: Change the way elements are compared with each other, by replacing each x_i with the pair, (x_i, i). Now do all comparisons lexicographically. This approach guarantees that if $x_i = x_j$, then the comparison will be resolved by using the lexicographic rule that $(x_i, i) < (x_j, j)$, if $i < j$.

C-9.2 Hint: Sort A and B first.

C-9.3 Hint: Think of alternate ways of viewing the elements.

Solution: To sort S, do a radix sort on the bits of each of the n elements, viewing them as pairs (i, j) such that i and j are integers in the range $[0, n-1]$.

C-9.4 Hint: Find a way of sorting them as a group that keeps each sequence contiguous in the final listing.

C-9.5 Hint: Sort first.

Solution: Sort the elements of S, by radix-sort, viewing the elements as triples of numbers in the range from 1 to n, which takes $O(n)$ time in this case. Then, step through the sequence looking for two consecutive elements that are equal, which takes an additional $O(n)$ time. Overall, this method takes $O(n)$ time.

C-9.6 Hint: Consider the graph of the equation $x = a + b$ for a fixed value of x.

Solution: Replace each a in A with $x - a$, and do a radix sort of these elements with the elements in B, viewing the integers as 4-tuples of numbers in the range from 1 to n. If there is a duplicate, with one element in A and the other in B, then this implies the answer is "yes."

C-9.7 Hint: Perform a selection first on some appropriate order statistics.

C-9.8 Hint: Think about what would be the perfect pivot.

C-9.9 Hint: Use linear-time selection in an appropriate way.

C-9.10 Hint: Review the binary search algorithm and try to do a coordinated search in A and B simultaneously.

C-9.11 Hint: Start by constructing a binary tree with all the elements initially stored in the external nodes.

C-9.12 Hint: Consider using bucket-sort first.

C-9.13 Hint: Consider the probability that all the pivots are bad.

Applications

A-9.1 Hint: Think about what would occur if you repeatedly asked for the same comparison until you get a valid answer.

A-9.2 Hint: Consider a type of radix-sort.

Solution: An efficient algorithm is to sort the tuples using radix sort, but using the ordering (w, r, d). This algorithm runs in time $O(n)$.

A-9.3 Hint: Consider how to use the selection algorithm to solve this problem.

A-9.4 Hint: Think of how to use the selection algorithm.

Solution: Use the linear-time selection algorithm to find the median number, x, in A. Then scan A to count the number of times x appears in A. If it appears more than $n/2$ times, then it is the majority. If not, then there is no majority number in A, since the majority number must be the median.

A-9.5 Hint: Think about how to use the linear-time selection algorithm.

Solution: Use the linear-time selection algorithm to search for the items, x and y, of rank $n/3$ and $2n/3$, respectively. Note that if a candidate has received over $n/3$ votes, then his or her student number must be included in at least one of the items at rank $n/3$ and $2n/3$. So, given x and y, we scan A once more to count how many times x and y appear in A. We then output which of these received more than $n/3$ votes (which could possibly be both).

A-9.6 Hint: Consider using the linear-time selection algorithm.

Solution: Use the linear-time selection algorithm to compute the median, c, of the z-coordinates of the points in S. The plane $z = c$ will minimize the flatness score, $F(S)$. Given c, it is a simple matter to then compute the value of $F(S)$ in $O(n)$ additional time by scanning through the points in S.

A-9.7 Hint: Consider using a weighted median here.

Chapter 10 The Greedy Method

Hints and Solutions

Reinforcement

R-10.1

R-10.2

R-10.3

R-10.4 Hint: Don't forget to include the space character.

R-10.5 Hint: Think about powers of 2.

Solution: A set of characters with the following frequencies will work:

$$\{1, 1, 2, 4, 8, 16, 32, 64, 128, 256\}.$$

R-10.6 Hint: Recall the lemma that the two lowest-frequency characters are at the lowest depth in a Huffman tree.

Solution: The solution is any set of 8 characters where each character has exactly the same frequency.

R-10.7 Hint: Consider drawing the Huffman tree for these code words.

R-10.8 Hint: Consider drawing the Huffman tree for these code words.

R-10.9 Hint: Consider drawing the Huffman tree for these code words.

R-10.10 Hint: Review the definitions for these terms and then study the proof of correctness for the Huffman coding algorithm.

Solution: The proof of correctness for the Huffman coding algorithm uses exchange arguments in the proofs of both lemmas.

Creativity

C-10.1 Hint: Think about how to create items with large weights and high benefit.

Solution: With a knapsack of weight 10, consider items with weight-benefit pairs, $(1, 40)$ and $(10, 50)$. This greedy choice picks all of item 2, with benefit 50, whereas choosing item 1 first, and 9 units of item 2, gives benefit $40 + 45 = 95$.

C-10.2 Hint: You may use the fact that it is possible to solve the selection problem for n numbers in $O(n)$ time.

Solution: Suppose the weight of each item is w and we have a knapsack of size W. Then we need to fill the knapsack with the top $\lfloor W/w \rfloor$ items based on their benefits, and then possibly a fraction of the item with benefit rank $\lfloor W/w \rfloor$. Thus, if we used linear-time selection to find the item with benefit rank $\lfloor W/w \rfloor$ in linear time, using the selection algorithm given in the chapter on Selection, then we can scan through the items to find the ones with benefits greater than this item, which all go in the knapsack, and then we can determine the fraction of this item that should go in the knapsack as well. All of this computation therefore runs in $O(n)$ time.

C-10.3 Hint: Use a lookup table.

Solution: Use a lookup or hash table, F, for the character set for X, where each $F[c]$ is initially 0. Then scan the string, X, and, for each $X[i]$, increment the value of $F[X[i]]$. When the scan for X is complete, output each c and $F[c]$ where $F[c] > 0$.

C-10.4 Solution: Anatjari should use a greedy algorithm. He should draw a line between every pair of watering holes that are within k miles of one another, for he can get from one to the other with one bottle of water. Anatjari should then use a path that uses the fewest number of lines and leads from his present position to a watering hole that is within with k miles of the other side. There can be no path with fewer stops, for he is including in his minimization all pairs of watering holes that can be reached with one bottle of water.

C-10.5 Solution: First give as many quarters as possible, then dimes, then nickles and finally pennies.

C-10.6 Solution: If the denominations are \$0.25, \$0.24, \$0.01, then a greedy algorithm for making change for 48 cents would give 1 quarter and 23 pennies.

C-10.7 Hint: Recall the way the Huffman algorithm processes characters based on their frequencies.

Solution: Sketch: Let F_i be the sum of the first i nonzero Fibonacci numbers. We can show, by induction, that $f_i > F_{i-2}$. Thus, at every step of Huffman's algorithm, the two lowest-frequency trees are going to be the one with weight F_{i-2} and f_{i-1}.

C-10.8 Hint: Try out some examples with just handful of men and spears.

Solution: This approach doesn't work, which can be seen with a small example of two men of height 3 and 6 and spears of height 1 and 4. This greedy algorithm will match the 4-spear with the 3-man and the 1-spear with the 6-man, for a cost of 6, whereas the optimal solution is to match the 1-spear with the 3-man and the 4-spear with the 6-man, for a cost of 4.

C-10.9 Hint: Consider a proof based on the exchange principle, where you have an optimal alternative solution, which you can improve by performing an exchange.

C-10.10 Hint: Consider an example where the enemies of the friend with the fewest enemies all get along.

Solution: Here is a counter-example set of friends and their enemies lists. Alice: Bob, Charles, David. Bob: Alice, Eve, Frank, George. Charles: Alice, Eve, Frank, George. David: Alice, Eve, Frank, George. Eve: Bob, Charles, David, Frank, George. Frank: Bob, Charles, David, Frank, George. George: Bob, Charles, David, Frank, George. The greedy strategy will first choose Alice and then one of Eve, Frank, or George, for an invitee list of size 2. The optimal solution, however, is Bob, Charles, and David, which has size 3.

Applications

A-10.1 Solution: We can use a greedy algorithm, which seeks to cover all the designated points on L with the fewest number of length-2 intervals (for such an interval is the distance one guard can protect). This greedy algorithm starts with x_0 and covers all the points that are within distance 2 of x_0. If x_i is the next uncovered point, then we repeat this same covering step starting from x_i. We then repeat this process until we have covered all the points in X.

A-10.2 Hint: Consider a greedy choice based on considering tasks by increasing finish times.

A-10.3 Hint: Notice the similarity of this problem to that of Huffman coding.

Solution: Let T be a Huffman tree, which is constructed using the x_i's as the weights of their associated external nodes. Then this tree will minimize the total path weight over all such trees, which in turn will provide a minimum value for e_n. The running time of this method is $O(n \log n)$.

A-10.4 Hint: Use a simple greedy method and then prove it is correct using an exchange argument.

A-10.5 Hint: Consider a simple case with three lines of text.

A-10.6 Hint: Draw the Huffman trees that can arise from different scenarios.

A-10.7 Hint: Consider using a greedy approach here.

Chapter 11 Divide-and-Conquer

Hints and Solutions

Reinforcement

R-11.1 Solution:

a. $T(n)$ is $O(n)$ (case 1).
b. $T(n)$ is $O(n^3)$ (case 1).
c. $T(n)$ is $O(n^4 \log^5 n)$ (case 2).
d. $T(n)$ is $O(n^{\log_3 7})$ (case 1).
e. $T(n)$ is $O(n^3 \log n)$ (case 3).

R-11.2

R-11.3

R-11.4

R-11.5 Hint: Recall when a point is dominated by another.

Solution: $\{(6,12),(8,6),(9,3)\}$.

R-11.6 Hint: Review the concept of when one point dominates another.

Solution: $\{(1,n),(2,n-1),\ldots,(i,n-i+1),\ldots,(n,1)\}$.

Creativity

C-11.1 Hint: Use a good induction hypothesis, such as $T(n) = 2^n - 1$.

C-11.2 Hint: Generalize case 2 of the master theorem.

C-11.3 Hint: Note that there are 3 recursive calls of size $2n/3$ each.

Solution: The proof of correctness is a simple induction proof and is omitted here. The recurrence for $T(n)$ is $T(n) = 3T(2n/3)+bn$, or $T(n) = 3T(2n/3)+b$ (if we can do constant-time array passing), which, by the master theorem, is $O(n^{\log 3/\log(3/2)})$, which is roughly $O(n^{2.7})$.

C-11.4 Hint: Note that there are 3 recursive calls of size $3n/4$ each.

Solution: The recurrence for $T(n)$ is $T(n) = 3T(3n/4) + bn$, or $T(n) = 3T(3n/4) + b$ (if we can do constant-time array passing), for the general case, and is a constant for $n \leq 4$. By the master theorem, the implies $T(n)$ is $O(n^{\log 3/\log(4/3)})$, which is roughly $O(n^{3.8})$.

C-11.5 Hint: Use a proof by induction.

Applications

A-11.1 Hint: Think about what happens in the base case.

Solution: Divide the set of numbers into $n/2$ groups of two elements each. With $n/2$ comparisons we will determine a minimum and a maximum in each group. Separate the maximums and minimums, and find the minimum of the minimums and the maximum of the maximums using the standard algorithm. These additional scans use $n/2$ comparisons each.

A-11.2 Hint: Use divide-and-conquer.

A-11.3 Hint: Use divide-and-conquer. If you need more intuition, draw a picture of the function, f, for a set of 5–8 linear inequalities.

A-11.4 Hint: First determine the probability any given point, (i, y_i), is a maximum point.

Solution: Since all the points in the set R have distinct x- and y-coordinates, and the x-coordinate of the i-th point is i, the probability that a point, (i, y_i), is a maximum is $1/(n - i + 1)$, since this is the probability that y_i is the maximum point among the set $\{y_i, \ldots, y_n\}$. Thus, the expected size of the maxima set is

$$\sum_{i=1}^{n} = \frac{1}{n-i+1} = \sum_{i=1}^{n} \frac{1}{i},$$

which is the nth harmonic number. Thus, the expected size of the maxima set of a random set of points in the plane is $O(\log n)$.

A-11.5 Hint: Consider using a k-way merge-sort algorithm, with k recursive subproblems.

Solution: Perform a k-way merge-sort, where you subdivide the input sequence into k lists of size n/k each, and recursively sort each one. Once these are done, you can do a k-way merge of the sorted lists using the bank of registers. Load the front of the list i as a pair (x, i), where x is its key. Removing the minimum each time is the element that needs to go in the output list, and its index, i, is the list it came from. So you can then load in the next smallest element, x', in that list, as (x', i). This allows us to merge k lists in $O(n)$ time. Therefore, the recurrence equation is $T(n) = kT(n/k) + bn$, where b is a constant. By the recursion-tree method, this running time is $O(n \log_k n)$, which is $O(n \log n / \log k)$.

A-11.6 Hint: Use divide-and-conquer.

Chapter 12 Dynamic Programming

Hints and Solutions

Reinforcement

R-12.1 Hint: Review the matrix chain-product algorithm.

R-12.2 Hint: Recall the way to remember where the parenthesization is done.

R-12.3 Hint: Bob making a bad choice improves Alice's potential winnings, but does it change the value of any of the $M_{i,j}$'s for game states that remain?

Solution: Bob making a bad choice improves Alice's potential winnings, but it does not violate the assumptions used to build the $M_{i,j}$ values for states of the game that remain. So Alice does not need to rebuild her table of $M_{i,j}$ values. She simply needs to start following a different path in this table.

R-12.4 Hint: Don't try to be greedy; just simulate the dynamic programming algorithm on this instance.

R-12.5 Hint: Review the pseudo-polynomial-time algorithm for 0-1 knapsack.

R-12.6 Hint: Review the telescope scheduling algorithm.

R-12.7 Hint: Just work through the recurrence equation.

R-12.8 Hint: Review the LCS algorithm.

R-12.9 Hint: The weight of the sack is n.

Solution: This is a knapsack problem, where the weight of the sack is n, and each bid i corresponds to an item of weight k_i and value d_i. If each bidder i is unwilling to accept fewer than k_i widgets, then this is a 0/1 problem. If bidders are willing to accept partial lots, on the other hand, then this is a fractional version of the knapsack problem.

