Exercise 5.3-7

5.3-7

Suppose we want to create a ***random sample*** of the set $\{1, 2, 3, \ldots, n\}$, that is, an m-element subset S, where $0 \le m \le n$, such that each m-subset is equally likely to be created. One way would be to set $A[i] = i$ for $i = 1, 2, 3, \ldots, n$, call RANDOMIZE-IN-PLACE(A), and then take just the first m array elements. This method would make n calls to the RANDOM procedure. If n is much larger than m, we can create a random sample with fewer calls to RANDOM. Show that the following recursive procedure returns a random m-subset S of $\{1, 2, 3, \ldots, n\}$, in which each m-subset is equally likely, while making only m calls to RANDOM:

```
RANDOM-SAMPLE(m, n)
1  if m == 0
2      return ∅
3  else S = RANDOM-SAMPLE(m − 1, n − 1)
4      i = RANDOM(1, n)
5      if i ∈ S
6          S = S ∪ {n}
7      else S = S ∪ {i}
8      return S
```

sol)

Since each recursive call reduces m by 1 and makes only one call to RANDOM, it's easy to see that there are a total of m calls to RANDOM. Moreover, since each recursive call adds exactly one element to the set, it's easy to see that the resulting set S contains exactly m elements.

Because the elements of set S are chosen independently of each other, it suffices to show that each of the n values appears in S with probability m/n. We use an inductive proof. The inductive hypothesis is that a call to RANDOM-SUBSET(m, n) returns a set S of m elements, each appearing with probability m/n. The base cases are for $m = 0$ and $m = 1$. When $m = 0$, the returned set is empty, and so it contains each element with probability 0. When $m = 1$, the returned set has one element, and it is equally likely to be any number in $\{1, 2, 3, \ldots, n\}$.

For the inductive step, we assume that the call RANDOM-SUBSET$(m - 1, n - 1)$ returns a set S' of $m - 1$ elements in which each value in $\{1, 2, 3, \ldots, n - 1\}$ occurs with probability $(m - 1)/(n - 1)$. After the line $i =$ RANDOM$(1, n)$, i is equally likely to be any value in $\{1, 2, 3, \ldots, n\}$. We consider separately the probabilities

that S contains $j < n$ and that S contains n. Let R_j be the event that the call RANDOM$(1, n)$ returns j, so that $\Pr\{R_j\} = 1/n$.

For $j < n$, the event that $j \in S$ is the union of two disjoint events:

- $j \in S'$, and
- $j \notin S'$ and R_j (these events are independent),

Thus,

$$\begin{aligned}
&\Pr\{j \in S\} \\
&\quad = \Pr\{j \in S'\} + \Pr\{j \notin S' \text{ and } R_j\} && \text{(the events are disjoint)} \\
&\quad = \frac{m-1}{n-1} + \left(1 - \frac{m-1}{n-1}\right) \cdot \frac{1}{n} && \text{(by the inductive hypothesis)} \\
&\quad = \frac{m-1}{n-1} + \left(\frac{n-1}{n-1} - \frac{m-1}{n-1}\right) \cdot \frac{1}{n} \\
&\quad = \frac{m-1}{n-1} \cdot \frac{n}{n} + \frac{n-m}{n-1} \cdot \frac{1}{n} \\
&\quad = \frac{(m-1)n + (n-m)}{(n-1)n} \\
&\quad = \frac{mn - n + n - m}{(n-1)n} \\
&\quad = \frac{m(n-1)}{(n-1)n} \\
&\quad = \frac{m}{n} .
\end{aligned}$$

The event that $n \in S$ is also the union of two disjoint events:

- R_n, and
- R_j and $j \in S'$ for some $j < n$ (these events are independent).

