Selected Solutions for Chapter 2: Getting Started

Solution to Exercise 2.2-2

```
SELECTION-SORT($A$)
$n = A.length$
for $j = 1$ to $n - 1$
    $smallest = j$
    for $i = j + 1$ to $n$
        if $A[i] < A[smallest]$
            $smallest = i$
    exchange $A[j]$ with $A[smallest]$
```

The algorithm maintains the loop invariant that at the start of each iteration of the outer **for** loop, the subarray $A[1 \mathinner{.\,.} j-1]$ consists of the $j-1$ smallest elements in the array $A[1 \mathinner{.\,.} n]$, and this subarray is in sorted order. After the first $n-1$ elements, the subarray $A[1 \mathinner{.\,.} n-1]$ contains the smallest $n-1$ elements, sorted, and therefore element $A[n]$ must be the largest element.

The running time of the algorithm is $\Theta(n^2)$ for all cases.

Solution to Exercise 2.2-4

Modify the algorithm so it tests whether the input satisfies some special-case condition and, if it does, output a pre-computed answer. The best-case running time is generally not a good measure of an algorithm.

Solution to Exercise 2.3-5

Procedure BINARY-SEARCH takes a sorted array A, a value v, and a range $[low \mathinner{.\,.} high]$ of the array, in which we search for the value v. The procedure compares v to the array entry at the midpoint of the range and decides to eliminate half the range from further consideration. We give both iterative and recursive versions, each of which returns either an index i such that $A[i] = v$, or NIL if no entry of

$A[low \mathinner{..} high]$ contains the value v. The initial call to either version should have the parameters $A, v, 1, n$.

```
ITERATIVE-BINARY-SEARCH($A, v, low, high$)
while $low \le high$
    $mid = \lfloor (low + high)/2 \rfloor$
    if $v == A[mid]$
        return $mid$
    elseif $v > A[mid]$
        $low = mid + 1$
    else $high = mid - 1$
return NIL
```

```
RECURSIVE-BINARY-SEARCH($A, v, low, high$)
if $low > high$
    return NIL
$mid = \lfloor (low + high)/2 \rfloor$
if $v == A[mid]$
    return $mid$
elseif $v > A[mid]$
    return RECURSIVE-BINARY-SEARCH($A, v, mid + 1, high$)
else return RECURSIVE-BINARY-SEARCH($A, v, low, mid - 1$)
```

Both procedures terminate the search unsuccessfully when the range is empty (i.e., $low > high$) and terminate it successfully if the value v has been found. Based on the comparison of v to the middle element in the searched range, the search continues with the range halved. The recurrence for these procedures is therefore $T(n) = T(n/2) + \Theta(1)$, whose solution is $T(n) = \Theta(\lg n)$.

Solution to Problem 2-4

a. The inversions are $(1, 5), (2, 5), (3, 4), (3, 5), (4, 5)$. (Remember that inversions are specified by indices rather than by the values in the array.)

b. The array with elements from $\{1, 2, \ldots, n\}$ with the most inversions is $\langle n, n-1, n-2, \ldots, 2, 1 \rangle$. For all $1 \le i < j \le n$, there is an inversion (i, j). The number of such inversions is $\binom{n}{2} = n(n-1)/2$.

c. Suppose that the array A starts out with an inversion (k, j). Then $k < j$ and $A[k] > A[j]$. At the time that the outer **for** loop of lines 1–8 sets $key = A[j]$, the value that started in $A[k]$ is still somewhere to the left of $A[j]$. That is, it's in $A[i]$, where $1 \le i < j$, and so the inversion has become (i, j). Some iteration of the **while** loop of lines 5–7 moves $A[i]$ one position to the right. Line 8 will eventually drop *key* to the left of this element, thus eliminating the inversion. Because line 5 moves only elements that are less than *key*, it moves only elements that correspond to inversions. In other words, each iteration of the **while** loop of lines 5–7 corresponds to the elimination of one inversion.

d. We follow the hint and modify merge sort to count the number of inversions in $\Theta(n \lg n)$ time.

To start, let us define a ***merge-inversion*** as a situation within the execution of merge sort in which the MERGE procedure, after copying $A[p \mathinner{..} q]$ to L and $A[q+1 \mathinner{..} r]$ to R, has values x in L and y in R such that $x > y$. Consider an inversion (i, j), and let $x = A[i]$ and $y = A[j]$, so that $i < j$ and $x > y$. We claim that if we were to run merge sort, there would be exactly one merge-inversion involving x and y. To see why, observe that the only way in which array elements change their positions is within the MERGE procedure. Moreover, since MERGE keeps elements within L in the same relative order to each other, and correspondingly for R, the only way in which two elements can change their ordering relative to each other is for the greater one to appear in L and the lesser one to appear in R. Thus, there is at least one merge-inversion involving x and y. To see that there is exactly one such merge-inversion, observe that after any call of MERGE that involves both x and y, they are in the same sorted subarray and will therefore both appear in L or both appear in R in any given call thereafter. Thus, we have proven the claim.

We have shown that every inversion implies one merge-inversion. In fact, the correspondence between inversions and merge-inversions is one-to-one. Suppose we have a merge-inversion involving values x and y, where x originally was $A[i]$ and y was originally $A[j]$. Since we have a merge-inversion, $x > y$. And since x is in L and y is in R, x must be within a subarray preceding the subarray containing y. Therefore x started out in a position i preceding y's original position j, and so (i, j) is an inversion.

Having shown a one-to-one correspondence between inversions and merge-inversions, it suffices for us to count merge-inversions.

Consider a merge-inversion involving y in R. Let z be the smallest value in L that is greater than y. At some point during the merging process, z and y will be the "exposed" values in L and R, i.e., we will have $z = L[i]$ and $y = R[j]$ in line 13 of MERGE. At that time, there will be merge-inversions involving y and $L[i], L[i+1], L[i+2], \ldots, L[n_1]$, and these $n_1 - i + 1$ merge-inversions will be the only ones involving y. Therefore, we need to detect the first time that z and y become exposed during the MERGE procedure and add the value of $n_1 - i + 1$ at that time to our total count of merge-inversions.

The following pseudocode, modeled on merge sort, works as we have just described. It also sorts the array A.

```
COUNT-INVERSIONS($A, p, r$)
$inversions = 0$
if $p < r$
    $q = \lfloor (p + r)/2 \rfloor$
    $inversions = inversions +$ COUNT-INVERSIONS($A, p, q$)
    $inversions = inversions +$ COUNT-INVERSIONS($A, q + 1, r$)
    $inversions = inversions +$ MERGE-INVERSIONS($A, p, q, r$)
return $inversions$
```

```
MERGE-INVERSIONS$(A, p, q, r)$
$n_1 = q - p + 1$
$n_2 = r - q$
let $L[1..n_1 + 1]$ and $R[1..n_2 + 1]$ be new arrays
for $i = 1$ to $n_1$
    $L[i] = A[p + i - 1]$
for $j = 1$ to $n_2$
    $R[j] = A[q + j]$
$L[n_1 + 1] = \infty$
$R[n_2 + 1] = \infty$
$i = 1$
$j = 1$
$inversions = 0$
$counted = $ FALSE
for $k = p$ to $r$
    if $counted ==$ FALSE and $R[j] < L[i]$
        $inversions = inversions + n_1 - i + 1$
        $counted = $ TRUE
    if $L[i] \le R[j]$
        $A[k] = L[i]$
        $i = i + 1$
    else $A[k] = R[j]$
        $j = j + 1$
        $counted = $ FALSE
return $inversions$
```

The initial call is COUNT-INVERSIONS$(A, 1, n)$.

In MERGE-INVERSIONS, the boolean variable *counted* indicates whether we have counted the merge-inversions involving $R[j]$. We count them the first time that both $R[j]$ is exposed and a value greater than $R[j]$ becomes exposed in the L array. We set *counted* to FALSE upon each time that a new value becomes exposed in R. We don't have to worry about merge-inversions involving the sentinel ∞ in R, since no value in L will be greater than ∞.

Since we have added only a constant amount of additional work to each procedure call and to each iteration of the last **for** loop of the merging procedure, the total running time of the above pseudocode is the same as for merge sort: $\Theta(n \lg n)$.

Selected Solutions for Chapter 3: Growth of Functions

Solution to Exercise 3.1-2

To show that $(n+a)^b = \Theta(n^b)$, we want to find constants $c_1, c_2, n_0 > 0$ such that $0 \le c_1 n^b \le (n+a)^b \le c_2 n^b$ for all $n \ge n_0$.

Note that

$$\begin{aligned} n + a &\le n + |a| \\ &\le 2n \qquad \text{when } |a| \le n \ , \end{aligned}$$

and

$$\begin{aligned} n + a &\ge n - |a| \\ &\ge \frac{1}{2}n \qquad \text{when } |a| \le \tfrac{1}{2}n \ . \end{aligned}$$

Thus, when $n \ge 2|a|$,

$$0 \le \frac{1}{2}n \le n + a \le 2n \ .$$

Since $b > 0$, the inequality still holds when all parts are raised to the power b:

$$0 \le \left(\frac{1}{2}n\right)^b \le (n+a)^b \le (2n)^b \ ,$$

$$0 \le \left(\frac{1}{2}\right)^b n^b \le (n+a)^b \le 2^b n^b \ .$$

Thus, $c_1 = (1/2)^b$, $c_2 = 2^b$, and $n_0 = 2|a|$ satisfy the definition.

Solution to Exercise 3.1-3

Let the running time be $T(n)$. $T(n) \ge O(n^2)$ means that $T(n) \ge f(n)$ for some function $f(n)$ in the set $O(n^2)$. This statement holds for any running time $T(n)$, since the function $g(n) = 0$ for all n is in $O(n^2)$, and running times are always nonnegative. Thus, the statement tells us nothing about the running time.

Solution to Exercise 3.1-4

$2^{n+1} = O(2^n)$, but $2^{2n} \neq O(2^n)$.

To show that $2^{n+1} = O(2^n)$, we must find constants $c, n_0 > 0$ such that

$$0 \leq 2^{n+1} \leq c \cdot 2^n \text{ for all } n \geq n_0 .$$

Since $2^{n+1} = 2 \cdot 2^n$ for all n, we can satisfy the definition with $c = 2$ and $n_0 = 1$.

To show that $2^{2n} \neq O(2^n)$, assume there exist constants $c, n_0 > 0$ such that

$$0 \leq 2^{2n} \leq c \cdot 2^n \text{ for all } n \geq n_0 .$$

Then $2^{2n} = 2^n \cdot 2^n \leq c \cdot 2^n \Rightarrow 2^n \leq c$. But no constant is greater than all 2^n, and so the assumption leads to a contradiction.

Solution to Exercise 3.2-4

$\lceil \lg n \rceil!$ is not polynomially bounded, but $\lceil \lg \lg n \rceil!$ is.

Proving that a function $f(n)$ is polynomially bounded is equivalent to proving that $\lg(f(n)) = O(\lg n)$ for the following reasons.

- If f is polynomially bounded, then there exist constants c, k, n_0 such that for all $n \geq n_0$, $f(n) \leq cn^k$. Hence, $\lg(f(n)) \leq kc \lg n$, which, since c and k are constants, means that $\lg(f(n)) = O(\lg n)$.
- Similarly, if $\lg(f(n)) = O(\lg n)$, then f is polynomially bounded.

In the following proofs, we will make use of the following two facts:

1. $\lg(n!) = \Theta(n \lg n)$ (by equation (3.19)).
2. $\lceil \lg n \rceil = \Theta(\lg n)$, because
   - $\lceil \lg n \rceil \geq \lg n$
   - $\lceil \lg n \rceil < \lg n + 1 \leq 2 \lg n$ for all $n \geq 2$

$$\begin{aligned}
\lg(\lceil \lg n \rceil!) &= \Theta(\lceil \lg n \rceil \lg \lceil \lg n \rceil) \\
&= \Theta(\lg n \lg \lg n) \\
&= \omega(\lg n) .
\end{aligned}$$

Therefore, $\lg(\lceil \lg n \rceil!) \neq O(\lg n)$, and so $\lceil \lg n \rceil!$ is not polynomially bounded.

$$\begin{aligned}
\lg(\lceil \lg \lg n \rceil!) &= \Theta(\lceil \lg \lg n \rceil \lg \lceil \lg \lg n \rceil) \\
&= \Theta(\lg \lg n \lg \lg \lg n) \\
&= o((\lg \lg n)^2) \\
&= o(\lg^2(\lg n)) \\
&= o(\lg n) .
\end{aligned}$$

The last step above follows from the property that any polylogarithmic function grows more slowly than any positive polynomial function, i.e., that for constants $a, b > 0$, we have $\lg^b n = o(n^a)$. Substitute $\lg n$ for n, 2 for b, and 1 for a, giving $\lg^2(\lg n) = o(\lg n)$.

Therefore, $\lg(\lceil \lg \lg n \rceil !) = O(\lg n)$, and so $\lceil \lg \lg n \rceil !$ is polynomially bounded.

Selected Solutions for Chapter 4: Divide-and-Conquer

Solution to Exercise 4.2-4

If you can multiply 3×3 matrices using k multiplications, then you can multiply $n \times n$ matrices by recursively multiplying $n/3 \times n/3$ matrices, in time $T(n) = kT(n/3) + \Theta(n^2)$.

Using the master method to solve this recurrence, consider the ratio of $n^{\log_3 k}$ and n^2:

- If $\log_3 k = 2$, case 2 applies and $T(n) = \Theta(n^2 \lg n)$. In this case, $k = 9$ and $T(n) = o(n^{\lg 7})$.
- If $\log_3 k < 2$, case 3 applies and $T(n) = \Theta(n^2)$. In this case, $k < 9$ and $T(n) = o(n^{\lg 7})$.
- If $\log_3 k > 2$, case 1 applies and $T(n) = \Theta(n^{\log_3 k})$. In this case, $k > 9$. $T(n) = o(n^{\lg 7})$ when $\log_3 k < \lg 7$, i.e., when $k < 3^{\lg 7} \approx 21.85$. The largest such integer k is 21.

Thus, $k = 21$ and the running time is $\Theta(n^{\log_3 k}) = \Theta(n^{\log_3 21}) = O(n^{2.80})$ (since $\log_3 21 \approx 2.77$).

Solution to Exercise 4.4-6

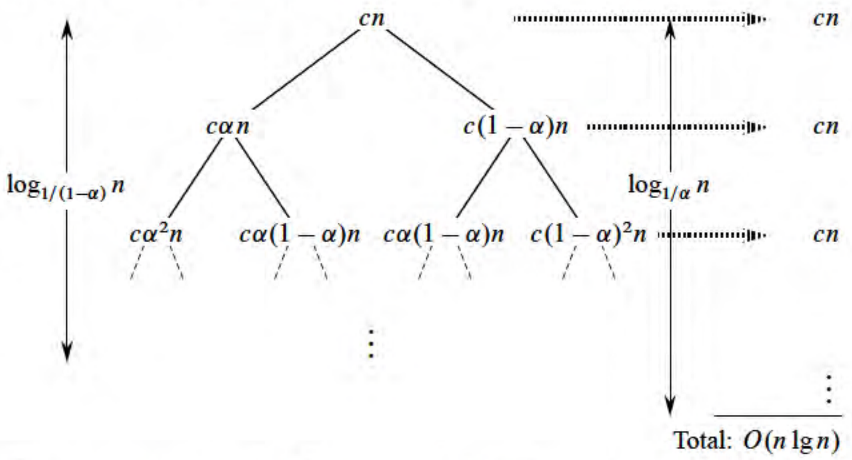
The shortest path from the root to a leaf in the recursion tree is $n \to (1/3)n \to (1/3)^2 n \to \cdots \to 1$. Since $(1/3)^k n = 1$ when $k = \log_3 n$, the height of the part of the tree in which every node has two children is $\log_3 n$. Since the values at each of these levels of the tree add up to cn, the solution to the recurrence is at least $cn \log_3 n = \Omega(n \lg n)$.

Solution to Exercise 4.4-9

$T(n) = T(\alpha n) + T((1-\alpha)n) + cn$

We saw the solution to the recurrence $T(n) = T(n/3) + T(2n/3) + cn$ in the text. This recurrence can be similarly solved.

Without loss of generality, let $\alpha \ge 1-\alpha$, so that $0 < 1-\alpha \le 1/2$ and $1/2 \le \alpha < 1$.



The recursion tree is full for $\log_{1/(1-\alpha)} n$ levels, each contributing cn, so we guess $\Omega(n \log_{1/(1-\alpha)} n) = \Omega(n \lg n)$. It has $\log_{1/\alpha} n$ levels, each contributing $\le cn$, so we guess $O(n \log_{1/\alpha} n) = O(n \lg n)$.