Creativity

C-12.1 Hint: Consider how Pascal's triangle applies to this problem.

Solution: (a) If we don't use memoization, then every application of the recursive equation doubles the number of calls that we make, until we hit a boundary condition.

(b) If we build Pascal's triangle, then we can apply memoization to this problem. The total number of arithmetic operations needed to compute $C(n,k)$ is $O(n^2)$ in this case.

C-12.2 Hint: Choosing the highest-valued available coin could reveal an even higher valued coin.

Solution: Consider the problem instance, $(10, 21, 12, 9)$. Using the greedy strategy, Alice will get 10 and 12, whereas Bob will get 21 and 9, to win.

C-12.3 Hint: Minimizing Bob's choice is not the same as maximizing the choices available to Alice.

Solution: Consider the problem instance $(1, 2, 1, 2, 1, 2, 1, 2, 3, 2)$. In order to deny Bob from getting the 3, Alice will choose 1, 1, 1, 1, and then get the 3, for a score of 7, whereas Bob gets 2, 2, 2, 2, 2, for a score of 10, and wins.

C-12.4 Hint: Consider the parity of the indices identifying character positions for the original copy of P.

Solution: If she wants to, Alice can take all the even-indexed coins and force Bob to take all the odd-indexed coins, or she can take all the odd-indexed coins and force Bob to take all the even-indexed coins. She can compute which of these choices is better in $O(n)$ time and then implement the strategy that gives her the better of the two.

C-12.5 Hint: Think of a fast way to sort the list by start and finish times.

Solution: Use radix-sort to sort the observations by increasing start and finish times in $O(n)$ time.

C-12.6 Hint: Review the LCS algorithm.

C-12.7 Hint: Use a balanced binary search tree.

C-12.8 Hint: Consider a proof by contradiction.

C-12.9 Hint: Review the LCS algorithm.

C-12.10 Hint: Review the method for longest common subsequence and modify the functions so that they work for substrings instead.

C-12.11 Hint: Extend the dynamic programming algorithm to three dimensions.

C-12.12 Hint: Think about how to add more information during updates.

Solution: Each time we update a value in the table, we need to additionally store the index of the item for which we are updating this entry. Once we have the optimal benefit value, we need only then trace back through these extra indices to determine the entire set of items to place in the knapsack.

C-12.13 Hint: Apply dynamic programming in a way similar to that used for the 0-1 knapsack problem.

C-12.14 Hint: There is a surprising similarity between this problem and the matrix chain-product problem.

C-12.15 Hint: Consider computing parameters of the form $N_{i,j}$, which is the set of nonterminal that generate the string $x_i x_{i+1} \cdots x_j$.

C-12.16 Hint: Index subproblems by nodes in T.

C-12.17 Hint: First, define $P(i,j) = \sum_{k=i}^{j} p_k$, for $1 \le i \le j \le n$. Then, consider how to compute $C(i,j)$, the cost of an optimal binary search tree for the keys that range from k_i to k_j.

Applications

A-12.1 Hint: To solve the making-change problem, use an approach similar to that used in the dynamic programming solution to the 0-1 knapsack problem.

A-12.2

A-12.3 Hint: Shoot for $O(n^2)$ time.

Solution: Define $D(j)$ to be **true** if and only if the first i characters of S can be divided into valid words. Then set $D(0) =$ **true**, and define $D(j)$ recursively so $D(j)$ is true if there is an $i < j$ such that $D(i-1) =$ **true** and $\text{valid}(S[i:j])$. Note that, in general, it takes $O(n)$ time to check if $D(i)$ is true, given D values for indices less than i. So the total running time of this algorithm is $O(n^2)$.

A-12.4 Hint: Use two indices to mark off a substring of S as the possible starting and ending places for a palindrome.

A-12.5 Hint: Define $W(i,j)$ to be the maximum weight way of transforming the first i characters of X into the first j characters of Y.

A-12.6

A-12.7 Hint: Consider altering the details of the telescope scheduling algorithm.

A-12.8 Hint: Define a cost function $C(i, j)$, which is the best way to match the first i numbers in X to the first j numbers in Y.

Solution: Define a cost function $C(i, j)$, which is the best way to match the first i numbers in X to the first j numbers in Y. Then $C(0,0) = 0$, $C(0,j) = \infty$, $C(i,0) = \infty$, and $C(i,j) = |x_i - y_j| + \min\{C(i-1,j), C(i,j-1), C(i-1,j-1)\}$. The solution is $C(n,m)$. This algorithm takes $O(nm)$ time, and if we store which of the choices in the above minimization we made, then we can determine the edge in the matching that is just prior to the edge (i,j), should we ultimately be able to reach this edge from the last edge, (n,m).

A-12.9 Hint: Think of how to generalize the solution to the coins-in-a-line game.

Solution: Let $M_{i,j}$ denote the maximum value of houses taken by Alice, from among the set of houses numbered i to j. Define $S[i,j]$ as the sum of the values from house i to house j. Then define $M_{i,j} = 0$, if $i > j$. This allows us to define, for $i \leq j$,

$$M_{i,j} = \max_{0 \leq k \leq j-i} \{M'_{i+k+1,j} + S[i, i+k], M'_{i,j-k+1} + S[j-k, j]\},$$

where

$$M'_{i,j} = \min_{0 \leq k \leq j-i} \{M_{i+k+1,j}, M_{i,j-k+1}\}.$$

Computing any $M_{i,j}$ takes time at most $O(n)$, and the value of $M_{1,n}$ gives us the optimal value for Alice. Therefore, the total running time of this algorithm is $O(n^3)$.

A-12.10 Hint: The Anteaters win the World Series if any of the events $(\lceil n/2 \rceil, j)$ occur, for $j < \lceil n/2 \rceil$.

Solution: $V(i,j) = pV(i-1,j) + (1-p)V(i,j-1)$, for $i, j < \lceil n/2 \rceil$.

$V(\lceil n/2 \rceil, j) = pV(\lceil n/2 \rceil - 1, j)$

$V(i, \lceil n/2 \rceil) = (1-p)V(i, \lceil n/2 \rceil - 1)$

The algorithm is to compute $V(\lceil n/2 \rceil, j)$ for all $j < \lceil n/2 \rceil$.

For instance, if $p = 1/2$ and $n = 5$, then $V(1,0) = .5$, $V(0,1) = .5$, $V(2,0) = 1/4$, $V(1,1) = 2/4$, $V(0,2) = 1/4$, $V(3,0) = 1/8$, $V(2,1) = 2/8 + 1/8 = 3/8$, $V(1,2) = 3/8$, $V(0,3) = 1/8$, $V(3,1) = .5V(2,1) = 3/16$, $V(3,2) = .5V(2,2) = .5(.5V(1,2) + .5V(2,1)) = .5(3/8) = 3/16$. The probability of $V(3,*) = V(3,0) + V(3,1) + V(3,2) = 1/8 + 3/16 + 3/16 = 1/2$.

A-12.11 Hint: Define subproblems in terms of the nodes in T.

Chapter 13 Graphs and Traversals

Hints and Solutions

Reinforcement

R-13.1 Hint: Review the combinatorial properties for graphs.

R-13.2 Hint: Review the definition of big-oh and the combinatorial property of how large m can be.

Solution: We know that $m \leq n \cdot (n-1)/2$ is $O(n^2)$. It follows, therefore, that $O(\log(n^2)) = O(2\log n) = O(\log n)$.

R-13.3 Hint: The Euler tour property is derived from the fact that the in-degree and out-degree of each vertex 2.

R-13.4 Hint: Model the courses and their prerequisites as a directed graph.

Solution: The topological sorting algorithm can help us solve this problem.

- Build a digraph to represent the course prerequisite requirements. The nine courses are vertices in the digraph. If a course A is a prerequisite for another course B, the digraph has a directed edge from A to B.
- Apply the topological sorting algorithm on this digraph. The result is one possible sequence of courses for Bob.

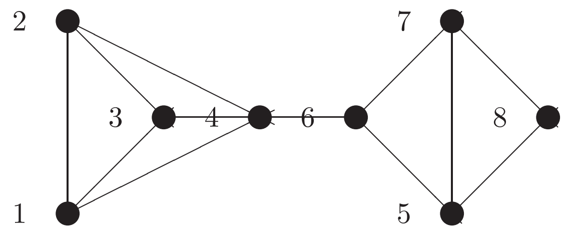
For the give set of prerequisites, the solution is not unique. For example, one possible solution is LA15, LA16, LA22, LA31, LA32, LA126, LA127, LA141, LA169. Another solution is LA15, LA16, LA127, LA31, LA32, LA169, LA22, LA126, LA141.

R-13.5 Hint: Consider the way that edges are represented.

Solution: Inserting an edges runs in $O(1)$ time since it is simply inserting an element for $e = (i, j)$ into the appropriate two locations, $A[i, j]$ and $A[j, i]$. To insert a vertex, on the other hand, we must construct a new version of the entire array, A. Clearly this will take $O(n^2)$ time.

R-13.6 Solution:

a. A drawing of graph G is given below:



b. A DFS traversal starting at vertex 1 visits the vertices in the following order: 1, 2, 3, 4, 6, 5, 7, 8.

c. A BFS traversal starting at vertex 1 visits the vertices in the following order: 1, 2, 3, 4, 6, 5, 7, 8.

R-13.7 Hint: Review the combinatorial properties for graphs.

Solution:

a. The adjacency list structure is preferable. Indeed, the adjacency matrix structure wastes a lot of space. It allocates entries for 100,000,000 edges while the graph has only 20,000 edges.

b. In general, both structures work well in this case. Regarding the space requirement, there is no clear winner. Note that the exact space usage of the two structures depends on the implementation details. The adjacency matrix structure is much better for operation areAdjacent, while the adjacency list structure is much better for operations insertVertex and removeVertex.

c. The adjacency matrix structure is preferable. Indeed, it supports operation areAdjacent in $O(1)$ time, irrespectively of the number of vertices or edges.

R-13.8

R-13.9 Hint: Review the definition of transitive closure.

R-13.10 Hint: Review the definition of topological ordering.

Solution: (BOS, JFK, MIA, ORD, DFW, SFO, LAX).

R-13.11 Hint: Review the code and how it would be implemented.

R-13.12 Hint: Review the DFS algorithm and how it labels edges.

R-13.13 Hint: Review the BFS algorithm and how it labels edges.

R-13.14 Hint: Review the directed DFS algorithm and how it labels edges.

R-13.15 Hint: Review the definition of biconnected component.

Solution: There would be three biconnected components in the graph after removing the edge (B,C) and the edge (N,K). One of these components is simply the edge (I,J).

Creativity

C-13.1 Hint: Consider using induction or contradiction.

C-13.2 Hint: Add the appropriate surrounding loop to your favorite graph traversal algorithm.

C-13.3 Hint: Suppose that such a nontree edge is a cross edge, and argue based upon the order the DFS visits the end vertices of this edge how this leads to a contradiction.

C-13.4 Hint: Use a data structure described in this book.

Solution: Use an adjacency list representation, but implement the adjacency list for each vertex using a balanced binary search tree. This would still use $O(n + m)$ space, but would allow an adjacency query for two vertices, v and w, to run in $O(\log n)$ time, by performing a query for w in the binary search tree of the adjacencies for v.

C-13.5 Hint: Prove that 1 implies 2, 2 implies 3, and 3 implies 1.

C-13.6 Hint: Draw some examples, for the sake of intuition, of graphs with separation edges.

C-13.7 Hint: Review the definition of a link component.

C-13.8 Hint: Start with the definition of a biconnected component and then combine the implications from the lemmas from the section that describes the LinkComponents algorithm.

C-13.9 Hint: Modify the list insertion and removal into enqueue and dequeue operations.

C-13.10 Hint: Justify this by induction on the length of a shortest path from the start vertex.

C-13.11 Hint: Consider first constructing an Euler tour traversal of the tree T.

Solution: Construct an Euler tour, P, of T, which visits each vertex first on the left, then possibly multiple times from below, and then finally on the right. Label each vertex, v, with the index, L_v, in P of the first visit for v (on the left) and the index, R_v, of the last visit for v (on the right). Notice that a node, v, is an ancestor of a node, w, if and only if the interval $[L_w, R_w]$ is contained inside the interval $[L_v, R_v]$. Thus, given the labels of the vertices, we can perform a scan of the edges in E' and label each edge as being either a back edge or a cross edge in $O(1)$ time each. Therefore, the running time of this labeling algorithm is $O(n + m)$.

C-13.12 Hint: Suppose there is an edge that is a forward edge and show why it would not be a nontree edge.

Solution: Suppose there is a nontree edge, e, that is a forward edge. Thus, e must go from a node, v, to a descendant, w. Note that w cannot be child of v, for otherwise v would have added w to the level list on the next level. Hence, w was added to a level list by a descendant of v to a level at least 2 away from v. But v could "see" w when v added all its outgoing adjacent unexplored vertices to the level list on the next level. Thus, w would have been added to this level list by v. Therefore, e cannot exist as a forward edge.

C-13.13 Hint: You may assume that you have access to both the incoming edges and outgoing edges for any vertex, v.

Solution: The directed DFS algorithm given in the book correctly identifies discovery edges and back edges, but it labels some edges as forward/cross edges. To distinguish between these two, we can add another for-loop to the algorithm, which is executed after the first for-loop. This second for-loop considers each incoming edge, $e = (w, v)$, for v, and if w is active, then it labels e as a forward edge. Then, in the place where we were previously labeling an edge as forward/cross edge, we would now label it as a cross edge.

C-13.14 Hint: Explain how there is a path from v to w and from w to v if the algorithm says that the graph is strongly connected.

C-13.15 Hint: There has to be a good traversal that does this.

Solution: Perform a DFS on G. G will have at least two cycles if and only if this DFS encounters at least two back edges.

C-13.16 Hint: Start with a DFS.