Thus,

$$\begin{aligned}
&\Pr\{n \in S\} \\
&\quad = \Pr\{R_n\} + \Pr\{R_j \text{ and } j \in S' \text{ for some } j < n\} && \text{(the events are disjoint)} \\
&\quad = \frac{1}{n} + \frac{n-1}{n} \cdot \frac{m-1}{n-1} && \text{(by the inductive hypothesis)} \\
&\quad = \frac{1}{n} \cdot \frac{n-1}{n-1} + \frac{n-1}{n} \cdot \frac{m-1}{n-1} \\
&\quad = \frac{n - 1 + nm - n - m + 1}{n(n-1)} \\
&\quad = \frac{nm - m}{n(n-1)} \\
&\quad = \frac{m(n-1)}{n(n-1)} \\
&\quad = \frac{m}{n} .
\end{aligned}$$

Exercise 5.4-6

5.4-6 ★

Suppose that n balls are tossed into n bins, where each toss is independent and the ball is equally likely to end up in any bin. What is the expected number of empty bins? What is the expected number of bins with exactly one ball?

sol)

First we determine the expected number of empty bins. We define a random variable X to be the number of empty bins, so that we want to compute $\mathrm{E}[X]$. Next, for $i = 1, 2, \ldots, n$, we define the indicator random variable $Y_i = \mathrm{I}\{\text{bin } i \text{ is empty}\}$. Thus,

$$X = \sum_{i=1}^{n} Y_i \ ,$$

and so

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} Y_i\right] \\
&= \sum_{i=1}^{n} \mathrm{E}[Y_i] \qquad \text{(by linearity of expectation)} \\
&= \sum_{i=1}^{n} \Pr\{\text{bin } i \text{ is empty}\} \quad \text{(by Lemma 5.1)} \ .
\end{aligned}$$

Let us focus on a specific bin, say bin i. We view a toss as a success if it misses bin i and as a failure if it lands in bin i. We have n independent Bernoulli trials, each with probability of success $1 - 1/n$. In order for bin i to be empty, we need n successes in n trials. Using a binomial distribution, therefore, we have that

$$\begin{aligned}
\Pr\{\text{bin } i \text{ is empty}\} &= \binom{n}{n}\left(1 - \frac{1}{n}\right)^{n}\left(\frac{1}{n}\right)^{0} \\
&= \left(1 - \frac{1}{n}\right)^{n} \ .
\end{aligned}$$

Thus,

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{i=1}^{n}\left(1 - \frac{1}{n}\right)^{n} \\
&= n\left(1 - \frac{1}{n}\right)^{n} \ .
\end{aligned}$$

By equation (3.14), as n approaches ∞, the quantity $(1 - 1/n)^n$ approaches $1/e$, and so $\mathrm{E}[X]$ approaches n/e.

Now we determine the expected number of bins with exactly one ball. We redefine X to be number of bins with exactly one ball, and we redefine Y_i to be $\mathrm{I}\{\text{bin } i \text{ gets exactly one ball}\}$. As before, we find that

$$\mathrm{E}[X] = \sum_{i=1}^{n} \Pr\{\text{bin } i \text{ gets exactly one ball}\} \ .$$

Again focusing on bin i, we need exactly $n-1$ successes in n independent Bernoulli trials, and so

$$\begin{aligned}\Pr\{\text{bin } i \text{ gets exactly one ball}\} &= \binom{n}{n-1}\left(1-\frac{1}{n}\right)^{n-1}\left(\frac{1}{n}\right)^{1} \\ &= n\cdot\left(1-\frac{1}{n}\right)^{n-1}\frac{1}{n} \\ &= \left(1-\frac{1}{n}\right)^{n-1},\end{aligned}$$

and so

$$\begin{aligned}\mathrm{E}[X] &= \sum_{i=1}^{n}\left(1-\frac{1}{n}\right)^{n-1} \\ &= n\left(1-\frac{1}{n}\right)^{n-1}.\end{aligned}$$

Because

$$n\left(1-\frac{1}{n}\right)^{n-1} = \frac{n\left(1-\frac{1}{n}\right)^{n}}{1-\frac{1}{n}},$$

as n approaches ∞, we find that $\mathrm{E}[X]$ approaches

$$\frac{n/e}{1-1/n} = \frac{n^2}{e(n-1)}.$$

Problem 5-1

5-1 Probabilistic counting

With a b-bit counter, we can ordinarily only count up to $2^b - 1$. With R. Morris's ***probabilistic counting***, we can count up to a much larger value at the expense of some loss of precision.