Now we show that $T(n) = \Theta(n \lg n)$ by substitution. To prove the upper bound, we need to show that $T(n) \le dn \lg n$ for a suitable constant $d > 0$.

$$\begin{aligned}
T(n) &= T(\alpha n) + T((1-\alpha)n) + cn \\
&\le d\alpha n \lg(\alpha n) + d(1-\alpha)n \lg((1-\alpha)n) + cn \\
&= d\alpha n \lg \alpha + d\alpha n \lg n + d(1-\alpha)n \lg(1-\alpha) + d(1-\alpha)n \lg n + cn \\
&= dn \lg n + dn(\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)) + cn \\
&\le dn \lg n ,
\end{aligned}$$

if $dn(\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)) + cn \le 0$. This condition is equivalent to

$$d(\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)) \le -c .$$

Since $1/2 \le \alpha < 1$ and $0 < 1-\alpha \le 1/2$, we have that $\lg \alpha < 0$ and $\lg(1-\alpha) < 0$. Thus, $\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha) < 0$, so that when we multiply both sides of the inequality by this factor, we need to reverse the inequality:

$$d \ge \frac{-c}{\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)}$$

or

$$d \ge \frac{c}{-\alpha \lg \alpha + -(1-\alpha)\lg(1-\alpha)} .$$

The fraction on the right-hand side is a positive constant, and so it suffices to pick any value of d that is greater than or equal to this fraction.

To prove the lower bound, we need to show that $T(n) \ge dn \lg n$ for a suitable constant $d > 0$. We can use the same proof as for the upper bound, substituting $\ge$ for $\le$, and we get the requirement that

$$0 < d \le \frac{c}{-\alpha \lg \alpha - (1-\alpha)\lg(1-\alpha)} .$$

Therefore, $T(n) = \Theta(n \lg n)$.

Selected Solutions for Chapter 5: Probabilistic Analysis and Randomized Algorithms

Solution to Exercise 5.2-1

Since HIRE-ASSISTANT always hires candidate 1, it hires exactly once if and only if no candidates other than candidate 1 are hired. This event occurs when candidate 1 is the best candidate of the n, which occurs with probability $1/n$.

HIRE-ASSISTANT hires n times if each candidate is better than all those who were interviewed (and hired) before. This event occurs precisely when the list of ranks given to the algorithm is $\langle 1, 2, \ldots, n \rangle$, which occurs with probability $1/n!$.

Solution to Exercise 5.2-4

Another way to think of the hat-check problem is that we want to determine the expected number of fixed points in a random permutation. (A ***fixed point*** of a permutation π is a value i for which $\pi(i) = i$.) We could enumerate all $n!$ permutations, count the total number of fixed points, and divide by $n!$ to determine the average number of fixed points per permutation. This would be a painstaking process, and the answer would turn out to be 1. We can use indicator random variables, however, to arrive at the same answer much more easily.

Define a random variable X that equals the number of customers that get back their own hat, so that we want to compute $\mathrm{E}[X]$.

For $i = 1, 2, \ldots, n$, define the indicator random variable

$$X_i = \mathrm{I}\{\text{customer } i \text{ gets back his own hat}\} .$$

Then $X = X_1 + X_2 + \cdots + X_n$.

Since the ordering of hats is random, each customer has a probability of $1/n$ of getting back his or her own hat. In other words, $\Pr\{X_i = 1\} = 1/n$, which, by Lemma 5.1, implies that $\mathrm{E}[X_i] = 1/n$.

Thus,

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] \\
&= \sum_{i=1}^{n} \mathrm{E}[X_i] \qquad \text{(linearity of expectation)} \\
&= \sum_{i=1}^{n} 1/n \\
&= 1 ,
\end{aligned}$$

and so we expect that exactly 1 customer gets back his own hat.

Note that this is a situation in which the indicator random variables are *not* independent. For example, if $n = 2$ and $X_1 = 1$, then X_2 must also equal 1. Conversely, if $n = 2$ and $X_1 = 0$, then X_2 must also equal 0. Despite the dependence, $\Pr\{X_i = 1\} = 1/n$ for all i, and linearity of expectation holds. Thus, we can use the technique of indicator random variables even in the presence of dependence.

Solution to Exercise 5.2-5

Let X_{ij} be an indicator random variable for the event where the pair $A[i], A[j]$ for $i < j$ is inverted, i.e., $A[i] > A[j]$. More precisely, we define $X_{ij} = \mathrm{I}\{A[i] > A[j]\}$ for $1 \le i < j \le n$. We have $\Pr\{X_{ij} = 1\} = 1/2$, because given two distinct random numbers, the probability that the first is bigger than the second is $1/2$. By Lemma 5.1, $\mathrm{E}[X_{ij}] = 1/2$.

Let X be the the random variable denoting the total number of inverted pairs in the array, so that

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} .$$

We want the expected number of inverted pairs, so we take the expectation of both sides of the above equation to obtain

$$\mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] .$$

We use linearity of expectation to get

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}[X_{ij}] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1/2
\end{aligned}$$

$$\begin{aligned} &= \binom{n}{2}\frac{1}{2} \\ &= \frac{n(n-1)}{2}\cdot\frac{1}{2} \\ &= \frac{n(n-1)}{4}\ . \end{aligned}$$

Thus the expected number of inverted pairs is $n(n-1)/4$.

Solution to Exercise 5.3-2

Although PERMUTE-WITHOUT-IDENTITY will not produce the identity permutation, there are other permutations that it fails to produce. For example, consider its operation when $n = 3$, when it should be able to produce the $n! - 1 = 5$ non-identity permutations. The **for** loop iterates for $i = 1$ and $i = 2$. When $i = 1$, the call to RANDOM returns one of two possible values (either 2 or 3), and when $i = 2$, the call to RANDOM returns just one value (3). Thus, PERMUTE-WITHOUT-IDENTITY can produce only $2 \cdot 1 = 2$ possible permutations, rather than the 5 that are required.

Solution to Exercise 5.3-4

PERMUTE-BY-CYCLIC chooses *offset* as a random integer in the range $1 \leq \textit{offset} \leq n$, and then it performs a cyclic rotation of the array. That is, $B[((i + \textit{offset} - 1) \bmod n) + 1] = A[i]$ for $i = 1, 2, \ldots, n$. (The subtraction and addition of 1 in the index calculation is due to the 1-origin indexing. If we had used 0-origin indexing instead, the index calculation would have simplied to $B[(i + \textit{offset}) \bmod n] = A[i]$ for $i = 0, 1, \ldots, n-1$.)

Thus, once *offset* is determined, so is the entire permutation. Since each value of *offset* occurs with probability $1/n$, each element $A[i]$ has a probability of ending up in position $B[j]$ with probability $1/n$.

This procedure does not produce a uniform random permutation, however, since it can produce only n different permutations. Thus, n permutations occur with probability $1/n$, and the remaining $n! - n$ permutations occur with probability 0.

Selected Solutions for Chapter 6: Heapsort

Solution to Exercise 6.1-1

Since a heap is an almost-complete binary tree (complete at all levels except possibly the lowest), it has at most $2^{h+1} - 1$ elements (if it is complete) and at least $2^h - 1 + 1 = 2^h$ elements (if the lowest level has just 1 element and the other levels are complete).

Solution to Exercise 6.1-2

Given an n-element heap of height h, we know from Exercise 6.1-1 that

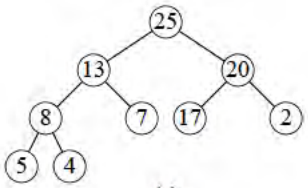
$$2^h \le n \le 2^{h+1} - 1 < 2^{h+1} .$$

Thus, $h \le \lg n < h + 1$. Since h is an integer, $h = \lfloor \lg n \rfloor$ (by definition of $\lfloor\ \rfloor$).

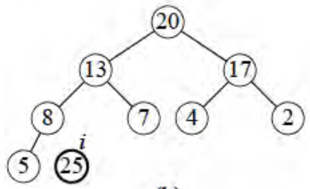
Solution to Exercise 6.2-6

If you put a value at the root that is less than every value in the left and right subtrees, then MAX-HEAPIFY will be called recursively until a leaf is reached. To make the recursive calls traverse the longest path to a leaf, choose values that make MAX-HEAPIFY always recurse on the left child. It follows the left branch when the left child is greater than or equal to the right child, so putting 0 at the root and 1 at all the other nodes, for example, will accomplish that. With such values, MAX-HEAPIFY will be called h times (where h is the heap height, which is the number of edges in the longest path from the root to a leaf), so its running time will be $\Theta(h)$ (since each call does $\Theta(1)$ work), which is $\Theta(\lg n)$. Since we have a case in which MAX-HEAPIFY's running time is $\Theta(\lg n)$, its worst-case running time is $\Omega(\lg n)$.

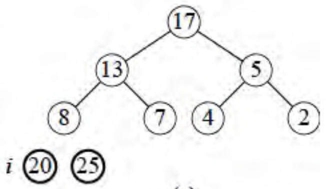
Solution to Exercise 6.4-1



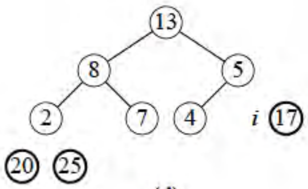
(a)



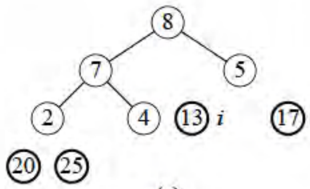
(b)



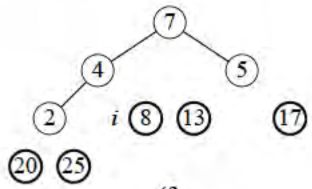
(c)



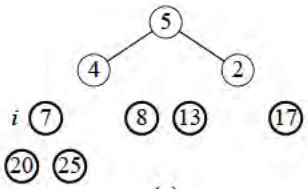
(d)



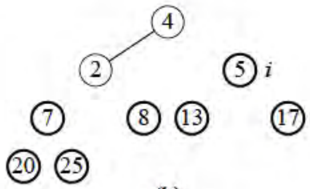
(e)



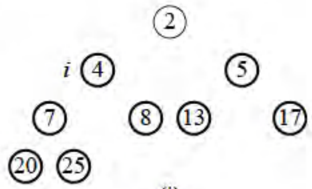
(f)



(g)



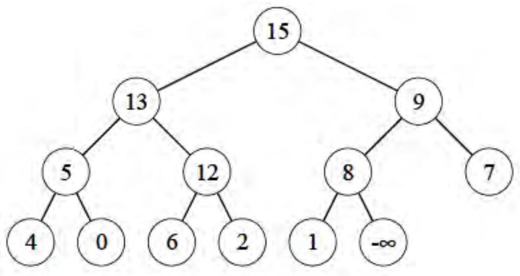
(h)



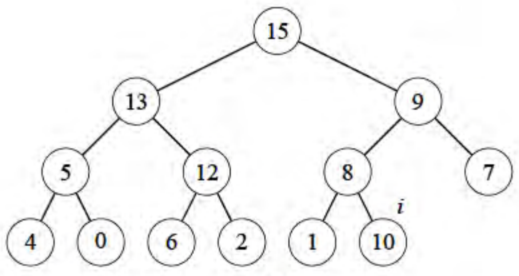
(i)

A	2	4	5	7	8	13	17	20	25

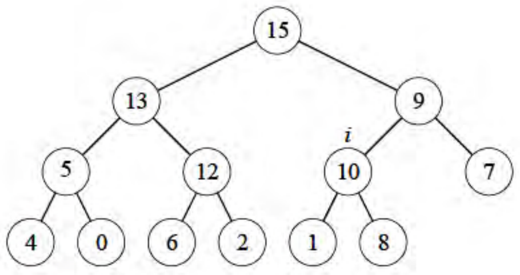
Solution to Exercise 6.5-2



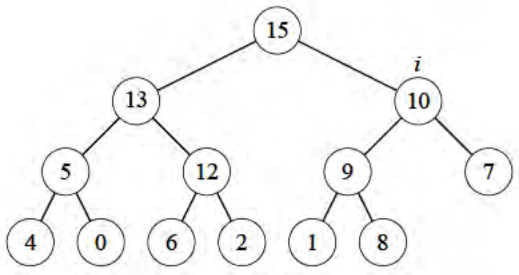
(a)



(b)



(c)



(d)

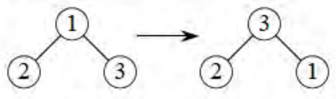
Solution to Problem 6-1

a. The procedures BUILD-MAX-HEAP and BUILD-MAX-HEAP′ do not always create the same heap when run on the same input array. Consider the following counterexample.

Input array A:

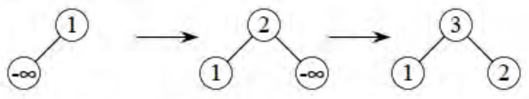
A	1	2	3

BUILD-MAX-HEAP(A):



A	3	2	1

BUILD-MAX-HEAP′(A):



A	3	1	2

b. An upper bound of $O(n \lg n)$ time follows immediately from there being $n - 1$ calls to MAX-HEAP-INSERT, each taking $O(\lg n)$ time. For a lower bound of $\Omega(n \lg n)$, consider the case in which the input array is given in strictly increasing order. Each call to MAX-HEAP-INSERT causes HEAP-INCREASE-KEY to go all the way up to the root. Since the depth of node i is $\lfloor \lg i \rfloor$, the total time is

$$\begin{aligned}
\sum_{i=1}^{n} \Theta(\lfloor \lg i \rfloor) &\geq \sum_{i=\lceil n/2 \rceil}^{n} \Theta(\lfloor \lg \lceil n/2 \rceil \rfloor) \\
&\geq \sum_{i=\lceil n/2 \rceil}^{n} \Theta(\lfloor \lg(n/2) \rfloor) \\
&= \sum_{i=\lceil n/2 \rceil}^{n} \Theta(\lfloor \lg n - 1 \rfloor) \\
&\geq n/2 \cdot \Theta(\lg n) \\
&= \Omega(n \lg n) .
\end{aligned}$$

In the worst case, therefore, BUILD-MAX-HEAP$'$ requires $\Theta(n \lg n)$ time to build an n-element heap.

Selected Solutions for Chapter 7: Quicksort

Solution to Exercise 7.2-3

PARTITION does a "worst-case partitioning" when the elements are in decreasing order. It reduces the size of the subarray under consideration by only 1 at each step, which we've seen has running time $\Theta(n^2)$.

In particular, PARTITION, given a subarray $A[p \mathinner{.\,.} r]$ of distinct elements in decreasing order, produces an empty partition in $A[p \mathinner{.\,.} q-1]$, puts the pivot (originally in $A[r]$) into $A[p]$, and produces a partition $A[p+1 \mathinner{.\,.} r]$ with only one fewer element than $A[p \mathinner{.\,.} r]$. The recurrence for QUICKSORT becomes $T(n) = T(n-1) + \Theta(n)$, which has the solution $T(n) = \Theta(n^2)$.

Solution to Exercise 7.2-5

The minimum depth follows a path that always takes the smaller part of the partition—i.e., that multiplies the number of elements by α. One iteration reduces the number of elements from n to αn, and i iterations reduces the number of elements to $\alpha^i n$. At a leaf, there is just one remaining element, and so at a minimum-depth leaf of depth m, we have $\alpha^m n = 1$. Thus, $\alpha^m = 1/n$. Taking logs, we get $m \lg \alpha = -\lg n$, or $m = -\lg n / \lg \alpha$.

Similarly, maximum depth corresponds to always taking the larger part of the partition, i.e., keeping a fraction $1-\alpha$ of the elements each time. The maximum depth M is reached when there is one element left, that is, when $(1-\alpha)^M n = 1$. Thus, $M = -\lg n / \lg(1-\alpha)$.

All these equations are approximate because we are ignoring floors and ceilings.

Selected Solutions for Chapter 8: Sorting in Linear Time

Solution to Exercise 8.1-3

If the sort runs in linear time for m input permutations, then the height h of the portion of the decision tree consisting of the m corresponding leaves and their ancestors is linear.

Use the same argument as in the proof of Theorem 8.1 to show that this is impossible for $m = n!/2$, $n!/n$, or $n!/2^n$.

We have $2^h \geq m$, which gives us $h \geq \lg m$. For all the possible m's given here, $\lg m = \Omega(n \lg n)$, hence $h = \Omega(n \lg n)$.

In particular,

$$\begin{aligned}
\lg \frac{n!}{2} &= \lg n! - 1 \geq n \lg n - n \lg e - 1 , \\
\lg \frac{n!}{n} &= \lg n! - \lg n \geq n \lg n - n \lg e - \lg n , \\
\lg \frac{n!}{2^n} &= \lg n! - n \geq n \lg n - n \lg e - n .
\end{aligned}$$

Solution to Exercise 8.2-3

The following solution also answers Exercise 8.2-2.

Notice that the correctness argument in the text does not depend on the order in which A is processed. The algorithm is correct no matter what order is used!

But the modified algorithm is not stable. As before, in the final **for** loop an element equal to one taken from A earlier is placed before the earlier one (i.e., at a lower index position) in the output arrray B. The original algorithm was stable because an element taken from A later started out with a lower index than one taken earlier. But in the modified algorithm, an element taken from A later started out with a higher index than one taken earlier.

In particular, the algorithm still places the elements with value k in positions $C[k-1]+1$ through $C[k]$, but in the reverse order of their appearance in A.

Solution to Exercise 8.3-3

Basis: If $d = 1$, there's only one digit, so sorting on that digit sorts the array.

Inductive step: Assuming that radix sort works for $d - 1$ digits, we'll show that it works for d digits.

Radix sort sorts separately on each digit, starting from digit 1. Thus, radix sort of d digits, which sorts on digits $1, \ldots, d$ is equivalent to radix sort of the low-order $d - 1$ digits followed by a sort on digit d. By our induction hypothesis, the sort of the low-order $d - 1$ digits works, so just before the sort on digit d, the elements are in order according to their low-order $d - 1$ digits.