Solution: Perform a DFS on G. G will have at least two cycles if and only if this DFS encounters at least two back edges. If there are more than two back edges, then G has more than two cycles. So, for the first back edge, mark all the edges in the cycle determined by this edge and the DFS tree as "used." Then, for the second back edge, examine each of the edges in the cycle determined by this edge. If non of them is marked “used,” then these two cycles are disjoint. If one of them is “used,” then there are no two disjoint cycles.

C-13.17

C-13.18 Hint: Do a topological ordering first.

Applications

A-13.1 Hint: Start out with a natural traversal mentioned in this chapter and patch in the places this approach misses.

Solution:

First, even if it is not asked, we show that a connected directed graph has an Euler tour, if and only if, each vertex has the same in-degree and out-degree (degree property from now on).

If the graph has an Euler tour, then by following that tour, it is always able to leave any vertex that we visit. Thus, the in-degree of each vertex equals its out-degree.

Assume now that each vertex has the same in-degree and out-degree. we can construct an Euler tour using the following precedure. Start at any node and keep traversing edges until you reach the same vertex (the graph is connected and by the degree property we are always able to return at this node). If this cycle (not necessary simple) is a tour (contains all edges) we are done. Otherwise, delete the traversed edges from the graph and starting by any vertex of the cycle that has non-zero degree, find (by traversing edges) another cycle (observe that the degree property holds even after the edges’ removal). Remove the traversed edges and continue until no edges are left in the graph. All the cycles that where discovered can be conbined to give us an Euler tour: just keep track of the starting vertex of each cycle and insert this cycle into the previous cycle.

The last procedure corresponds to a well defined linear time algorithm for finding an Euler tour. However, we give another algorithm that, using a DFS traversal over the graph, constructs an Euler tour in a systematic way.

The algorithm is very similar to the one that traverses a connected undirected graph such that each edge is traversed exactly twice (once for each direction) (see problem 2 of homework 7a - Collaborative). there by marking the entrance edge we had been able to succesfully (that is, so that no edges were left untouched) leave a vertex and never visit it again.

This, however, can not be done here because an edge has only one direction. We, instead, “preprocess” the graph, by performing a DFS traversal in the graph.

however, traversing edges in the opposite direction (that is, we have a DFS traversal where each edge is trvaersed in its opposite direction (backwards)). We only label the discovering edges in this DFS. Thus, when the DFS is over, each vertex will have a unique outgoing labeled edge (the one connecting this vertex with its perent in the discovering tree of DFS). We then simulate the algorithm for homework 7a, problem 2: starting from any node, we perform a “blind” traversal, where we do not cross any label edge unless we are forced (there are no other way to leave the current vertex).

A-13.2 Hint: Consider how to use a traversal method described in this chapter.

Solution: Perform a DFS of G, and traverse an edge each time it is considered, where you traverse each discovery edge in its specified direction, each back edge in both directions, and then backwards along a discovery edge, when the recursive DFS returns back to the vertex that called it. This method runs in $O(n+m)$ time.

A-13.3 Hint: Be greedy.

Solution: Assume $V = \{v_1, \dots, v_n\}$. An algorithm would be:

```
Initialize $I$ to empty
for $i = 1$ to $n$
    if $\nexists(v_i, x) \in E$ with $x \in I$, then
        add $v_i$ to $I$.
```

This algorithm can be implemented in $O(n+m)$ time, by removing from G each vertex v_i added to I together with all of its adjacent vertices.

A-13.4 Hint: Perform a traversal of T to compute distances to leaves.

Solution:

a. No. Not always unique. It’s possible that the remaining tree has two nodes. Consider again, a tree that consists of a unique path having an even number of nodes. And, of course, we don’t like to remove the leaves of a two-node tree (there will be nothing left!).
b. First, observe that the center of a tree T does not change, if we remove all leaves of the tree (or, if we add some children to every leaf of the tree). Moreover, a leaf can not be the center of a tree of at least 3 nodes. The idea here is, thus, to keep pruning the leaves until one or two nodes are left:
   (a) Remove all leaves of T. Let the remaining tree be T_1.
   (b) Remove all leaves of T_1. Let the remaining tree be T_2.
   (c) Repeat the “remove” operation as follows: Remove all leaves of T_i. Let remaining tree be T_{i+1}.
   (d) Once the remaining tree has only one node or two nodes, stop. Suppose now the remaining tree is T_k.

(e) If T_k has only one node, that is the center of T. The *eccentricity* of the center node is k.

(f) If T_k has two nodes, either can be the center of T. The *eccentricity* of the center node is $k+1$.

This high level algorithm description gives us a linear time algorithm. (Worst case, when the tree is just a path.)

Now, we can either really perform removals in a copy of our tree, or simulate the removal by marking the leaves. Any "remove" operation can be performed by traversing the tree (starting from an arbitrary node) and deleting/marking the leaves. However, a "remove" operation may require $O(n)$ time, yielding a total quadratic time complexity. Another way to see that: consider a degenerate tree which is actually a path (or list). At each step only two leaves are removed, and these two leaves are found in time proportional to the current size of the path. This worst case example shows that the required time can be of the order of $\sum_{i=1}^{n} i = O(n^2)$.

A-13.5 Hint: Consider a process where we repeatedly remove the leaves of T, that is, the external nodes of T.

Solution: We can compute the *diameter* of the tree T as follows:

a. Remove all leaves of T. Let the remaining tree be T_1.
b. Remove all leaves of T_1. Let the remaining tree be T_2.
c. Repeat the "remove" operation as follows: Remove all leaves of T_i. Let remaining tree be T_{i+1}.
d. When the remaining tree has only one node or two nodes, stop! Suppose now the remaining tree is T_k.
e. If T_k has only one node, that is the center of T. The *diameter* of T is $2k$.
f. If T_k has two nodes, either can be the center of T. The *diameter* of T is $2k+1$.

A-13.6 Hint: Perform "baby" searches from each station.

Solution: For each vertex v, perform a modified BFS traversal starting a v that stops as soon as level 4 is computed.

Alternatively, for each vertex v, call method DFS$(v, 4)$ given below, which is a modified DFS traversal starting at v:

```
Algorithm DFS($v, i$):
  if $i > 0$ and $v$ is not marked then
      Mark $v$.
      Print $v$.
      for all vertices $w$ adjacent to $v$ do
          DFS($w, i-1$).
```

A-13.7 Hint: Consider how an algorithm for finding biconnected components could help here.

Solution: The central kissers in G correspond exactly to the separation vertices in G. Thus, we can identify all the central kissers in $O(n+m)$ time by running an efficient biconnected components algorithm on G and then using that to identify all the separation vertices in G.

Chapter 14 Shortest Paths

Hints and Solutions

Reinforcement

R-14.1 Hint: The fact that the edge weights are distinct means that no edge weight is unique.

R-14.2 Hint: Think about how we modified DFS and BFS for directed graphs in the previous chapter.

R-14.3 Hint: Think about how to update this tree each time we do a relaxation step.

R-14.4 Hint: Recall the way that the Bellman-Ford algorithm finds negative-weight cycles.

R-14.5 Hint: Think about how you can reuse space.

R-14.6 Hint: Think about what extra information to store in the table after doing a comparison.

R-14.7 Hint: Consider the size of Q each time the minimum element is extracted.

R-14.8

Creativity

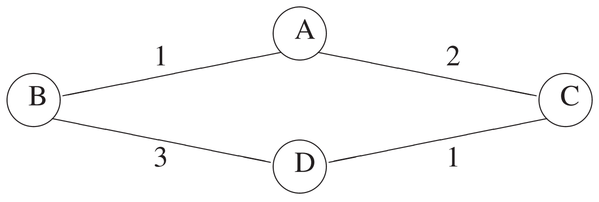
C-14.1 Hint: Give G lots of edges and make every edge count for lots updates in the heap.

C-14.2 Hint: Examine closely the justification for why Dijkstra's algorithm works correctly and note the place where it breaks if negative-weight edges.

C-14.3 Hint: Think about what extra work we need to do when we remove a vertex with smallest key from Q.

Solution: The main change to the description of Dijkstra's algorithm is that now when we remove a vertex, v, with smallest key, we need to check if v has already been removed earlier in the algorithm. To perform this check, we could add a Boolean flag to each vertex that is true if and only if that vertex has already been processed (and added to the cloud). So when we remove a vertex, we test if this flag is true. If it is not, then this is the first removal for v—so we do all the relaxation operations for v and set its flag to true. If the flag for v is true, then we just ignore this vertex and repeat the removeMin operation. The running time for Dijkstra's algorithm is still $O(m \log n)$ in this case, because $O(\log m)$ is $O(\log n)$, and each heap operation is done once for each edge, just as before.

C-14.4 Solution: The greedy algorithm presented in this problem is not guaranteed to find the shortest path between vertices in graph. This can be seen by counterexample. Consider the following weighted graph:



Suppose we wish to find the shortest path from $start = A$ to $goal = D$ and we make use of the proposed greedy strategy. Starting at A and with $path$ initialized to A, the algorithm will next place B in $path$ (because (A, B) is the minimum cost edge incident on A) and set $start$ equal to B. The lightest edge incident on $start = B$ which has not yet been traversed is (B, D). As a result, D will be appended to $path$ and $start$ will be assigned D. At this point the algorithm will terminate (since $start = goal = D$). In the end we have that $path = A, B, D$, which clearly is not the shortest path from A to D. (Note: in fact the algorithm is not guaranteed to find any path between two given nodes since it makes no provisions for backing up. In an appropriate situation, it will fail to halt.)

C-14.5

C-14.6 Solution:

a. If $M^2(i, j) = 1$, then there is a path of length 2 (a path traversing exactly 2 edges) from vertex i to vertex j in the graph G. Alternatively, if $M^2(i, j) = 0$, then there is no such path.

b. Similarly, if $M^4(i,j) = 1$, then there exists a path of length 4 from v_i to v_j, otherwise no such path exists. The situation with M^5 is analogous to that of M^4 and M^2. In general, M^p gives us all the vertex pairs of G which are connected by paths of length p.

c. if $M^2(i,j) = k$, then we can conclude that the shortest path connecting vertices i and j of length ≤ 2 has weight k. That is, if $k = \infty$, then i and j are not connected with a path of length at most 2, and, if $k \neq \infty$, then they are connected with a path of length at most 2 that has weight k and is a shortest path of length ≤ 2.

C-14.7 Hint: Think about how to exploit the fact that the distance from v to any other vertex in G can be at most $O(cn) = O(n)$.

Solution: In this case, we can implement the priority queue, Q, as lookup table, T, of size $O(cn) = O(n)$, where $T[i]$ is a set representing all the vertices with $D[v]$ label equal to i. Since distances from the start are monotonically increasing as we are iterating through Dijkstra's algorithm, we can always keep track of the nonempty cell, $T[i]$, in T with smallest index, i, in amortized $O(1)$ time. This allows us to perform removeMin operations in amortized constant time (by removing from the nonempty $T[i]$ cell with smallest i) and also to update the key value for any vertex in $O(1)$ time (just by moving the vertex from one cell in T to another). Thus, the total running time for Dijkstra's algorithm becomes $O(n+m)$. Another approach is to replace each edge, e, with $w(e)$ edges linked in a path, where $w(e)$ is the weight of e, and then run a BFS from the v.

C-14.8 Hint: Review the proof of correctness for the Bellman-Ford algorithm.

Applications

A-14.1 Hint: Consider turning G into a directed graph.

Solution: Orient each edge in G as a directed edge that goes from left to right. This results in G becoming a directed acyclic graph, $\vec{G}$. So that if we find a shortest path in $\vec{G}$, using the cost of each edge as its weight, then we will have a solution to the minimum-cost monotone path in G. The running time of this algorithm is $O(n+m)$ on a graph with n vertices and m edges.

A-14.2 Hint: Think about how you can assign weights to edges.

Solution: Since the only thing that matters is the number of compromised edges along a path, we can give a weight of 0 to each uncompromised edge and a weight of 1 to each compromised edge. Then a shortest path from s to t will be minimizing the number of compromised edges along this path. We can apply the usual Dijkstra's algorithm to find such a shortest path in $O((n+m)\log n)$ time, and this can even be improved to $O(n+m)$ time, by exploiting an idea from an earlier creative exercise.

A-14.3 Hint: Think about how to modify Dijkstra's algorithm to work in this context.

Solution: The main idea is to modify Dijkstra's algorithm to work in this context. We initialize as in the standard algorithm, so $D[s] = 0$ for our start vertex, s, and $D[v] = +\infty$, for each vertex $v \neq s$. Given the time, t_0, to start, we can do our initial relaxations for each edge, (s, j), incident on the start vertex, s, by setting

$$D[j] = f_{s,j}(t_0).$$

Then, we continue as in the normal Dijkstra's algorithm. Note that each $D[v]$ value is the amount of time that it takes us to get from s to v, based on the paths discovered so far in the algorithm. Thus, we have to modify the relaxation operation for an edge (v, w) so that it becomes a comparison of $D[w]$ and

$$D[v] + f_{v,w}(t_0 + D[v]).$$

Then the rest of the algorithm is as in the normal Dijkstra's algorithm. Therefore, the running time for a graph with n vertices and m edges is $O(m \log n)$, assuming the priority queue in Dijkstra's algorithm is implemented with a heap.

A-14.4 Solution:

We describe two solutions for the flight scheduling problem.

a. We reduce the flight scheduling problem to the shortest paths problem. That means that we will construct an instance for the shortest paths problem, that when solved, gives us a solution for the flight scheduling problem. Given the set of airports $\mathcal{A}$ and the set of flights $\mathcal{F}$, we consider a weighted directed graph $\overrightarrow{G}$ that is constructed as follows.

   - For each airport $a_i \in \mathcal{A}$, draw a circle $circle_i$ to represent the time (24 hours).
   - For each flight $f_i \in \mathcal{F}$, find the origin airport a_o, the destination airport a_d, the departure time t_d, and the arrival time t_a. Draw a vertex v_1 on $circle_o$ marked the time t_d and also draw a vertex v_2 on $circle_d$ marked the time (t_a+c(a_d)). Draw an directed edge from v_1 to v_2 with weight t_a+c(a_d)-t_d (of course we must compute the correct weight (flight time) in case like the departure time is 10:00PM and the arrival time is 1:00AM next day). The direction of edges on a circle should be clockwise.