We let a counter value of i represent a count of n_i for $i = 0, 1, \ldots, 2^b - 1$, where the n_i form an increasing sequence of nonnegative values. We assume that the initial value of the counter is 0, representing a count of $n_0 = 0$. The INCREMENT operation works on a counter containing the value i in a probabilistic manner. If $i = 2^b - 1$, then the operation reports an overflow error. Otherwise, the INCREMENT operation increases the counter by 1 with probability $1/(n_{i+1} - n_i)$, and it leaves the counter unchanged with probability $1 - 1/(n_{i+1} - n_i)$.

If we select $n_i = i$ for all $i \geq 0$, then the counter is an ordinary one. More interesting situations arise if we select, say, $n_i = 2^{i-1}$ for $i > 0$ or $n_i = F_i$ (the ith Fibonacci number—see Section 3.2).

For this problem, assume that n_{2^b-1} is large enough that the probability of an overflow error is negligible.

a. Show that the expected value represented by the counter after n INCREMENT operations have been performed is exactly n.

b. The analysis of the variance of the count represented by the counter depends on the sequence of the n_i. Let us consider a simple case: $n_i = 100i$ for all $i \geq 0$. Estimate the variance in the value represented by the register after n INCREMENT operations have been performed.

sol)

a. To determine the expected value represented by the counter after n INCREMENT operations, we define some random variables:

- For $j = 1, 2, \ldots, n$, let X_j denote the increase in the value represented by the counter due to the jth INCREMENT operation.
- Let V_n be the value represented by the counter after n INCREMENT operations.

Then $V_n = X_1 + X_2 + \cdots + X_n$. We want to compute $\mathrm{E}[V_n]$. By linearity of expectation,

$$\mathrm{E}[V_n] = \mathrm{E}[X_1 + X_2 + \cdots + X_n] = \mathrm{E}[X_1] + \mathrm{E}[X_2] + \cdots + \mathrm{E}[X_n] \ .$$

We shall show that $\mathrm{E}[X_j] = 1$ for $j = 1, 2, \ldots, n$, which will prove that $\mathrm{E}[V_n] = n$.

We actually show that $\mathrm{E}[X_j] = 1$ in two ways, the second more rigorous than the first:

1. Suppose that at the start of the jth INCREMENT operation, the counter holds the value i, which represents n_i. If the counter increases due to this INCREMENT operation, then the value it represents increases by $n_{i+1} - n_i$. The counter increases with probability $1/(n_{i+1} - n_i)$, and so

$$\begin{aligned}
\mathrm{E}[X_j] &= (0 \cdot \Pr\{\text{counter does not increase}\}) \\
&\quad + ((n_{i+1} - n_i) \cdot \Pr\{\text{counter increases}\}) \\
&= \left(0 \cdot \left(1 - \frac{1}{n_{i+1} - n_i}\right)\right) + \left((n_{i+1} - n_i) \cdot \frac{1}{n_{i+1} - n_i}\right) \\
&= 1 \,,
\end{aligned}$$

and so $\mathrm{E}[X_j] = 1$ regardless of the value held by the counter.

2. Let C_j be the random variable denoting the value held in the counter at the start of the jth INCREMENT operation. Since we can ignore values of C_j greater than $2^b - 1$, we use a formula for conditional expectation:

$$\begin{aligned}
\mathrm{E}[X_j] &= \mathrm{E}[\mathrm{E}[X_j \mid C_j]] \\
&= \sum_{i=0}^{2^b-1} \mathrm{E}[X_j \mid C_j = i] \cdot \Pr\{C_j = i\} \,.
\end{aligned}$$

To compute $\mathrm{E}[X_j \mid C_j = i]$, we note that

- $\Pr\{X_j = 0 \mid C_j = i\} = 1 - 1/(n_{i+1} - n_i)$,
- $\Pr\{X_j = n_{i+1} - n_i \mid C_j = i\} = 1/(n_{i+1} - n_i)$, and
- $\Pr\{X_j = k \mid C_j = i\} = 0$ for all other k.