The sort on digit d will order the elements by their dth digit. Consider two elements, a and b, with dth digits a_d and b_d respectively.

- If $a_d < b_d$, the sort will put a before b, which is correct, since $a < b$ regardless of the low-order digits.
- If $a_d > b_d$, the sort will put a after b, which is correct, since $a > b$ regardless of the low-order digits.
- If $a_d = b_d$, the sort will leave a and b in the same order they were in, because it is stable. But that order is already correct, since the correct order of a and b is determined by the low-order $d - 1$ digits when their dth digits are equal, and the elements are already sorted by their low-order $d - 1$ digits.

If the intermediate sort were not stable, it might rearrange elements whose dth digits were equal—elements that *were* in the right order after the sort on their lower-order digits.

Solution to Exercise 8.3-4

Treat the numbers as 3-digit numbers in radix n. Each digit ranges from 0 to $n - 1$. Sort these 3-digit numbers with radix sort.

There are 3 calls to counting sort, each taking $\Theta(n + n) = \Theta(n)$ time, so that the total time is $\Theta(n)$.

Solution to Problem 8-1

a. For a comparison algorithm A to sort, no two input permutations can reach the same leaf of the decision tree, so there must be at least $n!$ leaves reached in T_A, one for each possible input permutation. Since A is a deterministic algorithm, it must always reach the same leaf when given a particular permutation as input, so at most $n!$ leaves are reached (one for each permutation). Therefore exactly $n!$ leaves are reached, one for each input permutation.

These $n!$ leaves will each have probability $1/n!$, since each of the $n!$ possible permutations is the input with the probability $1/n!$. Any remaining leaves will have probability 0, since they are not reached for any input.

Without loss of generality, we can assume for the rest of this problem that paths leading only to 0-probability leaves aren't in the tree, since they cannot affect the running time of the sort. That is, we can assume that T_A consists of only the $n!$ leaves labeled $1/n!$ and their ancestors.

b. If $k > 1$, then the root of T is not a leaf. This implies that all of T's leaves are leaves in LT and RT. Since every leaf at depth h in LT or RT has depth $h+1$ in T, $D(T)$ must be the sum of $D(LT)$, $D(RT)$, and k, the total number of leaves. To prove this last assertion, let $d_T(x) =$ depth of node x in tree T. Then,

$$\begin{aligned}
D(T) &= \sum_{x \in \text{leaves}(T)} d_T(x) \\
&= \sum_{x \in \text{leaves}(LT)} d_T(x) + \sum_{x \in \text{leaves}(RT)} d_T(x) \\
&= \sum_{x \in \text{leaves}(LT)} (d_{LT}(x) + 1) + \sum_{x \in \text{leaves}(RT)} (d_{RT}(x) + 1) \\
&= \sum_{x \in \text{leaves}(LT)} d_{LT}(x) + \sum_{x \in \text{leaves}(RT)} d_{RT}(x) + \sum_{x \in \text{leaves}(T)} 1 \\
&= D(LT) + D(RT) + k\ .
\end{aligned}$$

c. To show that $d(k) = \min_{1 \le i \le k-1} \{d(i) + d(k-i) + k\}$ we will show separately that

$$d(k) \le \min_{1 \le i \le k-1} \{d(i) + d(k-i) + k\}$$

and

$$d(k) \ge \min_{1 \le i \le k-1} \{d(i) + d(k-i) + k\}\ .$$

- To show that $d(k) \le \min_{1 \le i \le k-1} \{d(i) + d(k-i) + k\}$, we need only show that $d(k) \le d(i) + d(k-i) + k$, for $i = 1, 2, \ldots, k-1$. For any i from 1 to $k-1$ we can find trees RT with i leaves and LT with $k-i$ leaves such that $D(RT) = d(i)$ and $D(LT) = d(k-i)$. Construct T such that RT and LT are the right and left subtrees of T's root respectively. Then

$$\begin{aligned}
d(k) &\le D(T) && \text{(by definition of } d \text{ as min } D(T) \text{ value)} \\
&= D(RT) + D(LT) + k && \text{(by part (b))} \\
&= d(i) + d(k-i) + k && \text{(by choice of } RT \text{ and } LT)\ .
\end{aligned}$$

- To show that $d(k) \ge \min_{1 \le i \le k-1} \{d(i) + d(k-i) + k\}$, we need only show that $d(k) \ge d(i) + d(k-i) + k$, for some i in $\{1, 2, \ldots, k-1\}$. Take the tree T with k leaves such that $D(T) = d(k)$, let RT and LT be T's right and left subtree, respecitvely, and let i be the number of leaves in RT. Then $k-i$ is the number of leaves in LT and

$$\begin{aligned}
d(k) &= D(T) && \text{(by choice of } T) \\
&= D(RT) + D(LT) + k && \text{(by part (b))} \\
&\ge d(i) + d(k-i) + k && \text{(by defintion of } d \text{ as min } D(T) \text{ value)}\ .
\end{aligned}$$

Neither i nor $k-i$ can be 0 (and hence $1 \le i \le k-1$), since if one of these were 0, either RT or LT would contain all k leaves of T, and that k-leaf subtree would have a D equal to $D(T) - k$ (by part (b)), contradicting the choice of T as the k-leaf tree with the minimum D.

d. Let $f_k(i) = i \lg i + (k-i)\lg(k-i)$. To find the value of i that minimizes f_k, find the i for which the derivative of f_k with respect to i is 0:

$$\begin{aligned} f_k'(i) &= \frac{d}{di}\left(\frac{i \ln i + (k-i)\ln(k-i)}{\ln 2}\right) \\ &= \frac{\ln i + 1 - \ln(k-i) - 1}{\ln 2} \\ &= \frac{\ln i - \ln(k-i)}{\ln 2} \end{aligned}$$

is 0 at $i = k/2$. To verify this is indeed a minimum (not a maximum), check that the second derivative of f_k is positive at $i = k/2$:

$$\begin{aligned} f_k''(i) &= \frac{d}{di}\left(\frac{\ln i - \ln(k-i)}{\ln 2}\right) \\ &= \frac{1}{\ln 2}\left(\frac{1}{i} + \frac{1}{k-i}\right) . \end{aligned}$$

$$\begin{aligned} f_k''(k/2) &= \frac{1}{\ln 2}\left(\frac{2}{k} + \frac{2}{k}\right) \\ &= \frac{1}{\ln 2} \cdot \frac{4}{k} \\ &> 0 \qquad \text{since } k > 1 . \end{aligned}$$

Now we use substitution to prove $d(k) = \Omega(k \lg k)$. The base case of the induction is satisfied because $d(1) \ge 0 = c \cdot 1 \cdot \lg 1$ for any constant c. For the inductive step we assume that $d(i) \ge ci \lg i$ for $1 \le i \le k-1$, where c is some constant to be determined.

$$\begin{aligned} d(k) &= \min_{1 \le i \le k-1} \{d(i) + d(k-i) + k\} \\ &\ge \min_{1 \le i \le k-1} \{c(i \lg i + (k-i)\lg(k-i)) + k\} \\ &= \min_{1 \le i \le k-1} \{c f_k(i) + k\} \\ &= c\left(\frac{k}{2}\lg\frac{k}{2}\left(k - \frac{k}{2}\right)\lg\left(k - \frac{k}{2}\right)\right) + k \\ &= ck \lg\left(\frac{k}{2}\right) + k \\ &= c(k \lg k - k) + k \\ &= ck \lg k + (k - ck) \\ &\ge ck \lg k \qquad \text{if } c \le 1 , \end{aligned}$$

and so $d(k) = \Omega(k \lg k)$.

e. Using the result of part (d) and the fact that T_A (as modified in our solution to part (a)) has $n!$ leaves, we can conclude that

$$D(T_A) \ge d(n!) = \Omega(n! \lg(n!)) .$$

$D(T_A)$ is the sum of the decision-tree path lengths for sorting all input permutations, and the path lengths are proportional to the run time. Since the $n!$ permutations have equal probability $1/n!$, the expected time to sort n random elements (1 input permutation) is the total time for all permutations divided by $n!$:

$$\frac{\Omega(n! \lg(n!))}{n!} = \Omega(\lg(n!)) = \Omega(n \lg n) \ .$$

f. We will show how to modify a randomized decision tree (algorithm) to define a deterministic decision tree (algorithm) that is at least as good as the randomized one in terms of the average number of comparisons.

At each randomized node, pick the child with the smallest subtree (the subtree with the smallest average number of comparisons on a path to a leaf). Delete all the other children of the randomized node and splice out the randomized node itself.

The deterministic algorithm corresponding to this modified tree still works, because the randomized algorithm worked no matter which path was taken from each randomized node.

The average number of comparisons for the modified algorithm is no larger than the average number for the original randomized tree, since we discarded the higher-average subtrees in each case. In particular, each time we splice out a randomized node, we leave the overall average less than or equal to what it was, because

- the same set of input permutations reaches the modified subtree as before, but those inputs are handled in less than or equal to average time than before, and
- the rest of the tree is unmodified.

The randomized algorithm thus takes at least as much time on average as the corresponding deterministic one. (We've shown that the expected running time for a deterministic comparison sort is $\Omega(n \lg n)$, hence the expected time for a randomized comparison sort is also $\Omega(n \lg n)$.)

Selected Solutions for Chapter 9: Medians and Order Statistics

Solution to Exercise 9.3-1

For groups of 7, the algorithm still works in linear time. The number of elements greater than x (and similarly, the number less than x) is at least

$$4\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{7} \right\rceil \right\rceil - 2\right) \ge \frac{2n}{7} - 8 ,$$

and the recurrence becomes

$$T(n) \le T(\lceil n/7 \rceil) + T(5n/7 + 8) + O(n) ,$$

which can be shown to be $O(n)$ by substitution, as for the groups of 5 case in the text.

For groups of 3, however, the algorithm no longer works in linear time. The number of elements greater than x, and the number of elements less than x, is at least

$$2\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{3} \right\rceil \right\rceil - 2\right) \ge \frac{n}{3} - 4 ,$$

and the recurrence becomes

$$T(n) \le T(\lceil n/3 \rceil) + T(2n/3 + 4) + O(n) ,$$

which does not have a linear solution.

We can prove that the worst-case time for groups of 3 is $\Omega(n \lg n)$. We do so by deriving a recurrence for a particular case that takes $\Omega(n \lg n)$ time.

In counting up the number of elements greater than x (and similarly, the number less than x), consider the particular case in which there are exactly $\left\lceil \frac{1}{2}\left\lceil \frac{n}{3} \right\rceil \right\rceil$ groups with medians $\ge x$ and in which the "leftover" group does contribute 2 elements greater than x. Then the number of elements greater than x is exactly $2\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{3} \right\rceil \right\rceil - 1\right) + 1$ (the -1 discounts x's group, as usual, and the $+1$ is contributed by x's group) $= 2\lceil n/6 \rceil - 1$, and the recursive step for elements $\le x$ has $n - (2\lceil n/6 \rceil - 1) \ge n - (2(n/6 + 1) - 1) = 2n/3 - 1$ elements. Observe also that the $O(n)$ term in the recurrence is really $\Theta(n)$, since the partitioning in step 4 takes $\Theta(n)$ (not just $O(n)$) time. Thus, we get the recurrence

$$T(n) \ge T(\lceil n/3 \rceil) + T(2n/3 - 1) + \Theta(n) \ge T(n/3) + T(2n/3 - 1) + \Theta(n) ,$$

from which you can show that $T(n) \ge cn \lg n$ by substitution. You can also see that $T(n)$ is nonlinear by noticing that each level of the recursion tree sums to n.

In fact, any odd group size ≥ 5 works in linear time.

Solution to Exercise 9.3-3

A modification to quicksort that allows it to run in $O(n \lg n)$ time in the worst case uses the deterministic PARTITION algorithm that was modified to take an element to partition around as an input parameter.

SELECT takes an array A, the bounds p and r of the subarray in A, and the rank i of an order statistic, and in time linear in the size of the subarray $A[p \mathinner{..} r]$ it returns the ith smallest element in $A[p \mathinner{..} r]$.

```
BEST-CASE-QUICKSORT($A, p, r$)
if $p < r$
    $i = \lfloor (r - p + 1)/2 \rfloor$
    $x = $ SELECT($A, p, r, i$)
    $q = $ PARTITION($x$)
    BEST-CASE-QUICKSORT($A, p, q - 1$)
    BEST-CASE-QUICKSORT($A, q + 1, r$)
```

For an n-element array, the largest subarray that BEST-CASE-QUICKSORT recurses on has $n/2$ elements. This situation occurs when $n = r - p + 1$ is even; then the subarray $A[q + 1 \mathinner{..} r]$ has $n/2$ elements, and the subarray $A[p \mathinner{..} q - 1]$ has $n/2 - 1$ elements.

Because BEST-CASE-QUICKSORT always recurses on subarrays that are at most half the size of the original array, the recurrence for the worst-case running time is $T(n) \le 2T(n/2) + \Theta(n) = O(n \lg n)$.

Solution to Exercise 9.3-5

We assume that are given a procedure MEDIAN that takes as parameters an array A and subarray indices p and r, and returns the value of the median element of $A[p \mathinner{..} r]$ in $O(n)$ time in the worst case.

Given MEDIAN, here is a linear-time algorithm SELECT$'$ for finding the ith smallest element in $A[p \mathinner{..} r]$. This algorithm uses the deterministic PARTITION algorithm that was modified to take an element to partition around as an input parameter.

```
SELECT′(A, p, r, i)
if p == r
    return A[p]
x = MEDIAN(A, p, r)
q = PARTITION(x)
k = q − p + 1
if i == k
    return A[q]
elseif i < k
    return SELECT′(A, p, q − 1, i)
else return SELECT′(A, q + 1, r, i − k)
```

Because x is the median of $A[p \mathinner{..} r]$, each of the subarrays $A[p \mathinner{..} q-1]$ and $A[q+1 \mathinner{..} r]$ has at most half the number of elements of $A[p \mathinner{..} r]$. The recurrence for the worst-case running time of SELECT′ is $T(n) \le T(n/2) + O(n) = O(n)$.

Solution to Problem 9-1

We assume that the numbers start out in an array.

a. Sort the numbers using merge sort or heapsort, which take $\Theta(n \lg n)$ worst-case time. (Don't use quicksort or insertion sort, which can take $\Theta(n^2)$ time.) Put the i largest elements (directly accessible in the sorted array) into the output array, taking $\Theta(i)$ time.

Total worst-case running time: $\Theta(n \lg n + i) = \Theta(n \lg n)$ (because $i \le n$).

b. Implement the priority queue as a heap. Build the heap using BUILD-HEAP, which takes $\Theta(n)$ time, then call HEAP-EXTRACT-MAX i times to get the i largest elements, in $\Theta(i \lg n)$ worst-case time, and store them in reverse order of extraction in the output array. The worst-case extraction time is $\Theta(i \lg n)$ because

- i extractions from a heap with $O(n)$ elements takes $i \cdot O(\lg n) = O(i \lg n)$ time, and
- half of the i extractions are from a heap with $\ge n/2$ elements, so those $i/2$ extractions take $(i/2)\Omega(\lg(n/2)) = \Omega(i \lg n)$ time in the worst case.

Total worst-case running time: $\Theta(n + i \lg n)$.

c. Use the SELECT algorithm of Section 9.3 to find the ith largest number in $\Theta(n)$ time. Partition around that number in $\Theta(n)$ time. Sort the i largest numbers in $\Theta(i \lg i)$ worst-case time (with merge sort or heapsort).

Total worst-case running time: $\Theta(n + i \lg i)$.

Note that method (c) is always asymptotically at least as good as the other two methods, and that method (b) is asymptotically at least as good as (a). (Comparing (c) to (b) is easy, but it is less obvious how to compare (c) and (b) to (a). (c) and (b) are asymptotically at least as good as (a) because n, $i \lg i$, and $i \lg n$ are all $O(n \lg n)$. The sum of two things that are $O(n \lg n)$ is also $O(n \lg n)$.)

Selected Solutions for Chapter 11: Hash Tables

Solution to Exercise 11.2-1

For each pair of keys k, l, where $k \neq l$, define the indicator random variable $X_{kl} = \text{I}\{h(k) = h(l)\}$. Since we assume simple uniform hashing, $\Pr\{X_{kl} = 1\} = \Pr\{h(k) = h(l)\} = 1/m$, and so $\text{E}[X_{kl}] = 1/m$.

Now define the random variable Y to be the total number of collisions, so that $Y = \sum_{k \neq l} X_{kl}$. The expected number of collisions is

$$\begin{aligned}
\text{E}[Y] &= \text{E}\left[\sum_{k \neq l} X_{kl}\right] \\
&= \sum_{k \neq l} \text{E}[X_{kl}] \qquad \text{(linearity of expectation)} \\
&= \binom{n}{2} \frac{1}{m} \\
&= \frac{n(n-1)}{2} \cdot \frac{1}{m} \\
&= \frac{n(n-1)}{2m} .
\end{aligned}$$

Solution to Exercise 11.2-4

The flag in each slot will indicate whether the slot is free.

- A free slot is in the free list, a doubly linked list of all free slots in the table. The slot thus contains two pointers.
- A used slot contains an element and a pointer (possibly NIL) to the next element that hashes to this slot. (Of course, that pointer points to another slot in the table.)