   Note that we have included the minimum connecting time in the total flight time, so we can only consider the modified new flights without worrying about the connecting times.

   Given graph $\overrightarrow{G}$, in order to solve the flight scheduling problem, we can just find a shortest path from the first vertex on $circle(a)$ (origin airport a) representing time t or after t to one vertex on $circle(b)$ (destination airport

b) in the graph. The flights' sequence can be obtained from the shortest path from origin to destination.

We can use Dijkstra's algorithm to find the desired shortest path (observe that $\overrightarrow{G}$ has only positive weights), but we need to slightly modify the algorithm (as it is presented in the textbook), so that it is applicable to directed graphs. The modifications should be straightforward: in the relaxation step, we only consider edges with orientation from the previously selected vertex to some other vertex. Finally, we need keep track (mark) the shortest path, so we include a parent-child relationship to the visited nodes (value p).

The time complexity of the algorithm essentially the time complexity of Dijkstra's algorithm (since the construction of the graph can be done in linear ($O(n+m)$) time). Using a heap as the priority queue, we achieve a total $O((N+M)\log N)$ time complexity, where N the number of vertices of $\overrightarrow{G}$ and M the number of edges of $\overrightarrow{G}$. Note that, generally, $M = O(m)$ and $N = O(m)$, so the running time of the algorithm is $O(m \log m)$.

b. The following algorithm finds the minimum travel time path from airport $a \in \mathcal{A}$ to airport $b \in \mathcal{A}$. Recall, we must depart from a at or after time t. Note, "$\oplus$" and "$\preceq$" are operations which we must implement. We define these operations as follows: If $x \oplus y = z$, then z is the point in time (with date taken into consideration) which follows x by y time units. Also, if $a \preceq b$, then a is a point in time which preceeds or equals b. These operations can be implemented in constant time.

The algorithm essentially performs Dijkstra's Shortest Path Algorithm (the cities are the vertices and the flights are the edges). The only small alterations are that we restrict edge traversals (an edge can only be traversed at a certain time), we stop at a certain goal, and we output the path (a sequence of flights) to this goal. The only change of possible consequence to the time complexity is the flight sequence computation (the last portion of the algorithm). A minimum time path will certainly never contain more that m flights (since there are only m total flights). Thus, we may discover the path (tracing from b back to a) in $O(m)$ time. Reversal of this path will require $O(m)$ time as well. Thus, since the time complexity of Dijkstra's algorithm is $O(m \log n)$, the time complexity of our algorithm is $O(m \log n)$.

Note: Prior to execution of this algorithm, we may (if not given to us) divide $\mathcal{F}$ into subsets $F_1, F_2, \ldots, F_N$, where F_i contains the flights which depart from airport i. Then when we say "for each flight $f \in \mathcal{F}$ such that $a_1(f) = x$", we will not have to waste time looking through flights which do not depart from x (we will actually compute "for each flight $f \in F_x$"). This division of $\mathcal{F}$ will require $O(m)$ time (since there are m total flights), and thus will not slow down the computation of the minimum time path (which requires $O(m \log n)$ time).

A-14.5 Hint: Model this problem as an optimal path problem that goes between two vertices in a directed weighted graph without cycles.

Solution: Model the maze as a directed acyclic graph. Make the weight of each edge, e, be $-g$ if e has g units of gold. Then run the shortest-path algorithm for a dag. Since all paths are negative, this algorithm will find the "most negative" path from the start to finish. In this case the most negative path is the one with the most gold.

A-14.6 Hint: Think about how to build a dynamic programming algorithm along the lines of the Floyd-Warshall algorithm.

Solution: Let us build a dynamic programming algorithm along the lines of the Floyd-Warshall algorithm, but rather than construct a label for pairwise distances between vertices, we will construct labels for best exchange rates. In particular, number the vertices in $\vec{G}$ from 1 to n, and let $r_k(v,w)$, denote the best exchange rate for converting from currency v to currency w, using currencies in the set, $\{1,2,\ldots,k\}$, for intermediate trades. Initially, $r_0(v,v) = 1$, for every v in $\vec{G}$, and $r_0(v,w) = r(v,w)$, for each edge in $\vec{G}$. In iteration $k \geq 1$, we perform the following assignment, for each pair, (v,w),

$$r_k(v,w) \leftarrow \max\left\{r_{k-1}(v,w),\ r_{k-1}(v,k)\cdot r_{k-1}(k,w)\right\},$$

including for the case when $w = v$. When we are done, we check if there is a vertex, v, such that $r_n(v,v) > 1$. If so, then there is an arbitrage opportunity for currency v. In order to identify the actual cycle, we need to just backtrack through the assingments that led to this value, assuming we kept around the set of $r_k(v,w)$ values. The running time for this algorithm is $O(n^3)$, assuming that multiplication is a constant-time operation.

Chapter 15 Minimum Spanning Trees

Hints and Solutions

Reinforcement

R-15.1 Hint: Review the MST algorithm in question.

R-15.2 Hint: Review the MST algorithm in question.

R-15.3 Hint: Review the MST algorithm in question.

R-15.4 Hint: Review the MST algorithm in question.

R-15.5 Hint: Review the MST algorithm in question.

R-15.6 Hint: Review the MST algorithm in question.

R-15.7 Hint: Review the meaning of the union and find operations from the chapter that describes these methods.

R-15.8 Hint: Consider the proofs of correctness. Do they change if the graph has negative-weight edges?

R-15.9 Hint: One of the graphs given in this chapter is an example.

R-15.10 Hint: Consider an MST that has two edges that go from V_1 to V_2.

Creativity

C-15.1 Hint: Be sure to consider the case where there might be another edge with the same weight as e.

C-15.2 Hint: Consider the case when G is very sparse.

Solution: This claim is false. For instance, if G is already a tree, then even the largest-weight edge must belong to its minimum spanning tree.

C-15.3 Hint: Use a crucial fact about minimum spanning trees.

C-15.4 Hint: Think about how this change would effect the running of Kruskal's algorithm.

C-15.5 Hint: Think about how this change would effect the running of Kruskal's algorithm.

C-15.6 Hint: Think about an alternative way of implementing the priority queue, Q, that would support fast insertions and updates at the expense of the removeMind operation.

Solution: The main modification in this case is to implement the priority queue, Q, as an unordered doubly linked list. This allows us to insert and update the key for any vertex in $O(1)$ time and remove the minimum in $O(n)$ time. We perform $O(m)$ insertions and updates and $O(n)$ removeMin operations in the algorithm, and these computations are the bottlenecks. Thus, since m is $O(n^2)$, the total running time is $O(n^2)$ in this case in the worst case.

C-15.7 Hint: Consider contracting the representation of the graph so that each connected component "cluster" is reduced to a single "super" vertex, with self-loops and parallel edges removed with each iteration. Then characterize the running time using a recurrence relation involving only n and show that this running time is $O(n^2)$.

C-15.8 Hint: Think about first bounding the expected number of times the Prim-Jarník algorithm would change the value of $D[v]$ for any given vertex, v, in G.

Solution: For any vertex, v, in G, the expected number of times that we would change the value of $D[v]$ during the Prim-Jarník algorithm is bounded by the number of times we would update a running minimum in a loop that scans a random sequence of $\deg(v)$ numbers. This number is $O(\log \deg(v))$, which is $O(\log n)$. Thus, by the linearity of expectation, the number of times we need to do updates on Q for any vertex is $O(\log n)$, each of which can be done in $O(\log n)$ time. Thus, the expected running time of the algorithm is $O(n \log^2 n + m)$.

C-15.9 Hint: Think about how Kruskal's algorithm would operate on G and G'.

C-15.10 Hint: Try to apply one of the crucial facts about minimum spanning trees.

Solution: First, observe that Peter's reverse-greedy algorithm results in a tree, since G starts out connected and we consider every edge in a sequential fashion and remove it if this action doesn't disconnect G. Also, observe that the MST is unique, since the edge weights are distinct. Consider each edge, e, that is included in Peter's tree. When we considered e, removing it would disconnect the graph, into two components V_1 and V_2. Moreover, since we considered the edges in reverse order, and have removed edges that can be removed without disconnecting V_1 and V_2, e must be the minimum-weight edge that connects V_1 and V_2. Thus, by a crucial fact about MSTs, e belongs to the MST.

C-15.11 Hint: Try to come up with an example input and set of division steps that result in an incorrect output.

Solution: Stanislaw's algorithm is not correct. Consider an example graph that is a square, $abcd$, with weights $(1, 2, 3, 4)$ in order around its boundary. Suppose the first divide is bc and da, with the edges of weight 1 and 3 joining these sets, and so that the edges bc and da have weights 2 and 4, respectively. Then Stanislaw's algorithm outputs the edge of weight 1, then recursively outputs the edges with weights 2 and 4. But the MST consists of the edges of weights 1, 2, and 3.

Applications

A-15.1 Hint: Consider a transformation of the edge weights to turn this into a minimum spanning tree problem.

Solution: We can perform a simple transformation of the edge weights to turn this into a minimum spanning tree problem. Namely, replace each edge weight, $w(e)$, with $-w(e)$. Now a minimum spanning tree for the transformed graph will be a maximum spanning tree for the original graph.

A-15.2 Hint: Think about how this problem could be solved with a maximum spanning tree.

Solution:

We can model this problem using a graph. We associate a vertex of the graph with each switching center and an edge of the graph with each line between two switching centers. We assign the weight of each edge to be its bandwidth. Vertices that represent switching centers that are not connected by a line do not have an edge between them.

We use the same basic idea as in Dijkstra's algorithm. We keep a variable $d[v]$ associated with each vertex v that is the bandwidth on any path from a to this vertex. We initialize the d values of all vertices to 0, except for the value of the source (the vertex corresponding to a) that is initialized to infinity. We also keep a π value associated with each vertex (that contains the predecessor vertex).

Assume that we have an edge (u, v). If $\min\{d[u], w(u, v)\} > d[v]$ then we should update $d[v]$ to $\min\{d[u], w(u, v)\}$ (because the path from a to u and then to v has bandwidth $\min\{d[u], w(u, v)\}$, which is more than the one we have currently).

The total running time is $O((n + m) \log n)$.

A-15.3 Hint: Consider defining a weighted graph $G = (V, E)$, where V is the set of stations and E is the set of possible channels between the stations. Define the weight $w(e)$ of an edge $e \in E$ as the energy needed for the corresponding channel.

Solution: Consider the weighted graph $G = (V, E)$, where V is the set of stations and E is the set of channels between the stations. Define the weight $w(e)$ of an edge $e \in E$ as the energy needed the corresponding channel. Then a minimum spanning tree in G will be a minimum-energy method for connecting the various stations. We can construct such a minimum spanning tree for G in $O(n^2)$ time by using the Prim-Jarník algorithm, where we implement the queue, Q, using an unordered list, so that insertions and updates take $O(1)$ time and removeMin operations take $O(n)$ time.

A-15.4 Hint: Think about how the new edge effects the minimum spanning tree, T.

A-15.5 Hint: Think about how you can satisfy the conditions that would allow you to get a faster running time for one of the MST algorithms given in the chapter.

A-15.6 Hint: Consider using the Prim-Jarník algorithm, but with a simpler implementation of the priority queue, Q.

Solution: Use the Prim-Jarník algorithm, but with a simpler implementation of the priority queue, Q. Namely, we can implement the priority queue as two doubly linked lists, one for the items with cost 10 and the other for items with cost 30. Insertions and updates can therefore be done in $O(1)$ time, as can the removal of an item with smallest key. Therefore, the running time of the algorithm is $O(n + m)$.

A-15.7 Hint: Consider how a minimum spanning tree could be used.

Solution: Construct a graph, G, having the n points as its vertices and the distances between pairs of them as weights on its edges. Find an MST, T, for G and remove the largest-weight edge in T. Let the vertices in the two remaining components of T be the two sets, A and B. Note that by the crucial fact of MSTs, every point in A or B is closer to another point in its set than it is to any point in the other set. The running time for this algorithm is $O(n^2)$ if we use the Prim-Jarník algorithm implemented with a priority queue implemented with an unordered list, so as to support constant-time updates and $O(n)$-time removals of the minimum item.

Chapter 16 Network Flow and Matching

Hints and Solutions

Reinforcement

R-16.1 Solution: Perform a depth-first search from s, limiting the search to edges that have some residual flow left, much like we are looking for an augmenting path. Define the cut $\chi = (V_s, V_t)$ to be defined so that V_s is the set of all vertices reached during this DFS. By the Max-Flow, Min-Cut Theorem, this cut is a minimum cut.

R-16.2 Solution: The forward edges in π are (s, v_2), (v_2, v_3), (v_1, v_4), and (v_4, t). The backward edge is (v_1, v_3). We can form augmenting paths different from π by replacing (s, v_2, v_3) with (s, v_3), by replacing (v_3, v_1, v_4) with (v_3, v_4), or by replacing (v_3, v_1, v_4, t) with (v_3, t), but these last two substitutions conflict with each other. Thus, there are $2 \cdot 3 = 6$ possible augmenting paths. The maximum flow starts with the flow of the (b)-figure and adds one unit of flow along (s, v_3, t) and one additional unit of flow along (s, v_2, v_3, v_4, t). At this point, there are no augmenting paths left. Therefore, the maximum flow has value 14.

R-16.3

R-16.4

R-16.5

R-16.6

R-16.7

R-16.8

R-16.9 Solution: Let $X = \{x_1, x_2, \ldots, x_n\}$ and $Y = \{y_1, y_2, \ldots, y_n\}$. We can count all possible matchings inductively. There are n ways to match x_1 with vertices in Y. Given this match, there are still $n - 1$ ways to match x_2 with vertices in Y. Given the previous matches, there are still $n - 2$ ways to match x_3

with vertices in Y, and so on. Ultimately, given previous matches, there will be $n - (n-1) = 1$ way to match x_n. Thus, there are $n!$ possible matches.