Thus,

$$\begin{aligned}
\mathrm{E}[X_j \mid C_j = i] &= \sum_k k \cdot \Pr\{X_j = k \mid C_j = i\} \\
&= \left(0 \cdot \left(1 - \frac{1}{n_{i+1} - n_i}\right)\right) + \left((n_{i+1} - n_i) \cdot \frac{1}{n_{i+1} - n_i}\right) \\
&= 1 \,.
\end{aligned}$$

Therefore, noting that

$$\sum_{i=0}^{2^b-1} \Pr\{C_j = i\} = 1 \,,$$

we have

$$\begin{aligned}
\mathrm{E}[X_j] &= \sum_{i=0}^{2^b-1} 1 \cdot \Pr\{C_j = i\} \\
&= 1 \,.
\end{aligned}$$

Why is the second way more rigorous than the first? Both ways condition on the value held in the counter, but only the second way incorporates the conditioning into the expression for $\mathrm{E}[X_j]$.

b. Defining V_n and X_j as in part (a), we want to compute $\mathrm{Var}[V_n]$, where $n_i = 100i$. The X_j are pairwise independent, and so by equation (C.29), $\mathrm{Var}[V_n] = \mathrm{Var}[X_1] + \mathrm{Var}[X_2] + \cdots + \mathrm{Var}[X_n]$.

Since $n_i = 100i$, we see that $n_{i+1} - n_i = 100(i+1) - 100i = 100$. Therefore, with probability 99/100, the increase in the value represented by the counter due to the jth INCREMENT operation is 0, and with probability 1/100, the

value represented increases by 100. Thus, by equation (C.27),

$$\begin{aligned}\mathrm{Var}[X_j] &= \mathrm{E}\left[X_j^2\right] - \mathrm{E}^2[X_j] \\ &= \left(\left(0^2 \cdot \frac{99}{100}\right) + \left(100^2 \cdot \frac{1}{100}\right)\right) - 1^2 \\ &= 100 - 1 \\ &= 99\,.\end{aligned}$$

Summing up the variances of the X_j gives $\mathrm{Var}[V_n] = 99n$.

Exercise 6.3-3

6.3-3

Show that there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in any n-element heap.

sol)

Let H be the height of the heap.

Two subtleties to beware of:

- Be careful not to confuse the height of a node (longest distance from a leaf) with its depth (distance from the root).
- If the heap is not a complete binary tree (bottom level is not full), then the nodes at a given level (depth) don't all have the same height. For example, although all nodes at depth H have height 0, nodes at depth $H-1$ can have either height 0 or height 1.

For a complete binary tree, it's easy to show that there are $\lceil n/2^{h+1} \rceil$ nodes of height h. But the proof for an incomplete tree is tricky and is not derived from the proof for a complete tree.

Proof By induction on h.

Basis: Show that it's true for $h = 0$ (i.e., that # of leaves $\leq \lceil n/2^{h+1} \rceil = \lceil n/2 \rceil$). In fact, we'll show that the # of leaves $= \lceil n/2 \rceil$.

The tree leaves (nodes at height 0) are at depths H and $H-1$. They consist of

- all nodes at depth H, and
- the nodes at depth $H-1$ that are not parents of depth-H nodes.

Let x be the number of nodes at depth H—that is, the number of nodes in the bottom (possibly incomplete) level.

Note that $n-x$ is odd, because the $n-x$ nodes above the bottom level form a complete binary tree, and a complete binary tree has an odd number of nodes (1 less than a power of 2). Thus if n is odd, x is even, and if n is even, x is odd.

To prove the base case, we must consider separately the case in which n is even (x is odd) and the case in which n is odd (x is even). Here are two ways to do this: The first requires more cleverness, and the second requires more algebraic manipulation.

1. First method of proving the base case:

   - If n is odd, then x is even, so all nodes have siblings—i.e., all internal nodes have 2 children. Thus (see Exercise B.5-3), # of internal nodes $=$ # of leaves $- 1$.
     So, $n =$ # of nodes $=$ # of leaves $+$ # of internal nodes $= 2 \cdot$ # of leaves $- 1$.
     Thus, # of leaves $= (n+1)/2 = \lceil n/2 \rceil$. (The latter equality holds because n is odd.)
   - If n is even, then x is odd, and some leaf doesn't have a sibling. If we gave it a sibling, we would have $n + 1$ nodes, where $n + 1$ is odd, so the case we analyzed above would apply. Observe that we would also increase the number of leaves by 1, since we added a node to a parent that already had a child. By the odd-node case above, # of leaves $+ 1 = \lceil (n + 1)/2 \rceil = \lceil n/2 \rceil + 1$. (The latter equality holds because n is even.)