Operations

- ***Insertion:***
  - If the element hashes to a free slot, just remove the slot from the free list and store the element there (with a NIL pointer). The free list must be doubly linked in order for this deletion to run in $O(1)$ time.
  - If the element hashes to a used slot j, check whether the element x already there "belongs" there (its key also hashes to slot j).
    - If so, add the new element to the chain of elements in this slot. To do so, allocate a free slot (e.g., take the head of the free list) for the new element and put this new slot at the head of the list pointed to by the hashed-to slot (j).
    - If not, E is part of another slot's chain. Move it to a new slot by allocating one from the free list, copying the old slot's (j's) contents (element x and pointer) to the new slot, and updating the pointer in the slot that pointed to j to point to the new slot. Then insert the new element in the now-empty slot as usual.
      To update the pointer to j, it is necessary to find it by searching the chain of elements starting in the slot x hashes to.
- ***Deletion:*** Let j be the slot the element x to be deleted hashes to.
  - If x is the only element in j (j doesn't point to any other entries), just free the slot, returning it to the head of the free list.
  - If x is in j but there's a pointer to a chain of other elements, move the first pointed-to entry to slot j and free the slot it was in.
  - If x is found by following a pointer from j, just free x's slot and splice it out of the chain (i.e., update the slot that pointed to x to point to x's successor).
- ***Searching:*** Check the slot the key hashes to, and if that is not the desired element, follow the chain of pointers from the slot.

  All the operations take expected $O(1)$ times for the same reason they do with the version in the book: The expected time to search the chains is $O(1 + \alpha)$ regardless of where the chains are stored, and the fact that all the elements are stored in the table means that $\alpha \le 1$. If the free list were singly linked, then operations that involved removing an arbitrary slot from the free list would not run in $O(1)$ time.

Solution to Problem 11-2

a. A particular key is hashed to a particular slot with probability $1/n$. Suppose we select a specific set of k keys. The probability that these k keys are inserted into the slot in question and that all other keys are inserted elsewhere is

$$\left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} .$$

Since there are $\binom{n}{k}$ ways to choose our k keys, we get

$$Q_k = \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \binom{n}{k} .$$

b. For $i = 1, 2, \ldots, n$, let X_i be a random variable denoting the number of keys that hash to slot i, and let A_i be the event that $X_i = k$, i.e., that exactly k keys hash to slot i. From part (a), we have $\Pr\{A\} = Q_k$. Then,

$$\begin{aligned} P_k &= \Pr\{M = k\} \\ &= \Pr\left\{\left(\max_{1 \le i \le n} X_i\right) = k\right\} \\ &= \Pr\{\text{there exists } i \text{ such that } X_i = k \text{ and that } X_i \le k \text{ for } i = 1, 2, \ldots, n\} \\ &\le \Pr\{\text{there exists } i \text{ such that } X_i = k\} \\ &= \Pr\{A_1 \cup A_2 \cup \cdots \cup A_n\} \\ &\le \Pr\{A_1\} + \Pr\{A_2\} + \cdots + \Pr\{A_n\} \qquad \text{(by inequality (C.19))} \\ &= nQ_k . \end{aligned}$$

c. We start by showing two facts. First, $1 - 1/n < 1$, which implies $(1 - 1/n)^{n-k} < 1$. Second, $n!/(n-k)! = n \cdot (n-1) \cdot (n-2) \cdots (n-k+1) < n^k$. Using these facts, along with the simplification $k! > (k/e)^k$ of equation (3.18), we have

$$\begin{aligned} Q_k &= \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \frac{n!}{k!(n-k)!} \\ &< \frac{n!}{n^k k!(n-k)!} && ((1 - 1/n)^{n-k} < 1) \\ &< \frac{1}{k!} && (n!/(n-k)! < n^k) \\ &< \frac{e^k}{k^k} && (k! > (k/e)^k) . \end{aligned}$$

d. Notice that when $n = 2$, $\lg\lg n = 0$, so to be precise, we need to assume that $n \ge 3$.

In part (c), we showed that $Q_k < e^k/k^k$ for any k; in particular, this inequality holds for k_0. Thus, it suffices to show that $e^{k_0}/{k_0}^{k_0} < 1/n^3$ or, equivalently, that $n^3 < {k_0}^{k_0}/e^{k_0}$.

Taking logarithms of both sides gives an equivalent condition:

$$\begin{aligned} 3 \lg n &< k_0(\lg k_0 - \lg e) \\ &= \frac{c \lg n}{\lg\lg n}(\lg c + \lg\lg n - \lg\lg\lg n - \lg e) . \end{aligned}$$

Dividing both sides by $\lg n$ gives the condition

$$\begin{aligned} 3 &< \frac{c}{\lg\lg n}(\lg c + \lg\lg n - \lg\lg\lg n - \lg e) \\ &= c\left(1 + \frac{\lg c - \lg e}{\lg\lg n} - \frac{\lg\lg\lg n}{\lg\lg n}\right) . \end{aligned}$$

Let x be the last expression in parentheses:

$$x = \left(1 + \frac{\lg c - \lg e}{\lg\lg n} - \frac{\lg\lg\lg n}{\lg\lg n}\right) .$$

We need to show that there exists a constant $c > 1$ such that $3 < cx$.

Noting that $\lim_{n\to\infty} x = 1$, we see that there exists n_0 such that $x \geq 1/2$ for all $n \geq n_0$. Thus, any constant $c > 6$ works for $n \geq n_0$.

We handle smaller values of n—in particular, $3 \leq n < n_0$—as follows. Since n is constrained to be an integer, there are a finite number of n in the range $3 \leq n < n_0$. We can evaluate the expression x for each such value of n and determine a value of c for which $3 < cx$ for all values of n. The final value of c that we use is the larger of

- 6, which works for all $n \geq n_0$, and
- $\max_{3\leq n<n_0}\{c : 3 < cx\}$, i.e., the largest value of c that we chose for the range $3 \leq n < n_0$.

Thus, we have shown that $Q_{k_0} < 1/n^3$, as desired.

To see that $P_k < 1/n^2$ for $k \geq k_0$, we observe that by part (b), $P_k \leq nQ_k$ for all k. Choosing $k = k_0$ gives $P_{k_0} \leq nQ_{k_0} < n \cdot (1/n^3) = 1/n^2$. For $k > k_0$, we will show that we can pick the constant c such that $Q_k < 1/n^3$ for all $k \geq k_0$, and thus conclude that $P_k < 1/n^2$ for all $k \geq k_0$.

To pick c as required, we let c be large enough that $k_0 > 3 > e$. Then $e/k < 1$ for all $k \geq k_0$, and so e^k/k^k decreases as k increases. Thus,

$$\begin{aligned} Q_k &< e^k/k^k \\ &\leq e^{k_0}/k^{k_0} \\ &< 1/n^3 \end{aligned}$$

for $k \geq k_0$.

e. The expectation of M is

$$\begin{aligned} \mathrm{E}[M] &= \sum_{k=0}^{n} k \cdot \Pr\{M = k\} \\ &= \sum_{k=0}^{k_0} k \cdot \Pr\{M = k\} + \sum_{k=k_0+1}^{n} k \cdot \Pr\{M = k\} \\ &\leq \sum_{k=0}^{k_0} k_0 \cdot \Pr\{M = k\} + \sum_{k=k_0+1}^{n} n \cdot \Pr\{M = k\} \\ &\leq k_0 \sum_{k=0}^{k_0} \Pr\{M = k\} + n \sum_{k=k_0+1}^{n} \Pr\{M = k\} \\ &= k_0 \cdot \Pr\{M \leq k_0\} + n \cdot \Pr\{M > k_0\} , \end{aligned}$$

which is what we needed to show, since $k_0 = c \lg n / \lg\lg n$.

To show that $\mathrm{E}[M] = O(\lg n / \lg\lg n)$, note that $\Pr\{M \leq k_0\} \leq 1$ and

$$
\begin{aligned}
\Pr\{M > k_0\} &= \sum_{k=k_0+1}^{n} \Pr\{M = k\} \\
&= \sum_{k=k_0+1}^{n} P_k \\
&< \sum_{k=k_0+1}^{n} 1/n^2 \qquad \text{(by part (d))} \\
&< n \cdot (1/n^2) \\
&= 1/n \; .
\end{aligned}
$$

We conclude that

$$
\begin{aligned}
\mathrm{E}[M] &\le k_0 \cdot 1 + n \cdot (1/n) \\
&= k_0 + 1 \\
&= O(\lg n / \lg\lg n) \; .
\end{aligned}
$$

Selected Solutions for Chapter 12: Binary Search Trees

Solution to Exercise 12.1-2

In a heap, a node's key is $\geq$ both of its children's keys. In a binary search tree, a node's key is $\geq$ its left child's key, but $\leq$ its right child's key.

The heap property, unlike the binary-searth-tree property, doesn't help print the nodes in sorted order because it doesn't tell which subtree of a node contains the element to print before that node. In a heap, the largest element smaller than the node could be in either subtree.

Note that if the heap property could be used to print the keys in sorted order in $O(n)$ time, we would have an $O(n)$-time algorithm for sorting, because building the heap takes only $O(n)$ time. But we know (Chapter 8) that a comparison sort must take $\Omega(n \lg n)$ time.

Solution to Exercise 12.2-7

Note that a call to TREE-MINIMUM followed by $n-1$ calls to TREE-SUCCESSOR performs exactly the same inorder walk of the tree as does the procedure INORDER-TREE-WALK. INORDER-TREE-WALK prints the TREE-MINIMUM first, and by definition, the TREE-SUCCESSOR of a node is the next node in the sorted order determined by an inorder tree walk.

This algorithm runs in $\Theta(n)$ time because:

- It requires $\Omega(n)$ time to do the n procedure calls.
- It traverses each of the $n-1$ tree edges at most twice, which takes $O(n)$ time.

To see that each edge is traversed at most twice (once going down the tree and once going up), consider the edge between any node u and either of its children, node v. By starting at the root, we must traverse (u, v) downward from u to v, before traversing it upward from v to u. The only time the tree is traversed downward is in code of TREE-MINIMUM, and the only time the tree is traversed upward is in code of TREE-SUCCESSOR when we look for the successor of a node that has no right subtree.

Suppose that v is u's left child.

- Before printing u, we must print all the nodes in its left subtree, which is rooted at v, guaranteeing the downward traversal of edge (u, v).
- After all nodes in u's left subtree are printed, u must be printed next. Procedure TREE-SUCCESSOR traverses an upward path to u from the maximum element (which has no right subtree) in the subtree rooted at v. This path clearly includes edge (u, v), and since all nodes in u's left subtree are printed, edge (u, v) is never traversed again.

Now suppose that v is u's right child.

- After u is printed, TREE-SUCCESSOR(u) is called. To get to the minimum element in u's right subtree (whose root is v), the edge (u, v) must be traversed downward.
- After all values in u's right subtree are printed, TREE-SUCCESSOR is called on the maximum element (again, which has no right subtree) in the subtree rooted at v. TREE-SUCCESSOR traverses a path up the tree to an element after u, since u was already printed. Edge (u, v) must be traversed upward on this path, and since all nodes in u's right subtree have been printed, edge (u, v) is never traversed again.

Hence, no edge is traversed twice in the same direction.

Therefore, this algorithm runs in $\Theta(n)$ time.

Solution to Exercise 12.3-3

Here's the algorithm:

```
TREE-SORT(A)
let T be an empty binary search tree
for i = 1 to n
    TREE-INSERT(T, A[i])
INORDER-TREE-WALK(T.root)
```

Worst case: $\Theta(n^2)$—occurs when a linear chain of nodes results from the repeated TREE-INSERT operations.

Best case: $\Theta(n \lg n)$—occurs when a binary tree of height $\Theta(\lg n)$ results from the repeated TREE-INSERT operations.

Solution to Problem 12-2

To sort the strings of S, we first insert them into a radix tree, and then use a preorder tree walk to extract them in lexicographically sorted order. The tree walk outputs strings only for nodes that indicate the existence of a string (i.e., those that are lightly shaded in Figure 12.5 of the text).

Correctness

The preorder ordering is the correct order because:

- Any node's string is a prefix of all its descendants' strings and hence belongs before them in the sorted order (rule 2).
- A node's left descendants belong before its right descendants because the corresponding strings are identical up to that parent node, and in the next position the left subtree's strings have 0 whereas the right subtree's strings have 1 (rule 1).

Time

$\Theta(n)$.

- Insertion takes $\Theta(n)$ time, since the insertion of each string takes time proportional to its length (traversing a path through the tree whose length is the length of the string), and the sum of all the string lengths is n.
- The preorder tree walk takes $O(n)$ time. It is just like INORDER-TREE-WALK (it prints the current node and calls itself recursively on the left and right subtrees), so it takes time proportional to the number of nodes in the tree. The number of nodes is at most 1 plus the sum (n) of the lengths of the binary strings in the tree, because a length-i string corresponds to a path through the root and i other nodes, but a single node may be shared among many string paths.

Selected Solutions for Chapter 13: Red-Black Trees

Solution to Exercise 13.1-4

After absorbing each red node into its black parent, the degree of each node black node is

- 2, if both children were already black,
- 3, if one child was black and one was red, or
- 4, if both children were red.

All leaves of the resulting tree have the same depth.

Solution to Exercise 13.1-5

In the longest path, at least every other node is black. In the shortest path, at most every node is black. Since the two paths contain equal numbers of black nodes, the length of the longest path is at most twice the length of the shortest path.

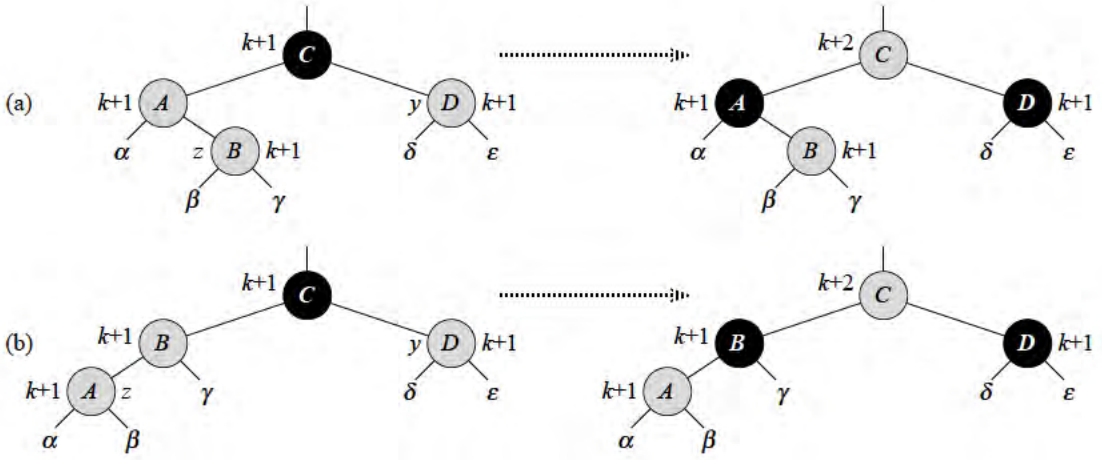
We can say this more precisely, as follows:

Since every path contains $\text{bh}(x)$ black nodes, even the shortest path from x to a descendant leaf has length at least $\text{bh}(x)$. By definition, the longest path from x to a descendant leaf has length $\text{height}(x)$. Since the longest path has $\text{bh}(x)$ black nodes and at least half the nodes on the longest path are black (by property 4), $\text{bh}(x) \geq \text{height}(x)/2$, so

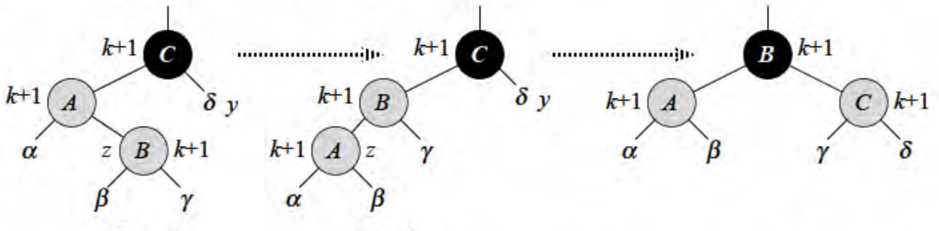
$$\text{length of longest path} = \text{height}(x) \leq 2 \cdot \text{bh}(x) \leq \text{ twice length of shortest path} .$$

Solution to Exercise 13.3-3

In Figure 13.5, nodes A, B, and D have black-height $k + 1$ in all cases, because each of their subtrees has black-height k and a black root. Node C has black-height $k + 1$ on the left (because its red children have black-height $k + 1$) and black-height $k+2$ on the right (because its black children have black-height $k+1$).



In Figure 13.6, nodes A, B, and C have black-height $k + 1$ in all cases. At left and in the middle, each of A's and B's subtrees has black-height k and a black root, while C has one such subtree and a red child with black-height $k + 1$. At the right, each of A's and C's subtrees has black-height k and a black root, while B's red children each have black-height $k + 1$.



Case 2 Case 3

Property 5 is preserved by the transformations. We have shown above that the black-height is well-defined within the subtrees pictured, so property 5 is preserved within those subtrees. Property 5 is preserved for the tree containing the subtrees pictured, because every path through these subtrees to a leaf contributes $k + 2$ black nodes.