R-16.10

R-16.11

R-16.12

R-16.13

R-16.14

R-16.15

R-16.16 Solution: Yes, its running time depends on both the size of the input (specified with variables n and m) and a numeric value derived from values in the input (the flow value f).

Creativity

C-16.1 Solution: If every edge capacity is bounded by a constant, then the maximum flow must be bounded by $O(m)$. Thus, the Ford-Fulkerson algorithm would run in worst-case time $O(m^2)$.

C-16.2 Hint: Use $d_f(u,t)$ in addition to $d_f(s,v)$.

C-16.3 Solution: The residual capacity of an augmenting path is the minimum capacity of one of its edges. Thus, we are interested in finding a maximum capacity path from s to t. We can perform this computation by using a maximum spanning tree algorithm, which is just like a minimum spanning tree algorithm with all the weights multiplied by -1. The path from s to t in this tree will be a maximum capacity path (using a similar argument used to prove the important fact about minimum spanning trees).

C-16.4

C-16.5

C-16.6

C-16.7 Solution: Start from some vertex v and perform a breadth-first search. If there are any non-tree edges joining vertices on the same level, then the graph is not bipartite (for we would have just found an odd-length cycle). If, after performing this test, there is an unvisited vertex, w, we repeat this algorithm with w. We continue this process until we have determined that the graph is not bipartite or we have visited all its vertices. If we have visited all its vertices and found no odd-length cycles, then the graph is bipartite. The running time is $O(n + m)$, since we are performing a BFS on each connected component.

C-16.8

C-16.9

C-16.10 Hint: Think of using the outcome of each maximum flow problem to determine which ones to do next.

Solution: Order all the teams by their $w_i + g_i$ values, and then do a binary search on this ordered list to find the breakpoint between teams that are eliminated and those that are not.

C-16.11 Hint: Use the max-flow, min-cut theorem, and the reduction of maximum bipartite matching to maximum flow to prove that, for any bipartite graph, G, the number of vertices in a minimum vertex cover for G equals the number of edges in a maximum matching in G.

Applications

A-16.1 Hint: Try to set this up as a problem that was studied in the chapter.

Solution: This is an example of a bipartite maximum matching problem, with vertices being residents and puppies, and edges being puppy preferences. Given this graph, a maximum matching algorithm will find the largest number of puppy adoptions.

A-16.2 Hint: Try to set this up as a network flow problem, similar to how network flow was used to solve the maximum matching problem.

Solution: Create a bipartite graph, with residents and puppies as its vertices and include an edge (i, j) if resident i would like to adopt puppy j. Give each such edge a capacity of 1. Next create a super node, s, and connect it with an edge to each resident, giving each such edge a capacity of 3. Finally, create a super node, t, and connect every puppy to it with an edge, giving each such edge a capacity of 1. Now perform a maximum flow algorithm on this network. The edges used for flow between residents and puppies will indicate which puppies each resident adopts, and the constraints on the edges from s guarantee that no resident can adopt more than three puppies.

A-16.3 Hint: Try to set this up as a network flow problem, similar to how network flow was used to solve the maximum matching problem.

Solution: Create a directed graph, with residents and puppies as its vertices. Give each such edge a capacity of 1. Next create a node, $v_{i,b}$, for each resident, i, and each breed, b, and connect $v_{i,b}$ to dog j if resident i likes dog j and this dog is of breed b. Give this edge capacity 1. Then, connect i to each $v_{i,b}$, and give this edge capacity 1. Next create a super node, s, and connect it with an edge to each resident, i, giving each such edge a capacity of 3. Finally, create a super node, t, and connect every puppy to it with an edge, giving each such edge a capacity of 1. Now perform a maximum flow algorithm on this network. The edges used for flow between residents and puppies will indicate which puppies each resident adopts, and the constraints on the edges from s guarantee that no resident can adopt more than three puppies and no more than one from any breed.

A-16.4 Hint: Think about how to reduce this problem to a traditional network flow problem.

Solution: The solution is to replace each vertex, v, with two vertices, v' and v'', and create a single edge from v' to v'' with capacity equal to the bandwidth constraint for v. Then, only have v's in-edges directed into v' and v's out-edges directed out from v''. Now a traditional maximum flow in this modified network will be a maximum flow in the original network with vertex constraints.

A-16.5 Hint: Try to set this up as a maximum flow problem.

A-16.6 Hint: Think about how to use the addition of new "super" nodes to solve this problem using network flow.

Solution: Create a source, x, and connect it to each populated region, r, giving the edge to region r a capacity of N_r. Similarly, create a sink, y, and connect each stronghold, s, to y, and give this edge a capacity of N_s. Next, connect each region to all the strongholds it can reach in three days' time and give each of these edges a capacity of the total population of Middle Earth. Now a maximum flow in this network will describe how to assign the maximum number of noble creatures to strongholds that are reachable in three days, while satisfying the population constraints of each stronghold.

A-16.7 Hint: Try to formulate this as a min-cost flow problem.

Solution: We are interested in a minimum-cost matching of limos to locations. We can produce such a solution by performing a minimum-cost flow algorithm on the flow network derived from the maximum matching problem, where we add an extra node s, connected to all the limos, and an extra node t, connected to all the locations.

Chapter 17 *NP*-Completeness

Hints and Solutions

Reinforcement

R-17.1 Solution: Professor Amongus reduced a problem in P to a problem in NP. To show $P = NP$, he would have to reduce a problem in NP to a problem in P. We already know that every problem in P is already in NP.

R-17.2

R-17.3 Hint: Use the restriction form of proof. Also, don't forget to show that SAT is in ***NP***.

Solution: SAT contains CNF-SAT and 3SAT as special cases; hence, by restriction, SAT is NP-hard. To see that the general form of SAT is in NP, note that we can guess an assignment of 0's and 1's to the variables in the formula and test in polynomial time if this assignment satisfies the formula.

R-17.4 Hint: Consider each clause of S in turn.

R-17.5 Hint: Consider using De Morgan's laws. Also, don't forget to show that DNF-DISSAT is in ***NP***.

R-17.6

R-17.7 Solution: To show that CLIQUE is in NP, we can guess a set of vertices of size k and then check that every pair of vertices in this set is adjacent.

R-17.8

R-17.9

R-17.10

R-17.11

R-17.12 Solution: This is not a polynomial-time algorithm, since k is a number in the input, not the input size. Moreover, k is not a constant, as would be required of a general polynomial-time algorithm for the CLIQUE problem.

R-17.13

R-17.14

R-17.15

R-17.16

R-17.17

Creativity

C-17.1 Hint: Consider how the multiplication of two numbers with $b/2$ bits each can increase the number of bits in the output.

C-17.2 Hint: Think about how much any operation can increase the complexity of representing numbers in a c-incremental algorithm.

Solution:

Proof: Note that $N \leq n$. Thus, $t(N) \leq t(n)$. Likewise, each primitive operation in the algorithm A, involving one or two objects represented with $b \geq 1$ bits, can be performed using at most db^2 bitwise operations, for some constant $d \geq 1$, since c is a constant. Such primitive operations include all comparison, control flow, and basic non-multiplicative arithmetic operations. Moreover, in N steps of a c-incremental algorithm, the largest any object's representation can become is $cN+b$, where b is the maximum size of any input object. But, $cN+b \leq (c+1)n$. Thus, every step in A will take at most $O(n^2)$ bit steps to complete. ■

C-17.3 Solution: Suppose that algorithm A makes at most $c \log n$ calls to the choose method. Create a deterministic version A_i of A for every possible outcome of these calls, and run A_i to see if it accepts the given input. If one of the A_i's accepts, then we accept the input. Otherwise, we reject it. If A runs nondeterministically in $p(n)$ steps, then this deterministic version of A will run in $p(n)2^{c \log n} = p(n)n^c$ steps, which is a polynomial.

C-17.4

C-17.5 Hint: Use recursion to repeat subexpressions in a way that doubles their size each time they are used.

C-17.6

C-17.7

C-17.8 Hint: Consider dealing with each variable one at a time.

Solution: Number the variables in a satisfiable formula B as $x_1, x_2, \ldots, x_n$. Then, for $i = 1$ to n, consider a version of B that is the formula derived from setting $x_i = 1$. If this formula is satisfiable, update B to now be this formula. Otherwise, set B to the formula derived from setting $x_i = 0$, and repeat. When we will have iteratively transformed B for each variable, then the remaining formula will just be the constant 1, and the variables assignments will be a satisfying assignment for the original formula B. Note that this algorithm makes $n+1$ calls to the oracle.

C-17.9 Hint: Reduce HAMILTONIAN-CYCLE or CLIQUE to this problem.

Solution: Suppose the number of vertices in H is r. To show this problem is in NP, we guess a subset of r vertices of G and we also guess a mapping (that is, a matching) of the vertices in H to the chosen vertices in G. Then, in polynomial time we can check that every edge of H has a corresponding edge in G.

To see that SUBGRAPH-ISOMORPHISM is ***NP***-hard, we will reduce the Hamiltonian cycle problem to this one. In particular, given an instance $G = (V, E)$ of the Hamiltonian cycle problem, we define a subgraph H to be a simple cycle of $|V|$ vertices. This graph H is a subgraph of G if and only if G is Hamiltonian. This reduction clearly takes polynomial time.

C-17.10 Hint: Reduce VERTEX-COVER to this problem.

Solution: To see that INDEPENDENT-SET is in NP, note that we can guess a subset of k vertices in G and confirm that no pair of vertices in this set are adjacent.

To show INDEPENDENT-SET is NP-hard, we will reduce VERTEX-COVER to this problem. Specifically, given an instance G, k to VERTEX-COVER, we note that G has a vertex cover of size k if and only if it has an independent set of size $n - k$, where n is the number of vertices in G.

C-17.11

C-17.12 Hint: Reduce SUBSET-SUM to this problem.

Solution: To show that PARTITION is in NP, note that we can guess a partition of the set of numbers and then check that they sum to the same value.

To show that PARTITION is ***NP***-hard, we will reduce SUBSET-SUM to this problem. So, suppose we are given a set of numbers S and a goal sum M that comprise an input to the SUBSET-SUM problem. We begin our transformation of this input into an instance of PARTITION by including every number in S as input numbers for PARTITION. Let N denote the total sum of all the numbers in S. Then, in addition to the numbers already included, we add the number $|N - 2M|$, which is a kind of ***enforcer***. If there is a subset T of S that sums to M, then T will make a partition such that T is on one side and $S - T$ (the rest of the elements) is on the other, with the enforcer element $|N - 2M|$ either on T's side or the other side so as to make the two sides sum to the same value.

C-17.13 Hint: Reduce the HAMILTONIAN-CYCLE problem for undirected graphs to this problem.

C-17.14

C-17.15

C-17.16

C-17.17 Hint: Consider a reduction from HAMILTONIAN-CIRCUIT.

Solution: A reduction from HAMILTONIAN-CIRCUIT could be as follows. Suppose we are given a graph G and asked whether it has a Hamiltonian circuit. Pick some vertex v in G, and create a graph, G', that is the same as G except that we replace v with two copies, v' and v'', such that v' and v'' have the same adjacencies as v. G will have a Hamiltonian circuit if and only if G' has a Hamiltonian path from v' to v''.

Applications

A-17.1 Hint: Model this problem using a graph.

Solution: Model the problem using a graph, where companies are vertices and there is an edge between two companies if they are not competitors. Then it is easy to see that the decision version of this problem is an instance of the CLIQUE problem, which was shown to be ***NP***-complete in the chapter.

A-17.2 Hint: Model this problem so that every trophy is represented as a pair, (w, s), where w is its width and s is its prestige score.

A-17.3 Hint: Think about how you could use dynamic programming to solve this problem.

A-17.4 Hint: Model this problem using a graph.

Solution: If we model this problem using a graph, so that each beer is a vertex and two beers are connected if they were drunk at the same party, then this problem is an instance of the HAMILTONIAN-CIRCUIT problem, which was shown to be ***NP***-complete in the chapter.

A-17.5 Hint: Model this as a problem involving sets.

Solution: This is an instance of the SET-COVER problem. In this case, the universe of objects is the collection of infected computers at CableClock. Each set, S_w, is defined to contain all the infected computers that visited a particular website, w. The optimization problem is to find the smallest number of websites visited by all the infected computers, which is the same as finding the smallest number of sets that cover the universe of infected computers.

A-17.6 Hint: Model this problem as a graph problem.

Solution: Model this as a graph, G, were the movie-goers are vertices and two movie-goers are connected by an edge if they have no previous movie in common. Then this problem of selecting a large set of compatible movie-goers is an instance of the CLIQUE problem.

A-17.7 Hint: Consider a reduction from SUBSET-SUM.

Solution: This problem can be shown to be ***NP***-complete by a reduction from SUBSET-SUM. Suppose you are given an instance of SUBSET-SUM, which is a set of n values, $x_1, \ldots, x_n$, and an integer k. Create a book for each value in the set and give it a monetary value equal to its numerical value. Then create two more books, b_1 and b_2, with b_1 having value $k + 2$ and b_2 having value $(\sum x_i) - k + 2$. Note that in any successful partition, b_1 and b_2 have to go to separate people, since they sum to $(\sum x_i) + 4$ and the total sum is $2(\sum x_i) + 4$. So, suppose you get b_2 in this case. The total value of all the books is $2(\sum x_i)+4$, so your books are worth $(\sum x_i) + 2$. But, since you have b_2, this means that the value of the rest of your books is k. Thus, there is a solution to SUBSET-SUM in this case if and only if there is a good book partition.

Chapter 18 Approximation Algorithms

Hints and Solutions

Reinforcement

R-18.1

R-18.2

R-18.3

R-18.4

R-18.5

R-18.6

R-18.7

R-18.8

R-18.9 Hint: Consider the cases for three vertices.

R-18.10 Hint: Review the definition of the triangle inequality.

R-18.11 Hint: Draw a Venn diagram for the sets.

Solution: The optimal solution is $\{S_3, S_4, S_5\}$, the greedy algorithm will produce the solution $\{S_1, S_4, S_5, S_3\}$, in this order.

R-18.12 Hint: Recall that the sum of all the degrees is even.