   In either case, # of leaves $= \lceil n/2 \rceil$.

2. Second method of proving the base case:

   Note that at any depth $d < H$ there are 2^d nodes, because all such tree levels are complete.

   - If x is even, there are $x/2$ nodes at depth $H - 1$ that are parents of depth H nodes, hence $2^{H-1} - x/2$ nodes at depth $H - 1$ that are not parents of depth-H nodes. Thus,

     $$\begin{aligned} \text{total \# of height-0 nodes} &= x + 2^{H-1} - x/2 \\ &= 2^{H-1} + x/2 \\ &= (2^H + x)/2 \\ &= \lceil (2^H + x - 1)/2 \rceil \qquad \text{(because } x \text{ is even)} \\ &= \lceil n/2 \rceil \ . \end{aligned}$$

     ($n = 2^H + x - 1$ because the complete tree down to depth $H - 1$ has $2^H - 1$ nodes and depth H has x nodes.)
   - If x is odd, by an argument similar to the even case, we see that

     $$\begin{aligned} \text{\# of height-0 nodes} &= x + 2^{H-1} - (x + 1)/2 \\ &= 2^{H-1} + (x - 1)/2 \\ &= (2^H + x - 1)/2 \\ &= n/2 \\ &= \lceil n/2 \rceil \qquad \text{(because } x \text{ odd} \Rightarrow n \text{ even)} \, . \end{aligned}$$

Inductive step: Show that if it's true for height $h - 1$, it's true for h.

Let n_h be the number of nodes at height h in the n-node tree T.

Consider the tree T' formed by removing the leaves of T. It has $n' = n - n_0$ nodes. We know from the base case that $n_0 = \lceil n/2 \rceil$, so $n' = n - n_0 = n - \lceil n/2 \rceil = \lfloor n/2 \rfloor$.

Note that the nodes at height h in T would be at height $h - 1$ if the leaves of the tree were removed—that is, they are at height $h - 1$ in T'. Letting n'_{h-1} denote the number of nodes at height $h - 1$ in T', we have

$$n_h = n'_{h-1} .$$

By induction, we can bound n'_{h-1}:

$$n_h = n'_{h-1} \le \lceil n'/2^h \rceil = \lceil \lfloor n/2 \rfloor / 2^h \rceil \le \lceil (n/2)/2^h \rceil = \lceil n/2^{h+1} \rceil . \quad \blacksquare$$

Alternative solution

An alternative solution relies on four facts:

1. Every node *not* on the unique simple path from the last leaf to the root is the root of a complete binary subtree.
2. A node that is the root of a complete binary subtree and has height h is the ancestor of 2^h leaves.
3. By Exercise 6.1-7, an n-element heap has $\lceil n/2 \rceil$ leaves.
4. For nonnegative reals a and b, we have $\lceil a \rceil \cdot b \ge \lceil ab \rceil$.

The proof is by contradiction. Assume that an n-element heap contains at least $\lceil n/2^{h+1} \rceil + 1$ nodes of height h. Exactly one node of height h is on the unique simple path from the last leaf to the root, and the subtree rooted at this node has at least one leaf (that being the last leaf). All other nodes of height h, of which the heap contains at least $\lceil n/2^{h+1} \rceil$, are the roots of complete binary subtrees, and each such node is the root of a subtree with 2^h leaves. Moreover, each subtree whose root is at height h is disjoint. Therefore, the number of leaves in the entire heap is at least

$$\begin{aligned} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot 2^h + 1 &\ge \left\lceil \frac{n}{2^{h+1}} \cdot 2^h \right\rceil + 1 \\ &= \left\lceil \frac{n}{2} \right\rceil + 1 , \end{aligned}$$