Solution to Problem 13-1

a. When inserting key k, all nodes on the path from the root to the added node (a new leaf) must change, since the need for a new child pointer propagates up from the new node to all of its ancestors.

When deleting a node, let y be the node actually removed and z be the node given to the delete procedure.

- If z has at most one child, it will be spliced out, so that all ancestors of z will be changed. (As with insertion, the need for a new child pointer propagates up from the removed node.)
- If z has two children, then its successor y will be spliced out and moved to z's position. Therefore all ancestors of both z and y must be changed.

Because z is an ancestor of y, we can just say that all ancestors of y must be changed.

In either case, y's children (if any) are unchanged, because we have assumed that there is no parent attribute.

b. We assume that we can call two procedures:

- MAKE-NEW-NODE(k) creates a new node whose *key* attribute has value k and with *left* and *right* attributes NIL, and it returns a pointer to the new node.
- COPY-NODE(x) creates a new node whose *key*, *left*, and *right* attributes have the same values as those of node x, and it returns a pointer to the new node.

Here are two ways to write PERSISTENT-TREE-INSERT. The first is a version of TREE-INSERT, modified to create new nodes along the path to where the new node will go, and to not use parent attributes. It returns the root of the new tree.

```
PERSISTENT-TREE-INSERT($T, k$)
$z$ = MAKE-NEW-NODE($k$)
$new\text{-}root$ = COPY-NODE($T.root$)
$y$ = NIL
$x$ = $new\text{-}root$
while $x \neq$ NIL
    $y = x$
    if $z.key < x.key$
        $x$ = COPY-NODE($x.left$)
        $y.left = x$
    else $x$ = COPY-NODE($x.right$)
        $y.right = x$
if $y$ == NIL
    $new\text{-}root = z$
elseif $z.key < y.key$
    $y.left = z$
else $y.right = z$
return $new\text{-}root$
```

The second is a rather elegant recursive procedure. The initial call should have $T.root$ as its first argument. It returns the root of the new tree.

```
PERSISTENT-TREE-INSERT($r, k$)
if $r$ == NIL
    $x$ = MAKE-NEW-NODE($k$)
else $x$ = COPY-NODE($r$)
    if $k < r.key$
        $x.left$ = PERSISTENT-TREE-INSERT($r.left, k$)
    else $x.right$ = PERSISTENT-TREE-INSERT($r.right, k$)
return $x$
```

c. Like TREE-INSERT, PERSISTENT-TREE-INSERT does a constant amount of work at each node along the path from the root to the new node. Since the length of the path is at most h, it takes $O(h)$ time.

Since it allocates a new node (a constant amount of space) for each ancestor of the inserted node, it also needs $O(h)$ space.

d. If there were parent attributes, then because of the new root, every node of the tree would have to be copied when a new node is inserted. To see why, observe that the children of the root would change to point to the new root, then their children would change to point to them, and so on. Since there are n nodes, this change would cause insertion to create $\Omega(n)$ new nodes and to take $\Omega(n)$ time.

e. From parts (a) and (c), we know that insertion into a persistent binary search tree of height h, like insertion into an ordinary binary search tree, takes worst-case time $O(h)$. A red-black tree has $h = O(\lg n)$, so insertion into an ordinary red-black tree takes $O(\lg n)$ time. We need to show that if the red-black tree is persistent, insertion can still be done in $O(\lg n)$ time. To do this, we will need to show two things:

- How to still find the parent pointers we need in $O(1)$ time without using a parent attribute. We cannot use a parent attribute because a persistent tree with parent attributes uses $\Omega(n)$ time for insertion (by part (d)).
- That the additional node changes made during red-black tree operations (by rotation and recoloring) don't cause more than $O(\lg n)$ additional nodes to change.

Each parent pointer needed during insertion can be found in $O(1)$ time without having a parent attribute as follows:

To insert into a red-black tree, we call RB-INSERT, which in turn calls RB-INSERT-FIXUP. Make the same changes to RB-INSERT as we made to TREE-INSERT for persistence. Additionally, as RB-INSERT walks down the tree to find the place to insert the new node, have it build a stack of the nodes it traverses and pass this stack to RB-INSERT-FIXUP. RB-INSERT-FIXUP needs parent pointers to walk back up the same path, and at any given time it needs parent pointers only to find the parent and grandparent of the node it is working on. As RB-INSERT-FIXUP moves up the stack of parents, it needs only parent pointers that are at known locations a constant distance away in the stack. Thus, the parent information can be found in $O(1)$ time, just as if it were stored in a parent attribute.

Rotation and recoloring change nodes as follows:

- RB-INSERT-FIXUP performs at most 2 rotations, and each rotation changes the child pointers in 3 nodes (the node around which we rotate, that node's parent, and one of the children of the node around which we rotate). Thus, at most 6 nodes are directly modified by rotation during RB-INSERT-FIXUP. In a persistent tree, all ancestors of a changed node are copied, so RB-INSERT-FIXUP's rotations take $O(\lg n)$ time to change nodes due to rotation. (Actually, the changed nodes in this case share a single $O(\lg n)$-length path of ancestors.)

- RB-INSERT-FIXUP recolors some of the inserted node's ancestors, which are being changed anyway in persistent insertion, and some children of ancestors (the "uncles" referred to in the algorithm description). There are at most $O(\lg n)$ ancestors, hence at most $O(\lg n)$ color changes of uncles. Recoloring uncles doesn't cause any additional node changes due to persistence, because the ancestors of the uncles are the same nodes (ancestors of the inserted node) that are being changed anyway due to persistence. Thus, recoloring does not affect the $O(\lg n)$ running time, even with persistence.

We could show similarly that deletion in a persistent tree also takes worst-case time $O(h)$.

- We already saw in part (a) that $O(h)$ nodes change.
- We could write a persistent RB-DELETE procedure that runs in $O(h)$ time, analogous to the changes we made for persistence in insertion. But to do so without using parent pointers we need to walk down the tree to the node to be deleted, to build up a stack of parents as discussed above for insertion. This is a little tricky if the set's keys are not distinct, because in order to find the path to the node to delete—a particular node with a given key—we have to make some changes to how we store things in the tree, so that duplicate keys can be distinguished. The easiest way is to have each key take a second part that is unique, and to use this second part as a tiebreaker when comparing keys.

Then the problem of showing that deletion needs only $O(\lg n)$ time in a persistent red-black tree is the same as for insertion.

- As for insertion, we can show that the parents needed by RB-DELETE-FIXUP can be found in $O(1)$ time (using the same technique as for insertion).
- Also, RB-DELETE-FIXUP performs at most 3 rotations, which as discussed above for insertion requires $O(\lg n)$ time to change nodes due to persistence. It also does $O(\lg n)$ color changes, which (as for insertion) take only $O(\lg n)$ time to change ancestors due to persistence, because the number of copied nodes is $O(\lg n)$.

Selected Solutions for Chapter 14: Augmenting Data Structures

Solution to Exercise 14.1-7

Let $A[1 \mathinner{..} n]$ be the array of n distinct numbers.

One way to count the inversions is to add up, for each element, the number of larger elements that precede it in the array:

$$\text{\# of inversions} = \sum_{j=1}^{n} |\mathit{Inv}(j)| \ ,$$

where $\mathit{Inv}(j) = \{i : i < j \text{ and } A[i] > A[j]\}$.

Note that $|\mathit{Inv}(j)|$ is related to $A[j]$'s rank in the subarray $A[1 \mathinner{..} j]$ because the elements in $\mathit{Inv}(j)$ are the reason that $A[j]$ is not positioned according to its rank. Let $r(j)$ be the rank of $A[j]$ in $A[1 \mathinner{..} j]$. Then $j = r(j) + |\mathit{Inv}(j)|$, so we can compute

$$|\mathit{Inv}(j)| = j - r(j)$$

by inserting $A[1], \ldots, A[n]$ into an order-statistic tree and using OS-RANK to find the rank of each $A[j]$ in the tree immediately after it is inserted into the tree. (This OS-RANK value is $r(j)$.)

Insertion and OS-RANK each take $O(\lg n)$ time, and so the total time for n elements is $O(n \lg n)$.

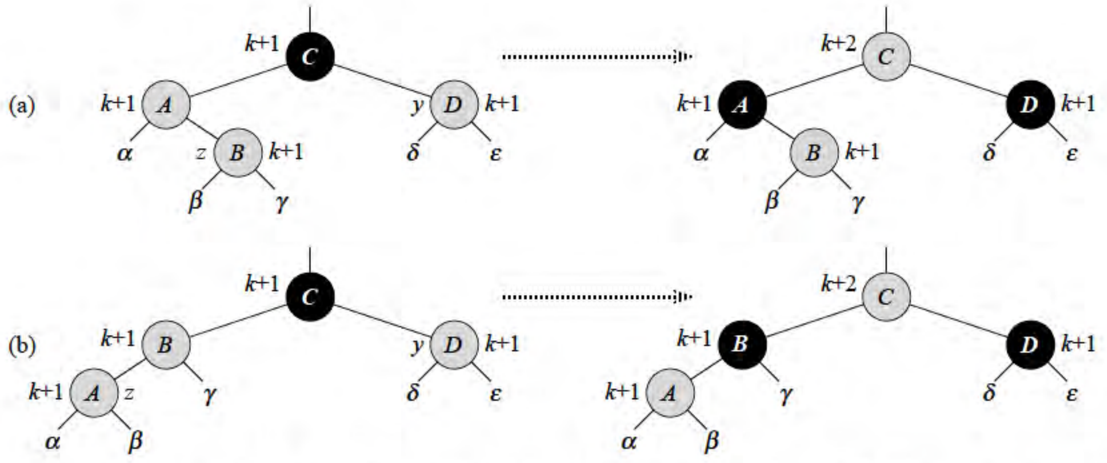
Solution to Exercise 14.2-2

Yes, we can maintain black-heights as attributes in the nodes of a red-black tree without affecting the asymptotic performance of the red-black tree operations. We appeal to Theorem 14.1, because the black-height of a node can be computed from the information at the node and its two children. Actually, the black-height can be computed from just one child's information: the black-height of a node is the black-height of a red child, or the black height of a black child plus one. The second child does not need to be checked because of property 5 of red-black trees.

Within the RB-INSERT-FIXUP and RB-DELETE-FIXUP procedures are color changes, each of which potentially cause $O(\lg n)$ black-height changes. Let us show that the color changes of the fixup procedures cause only local black-height changes and thus are constant-time operations. Assume that the black-height of each node x is kept in the attribute $x.bh$.

For RB-INSERT-FIXUP, there are 3 cases to examine.

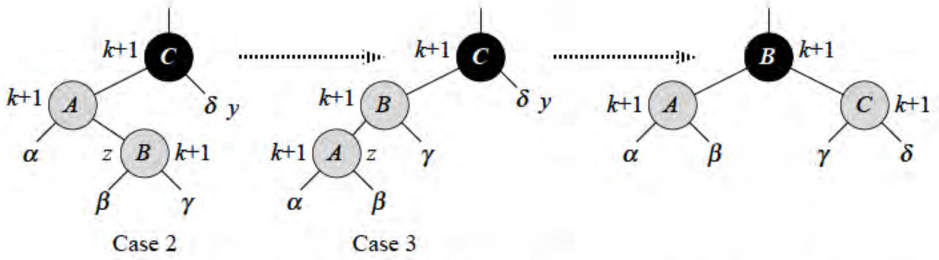
Case 1: z's uncle is red.



- Before color changes, suppose that all subtrees $\alpha, \beta, \gamma, \delta, \epsilon$ have the same black-height k with a black root, so that nodes A, B, C, and D have black-heights of $k + 1$.
- After color changes, the only node whose black-height changed is node C. To fix that, add $z.p.p.bh = z.p.p.bh + 1$ after line 7 in RB-INSERT-FIXUP.
- Since the number of black nodes between $z.p.p$ and z remains the same, nodes above $z.p.p$ are not affected by the color change.

Case 2: z's uncle y is black, and z is a right child.

Case 3: z''s uncle y is black, and z is a left child.

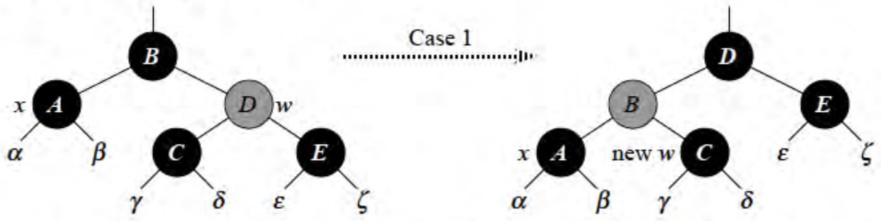


- With subtrees $\alpha, \beta, \gamma, \delta, \epsilon$ of black-height k, we see that even with color changes and rotations, the black-heights of nodes A, B, and C remain the same ($k + 1$).

Thus, RB-INSERT-FIXUP maintains its original $O(\lg n)$ time.

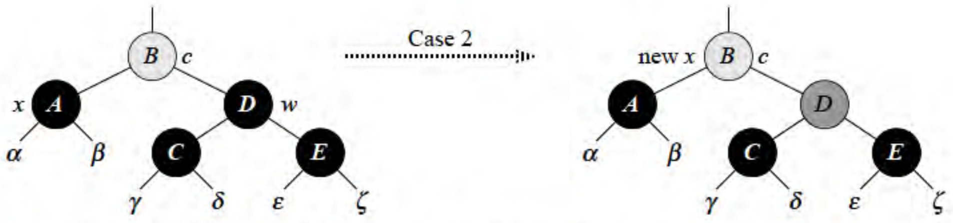
For RB-DELETE-FIXUP, there are 4 cases to examine.

Case 1: x's sibling w is red.



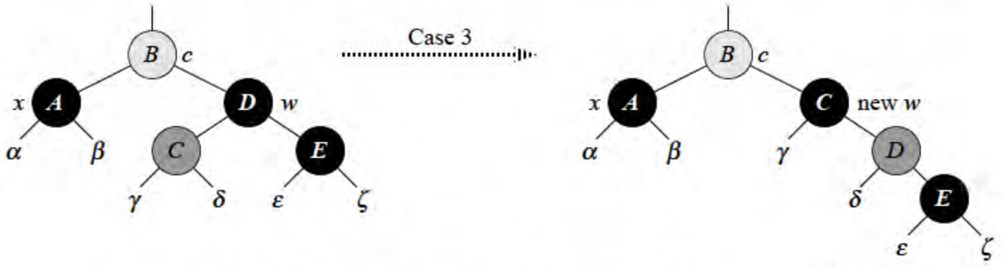
- Even though case 1 changes colors of nodes and does a rotation, black-heights are not changed.
- Case 1 changes the structure of the tree, but waits for cases 2, 3, and 4 to deal with the "extra black" on x.

Case 2: x's sibling w is black, and both of w's children are black.



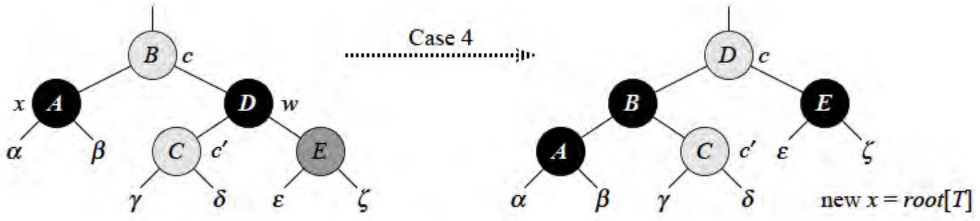
- w is colored red, and x's "extra" black is moved up to $x.p$.
- Now we can add $x.p.bh = x.bh$ after line 10 in RB-DELETE-FIXUP.
- This is a constant-time update. Then, keep looping to deal with the extra black on $x.p$.

Case 3: x's sibling w is black, w's left child is red, and w's right child is black.



- Regardless of the color changes and rotation of this case, the black-heights don't change.
- Case 3 just sets up the structure of the tree, so it can fall correctly into case 4.

Case 4: x's sibling w is black, and w's right child is red.



- Nodes A, C, and E keep the same subtrees, so their black-heights don't change.
- Add these two constant-time assignments in RB-DELETE-FIXUP after line 20:

  $$x.p.bh = x.bh + 1$$
  $$x.p.p.bh = x.p.bh + 1$$

- The extra black is taken care of. Loop terminates.

Thus, RB-DELETE-FIXUP maintains its original $O(\lg n)$ time.

Therefore, we conclude that black-heights of nodes can be maintained as attributes in red-black trees without affecting the asymptotic performance of red-black tree operations.

For the second part of the question, no, we cannot maintain node depths without affecting the asymptotic performance of red-black tree operations. The depth of a node depends on the depth of its parent. When the depth of a node changes, the depths of all nodes below it in the tree must be updated. Updating the root node causes $n - 1$ other nodes to be updated, which would mean that operations on the tree that change node depths might not run in $O(n \lg n)$ time.

Solution to Exercise 14.3-7

General idea: Move a sweep line from left to right, while maintaining the set of rectangles currently intersected by the line in an interval tree. The interval tree will organize all rectangles whose x interval includes the current position of the sweep line, and it will be based on the y intervals of the rectangles, so that any overlapping y intervals in the interval tree correspond to overlapping rectangles.