Creativity

C-18.1 Hint: Reduce HAMILTONIAN-CYCLE to this problem by defining a cost function for a complete graph H for the n-vertex input graph G so that edges of H also in G have cost 1 but edges of H not in G have cost δn more than 1.

Solution: Use the hint. That is, suppose we are given an instance G to the Hamiltonian cycle problem. Construct a complete graph H on the n vertices of G and weight these edges so that an edge of H also in G has cost 1 but an edge of H not in G has cost $1 + \delta n$. Note that if we include even just one of these edges in a cycle, then the cost of that cycle is at least $n + \delta n = (1 + \delta)n$. Thus, if we ask for a strict $(1 + \delta)$-approximation of a TSP of cost n, this can only be achieved using original edges of G that form a Hamiltonian cycle.

C-18.2

C-18.3

C-18.4

C-18.5

C-18.6 Hint: Show that it is possible to turn an Euler-tour traversal, E, of an MST for G into a tour visiting each vertex exactly once such that each edge of the tour skips at most two vertices of E.

C-18.7 Hint: Show that if there is a crossing pair of line segments, then you can uncross them and get a shorter tour.

C-18.8 Hint: Think about the set of items that are omitted from consideration in the greedy approach.

C-18.9 Hint: Use a bipartite graph where all the vertices on one side have the same degree.

C-18.10 Hint: Follow the general approach as the approximation algorithm for the set cover problem.

Applications

A-18.1 Hint: Start with a city of smallest degree.

A-18.2 Hint: Consider a greedy strategy of always taking the shortest route.

Solution: Consider a greedy strategy of always taking the shortest route. Let I be some interval between two adjacent routers. Since the greedy strategy always takes the shortest route, there is another interval, J, on the opposite side of the ring such that I and J have no transmissions in common in the greedy solution. Note that the optimal solution must route all of these messages as well, and each such transmission must go through I or J. Let g_i and g_j denote the number of the respective transmissions for the intervals I and J in the greedy solution and o_i and o_j denote the number of the respective transmissions for I and J in the optimal solution. Then

$$g_i + g_j \leq o_i + o_j.$$

This is not necessarily an equality, since there may be some transmissions through I or J in the optimal solution that do not span either I or J in the greedy solution. Suppose, without loss of generality, that I is the interval that determines the max load, M, for the greedy solution. We know that the max load, L, for the optimal solution is at least $\max\{o_i, o_j\}$. Thus, we have

$$M = g_i \leq o_i + o_j \leq 2L.$$

Therefore, this greedy solution is a 2-approximation.

A-18.3 Hint: Consider the greedy the strategy of always assigning the next box to the least loaded truck.

A-18.4 Hint: Fill up the trucks one at a time. For the analysis, consider boxes that weigh more than $M/2$ pounds and at most $M/2$ pounds separately

Solution: Consider a greedy algorithm that fills up the trucks one at a time. To see that this is a 2-approximation algorithm, consider boxes to be "big" if they weigh more than $M/2$ pounds and "small" otherwise. Note that every big box will go in a different truck, in both the greedy solution and the optimal solution. Let G_B be the number of trucks in the greedy solution containing big boxes and let H_B be the number of trucks in the optimal solution containing big boxes. Then $G_B = H_B$. Thus, let us consider the number of trucks containing only small boxes. In the greedy solution, any truck filled with only small boxes must be filled to have weight at least $M/2$. Otherwise, we could have put one more small box in that truck. Let G_S be the number of trucks in the greedy solution containing only small boxes and let H_S be the number of trucks in the optimal solution containing only small boxes. Note that the optimal solution might fit some small boxes into trucks that also contain a big box, whereas the greedy solution might not. Still, the most that the optimal solution can fill a truck with small boxes is to weight M and the most it can pack into a truck already containing a big box is $M/2$. Thus,

$$G_S \leq 2H_S + H_B.$$

In other words, the number of trucks used in the greedy solution, $G_S + G_B$, satisfies the following:

$$G_S + G_B \leq 2H_S + 2H_B = 2(H_S + H_B).$$

Therefore, this greedy solution is a 2-approximation algorithm.

A-18.5 Hint: Choose a set of cities that forces this greedy algorithm to make poor choices by sending it far away from a place it ultimately needs to go.

A-18.6 Hint: Notice the similarity of this algorithm with the Prim-Jarnik algorithm for constructing a minimum spanning tree.

Chapter 19 Randomized Algorithms

Hints and Solutions

Reinforcement

R-19.1 Hint: Review the analysis of the algorithm.

Solution: Change the choice of K to be the smallest power of 2 greater than or equal to n^6.

R-19.2 Hint: Use a Chernoff bound.

Solution: Let X denote the number of wins for the Bears, so $E[X] = n/3$. So, $\Pr(X > n/2) \leq (e^{1/2}/1.5^1.5)^n$, which is less than 0.9^n.

R-19.3 Hint: Review the equation for the Chernoff bound.

R-19.4 Hint: Use a Chernoff bound for a sum of geometric random variables.

R-19.5 Hint: Think about using independent repetitions.

R-19.6 Hint: Think about selecting a random sample of A.

R-19.7 Hint: Use the bound $\ln n \leq H_n \leq \ln n + 1$.

R-19.8 Solution:

$$\binom{n}{2}$$

R-19.9 Hint: Use the Master theorem after deriving the recurrence.

R-19.10 Hint: Consider the probability of finding a given minimum cut by a series of random contractions.

R-19.11 Hint: Combine the performance bounds of the Rabin-Miller algorithm with the bounds for finding prime numbers.

R-19.12 Hint: Use a Chernoff bound.

R-19.13 Hint: Review the insertion method for skip lists.

R-19.14 Hint: Review the removal method for skip lists.

Creativity

C-19.1 Hint: Think of the case when k is not a power of 2.

Solution: Suppose we take $k = 10$. Then before he performs his modular reduction to an integer in the range $[0, 9]$, Bob gets an integer in the range $[0, 15]$, each with equal probability, $1/16$. Thus, after performing the modular reduction, he will get the integers, $0, 1, 2, 3, 4, 5$, each with probability $1/8$ and the integers, $6, 7, 8, 9$, each with probability $1/16$.

C-19.2

C-19.3 Hint: Use an efficient data structure for testing whether a newly generated random value, r_i, is distinct from the previously generated values, r_j ($j < i$).

C-19.4 Hint: Consider the case when $n = 3$.

Solution: Suppose $n = 3$. We start with 123. After 1 step, we get 123, 132, and 321, each with probability $1/3$. In step 2, from 123, we get 123, 213, and 132, each with probability $1/9$, and so on. In the end we get each permutation of 123 occurring with some probability of the form $i/27$, where i is the number of times we can get that permutation following the depth-3 tree of possibilities, which has degree-3 and 27 external nodes. But we need each permutation to occur with probability $1/3! = 1/6$, and there is no way to make a fraction of the form $i/27$ equal to $1/6$ with i being an integer.

C-19.5 Hint: Think about using a random permutation algorithm.

C-19.6 Hint: Use Chernoff bounds.

C-19.7 Hint: Break the problem into two coupon collector problems.

Solution: If the colored tickets were dispensed from two different windows, then you would need nH_n trips to the red window and $2nH_{2n}$ trips to the blue window to get all $3n$ coupons. Thus, you need more trips to the blue window than the red. Now, when you go to the window, there is only a probability of $1/2$ that you get a blue coupon. Hence, you need an expected number of $4nH_{2n}$ trips to the window to get all the blue coupons. During these trips, we can expect that half the time you will be getting a red coupon. That is, the expected number of red tickets you will get is $2nH_{2n}$, and since $nH_n < 2nH_{2n}$, you will get enough red tickets to get all n in these trips as well. So the expected number to get all $3n$ tickets is $4nH_{2n}$.

C-19.8 Hint: Use the fact that if p is prime, then every nonzero integer less than p has a multiplicative inverse when we do arithmetic modulo p.

C-19.9 Hint: Recall the definition of a cut in a flow network and its relation to a maximum flow.

C-19.10 Hint: Use induction.

C-19.11 Hint: Review the analysis of the success probability of the Karger-Stein algorithm.

C-19.12 Hint: Use Fermat's Little Theorem.

C-19.13 Hint: Compose A and B into one algorithm.

C-19.14

C-19.15 Hint: Follow the general framework for the proof of the applicable Chernoff bound and use a calculator for the final calculations.

Solution: Let X be a sum of $(\alpha-t)n$ indicator random variables, each of which is 1 with probability p. Then

$$\Pr(Y < (\alpha - t)n) \; = \; \Pr(X \geq n),$$

For the probability that we use fewer than $(\alpha - t)n$ coin flips to get n heads is equal to the probability that we have at least n heads among exactly $(\alpha - t)n$ coin flips. So, letting $\mu = E[X]$, we get

$$(1+\delta)\mu = (1+\delta)\cdot(1-tp)n = n;$$

hence,

$$1+\delta = \frac{1}{1-tp} \quad \text{and} \quad \delta = \frac{tp}{1-tp}.$$

Thus, we can apply the Chernoff bound on the sum of indicator random variables to show

$$\Pr(Y < (\alpha - t)n) \; \leq \; \left[\frac{e^{tp/(1-tp)}}{[1/(1-tp)]^{1/(1-pt)}}\right]^{(1-tp)n},$$

which is equal to

$$e^{tpn}(1-tp)^n,$$

which, for $t = 3/4p$, is at most $(2/3)^n$.

C-19.16 Hint: Find the beginning and end of the range.

C-19.17 Hint: In the insertion algorithm, first repeatedly flip the coin to determine the level where you should start inserting the new item.

Solution: First, observe that the method SkipSearch(k) is a top-down, scan-forward traversal technique. The removal of an item can be performed in a similar traversing manner.

C-19.18 Hint: Use the Chernoff bound for a sum of geometric random variables.

Applications

A-19.1 Hint: Think about how this problem relates to the harmonic numbers.

A-19.2 Hint: Consider how the coupon collector problem applies.

Solution: This is an instance of the coupon collector problem, with $n = 175,711,536$, since the only way to guarantee a winner is to have every combination of numbers issued as a lottery ticket. Thus, the expected number is nH_n, which we can approximate as $n \ln n$, which, in this case, is $3,335,770,038$.

A-19.3 Hint: Think about how the coupon collector problem applies here.

Solution: (a) $p(1-p)^{d-1}$. Note that the exponent is $d-1$, not d, since the there are $d-1$ routers after the one that is farthest away.

(b) The probability that any router's ID makes it to the recipient is dominated by the probability for the farthest router. Thus, by the analysis for the coupon collector problem, the expected number of packets that must be received by the recipient to identify all the routers is at most

$$\frac{dH_d}{p(1-p)^{d-1}}.$$

A-19.4 Hint: Use a Chernoff bound and the value of 6 choose 5.

A-19.5 Hint: Think in terms of figuring how a cell in A remains 0.

A-19.6 Hint: Recall that when you multiply powers you add their exponents. Also, recall that the sum of 1 to m is $m(m-1)/2$.

Chapter 20 B-Trees and External Memory

Hints and Solutions

Reinforcement

R-20.1

R-20.2 Solution: T is a valid (a, b) tree for $2 \leq a \leq \min\{5, (b+1)/2\}$ and $b \geq 8$. For example, T could be a $(4, 8)$ tree, a $(5, 9)$ tree, but not a $(5, 8)$ tree.

R-20.3 Solution: T could be an order-8, order-9, or order-10 B-tree.

R-20.4

R-20.5

R-20.6

R-20.7 Solution: This sequence causes 5 page misses, on the first 5, second 3, second 4, third 2, and last 3.

R-20.8 Solution: This sequence causes 3 page misses, on the first 5, the last 2, and the last 3.

R-20.9

R-20.10

Creativity

C-20.1 Hint: Consider an alternate linked list implementation that uses "fat" nodes.

Solution: Consider a linked list implementation of the dictionary where every node is a block of size B. We would perform an insertion by first reading in the last block. If this block is not full, then we add the new element to the end and transfer this block back. If this block is full, then we allocate a new block for the new element, transferring this block back to the disk when we are done. We also create a link in the previous block to this new block and we transfer the previously last block back to the disk.

C-20.2

C-20.3

C-20.4

C-20.5

C-20.6 Hint: Consider what happens to a page that is accessed a lot and then never accessed again.

Solution: Consider a sequence where page 1 is accessed m times and then never accessed again, and then pages 2 through $m+1$ are each accessed once and then this same access sequence of pages 2 through $m+1$ is repeated. The LFU algorithm will never through out page 1; hence, will have a page miss on every access, starting with page $m+1$. But an optimal algorithm will throw out page 1 and then never have any more page misses after the misses for the first time pages 2 through $m+1$ are accessed.

C-20.7

C-20.8

C-20.9

C-20.10 Show that the Marker algorithm is H_m-competitive when the size of the cache is m and there are $m+1$ possible pages that can be accessed, where H_m denotes the mth Harmonic number.

Applications

A-20.1 Hint: Think of extending a linked list implementation to external memory.

Solution: Use a linked list where each node is a block of size B. We add elements to the end, using $O(1)$ disk transfers (creating a new block when we have filled the last block). We dequeue elements from the front, returning the block to the free memory heap when it becomes empty.

A-20.2 Hint: Think about how to do union by size in this context.

A-20.3 Hint: Use a good sorting algorithm first.

Solution: First sort the course numbers, using external-memory mergesort, then scan this list, counting the number of instances of each course.

A-20.4 Solution: Sort the elements of S using an external-memory sorting algorithm. This brings together the elements with the same keys. One more scan through this sorted order matches up elements with the same key and we can then write out all the files. The time for this algorithm is dominated by the time to sort, which can be done using $O((n/B)\log(n/B)/\log(M/B))$ block transfers.

A-20.5 Hint: Choose an appropriate day for Alice to flip her coin and decide to rent or buy based on its outcome.