Details:

1. Sort the rectangles by their x-coordinates. (Actually, each rectangle must appear twice in the sorted list—once for its left x-coordinate and once for its right x-coordinate.)
2. Scan the sorted list (from lowest to highest x-coordinate).
   - When an x-coordinate of a left edge is found, check whether the rectangle's y-coordinate interval overlaps an interval in the tree, and insert the rectangle (keyed on its y-coordinate interval) into the tree.
   - When an x-coordinate of a right edge is found, delete the rectangle from the interval tree.

   The interval tree always contains the set of "open" rectangles intersected by the sweep line. If an overlap is ever found in the interval tree, there are overlapping rectangles.

Time: $O(n \lg n)$

- $O(n \lg n)$ to sort the rectangles (we can use merge sort or heap sort).
- $O(n \lg n)$ for interval-tree operations (insert, delete, and check for overlap).

Selected Solutions for Chapter 15: Dynamic Programming

Solution to Exercise 15.2-5

Each time the l-loop executes, the i-loop executes $n - l + 1$ times. Each time the i-loop executes, the k-loop executes $j - i = l - 1$ times, each time referencing m twice. Thus the total number of times that an entry of m is referenced while computing other entries is $\sum_{l=2}^{n}(n - l + 1)(l - 1)2$. Thus,

$$
\begin{aligned}
\sum_{i=1}^{n}\sum_{j=i}^{n} R(i, j) &= \sum_{l=2}^{n}(n - l + 1)(l - 1)2 \\
&= 2\sum_{l=1}^{n-1}(n - l)l \\
&= 2\sum_{l=1}^{n-1} nl - 2\sum_{l=1}^{n-1} l^2 \\
&= 2\frac{n(n-1)n}{2} - 2\frac{(n-1)n(2n-1)}{6} \\
&= n^3 - n^2 - \frac{2n^3 - 3n^2 + n}{3} \\
&= \frac{n^3 - n}{3} .
\end{aligned}
$$

Solution to Exercise 15.3-1

Running RECURSIVE-MATRIX-CHAIN is asymptotically more efficient than enumerating all the ways of parenthesizing the product and computing the number of multiplications for each.

Consider the treatment of subproblems by the two approaches.

- For each possible place to split the matrix chain, the enumeration approach finds all ways to parenthesize the left half, finds all ways to parenthesize the right half, and looks at all possible combinations of the left half with the right half. The amount of work to look at each combination of left- and right-half subproblem results is thus the product of the number of ways to do the left half and the number of ways to do the right half.

- For each possible place to split the matrix chain, RECURSIVE-MATRIX-CHAIN finds the best way to parenthesize the left half, finds the best way to parenthesize the right half, and combines just those two results. Thus the amount of work to combine the left- and right-half subproblem results is $O(1)$.

Section 15.2 argued that the running time for enumeration is $\Omega(4^n/n^{3/2})$. We will show that the running time for RECURSIVE-MATRIX-CHAIN is $O(n3^{n-1})$.

To get an upper bound on the running time of RECURSIVE-MATRIX-CHAIN, we'll use the same approach used in Section 15.2 to get a lower bound: Derive a recurrence of the form $T(n) \le \dots$ and solve it by substitution. For the lower-bound recurrence, the book assumed that the execution of lines 1–2 and 6–7 each take at least unit time. For the upper-bound recurrence, we'll assume those pairs of lines each take at most constant time c. Thus, we have the recurrence

$$T(n) \le \begin{cases} c & \text{if } n = 1 , \\ c + \displaystyle\sum_{k=1}^{n-1}(T(k) + T(n-k) + c) & \text{if } n \ge 2 . \end{cases}$$

This is just like the book's $\ge$ recurrence except that it has c instead of 1, and so we can be rewrite it as

$$T(n) \le 2\sum_{i=1}^{n-1} T(i) + cn .$$

We shall prove that $T(n) = O(n3^{n-1})$ using the substitution method. (Note: Any upper bound on $T(n)$ that is $o(4^n/n^{3/2})$ will suffice. You might prefer to prove one that is easier to think up, such as $T(n) = O(3.5^n)$.) Specifically, we shall show that $T(n) \le cn3^{n-1}$ for all $n \ge 1$. The basis is easy, since $T(1) \le c = c \cdot 1 \cdot 3^{1-1}$. Inductively, for $n \ge 2$ we have

$$\begin{aligned} T(n) &\le 2\sum_{i=1}^{n-1} T(i) + cn \\ &\le 2\sum_{i=1}^{n-1} ci3^{i-1} + cn \\ &\le c \cdot \left(2\sum_{i=1}^{n-1} i3^{i-1} + n\right) \\ &= c \cdot \left(2 \cdot \left(\frac{n3^{n-1}}{3-1} + \frac{1-3^n}{(3-1)^2}\right) + n\right) \qquad \text{(see below)} \\ &= cn3^{n-1} + c \cdot \left(\frac{1-3^n}{2} + n\right) \\ &= cn3^{n-1} + \frac{c}{2}(2n + 1 - 3^n) \\ &\le cn3^{n-1} \text{ for all } c > 0, n \ge 1 . \end{aligned}$$

Running RECURSIVE-MATRIX-CHAIN takes $O(n3^{n-1})$ time, and enumerating all parenthesizations takes $\Omega(4^n/n^{3/2})$ time, and so RECURSIVE-MATRIX-CHAIN is more efficient than enumeration.

Note: The above substitution uses the following fact:

$$\sum_{i=1}^{n-1} ix^{i-1} = \frac{nx^{n-1}}{x-1} + \frac{1-x^n}{(x-1)^2} .$$

This equation can be derived from equation (A.5) by taking the derivative. Let

$$f(x) = \sum_{i=1}^{n-1} x^i = \frac{x^n - 1}{x-1} - 1 .$$

Then

$$\sum_{i=1}^{n-1} ix^{i-1} = f'(x) = \frac{nx^{n-1}}{x-1} + \frac{1-x^n}{(x-1)^2} .$$

Solution to Exercise 15.4-4

When computing a particular row of the c table, no rows before the previous row are needed. Thus only two rows—$2 \cdot Y.\mathit{length}$ entries—need to be kept in memory at a time. (Note: Each row of c actually has $Y.\mathit{length} + 1$ entries, but we don't need to store the column of 0's—instead we can make the program "know" that those entries are 0.) With this idea, we need only $2 \cdot \min(m, n)$ entries if we always call LCS-LENGTH with the shorter sequence as the Y argument.

We can thus do away with the c table as follows:

- Use two arrays of length $\min(m, n)$, *previous-row* and *current-row*, to hold the appropriate rows of c.
- Initialize *previous-row* to all 0 and compute *current-row* from left to right.
- When *current-row* is filled, if there are still more rows to compute, copy *current-row* into *previous-row* and compute the new *current-row*.

Actually only a little more than one row's worth of c entries—$\min(m, n) + 1$ entries—are needed during the computation. The only entries needed in the table when it is time to compute $c[i, j]$ are $c[i, k]$ for $k \leq j - 1$ (i.e., earlier entries in the current row, which will be needed to compute the next row); and $c[i-1, k]$ for $k \geq j - 1$ (i.e., entries in the previous row that are still needed to compute the rest of the current row). This is one entry for each k from 1 to $\min(m, n)$ except that there are two entries with $k = j - 1$, hence the additional entry needed besides the one row's worth of entries.

We can thus do away with the c table as follows:

- Use an array a of length $\min(m, n) + 1$ to hold the appropriate entries of c. At the time $c[i, j]$ is to be computed, a will hold the following entries:
  - $a[k] = c[i, k]$ for $1 \leq k < j - 1$ (i.e., earlier entries in the current "row"),
  - $a[k] = c[i-1, k]$ for $k \geq j - 1$ (i.e., entries in the previous "row"),

- $a[0] = c[i, j-1]$ (i.e., the previous entry computed, which couldn't be put into the "right" place in a without erasing the still-needed $c[i-1, j-1]$).

- Initialize a to all 0 and compute the entries from left to right.
  - Note that the 3 values needed to compute $c[i, j]$ for $j > 1$ are in $a[0] = c[i, j-1]$, $a[j-1] = c[i-1, j-1]$, and $a[j] = c[i-1, j]$.
  - When $c[i, j]$ has been computed, move $a[0]$ ($c[i, j-1]$) to its "correct" place, $a[j-1]$, and put $c[i, j]$ in $a[0]$.

Solution to Problem 15-4

Note: We assume that no word is longer than will fit into a line, i.e., $l_i \le M$ for all i.

First, we'll make some definitions so that we can state the problem more uniformly. Special cases about the last line and worries about whether a sequence of words fits in a line will be handled in these definitions, so that we can forget about them when framing our overall strategy.

- Define $\mathit{extras}[i, j] = M - j + i - \sum_{k=i}^{j} l_k$ to be the number of extra spaces at the end of a line containing words i through j. Note that *extras* may be negative.
- Now define the cost of including a line containing words i through j in the sum we want to minimize:

$$\mathit{lc}[i, j] = \begin{cases} \infty & \text{if } \mathit{extras}[i, j] < 0 \text{ (i.e., words } i, \ldots, j \text{ don't fit)}, \\ 0 & \text{if } j = n \text{ and } \mathit{extras}[i, j] \ge 0 \text{ (last line costs 0)}, \\ (\mathit{extras}[i, j])^3 & \text{otherwise}. \end{cases}$$

  By making the line cost infinite when the words don't fit on it, we prevent such an arrangement from being part of a minimal sum, and by making the cost 0 for the last line (if the words fit), we prevent the arrangement of the last line from influencing the sum being minimized.

We want to minimize the sum of *lc* over all lines of the paragraph.

Our subproblems are how to optimally arrange words $1, \ldots, j$, where $j = 1, \ldots, n$.

Consider an optimal arrangement of words $1, \ldots, j$. Suppose we know that the last line, which ends in word j, begins with word i. The preceding lines, therefore, contain words $1, \ldots, i-1$. In fact, they must contain an optimal arrangement of words $1, \ldots, i-1$. (The usual type of cut-and-paste argument applies.)

Let $c[j]$ be the cost of an optimal arrangement of words $1, \ldots, j$. If we know that the last line contains words $i, \ldots, j$, then $c[j] = c[i-1] + \mathit{lc}[i, j]$. As a base case, when we're computing $c[1]$, we need $c[0]$. If we set $c[0] = 0$, then $c[1] = \mathit{lc}[1, 1]$, which is what we want.

But of course we have to figure out which word begins the last line for the subproblem of words $1, \ldots, j$. So we try all possibilities for word i, and we pick the one that gives the lowest cost. Here, i ranges from 1 to j. Thus, we can define $c[j]$ recursively by

$$c[j] = \begin{cases} 0 & \text{if } j = 0 , \\ \min\limits_{1 \le i \le j} (c[i-1] + lc[i,j]) & \text{if } j > 0 . \end{cases}$$

Note that the way we defined lc ensures that

- all choices made will fit on the line (since an arrangement with $lc = \infty$ cannot be chosen as the minimum), and
- the cost of putting words $i, \ldots, j$ on the last line will not be 0 unless this really is the last line of the paragraph ($j = n$) or words $i \ldots j$ fill the entire line.

We can compute a table of c values from left to right, since each value depends only on earlier values.

To keep track of what words go on what lines, we can keep a parallel p table that points to where each c value came from. When $c[j]$ is computed, if $c[j]$ is based on the value of $c[k-1]$, set $p[j] = k$. Then after $c[n]$ is computed, we can trace the pointers to see where to break the lines. The last line starts at word $p[n]$ and goes through word n. The previous line starts at word $p[p[n]]$ and goes through word $p[n] - 1$, etc.

In pseudocode, here's how we construct the tables:

```
PRINT-NEATLY($l, n, M$)
let $extras[1..n, 1..n]$, $lc[1..n, 1..n]$, and $c[0..n]$ be new arrays
// Compute $extras[i, j]$ for $1 \le i \le j \le n$.
for $i = 1$ to $n$
    $extras[i, i] = M - l_i$
    for $j = i + 1$ to $n$
        $extras[i, j] = extras[i, j-1] - l_j - 1$
// Compute $lc[i, j]$ for $1 \le i \le j \le n$.
for $i = 1$ to $n$
    for $j = i$ to $n$
        if $extras[i, j] < 0$
            $lc[i, j] = \infty$
        elseif $j == n$ and $extras[i, j] \ge 0$
            $lc[i, j] = 0$
        else $lc[i, j] = (extras[i, j])^3$
// Compute $c[j]$ and $p[j]$ for $1 \le j \le n$.
$c[0] = 0$
for $j = 1$ to $n$
    $c[j] = \infty$
    for $i = 1$ to $j$
        if $c[i-1] + lc[i, j] < c[j]$
            $c[j] = c[i-1] + lc[i, j]$
            $p[j] = i$
return $c$ and $p$
```

Quite clearly, both the time and space are $\Theta(n^2)$.

In fact, we can do a bit better: we can get both the time and space down to $\Theta(nM)$. The key observation is that at most $\lceil M/2 \rceil$ words can fit on a line. (Each word is at least one character long, and there's a space between words.) Since a line with words $i, \ldots, j$ contains $j - i + 1$ words, if $j - i + 1 > \lceil M/2 \rceil$ then we know that $lc[i, j] = \infty$. We need only compute and store *extras*$[i, j]$ and $lc[i, j]$ for $j - i + 1 \leq \lceil M/2 \rceil$. And the inner **for** loop header in the computation of $c[j]$ and $p[j]$ can run from $\max(1, j - \lceil M/2 \rceil + 1)$ to j.

We can reduce the space even further to $\Theta(n)$. We do so by not storing the *lc* and *extras* tables, and instead computing the value of $lc[i, j]$ as needed in the last loop. The idea is that we could compute $lc[i, j]$ in $O(1)$ time if we knew the value of *extras*$[i, j]$. And if we scan for the minimum value in *descending* order of i, we can compute that as *extras*$[i, j] =$ *extras*$[i + 1, j] - l_i - 1$. (Initially, *extras*$[j, j] = M - l_j$.) This improvement reduces the space to $\Theta(n)$, since now the only tables we store are c and p.

Here's how we print which words are on which line. The printed output of GIVE-LINES(p, j) is a sequence of triples (k, i, j), indicating that words $i, \ldots, j$ are printed on line k. The return value is the line number k.

```
GIVE-LINES(p, j)
i = p[j]
if i == 1
    k = 1
else k = GIVE-LINES(p, i - 1) + 1
print (k, i, j)
return k
```

The initial call is GIVE-LINES(p, n). Since the value of j decreases in each recursive call, GIVE-LINES takes a total of $O(n)$ time.

Selected Solutions for Chapter 16: Greedy Algorithms

Solution to Exercise 16.1-4

Let S be the set of n activities.

The "obvious" solution of using GREEDY-ACTIVITY-SELECTOR to find a maximum-size set S_1 of compatible activities from S for the first lecture hall, then using it again to find a maximum-size set S_2 of compatible activities from $S - S_1$ for the second hall, (and so on until all the activities are assigned), requires $\Theta(n^2)$ time in the worst case. Moreover, it can produce a result that uses more lecture halls than necessary. Consider activities with the intervals $\{[1,4), [2,5), [6,7), [4,8)\}$. GREEDY-ACTIVITY-SELECTOR would choose the activities with intervals $[1,4)$ and $[6,7)$ for the first lecture hall, and then each of the activities with intervals $[2,5)$ and $[4,8)$ would have to go into its own hall, for a total of three halls used. An optimal solution would put the activities with intervals $[1,4)$ and $[4,8)$ into one hall and the activities with intervals $[2,5)$ and $[6,7)$ into another hall, for only two halls used.

There is a correct algorithm, however, whose asymptotic time is just the time needed to sort the activities by time—$O(n \lg n)$ time for arbitrary times, or possibly as fast as $O(n)$ if the times are small integers.

The general idea is to go through the activities in order of start time, assigning each to any hall that is available at that time. To do this, move through the set of events consisting of activities starting and activities finishing, in order of event time. Maintain two lists of lecture halls: Halls that are busy at the current event-time t (because they have been assigned an activity i that started at $s_i \le t$ but won't finish until $f_i > t$) and halls that are free at time t. (As in the activity-selection problem in Section 16.1, we are assuming that activity time intervals are half open—i.e., that if $s_i \ge f_j$, then activities i and j are compatible.) When t is the start time of some activity, assign that activity to a free hall and move the hall from the free list to the busy list. When t is the finish time of some activity, move the activity's hall from the busy list to the free list. (The activity is certainly in some hall, because the event times are processed in order and the activity must have started before its finish time t, hence must have been assigned to a hall.)

To avoid using more halls than necessary, always pick a hall that has already had an activity assigned to it, if possible, before picking a never-used hall. (This can be done by always working at the front of the free-halls list—putting freed halls onto the front of the list and taking halls from the front of the list—so that a new hall doesn't come to the front and get chosen if there are previously-used halls.)

This guarantees that the algorithm uses as few lecture halls as possible: The algorithm will terminate with a schedule requiring $m \le n$ lecture halls. Let activity i be the first activity scheduled in lecture hall m. The reason that i was put in the mth lecture hall is that the first $m - 1$ lecture halls were busy at time s_i. So at this time there are m activities occurring simultaneously. Therefore any schedule must use at least m lecture halls, so the schedule returned by the algorithm is optimal.

Run time:

- Sort the $2n$ activity-starts/activity-ends events. (In the sorted order, an activity-ending event should precede an activity-starting event that is at the same time.) $O(n \lg n)$ time for arbitrary times, possibly $O(n)$ if the times are restricted (e.g., to small integers).
- Process the events in $O(n)$ time: Scan the $2n$ events, doing $O(1)$ work for each (moving a hall from one list to the other and possibly associating an activity with it).

Total: $O(n + \text{time to sort})$

Solution to Exercise 16.2-2

The solution is based on the optimal-substructure observation in the text: Let i be the highest-numbered item in an optimal solution S for W pounds and items $1, \ldots, n$. Then $S' = S - \{i\}$ must be an optimal solution for $W - w_i$ pounds and items $1, \ldots, i - 1$, and the value of the solution S is v_i plus the value of the subproblem solution S'.

We can express this relationship in the following formula: Define $c[i, w]$ to be the value of the solution for items $1, \ldots, i$ and maximum weight w. Then

$$c[i, w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 , \\ c[i-1, w] & \text{if } w_i > w , \\ \max(v_i + c[i-1, w-w_i], c[i-1, w]) & \text{if } i > 0 \text{ and } w \ge w_i . \end{cases}$$

The last case says that the value of a solution for i items either includes item i, in which case it is v_i plus a subproblem solution for $i - 1$ items and the weight excluding w_i, or doesn't include item i, in which case it is a subproblem solution for $i - 1$ items and the same weight. That is, if the thief picks item i, he takes v_i value, and he can choose from items $1, \ldots, i - 1$ up to the weight limit $w - w_i$, and get $c[i - 1, w - w_i]$ additional value. On the other hand, if he decides not to take item i, he can choose from items $1, \ldots, i - 1$ up to the weight limit w, and get $c[i - 1, w]$ value. The better of these two choices should be made.

The algorithm takes as inputs the maximum weight W, the number of items n, and the two sequences $v = \langle v_1, v_2, \ldots, v_n \rangle$ and $w = \langle w_1, w_2, \ldots, w_n \rangle$. It stores the $c[i, j]$ values in a table $c[0 \,..\, n, 0 \,..\, W]$ whose entries are computed in row-major order. (That is, the first row of c is filled in from left to right, then the second row, and so on.) At the end of the computation, $c[n, W]$ contains the maximum value the thief can take.

```
DYNAMIC-0-1-KNAPSACK($v, w, n, W$)
let $c[0 \,..\, n, 0 \,..\, W]$ be a new array
for $w = 0$ to $W$
    $c[0, w] = 0$
for $i = 1$ to $n$
    $c[i, 0] = 0$
    for $w = 1$ to $W$
        if $w_i \le w$
            if $v_i + c[i-1, w-w_i] > c[i-1, w]$
                $c[i, w] = v_i + c[i-1, w-w_i]$
            else $c[i, w] = c[i-1, w]$
        else $c[i, w] = c[i-1, w]$
```

We can use the c table to deduce the set of items to take by starting at $c[n, W]$ and tracing where the optimal values came from. If $c[i, w] = c[i-1, w]$, then item i is not part of the solution, and we continue tracing with $c[i-1, w]$. Otherwise item i is part of the solution, and we continue tracing with $c[i-1, w-w_i]$.

The above algorithm takes $\Theta(nW)$ time total:

- $\Theta(nW)$ to fill in the c table: $(n+1) \cdot (W+1)$ entries, each requiring $\Theta(1)$ time to compute.
- $O(n)$ time to trace the solution (since it starts in row n of the table and moves up one row at each step).

Solution to Exercise 16.2-7

Sort A and B into monotonically decreasing order.

Here's a proof that this method yields an optimal solution. Consider any indices i and j such that $i < j$, and consider the terms ${a_i}^{b_i}$ and ${a_j}^{b_j}$. We want to show that it is no worse to include these terms in the payoff than to include ${a_i}^{b_j}$ and ${a_j}^{b_i}$, i.e., that ${a_i}^{b_i}{a_j}^{b_j} \ge {a_i}^{b_j}{a_j}^{b_i}$. Since A and B are sorted into monotonically decreasing order and $i < j$, we have $a_i \ge a_j$ and $b_i \ge b_j$. Since a_i and a_j are positive and $b_i - b_j$ is nonnegative, we have ${a_i}^{b_i - b_j} \ge {a_j}^{b_i - b_j}$. Multiplying both sides by ${a_i}^{b_j}{a_j}^{b_j}$ yields ${a_i}^{b_i}{a_j}^{b_j} \ge {a_i}^{b_j}{a_j}^{b_i}$.

Since the order of multiplication doesn't matter, sorting A and B into monotonically increasing order works as well.

Selected Solutions for Chapter 17: Amortized Analysis

Solution to Exercise 17.1-3

Let c_i = cost of ith operation.

$$c_i = \begin{cases} i & \text{if } i \text{ is an exact power of 2}, \\ 1 & \text{otherwise}. \end{cases}$$

Operation	Cost
1	1
2	2
3	1
4	4
5	1
6	1
7	1
8	8
9	1
10	1
⋮	⋮

n operations cost

$$\sum_{i=1}^{n} c_i \le n + \sum_{j=0}^{\lg n} 2^j = n + (2n - 1) < 3n .$$

(Note: Ignoring floor in upper bound of $\sum 2^j$.)

$$\text{Average cost of operation} = \frac{\text{Total cost}}{\text{\# operations}} < 3 .$$

By aggregate analysis, the amortized cost per operation $= O(1)$.

Solution to Exercise 17.2-2

Let c_i = cost of ith operation.

$$c_i = \begin{cases} i & \text{if } i \text{ is an exact power of 2 ,} \\ 1 & \text{otherwise .} \end{cases}$$

Charge each operation \$3 (amortized cost $\widehat{c}_i$).

- If i is not an exact power of 2, pay \$1, and store \$2 as credit.
- If i is an exact power of 2, pay \$$i$, using stored credit.

Operation	Cost	Actual cost	Credit remaining
1	3	1	2
2	3	2	3
3	3	1	5
4	3	4	4
5	3	1	6
6	3	1	8
7	3	1	10
8	3	8	5
9	3	1	7
10	3	1	9
⋮	⋮	⋮	⋮

Since the amortized cost is \$3 per operation, $\sum_{i=1}^{n} \widehat{c}_i = 3n$.

We know from Exercise 17.1-3 that $\sum_{i=1}^{n} c_i < 3n$.

Then we have $\sum_{i=1}^{n} \widehat{c}_i \geq \sum_{i=1}^{n} c_i \Rightarrow$ credit $=$ amortized cost $-$ actual cost ≥ 0.

Since the amortized cost of each operation is $O(1)$, and the amount of credit never goes negative, the total cost of n operations is $O(n)$.

Solution to Exercise 17.2-3

We introduce a new field $A.max$ to hold the index of the high-order 1 in A. Initially, $A.max$ is set to -1, since the low-order bit of A is at index 0, and there are initially no 1's in A. The value of $A.max$ is updated as appropriate when the counter is incremented or reset, and we use this value to limit how much of A must be looked at to reset it. By controlling the cost of RESET in this way, we can limit it to an amount that can be covered by credit from earlier INCREMENTs.

```
INCREMENT($A$)
$i = 0$
**while** $i < A.length$ and $A[i] == 1$
    $A[i] = 0$
    $i = i + 1$
**if** $i < A.length$
    $A[i] = 1$
    // Additions to book's INCREMENT start here.
    **if** $i > A.max$
        $A.max = i$
**else** $A.max = -1$
```

```
RESET($A$)
**for** $i = 0$ **to** $A.max$
    $A[i] = 0$
$A.max = -1$
```

As for the counter in the book, we assume that it costs \$1 to flip a bit. In addition, we assume it costs \$1 to update $A.max$.

Setting and resetting of bits by INCREMENT will work exactly as for the original counter in the book: \$1 will pay to set one bit to 1; \$1 will be placed on the bit that is set to 1 as credit; the credit on each 1 bit will pay to reset the bit during incrementing.

In addition, we'll use \$1 to pay to update *max*, and if *max* increases, we'll place an additional \$1 of credit on the new high-order 1. (If *max* doesn't increase, we can just waste that \$1—it won't be needed.) Since RESET manipulates bits at positions only up to $A.max$, and since each bit up to there must have become the high-order 1 at some time before the high-order 1 got up to $A.max$, every bit seen by RESET has \$1 of credit on it. So the zeroing of bits of A by RESET can be completely paid for by the credit stored on the bits. We just need \$1 to pay for resetting *max*.

Thus charging \$4 for each INCREMENT and \$1 for each RESET is sufficient, so the sequence of n INCREMENT and RESET operations takes $O(n)$ time.

Selected Solutions for Chapter 21: Data Structures for Disjoint Sets

Solution to Exercise 21.2-3

We want to show that we can assign $O(1)$ charges to MAKE-SET and FIND-SET and an $O(\lg n)$ charge to UNION such that the charges for a sequence of these operations are enough to cover the cost of the sequence—$O(m + n \lg n)$, according to the theorem. When talking about the charge for each kind of operation, it is helpful to also be able to talk about the number of each kind of operation.

Consider the usual sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, and let $l < n$ be the number of UNION operations. (Recall the discussion in Section 21.1 about there being at most $n - 1$ UNION operations.) Then there are n MAKE-SET operations, l UNION operations, and $m - n - l$ FIND-SET operations.

The theorem didn't separately name the number l of UNIONs; rather, it bounded the number by n. If you go through the proof of the theorem with l UNIONs, you get the time bound $O(m - l + l \lg l) = O(m + l \lg l)$ for the sequence of operations. That is, the actual time taken by the sequence of operations is at most $c(m + l \lg l)$, for some constant c.

Thus, we want to assign operation charges such that

$$\begin{array}{lcl} \text{(MAKE-SET charge)} & \cdot & n \\ +\text{(FIND-SET charge)} & \cdot & (m - n - l) \\ +\text{(UNION charge)} & \cdot & l \\ \hline \geq c(m + l \lg l) \ , & & \end{array}$$

so that the amortized costs give an upper bound on the actual costs.

The following assignments work, where c' is some constant $\geq c$:

- MAKE-SET: c'
- FIND-SET: c'
- UNION: $c'(\lg n + 1)$

Substituting into the above sum, we get

$$\begin{aligned} c'n + c'(m - n - l) + c'(\lg n + 1)l &= c'm + c'l \lg n \\ &= c'(m + l \lg n) \\ &> c(m + l \lg l) \ . \end{aligned}$$

Solution to Exercise 21.2-6

Let's call the two lists A and B, and suppose that the representative of the new list will be the representative of A. Rather than appending B to the end of A, instead splice B into A right after the first element of A. We have to traverse B to update pointers to the set object anyway, so we can just make the last element of B point to the second element of A.

Selected Solutions for Chapter 22: Elementary Graph Algorithms

Solution to Exercise 22.1-7

$$BB^T(i,j) = \sum_{e \in E} b_{ie} b^T_{ej} = \sum_{e \in E} b_{ie} b_{je}$$

- If $i = j$, then $b_{ie} b_{je} = 1$ (it is $1 \cdot 1$ or $(-1) \cdot (-1)$) whenever e enters or leaves vertex i, and 0 otherwise.
- If $i \neq j$, then $b_{ie} b_{je} = -1$ when $e = (i, j)$ or $e = (j, i)$, and 0 otherwise.

Thus,

$$BB^T(i,j) = \begin{cases} \text{degree of } i = \text{in-degree} + \text{out-degree} & \text{if } i = j\ , \\ -(\text{\# of edges connecting } i \text{ and } j) & \text{if } i \neq j\ . \end{cases}$$

Solution to Exercise 22.2-5

The correctness proof for the BFS algorithm shows that $u.d = \delta(s, u)$, and the algorithm doesn't assume that the adjacency lists are in any particular order.

In Figure 22.3, if t precedes x in *Adj*$[w]$, we can get the breadth-first tree shown in the figure. But if x precedes t in *Adj*$[w]$ and u precedes y in *Adj*$[x]$, we can get edge (x, u) in the breadth-first tree.

Solution to Exercise 22.3-12

The following pseudocode modifies the DFS and DFS-VISIT procedures to assign values to the *cc* attributes of vertices.

```
DFS($G$)
for each vertex $u \in G.V$
    $u.color = \text{WHITE}$
    $u.\pi = \text{NIL}$
$time = 0$
$counter = 0$
for each vertex $u \in G.V$
    if $u.color == \text{WHITE}$
        $counter = counter + 1$
        DFS-VISIT($G, u, counter$)
```

```
DFS-VISIT($G, u, counter$)
$u.cc = counter$                // label the vertex
$time = time + 1$
$u.d = time$
$u.color = \text{GRAY}$
for each $v \in G.Adj[u]$
    if $v.color == \text{WHITE}$
        $v.\pi = u$
        DFS-VISIT($G, v, counter$)
$u.color = \text{BLACK}$
$time = time + 1$
$u.f = time$
```

This DFS increments a counter each time DFS-VISIT is called to grow a new tree in the DFS forest. Every vertex visited (and added to the tree) by DFS-VISIT is labeled with that same counter value. Thus $u.cc = v.cc$ if and only if u and v are visited in the same call to DFS-VISIT from DFS, and the final value of the counter is the number of calls that were made to DFS-VISIT by DFS. Also, since every vertex is visited eventually, every vertex is labeled.

Thus all we need to show is that the vertices visited by each call to DFS-VISIT from DFS are exactly the vertices in one connected component of G.

- All vertices in a connected component are visited by one call to DFS-VISIT from DFS:

  Let u be the first vertex in component C visited by DFS-VISIT. Since a vertex becomes non-white only when it is visited, all vertices in C are white when DFS-VISIT is called for u. Thus, by the white-path theorem, all vertices in C become descendants of u in the forest, which means that all vertices in C are visited (by recursive calls to DFS-VISIT) before DFS-VISIT returns to DFS.

- All vertices visited by one call to DFS-VISIT from DFS are in the same connected component:

  If two vertices are visited in the same call to DFS-VISIT from DFS, they are in the same connected component, because vertices are visited only by following paths in G (by following edges found in adjacency lists, starting from some vertex).

Solution to Exercise 22.4-3

An undirected graph is acyclic (i.e., a forest) if and only if a DFS yields no back edges.

- If there's a back edge, there's a cycle.
- If there's no back edge, then by Theorem 22.10, there are only tree edges. Hence, the graph is acyclic.

Thus, we can run DFS: if we find a back edge, there's a cycle.

- Time: $O(V)$. (Not $O(V + E)$!)
  If we ever see $|V|$ distinct edges, we must have seen a back edge because (by Theorem B.2 on p. 1174) in an acyclic (undirected) forest, $|E| \leq |V| - 1$.

Solution to Problem 22-1

a. 1. Suppose (u, v) is a back edge or a forward edge in a BFS of an undirected graph. Then one of u and v, say u, is a proper ancestor of the other (v) in the breadth-first tree. Since we explore all edges of u before exploring any edges of any of u's descendants, we must explore the edge (u, v) at the time we explore u. But then (u, v) must be a tree edge.

2. In BFS, an edge (u, v) is a tree edge when we set $v.\pi = u$. But we only do so when we set $v.d = u.d + 1$. Since neither $u.d$ nor $v.d$ ever changes thereafter, we have $v.d = u.d + 1$ when BFS completes.

3. Consider a cross edge (u, v) where, without loss of generality, u is visited before v. At the time we visit u, vertex v must already be on the queue, for otherwise (u, v) would be a tree edge. Because v is on the queue, we have $v.d \leq u.d + 1$ by Lemma 22.3. By Corollary 22.4, we have $v.d \geq u.d$. Thus, either $v.d = u.d$ or $v.d = u.d + 1$.

b. 1. Suppose (u, v) is a forward edge. Then we would have explored it while visiting u, and it would have been a tree edge.

2. Same as for undirected graphs.

3. For any edge (u, v), whether or not it's a cross edge, we cannot have $v.d > u.d + 1$, since we visit v at the latest when we explore edge (u, v). Thus, $v.d \leq u.d + 1$.

4. Clearly, $v.d \geq 0$ for all vertices v. For a back edge (u, v), v is an ancestor of u in the breadth-first tree, which means that $v.d \leq u.d$. (Note that since self-loops are considered to be back edges, we could have $u = v$.)

Selected Solutions for Chapter 23: Minimum Spanning Trees

Solution to Exercise 23.1-1

Theorem 23.1 shows this.

Let A be the empty set and S be any set containing u but not v.

Solution to Exercise 23.1-4

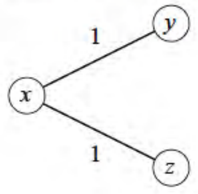
A triangle whose edge weights are all equal is a graph in which every edge is a light edge crossing some cut. But the triangle is cyclic, so it is not a minimum spanning tree.

Solution to Exercise 23.1-6

Suppose that for every cut of G, there is a unique light edge crossing the cut. Let us consider two minimum spanning trees, T and T', of G. We will show that every edge of T is also in T', which means that T and T' are the same tree and hence there is a unique minimum spanning tree.

Consider any edge $(u, v) \in T$. If we remove (u, v) from T, then T becomes disconnected, resulting in a cut $(S, V - S)$. The edge (u, v) is a light edge crossing the cut $(S, V - S)$ (by Exercise 23.1-3). Now consider the edge $(x, y) \in T'$ that crosses $(S, V - S)$. It, too, is a light edge crossing this cut. Since the light edge crossing $(S, V - S)$ is unique, the edges (u, v) and (x, y) are the same edge. Thus, $(u, v) \in T'$. Since we chose (u, v) arbitrarily, every edge in T is also in T'.