Solution: Alice can rent skis until day 14 and then flip her coin to decide on that day whether she should continue renting (and buy on day 20) or buy now. Assuming the rental cost is \$1, if this is the last day she skis, her expected cost is

$$(13+20)/2 + 14/2 = 16.5 + 7 = 23.5,$$

whereas the optimal strategy has cost 14. If she skis for 20 times, then her expected cost is

$$(13+20)/2 + (19+20)/2 = 16.5 + 19.5 = 36,$$

whereas the optimal strategy has cost 20.

Chapter 21 Multidimensional Searching

Hints and Solutions

Reinforcement

R-21.1 Hint: Think about how bad the height could be.

R-21.2 Hint: Think about how to do this with a traversal.

R-21.3 Hint: Review the heap-order property about priority queues.

R-21.4 Hint: Consider the different cases.

R-21.5 Hint: Notice the combinatorial condition that defines where the cut goes.

Solution: The depth is $O(\log n)$, even in higher dimensions, since the number of points is halved with each cut.

R-21.6 Hint: Try to characterize the depth in terms of N.

Solution: The worst-case depth is $O(\log N)$, since we divide the sides by two each time a square is cut.

R-21.7 Hint: Review the definition of a quadtree.

R-21.8 Hint: Review the definition of a k-d tree.

R-21.9 Hint: Review the definition of a priority search tree.

Creativity

C-21.1 Hint: Think about how to modify the search.

C-21.2 Hint: You could extend the AVL tree data structure, adding a new field to each internal node and ways of maintaining this field during updates.

Hint: Think about what additional information is needed at each node.

Solution: For each node of the tree, maintain the size of the corresponding subtree, defined as the number of internal nodes in that subtree. While performing the search operation in both the insertion and deletion, the subtree sizes can be either incremented or decremented. During the rebalancing, care must be taken to update the subtree sizes of the three nodes involved (labeled a, b, and c by the restructure algorithm).

To calculate the number of nodes in a range (k_1, k_2), search for both k_1 and k_2, and let P_1 and P_2 be the associated search paths. Call v the last node common to the two paths. Traverse path P_1 from v to k_1. For each internal node $w \neq v$ encountered, if the right child of w is in not in P_1, add one plus the size of the subtree of the child to the current sum. Similarly, traverse path P_2 from v to k_2. For each internal node $w \neq v$ encountered, if the left child of w is in not in P_2, add one plus the size of the subtree of the left to the current sum. Finally, add one to the current sum (for the key stored at node v).

C-21.3 Hint: Review the definition of a range tree and then think about the steps needed to construct it, including getting the points in the correct order.

C-21.4 Hint: Used a balanced binary tree.

Solution: Use a balanced binary tree, T, which stores the points of S in its external nodes, ordered by their keys. At each internal node, v, store the number of external nodes in the subtree rooted at v. With this additional information, we can perform a binary search for any rank in the set. Thus, we can use an algorithm similar to the 1DTreeRangeSearch to find all the items with ranks in the range $[a, b]$ in $O(\log n + k)$ time, where k is the number of answers.

C-21.5 Hint: Think about how to use a range tree for this.

Solution: Use a range tree, where in addition to the usual information, we also store the number of items stored in the subtree rooted in each auxiliary tree. But now instead of enumerating the items that are in the range, we simply count them (using the extra information).

C-21.6 Hint: Think of storing auxiliary structures at each node that are "linked" to the structures at neighboring nodes.

C-21.7 Hint: Design a recursive data structure that builds a d-dimensional structure using $(d-1)$-dimensional structures.

C-21.8 Hint: An enumeration includes all the reported points.

C-21.9 Hint: Think about reducing this to a two-dimensional problem.

Solution: Note that x is contained by $[a, b]$ if $a \leq x$ and $x \leq b$. That is, the point (a, b) is contained in the two-sided range of all points above and to the left of the point (x, x). Thus, we can use a priority search tree to enumerate the interval (viewed as two-dimensional points) in time $O(\log n + k)$, by viewing the two-sided range as a simple case of a 3-sided range.

C-21.10 Hint: Review the different ways of balancing a binary search tree.

Applications

A-21.1 Hint: Draw boxes that define where points of the form $(0.0\ldots, 0.0\ldots)$, $(0.0\ldots, 0.1\ldots)$, $(0.1\ldots, 0.0\ldots)$, and $(0.1\ldots, 0.1\ldots)$ would go.

A-21.2 Hint: Draw a picture that defines the different cases of how you would want to search in a k-d tree with respect to an axis-aligned rectangle, R, and then use a recurrence equation to characterize the running time of this search.

A-21.3 Hint: Think about the advantages that can come from having the points in sorted order.

Solution: Use a bucket sort to sort the input points by x-coordinates in $O(n)$ time and build a complete binary search tree on this set. Then perform a recursive algorithm similar to the heapify algorithm. This algorithm starts at the root, and recursively constructs a PST for the left and right children. Then it selects the child storing a point with larger y-coordinate, places that point at the root, and then repeates this down-heap bubbling action at the child where this point came from. The total time for this second phase is also $O(n)$.

A-21.4 Hint: If you are using more than $O(n)$ space, you are doing something wrong.

A-21.5 Hint: Think about the transformations that need to be made at each internal node.

A-21.6 Hint: Think about how to divide a single equilateral triangle into 4 equal-sized subtriangles.

A-21.7 Hint: If we divide a square into four equal-sized squares, and assign n points uniformly and independently at random to the square, consider the probability that any subsquare has more than $n/2$ of the points.

A-21.8 Hint: Think of how to characterize a rectangle using points.

Solution: Any rectangle is completely characterized by its left-lower and right-upper corners. These two points, (x_1, y_1) and (x_2, y_2), can be stored as a four-dimensional point, (x_1, y_1, x_2, y_2). Then a query range in two dimensions,

$$[(x_1', x_2'), (y_1', y_2')],$$

for two-dimensional objects, can be expressed as a four-dimensional range,

$$[(x_1', x_2'), (y_1', y_2'), (x_1', x_2'), (y_1', y_2')],$$

for the four-dimensional points. Thus, the data structure, D, can be used to store the bounding boxes as points and two-dimensional query ranges for bounding boxes can be converted into four-dimensional query ranges for the transformed points.

A-21.9 Hint: Think of starting from a priority range tree.

Chapter 22 Computational Geometry

Hints and Solutions

Reinforcement

R-22.1 Hint: Review the statement of the theorem and work out the details.

R-22.2 Hint: Review the statement of the theorem and work out the details.

R-22.3 Hint: Review the statement of the theorem and work out the details.

R-22.4 Hint: Review the statement of the theorem and work out the details.

R-22.5 Hint: Review the statement of the theorem and work out the details.

R-22.6 Hint: Review the statement of the theorem and work out the details.

R-22.7 Hint: Review the informal description of this algorithm given in the book.

R-22.8 Hint: Recall the definition of convex hull.

Creativity

C-22.1 Hint: Try to do this in terms of three orientation tests.

C-22.2 Hint: Use as a guide the figure illustrating the properties of the convex hull.

Solution: Project a vertical line through every vertex of our convex polygon P. Such a set of lines cuts the boundary of P through two points (except for the lines through the leftmost and rightmost vertices). These lines also subdivide the plane into vertical slabs, which can be ordered left-to-right by a simple sorting step (that can even be implemented in $O(n)$ time if we use the ordering information around P). Given this ordered set of slabs, we can perform polygon inclusion by first determining the slab that contains our query point q using a binary search. Then in constant additional time we can determine whether q is inside the part of P that this slab cuts (there are at most two more line comparisons to do this).

C-22.3 Hint: Work through the cases.

C-22.4 Hint: Use the fact established in the previous exercise.

C-22.5 Hint: Test the cumulative amount of winding that P does.

Solution: To determine whether P is convex, we first traverse the vertices of P in a counterclockwise order, making sure each one determines a left turn. This is not enough, however, for P could self-intersect. So, we next pick a vertex v on P and determine, for each other vertex w that the line vw is locally interior to P with respect to the edges incident on v and w respectively.

C-22.6 Hint: There is no simple formula for the area of an entire arbitrary convex polygon. But you should know how to compute the area of more simple shapes.

C-22.7 Hint: Think about how to modify the Graham scan algorithm to work on the vertices of P.

C-22.8 Hint: Recall that the edges in a simple polygon do not cross.

C-22.9 Hint: Use the setup for the Graham scan algorithm as a guide.

C-22.10 Hint: You may use a brute-force algorithm here.

Solution: We first use the listing of the polygon's vertices to create a set S of all the segments in the boundary of the polygon. Then we can compare every pair of segments to see if any two intersect.

C-22.11 Hint: Review the assumptions made by that algorithm.

C-22.12 Hint: Think about the points that are immediately available when we determine that a point is not on the convex hull.

C-22.13 Hint: Think about how you might modify the plane-sweep algorithm given in the book to solve this problem.

C-22.14 Hint: Think about how to generalize a binary search for this scenario.

Solution: We can use a binary search to locate q. The important observation is that the segments in A are ordered left to right. So, given any segment s_i in A, we can in constant time determine whether q is to the left or right of s_i, which in turn determines if we should continue searching among the segments to the left or right of s_i. Thus, we can ultimately locate the segment immediately right of q (if it exists) in $O(\log n)$ time.

C-22.15 Hint: Think about the possible cases for the second closest pair.

Solution: The second closest pair of points in the set S must either be a closest pair in $S - \{p, q\}$ or must include one of the points, p and q. So, run the closest pair algorithm on the set $S - \{p, q\}$ and compare the distance between the two points, (r, s), returned to the closest pair of points of the form (p, t) and (t, q), such that t is in S. One of these pairs will be the second closest pair of points. Moreover, this algorithm will run in $O(n \log n)$ time.

Applications

A-22.1 Hint: Think about how having the convex hull of S can help here.

A-22.2 Hint: Think about shooting a ray out from q and considering how it intersects with P.

Solution: Let r be a ray emanating vertically up from q. Scan through the edges of P and count how many of these edges intersect with r. If this number is odd, then q is inside P; if it is even, then q is outside P.

A-22.3 Hint: You should use sorting as a part of this solution.

Solution: By a sorting step on the segment endpoints, we can determine that each endpoint exists exactly twice. If this condition holds, then every endpoint forms a polygon vertex. If the condition does not hold, then S cannot form a polygon. So, if this condition is true, we then connect the edges of S according to their adjacencies and perform a traversal of these edges starting at some vertex v. If this traversal visits all the edges, then they form a polygon. If this traversal misses some edges, then the segments in S form at least two polygons. The running time of this algorithm is $O(n \log n)$, which is dominated by the sorting step.

A-22.4 Hint: Use the plane-sweep technique, including segments intersections as events. Note that you cannot know these events in advance, but it is always possible to know the next event to process as you are sweeping.

A-22.5 Hint: Think about how having the convex hull of S can help here.

A-22.6 Hint: Consider using the plane-sweep technique.

A-22.7 Hint: Think about how a convex hull algorithm could help here.

Solution: Construct the convex hull of the red points and the convex hull of the blue points. There is a line separating these sets of points if and only if these two convex hulls do not intersect. So, do a simple plane-sweeping algorithm to test if these two convex hulls intersect. The running time of this algorithm is $O(n \log n)$.

A-22.8 Hint: Think about how to modify the plane-sweep algorithm given in the book.

A-22.9

A-22.10

Chapter 23 String Algorithms

Hints and Solutions

Reinforcement

R-23.1 Solution: There are three: `a, aa, aaa`.

R-23.2

R-23.3

R-23.4
R-23.5

R-23.6

R-23.7
R-23.8

R-23.9

R-23.10

R-23.11 Solution: The longest prefix that is also a suffix of this string is `"cgtacg"`.

R-23.12 Hint: Try to devise an input instance where there is a match for every substring of the text.

Solution: Consider the pattern, $P = \mathtt{a}^m$, and pattern, $T = \mathtt{a}^n$. Since there is a match for every substring of T in this case, and the Karp-Rabin algorithm verifies every possible match, this algorithm will take $\Omega(nm)$ time for this input instance.

R-23.13 Hint: Look at how the table values are indexed.

R-23.14 Hint: Review how this function was defined for the polynomial hash function.

Solution: shiftHash$(h(X[i..i+m-1]), X, i) = (h(X[i..i+m-1]) - X[i] + X[i+m]) \bmod p$.

Creativity

C-23.1 Hint: Make the text and the pattern very periodic.

Solution: $T =$ "`aaaaaaaaaaaaaaaaaaa`", $P =$ "`aaaab`".

C-23.2 Solution:

The justification is very similar to the argument that the number of iterations in KMPMatch is $O(n)$.

Define $k = i - j$ for the sake of analysis. One of the following conditions occurs at each iteration of the loop:

- If $P[i] = P[j]$, then i increases by 1, and k does not change, since j also increases by 1.
- If $P[i] \neq P[j]$ and $j > 0$, then i does not change and k increases by at least 1, since in this case k changes from $i - j$ to $i - f(j-1)$, which is an addition of $j - f(j-1)$, which is positive because $f(j-1) < j$.
- If $P[i] \neq P[j]$ and $j = 0$, then i increases by 1 and k increases by 1, since j does not change.

As a result, the number of iterations is at most $2m$. Therefore, KMPFailureFunction runs in $O(m)$ time.

C-23.3 Solution: Instead of returning when a match is found, store the index and set $i = i + 1$ and $j = f(m)$.

C-23.4 Hint: Consider using a suffix trie.

Solution: Modify the KMPMatch algorithm to maintain a variable $maxIndex$ which is the index of the longest prefix found, $maxLen$ which is the length of the longest prefix found, and $currentLen$ which is the length of the current prefix. Initialize all three variables to zero and modify the loop in KMPMatch as follows:

- If $T[i] = P[j]$, increment $currentLen$
- If $T[i] \neq P[j]$ and $j > 0$, if $currentLen > maxLen$, then set $maxLen = currentLen$ and $maxIndex = i - j$. In any case, reset $currentLen = 0$.

When the algorithm terminates, $maxIndex$ and $maxLen$ will hold the location and length of the longest prefix.

C-23.5 Hint: Convert this problem to a non-circular pattern matching problem.

Solution: Generate a new text $T' = T[n-m \ldots n]+T[0 \ldots m]$. Run KMPMatch on T' and P.