Here's a counterexample for the converse:



Here, the graph is its own minimum spanning tree, and so the minimum spanning tree is unique. Consider the cut $(\{x\}, \{y, z\})$. Both of the edges (x, y) and (x, z) are light edges crossing the cut, and they are both light edges.

Selected Solutions for Chapter 24: Single-Source Shortest Paths

Solution to Exercise 24.1-3

If the greatest number of edges on any shortest path from the source is m, then the path-relaxation property tells us that after m iterations of BELLMAN-FORD, every vertex v has achieved its shortest-path weight in $v.d$. By the upper-bound property, after m iterations, no d values will ever change. Therefore, no d values will change in the $(m + 1)$st iteration. Because we do not know m in advance, we cannot make the algorithm iterate exactly m times and then terminate. But if we just make the algorithm stop when nothing changes any more, it will stop after $m + 1$ iterations.

```
BELLMAN-FORD-(M+1)($G, w, s$)
INITIALIZE-SINGLE-SOURCE($G, s$)
changes = TRUE
while changes == TRUE
    changes = FALSE
    for each edge $(u, v) \in G.E$
        RELAX-M($u, v, w$)
```

```
RELAX-M($u, v, w$)
if $v.d > u.d + w(u, v)$
    $v.d = u.d + w(u, v)$
    $v.\pi = u$
    changes = TRUE
```

The test for a negative-weight cycle (based on there being a d value that would change if another relaxation step was done) has been removed above, because this version of the algorithm will never get out of the **while** loop unless all d values stop changing.

Solution to Exercise 24.3-3

Yes, the algorithm still works. Let u be the leftover vertex that does not get extracted from the priority queue Q. If u is not reachable from s, then

$u.d = \delta(s, u) = \infty$. If u is reachable from s, then there is a shortest path $p = s \leadsto x \to u$. When the node x was extracted, $x.d = \delta(s, x)$ and then the edge (x, u) was relaxed; thus, $u.d = \delta(s, u)$.

Solution to Exercise 24.3-6

To find the most reliable path between s and t, run Dijkstra's algorithm with edge weights $w(u, v) = -\lg r(u, v)$ to find shortest paths from s in $O(E + V \lg V)$ time. The most reliable path is the shortest path from s to t, and that path's reliability is the product of the reliabilities of its edges.

Here's why this method works. Because the probabilities are independent, the probability that a path will not fail is the product of the probabilities that its edges will not fail. We want to find a path $s \overset{p}{\leadsto} t$ such that $\prod_{(u,v)\in p} r(u, v)$ is maximized. This is equivalent to maximizing $\lg(\prod_{(u,v)\in p} r(u, v)) = \sum_{(u,v)\in p} \lg r(u, v)$, which is in turn equivalent to minimizing $\sum_{(u,v)\in p} -\lg r(u, v)$. (Note: $r(u, v)$ can be 0, and $\lg 0$ is undefined. So in this algorithm, define $\lg 0 = -\infty$.) Thus if we assign weights $w(u, v) = -\lg r(u, v)$, we have a shortest-path problem.

Since $\lg 1 = 0$, $\lg x < 0$ for $0 < x < 1$, and we have defined $\lg 0 = -\infty$, all the weights w are nonnegative, and we can use Dijkstra's algorithm to find the shortest paths from s in $O(E + V \lg V)$ time.

Alternate answer

You can also work with the original probabilities by running a modified version of Dijkstra's algorithm that maximizes the product of reliabilities along a path instead of minimizing the sum of weights along a path.

In Dijkstra's algorithm, use the reliabilities as edge weights and substitute

- max (and EXTRACT-MAX) for min (and EXTRACT-MIN) in relaxation and the queue,
- $\cdot$ for $+$ in relaxation,
- 1 (identity for $\cdot$) for 0 (identity for $+$) and $-\infty$ (identity for min) for ∞ (identity for max).

For example, we would use the following instead of the usual RELAX procedure:

```
RELAX-RELIABILITY($u, v, r$)
if $v.d < u.d \cdot r(u, v)$
    $v.d = u.d \cdot r(u, v)$
    $v.\pi = u$
```

This algorithm is isomorphic to the one above: it performs the same operations except that it is working with the original probabilities instead of the transformed ones.

Solution to Exercise 24.4-7

Observe that after the first pass, all d values are at most 0, and that relaxing edges (v_0, v_i) will never again change a d value. Therefore, we can eliminate v_0 by running the Bellman-Ford algorithm on the constraint graph without the v_0 node but initializing all shortest path estimates to 0 instead of ∞.

Solution to Exercise 24.5-4

Whenever RELAX sets π for some vertex, it also reduces the vertex's d value. Thus if $s.\pi$ gets set to a non-NIL value, $s.d$ is reduced from its initial value of 0 to a negative number. But $s.d$ is the weight of some path from s to s, which is a cycle including s. Thus, there is a negative-weight cycle.

Solution to Problem 24-3

a. We can use the Bellman-Ford algorithm on a suitable weighted, directed graph $G = (V, E)$, which we form as follows. There is one vertex in V for each currency, and for each pair of currencies c_i and c_j, there are directed edges (v_i, v_j) and (v_j, v_i). (Thus, $|V| = n$ and $|E| = n(n-1)$.)

To determine edge weights, we start by observing that

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1$$

if and only if

$$\frac{1}{R[i_1, i_2]} \cdot \frac{1}{R[i_2, i_3]} \cdots \frac{1}{R[i_{k-1}, i_k]} \cdot \frac{1}{R[i_k, i_1]} < 1 .$$

Taking logs of both sides of the inequality above, we express this condition as

$$\lg \frac{1}{R[i_1, i_2]} + \lg \frac{1}{R[i_2, i_3]} + \cdots + \lg \frac{1}{R[i_{k-1}, i_k]} + \lg \frac{1}{R[i_k, i_1]} < 0 .$$

Therefore, if we define the weight of edge (v_i, v_j) as

$$\begin{aligned} w(v_i, v_j) &= \lg \frac{1}{R[i, j]} \\ &= -\lg R[i, j] , \end{aligned}$$

then we want to find whether there exists a negative-weight cycle in G with these edge weights.

We can determine whether there exists a negative-weight cycle in G by adding an extra vertex v_0 with 0-weight edges (v_0, v_i) for all $v_i \in V$, running BELLMAN-FORD from v_0, and using the boolean result of BELLMAN-FORD (which is TRUE if there are no negative-weight cycles and FALSE if there is a negative-weight cycle) to guide our answer. That is, we invert the boolean result of BELLMAN-FORD.

This method works because adding the new vertex ν_0 with 0-weight edges from ν_0 to all other vertices cannot introduce any new cycles, yet it ensures that all negative-weight cycles are reachable from ν_0.

It takes $\Theta(n^2)$ time to create G, which has $\Theta(n^2)$ edges. Then it takes $O(n^3)$ time to run BELLMAN-FORD. Thus, the total time is $O(n^3)$.

Another way to determine whether a negative-weight cycle exists is to create G and, without adding ν_0 and its incident edges, run either of the all-pairs shortest-paths algorithms. If the resulting shortest-path distance matrix has any negative values on the diagonal, then there is a negative-weight cycle.

b. Assuming that we ran BELLMAN-FORD to solve part (a), we only need to find the vertices of a negative-weight cycle. We can do so as follows. First, relax all the edges once more. Since there is a negative-weight cycle, the d value of some vertex u will change. We just need to repeatedly follow the π values until we get back to u. In other words, we can use the recursive method given by the PRINT-PATH procedure of Section 22.2, but stop it when it returns to vertex u.

The running time is $O(n^3)$ to run BELLMAN-FORD, plus $O(n)$ to print the vertices of the cycle, for a total of $O(n^3)$ time.

Selected Solutions for Chapter 25: All-Pairs Shortest Paths

Solution to Exercise 25.1-3

The matrix $L^{(0)}$ corresponds to the identity matrix

$$I = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix}$$

of regular matrix multiplication. Substitute 0 (the identity for $+$) for ∞ (the identity for min), and 1 (the identity for $\cdot$) for 0 (the identity for $+$).

Solution to Exercise 25.1-5

The all-pairs shortest-paths algorithm in Section 25.1 computes

$$L^{(n-1)} = W^{n-1} = L^{(0)} \cdot W^{n-1} ,$$

where $l_{ij}^{(n-1)} = \delta(i, j)$ and $L^{(0)}$ is the identity matrix. That is, the entry in the ith row and jth column of the matrix "product" is the shortest-path distance from vertex i to vertex j, and row i of the product is the solution to the single-source shortest-paths problem for vertex i.

Notice that in a matrix "product" $C = A \cdot B$, the ith row of C is the ith row of A "multiplied" by B. Since all we want is the ith row of C, we never need more than the ith row of A.

Thus the solution to the single-source shortest-paths from vertex i is $L_i^{(0)} \cdot W^{n-1}$, where $L_i^{(0)}$ is the ith row of $L^{(0)}$—a vector whose ith entry is 0 and whose other entries are ∞.

Doing the above "multiplications" starting from the left is essentially the same as the BELLMAN-FORD algorithm. The vector corresponds to the d values in BELLMAN-FORD—the shortest-path estimates from the source to each vertex.

- The vector is initially 0 for the source and ∞ for all other vertices, the same as the values set up for d by INITIALIZE-SINGLE-SOURCE.

- Each "multiplication" of the current vector by W relaxes all edges just as BELLMAN-FORD does. That is, a distance estimate in the row, say the distance to v, is updated to a smaller estimate, if any, formed by adding some $w(u, v)$ to the current estimate of the distance to u.
- The relaxation/multiplication is done $n - 1$ times.

Solution to Exercise 25.2-4

With the superscripts, the computation is $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$. If, having dropped the superscripts, we were to compute and store d_{ik} or d_{kj} before using these values to compute d_{ij}, we might be computing one of the following:

$$\begin{aligned}
d_{ij}^{(k)} &= \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k)} + d_{kj}^{(k-1)}\right) , \\
d_{ij}^{(k)} &= \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k)}\right) , \\
d_{ij}^{(k)} &= \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k)} + d_{kj}^{(k)}\right) .
\end{aligned}$$

In any of these scenarios, we're computing the weight of a shortest path from i to j with all intermediate vertices in $\{1, 2, \ldots, k\}$. If we use $d_{ik}^{(k)}$, rather than $d_{ik}^{(k-1)}$, in the computation, then we're using a subpath from i to k with all intermediate vertices in $\{1, 2, \ldots, k\}$. But k cannot be an *intermediate* vertex on a shortest path from i to k, since otherwise there would be a cycle on this shortest path. Thus, $d_{ik}^{(k)} = d_{ik}^{(k-1)}$. A similar argument applies to show that $d_{kj}^{(k)} = d_{kj}^{(k-1)}$. Hence, we can drop the superscripts in the computation.

Solution to Exercise 25.3-4

It changes shortest paths. Consider the following graph. $V = \{s, x, y, z\}$, and there are 4 edges: $w(s, x) = 2$, $w(x, y) = 2$, $w(s, y) = 5$, and $w(s, z) = -10$. So we'd add 10 to every weight to make $\hat{w}$. With w, the shortest path from s to y is $s \to x \to y$, with weight 4. With $\hat{w}$, the shortest path from s to y is $s \to y$, with weight 15. (The path $s \to x \to y$ has weight 24.) The problem is that by just adding the same amount to every edge, you penalize paths with more edges, even if their weights are low.

Selected Solutions for Chapter 26: Maximum Flow

Solution to Exercise 26.2-11

For any two vertices u and v in G, we can define a flow network G_{uv} consisting of the directed version of G with $s = u$, $t = v$, and all edge capacities set to 1. (The flow network G_{uv} has V vertices and $2\,|E|$ edges, so that it has $O(V)$ vertices and $O(E)$ edges, as required. We want all capacities to be 1 so that the number of edges of G crossing a cut equals the capacity of the cut in G_{uv}.) Let f_{uv} denote a maximum flow in G_{uv}.

We claim that for any $u \in V$, the edge connectivity k equals $\min\limits_{v \in V - \{u\}} \{|f_{uv}|\}$. We'll show below that this claim holds. Assuming that it holds, we can find k as follows:

```
EDGE-CONNECTIVITY($G$)
$k = \infty$
select any vertex $u \in G.V$
for each vertex $v \in G.V - \{u\}$
    set up the flow network $G_{uv}$ as described above
    find the maximum flow $f_{uv}$ on $G_{uv}$
    $k = \min(k, |f_{uv}|)$
return $k$
```

The claim follows from the max-flow min-cut theorem and how we chose capacities so that the capacity of a cut is the number of edges crossing it. We prove that $k = \min\limits_{v \in V - \{u\}} \{|f_{uv}|\}$, for any $u \in V$ by showing separately that k is at least this minimum and that k is at most this minimum.

- Proof that $k \ge \min\limits_{v \in V - \{u\}} \{|f_{uv}|\}$:

  Let $m = \min\limits_{v \in V - \{u\}} \{|f_{uv}|\}$. Suppose we remove only $m - 1$ edges from G. For any vertex v, by the max-flow min-cut theorem, u and v are still connected. (The max flow from u to v is at least m, hence any cut separating u from v has capacity at least m, which means at least m edges cross any such cut. Thus at least one edge is left crossing the cut when we remove $m - 1$ edges.) Thus every node is connected to u, which implies that the graph is still connected. So at least m edges must be removed to disconnect the graph—i.e., $k \ge \min\limits_{v \in V - \{u\}} \{|f_{uv}|\}$.

- Proof that $k \le \min_{v \in V - \{u\}} \{|f_{uv}|\}$:

  Consider a vertex v with the minimum $|f_{uv}|$. By the max-flow min-cut theorem, there is a cut of capacity $|f_{uv}|$ separating u and v. Since all edge capacities are 1, exactly $|f_{uv}|$ edges cross this cut. If these edges are removed, there is no path from u to v, and so our graph becomes disconnected. Hence $k \le \min_{v \in V - \{u\}} \{|f_{uv}|\}$.
- Thus, the claim that $k = \min_{v \in V - \{u\}} \{|f_{uv}|\}$, for any $u \in V$ is true.

Solution to Exercise 26.3-3

By definition, an augmenting path is a simple path $s \leadsto t$ in the residual network G'_f. Since G has no edges between vertices in L and no edges between vertices in R, neither does the flow network G' and hence neither does G'_f. Also, the only edges involving s or t connect s to L and R to t. Note that although edges in G' can go only from L to R, edges in G'_f can also go from R to L.

Thus any augmenting path must go

$$s \to L \to R \to \cdots \to L \to R \to t ,$$

crossing back and forth between L and R at most as many times as it can do so without using a vertex twice. It contains s, t, and equal numbers of distinct vertices from L and R—at most $2 + 2 \cdot \min(|L|, |R|)$ vertices in all. The length of an augmenting path (i.e., its number of edges) is thus bounded above by $2 \cdot \min(|L|, |R|) + 1$.

Solution to Problem 26-4

a. Just execute one iteration of the Ford-Fulkerson algorithm. The edge (u, v) in E with increased capacity ensures that the edge (u, v) is in the residual network. So look for an augmenting path and update the flow if a path is found.

Time

$O(V + E) = O(E)$ if we find the augmenting path with either depth-first or breadth-first search.

To see that only one iteration is needed, consider separately the cases in which (u, v) is or is not an edge that crosses a minimum cut. If (u, v) does not cross a minimum cut, then increasing its capacity does not change the capacity of any minimum cut, and hence the value of the maximum flow does not change. If (u, v) does cross a minimum cut, then increasing its capacity by 1 increases the capacity of that minimum cut by 1, and hence possibly the value of the maximum flow by 1. In this case, there is either no augmenting path (in which case there was some other minimum cut that (u, v) does not cross), or the augmenting path increases flow by 1. No matter what, one iteration of Ford-Fulkerson suffices.

b. Let f be the maximum flow before reducing $c(u, v)$.

If $f(u, v) = 0$, we don't need to do anything.

If $f(u, v) > 0$, we will need to update the maximum flow. Assume from now on that $f(u, v) > 0$, which in turn implies that $f(u, v) \geq 1$.

Define $f'(x, y) = f(x, y)$ for all $x, y \in V$, except that $f'(u, v) = f(u, v) - 1$. Although f' obeys all capacity contraints, even after $c(u, v)$ has been reduced, it is not a legal flow, as it violates flow conservation at u (unless $u = s$) and v (unless $v = t$). f' has one more unit of flow entering u than leaving u, and it has one more unit of flow leaving v than entering v.

The idea is to try to reroute this unit of flow so that it goes out of u and into v via some other path. If that is not possible, we must reduce the flow from s to u and from v to t by one unit.

Look for an augmenting path from u to v (note: *not* from s to t).

- If there is such a path, augment the flow along that path.
- If there is no such path, reduce the flow from s to u by augmenting the flow from u to s. That is, find an augmenting path $u \leadsto s$ and augment the flow along that path. (There definitely is such a path, because there is flow from s to u.) Similarly, reduce the flow from v to t by finding an augmenting path $t \leadsto v$ and augmenting the flow along that path.

Time

$O(V + E) = O(E)$ if we find the paths with either DFS or BFS.