C-23.6

C-23.7

C-23.8 Hint: Think about how to modify the shiftHash function for the position that contains the wild-card symbol.

C-23.9

C-23.10 Solution: Locate the external node with corresponds to the end of the string. While walking back to the root of the trie, delete every external node encountered. The running time of this algorithm is $O(s)$ where s is the length of the string to be deleted.

C-23.11

C-23.12

C-23.13 Hint: Try to implement a rolling hash for all k possible sizes at the same time.

Applications

A-23.1 Hint: Think about how to use a data structure described in the chapter for this purpose.

Solution: This performance can be achieved by storing all the encountered web pages as strings in a compressed trie, T. Each search can then be done in $O(n)$ time, to reach the node in T where the string belongs. If there is already a leaf node for that location, then this is a previously encountered web page. If not, then we can create such a node in $O(1)$ time and add it to T.

A-23.2 Hint: Think of using an efficient data structure.

Solution: One can either use a compressed trie or a cuckoo hash table for this purpose. With either case, you will get constant-time stop-word identification for any constant-length stop word.

A-23.3 Hint: Consider using a prefix trie.

A-23.4

A-23.5 Hint: Consider using a greedy algorithm.

A-23.6 Hint: Consider using dynamic programming for this problem.

A-23.7 Hint: This is a remarkably accurate algorithm when m is large.

A-23.8 Hint: Consider modifying the Karp-Rabin algorithm for this problem.

Chapter 24 Cryptography

Hints and Solutions

Reinforcement

R-24.1 Hint: Use a spreadsheet.

R-24.2

R-24.3 Hint: This one is easy if you remember Euler's Theorem.

Solution: By Euler's Theorem, the answer is 1.

R-24.4

R-24.5 Hint: Use a spreadsheet.

Solution:

p	12	6	3	1	0
r	1	12	8	5	1

R-24.6

R-24.7

R-24.8 Hint: Use a spreadsheet.

Solution:

a	412	113	73	40	33	7	5	2	1
b	113	73	40	33	7	5	2	1	0
r	3	1	1	1	4	1	2	2	
i	48	-31	17	-14	3	-2	1	0	1
j	-175	48	-31	17	-14	3	-2	1	0

R-24.9 Solution:

$$113^{-1} \equiv 172 \bmod 299$$
$$114^{-1} \equiv 160 \bmod 299$$
$$127^{-1} \equiv 228 \bmod 299$$

R-24.10 Hint: Write a small Java program or use a spreadsheet.

Solution:

M	10	11	12	13	14	15	16	17	18	19	20
C	193	197	122	166	29	2	118	272	18	304	39

R-24.11 Hint: Follow the formulas for computing a and b as described in the book.

Creativity

C-24.1 Hint: Review the basic facts of arithmetic.

C-24.2 Hint: Give an exact equation for the inverse.

C-24.3 Hint: Use Fermat's Little Theorem.

C-24.4 Hint: Start by proving the theorem about the existence of generators.

Solution: Consider the numbers $1, 2, \cdots, p-1$. We claim that multiplication by x yields distinct values (mod p) for each of the numbers between 1 and $p-1$. Why? Well, suppose not. Then there are some i and j, $p > i > j$ such that $ai \equiv aj \bmod p$. (If $i < j$, then we can just switch the two, and continue with the proof.) From our discussion above, this means that $ai - aj = kp$, or $a(i-j) = kp$. Now, a and p are relatively prime. Thus, a must divide k. This means that $i - j$ is a multiple of p, a contradiction, since $i - j < p$, so this can't be true. Therefore, we have proved the existence of generators. Since the set $\{ix \bmod p, i = 0, 1, \cdots, p-1\}$ is the same as Z_p, for $x > 0$, there must be an i such that $ix \equiv 1 \bmod p$, which completes the proof.

C-24.5 Hint: Think about how to make sure each step uses just this many bits.

Solution: Replace the statement

return $(d, l, k - lr)$

with

return $(d, l, (k - (lr \bmod n)) \bmod n)$

C-24.6 Hint: Use the mathematical properties of RSA encryption.

C-24.7 Hint: Use the mathematical properties of RSA encryption.

C-24.8 Hint: The number of primes to test for factoring is greatly reduced here. Use this information to break RSA.

C-24.9 Hint: What will d be in this case?

Applications

A-24.1 Hint: Note that $A \oplus A = 0$.

A-24.2 Hint: Consider what the owner of the document, D_i, needs to reconstruct h_r.

A-24.3 Hint: Notice that only $O(\log n)$ numbers need to change in T.

A-24.4 Hint: Think of using y and H in a backwards fashion.

Solution: Each day Alice should reveal the previous number in the chain of hashes that leads to y. The handler can apply the hash function the appropriate number of times to prove that the result is equal to y. Since H is one-way, only Alice could have effectively done this inversion, since she knows x.

A-24.5 Hint: Think about how to combine concepts from this chapter and a dictionary data structure.

Solution: One possible solution is to hold the revoked certificates in a binary search tree, and have the authority digitally sign each response. In this case a query takes $O(\log n)$ time plus the time for producting a digital signature, and any third party can verify that signature later.

A-24.6 Hint: Think about how Alice and Bob can exchange the secret key, based on the use of public keys.

A-24.7 Hint: You need to get rid of the random number used to obfuscate what Bob signed.

Chapter 25 The Fast Fourier Transform

Hints and Solutions

Reinforcement

R-25.1 **Hint:** Review the (forward) FFT algorithm.

R-25.2

R-25.3 **Solution:** $(0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15)$.

R-25.4 **Hint:** Use a calculator program that has a mod function.

R-25.5

R-25.6

R-25.7

R-25.8 **Hint:** Review the divide-and-conquer algorithm provided.

R-25.9 **Hint:** Draw the recursion trees.

Creativity

C-25.1 **Hint:** Review the definition of a primitive root of unity.

Solution: If ω is a primitive (cn)th root of unity, then 1, ω, ω^2, ..., ω^{cn-1} are all distinct. Thus, the numbers 1, ω^c, $(\omega^c)^2$, ..., $(\omega^c)^{n-1}$ are all distinct. In addition, $(\omega^c)^n = \omega^{cn} = 1$.

C-25.2 **Hint:** Review the definition of a primitive root of unity.

C-25.3 **Hint:** Use the FFT.

Solution: Compute $p'(x)$ and $q'(x)$ using the standard calculus rule for computing the derivative of a polynomial. Then use the FFT to compute the coefficients of the polynomial $p' \cdot q'$.

C-25.4

C-25.5 Hint: You don't need the FFT for this one.

Applications

A-25.1 Hint: Use some of the principles developed in this chapter regarding polynomials.

A-25.2 Hint: Use a divide-and-conquer algorithm.

Solution: Build a complete binary tree, T, having each x_i as a leaf. For each internal node, v, construct a coefficient-form representation of the polynomial $q_{i,j}(x)$, where v's descendants are $\{x_i, \ldots, x_j\}$. This can be done bottom-up in T using the FFT polynomial multiplication algorithm, which uses $O(n \log n)$ arithmetic operations per level; hence, a total number of $O(n \log^2 n)$ arithmetic operations. Finally, perform a top-down computation in T, by using these representations and the "black box" to compute $p(x_i)$ for each leaf, x_i.

A-25.3 Hint: Note that, if $p(x)$ is zero on a set of values X and $q(x)$ is zero on a set of values Y, then $p(x) \cdot q(x)$ is zero on the set of values $X \cup Y$.

Solution: Divide X into two sets X_1 and X_2 of equal size ($n/2$), and recursively construct polynomials $p_1(n)$ and $p_2(n)$, such that $p_1(X_1) = 0$ and $p_2(X_2) = 0$. Then use the FFT algorithm to compute the polynomial $p_1 \cdot p_2$. This algorithm satisfies the recurrence $T(n) = 2T(n/2) + bn \log n$, for some constant b, which implies that $T(n)$ is $O(n \log^2 n)$ by the master theorem.

A-25.4 Hint: Note that the kth position in the convolution of two bit strings, A and B, counts the number of 1's that match among the first $k-1$ places in A with the last $k-1$ places in the reversal of B.

Solution: Pad P with 0's to form P' and then compute the reversal of P' to form R. Now compute the convolution of T and R. Each index, $m \le k \le n-m$, in the convolution will count the number of places where the 1's in P match the 1's in T. Next, do this computation again on the complements of T and P (trading the 0's and 1's). Each index, $m \le k \le n-m$, in this convolution will count the number of places where the 0's in P match the 0's in T. The places where they two numbers sum to m are the places where P appears in T.

A-25.5 Hint: Think about how to convert the strings P and T to binary.

Solution: Convert the strings P and T to binary, e.g., using a standard binary encoding of the alphabet Σ.

A-25.6 Hint: Think about how to modify the strings given to the input to the Conv method.

Solution: Do the reduction as before, but this time, in each place where there is a $\star$ in P, convert all the bits associated with this symbol to 1's (even if we are counting 0's).

A-25.7 Hint: Think about how to set up the linear equations.

Solution: The coefficients of $p(x)$ define the variables, with the coefficients now being powers of x, for each (x, y) in S. To set up these equations, then, you need to compute all the powers, $x, x^2, \ldots, x^{n-1}$, for each (x, y) pair in S. Doing these iteratively for each such pair takes $O(n)$ time; hence, the setup for all the equations can be done in $O(n^2)$ time.

A-25.8 Hint: Try to set this up as a convolution computation.

Chapter 26 Linear Programming

Hints and Solutions

Reinforcement

R-26.1 Hint: Read the chapter.

R-26.2 Hint: Read the chapter and note the geometry of LPs.

R-26.3 Hint: Read the chapter and note the geometric way of finding an LP solution.

R-26.4 Hint: Recall the conditions where there is no solution.

R-26.5 Hint: Read the chapter, focusing on the simplex method.

R-26.6 Hint: Read the chapter.

R-26.7 Hint: Read the chapter.
R-26.8 Hint: Read the chapter.

R-26.9

R-26.10 Hint: Read the chapter.

Solution: For $(3, 9)$: maximize $z = 0.1x+y$. For $(8, 6)$: maximize $z = x+0.1y$.

R-26.11 Hint: Read the chapter, focusing on the condition that causes an infinite number of solutions.

Solution: maximize $z = 0.6x + y$. The value of the objective function in this case is 10.8.

R-26.12 Hint: Review the sections of the dual and the maximum flow problem.

R-26.13 Hint: Recall the definition of LP duality.

Creativity

C-26.1 Hint: Think about how many points should be contained in such a linear program's feasible region.

Solution: There are many solutions. Here is one: maximize $z = x + y$, subject to $x \geq 0$, $y \geq 0$, and $-2x - y \geq 0$.

C-26.2 Hint: What do we know about the intersection of the feasible region and the objective function if there is more than one optimal solution?

Solution: Suppose there are two optimal solutions, $\vec{x}$ and $\vec{y}$, to a linear objective function. Since a feasible region is convex, this implies that every point on the line segment from $\vec{x}$ to $\vec{y}$ must also be a solution.

C-26.3 Hint: Recall the definition of ***convex***: that for any two points p, q in a convex set, the line segment $\overline{pq}$ is also in the set. Observe that the feasible region defined by any single constraint (that is, a half-plane) is convex. How does the feasible region defined by multiple constraints relate to the feasible regions of the individual constraints?

C-26.4 Hint: It doesn't have to be a regular tetrahedron. An easy way is to let three of the faces be defined by the xy, xz and yz planes.

Solution: There are many solutions. Here is one: maximize $z = x + y + w$, subject to $x, y, w \geq 0$ and $x + y + w \leq 10$.

C-26.5 Hint: Make sure you understand the simplex algorithm.

C-26.6 Hint: Make sure you understand the simplex algorithm.

C-26.7 Hint: Start by showing that $P^{2*} = P$. What happens to the variables in $\vec{b}$, $\vec{c}$, B, and F with each application of the dual?

C-26.8 Hint: What happens at the boundary of the feasible region?

Solution: Here is an example of an LP with strict inequalities that is feasible and bounded, but there is no finite optimal solution: maximize $z = x + y$, subject to $x, y \geq 0$ and $2x + y < 10$. Since $(0, 10)$ is not feasible, for any feasible point (x', y'), there is another feasible point, $(x', y' + (10 - y')/2)$, which improves the objective function.

C-26.9 Hint: Start by thinking about the feasible region of the primal and dual form of a one-dimensional linear program.

C-26.10 Hint: What are the possible values for i_v, and what do they correspond to in the original problem?

C-26.11 Hint: The algorithm our linear program is based on may also have interesting behavior if the graph has negative cycles.

Solution: If the graph has a negative cycle, then the feasible region is empty, since in this case, it is not possible to simultaneously satisfy the triangle inequality for every edge.

C-26.12 Hint: Take inspiration from the linear program for the single-source shortest-path problem

C-26.13 Hint: Use an indicator variable for each edge. What is the objective we are trying to minimize, and what are the constraints on the edges? Use the following spanning tree facts: (1) in a spanning graph, each vertex must be incident to at least one edge (why?), and (2) every subgraph F of a tree T has at most $|F| - 1$ edges. (why?).

C-26.14 Hint: Make sure you understand the first part of the proof given in the chapter.

C-26.15 Hint: Make sure you understand the first part of the proof given in the chapter.

Hint: Try to isolate each term $a_i x_i$

Applications

A-26.1 Hint: Assume the total number of bits required to encode the points is a function of the total size of all ϵ_i.

A-26.2 Hint: Create a variable for each category of item.

A-26.3 Hint: Use a variable for each type of ad.

A-26.4 Hint: Use a variable for each pizza ingredient.

A-26.5 Hint: There are constraints on the amount of goods shipped from each wherehouse and amount shipped to each retail store.

A-26.6 Hint: Each power plant is a source of power that may flow through the electrical network to each city.

A-26.7 Hint: The constraints of the linear program must partition the items among the minimum number of camels. Imagine you select some number of camels to carry your gear from at most n camels. Use variables to indicate which camels were selected, and which pieces of gear are assigned to which camels.