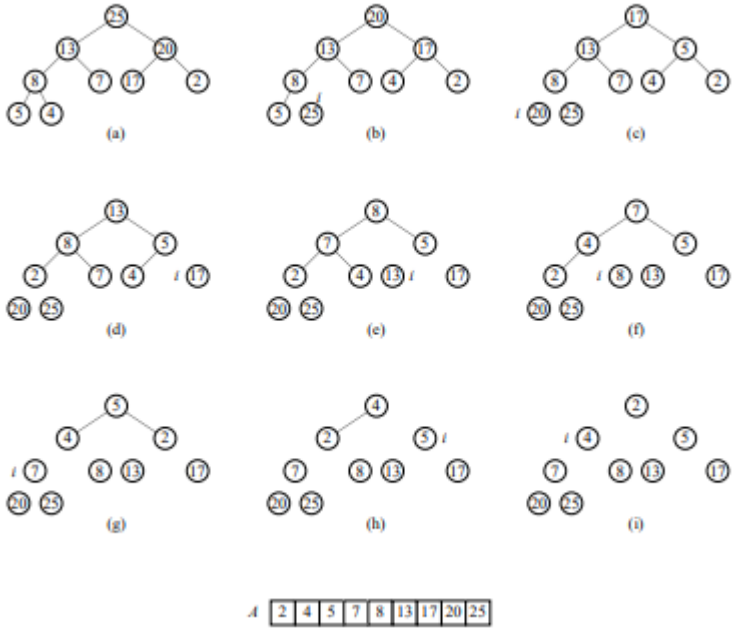
which contradicts the property that an n-element heap has $\lceil n/2 \rceil$ leaves. $\blacksquare$

Exercise 6.4-1

6.4-1

Using Figure 6.4 as a model, illustrate the operation of HEAPSORT on the array $A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle$.

sol)



Exercise 6.5-6

6.5-6

Each exchange operation on line 5 of HEAP-INCREASE-KEY typically requires three assignments. Show how to use the idea of the inner loop of INSERTION-SORT to reduce the three assignments down to just one assignment.

sol)

Change the procedure to the following:

```
HEAP-INCREASE-KEY(A, i, key)
 if key < A[i]
     error "new key is smaller than current key"
 A[i] = key
 while i > 1 and A[PARENT(i)] < A[i]
     A[i] = A[PARENT(i)]
     i = PARENT(i)
 A[i] = key
```

Problem 6-1

6-1 Building a heap using insertion

We can build a heap by repeatedly calling MAX-HEAP-INSERT to insert the elements into the heap. Consider the following variation on the BUILD-MAX-HEAP procedure:

BUILD-MAX-HEAP′(A)

1 $A.heap\text{-}size = 1$
2 **for** $i = 2$ **to** $A.length$
3 MAX-HEAP-INSERT($A, A[i]$)

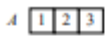
a. Do the procedures BUILD-MAX-HEAP and BUILD-MAX-HEAP′ always create the same heap when run on the same input array? Prove that they do, or provide a counterexample.

b. Show that in the worst case, BUILD-MAX-HEAP′ requires $\Theta(n \lg n)$ time to build an n-element heap.

sol)

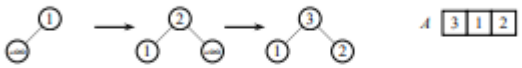
a. The procedures BUILD-MAX-HEAP and BUILD-MAX-HEAP′ do not always create the same heap when run on the same input array. Consider the following counterexample.

Input array A:



BUILD-MAX-HEAP(A):

BUILD-MAX-HEAP′(A):



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.

Problem 7-2

7-2 *Quicksort with equal element values*

The analysis of the expected running time of randomized quicksort in Section 7.4.2 assumes that all element values are distinct. In this problem, we examine what happens when they are not.

a. Suppose that all element values are equal. What would be randomized quicksort's running time in this case?

b. The PARTITION procedure returns an index q such that each element of $A[p..q-1]$ is less than or equal to $A[q]$ and each element of $A[q+1..r]$ is greater than $A[q]$. Modify the PARTITION procedure to produce a procedure PARTITION$'(A, p, r)$, which permutes the elements of $A[p..r]$ and returns two indices q and t, where $p \le q \le t \le r$, such that

- all elements of $A[q..t]$ are equal,
- each element of $A[p..q-1]$ is less than $A[q]$, and
- each element of $A[t+1..r]$ is greater than $A[q]$.

Like PARTITION, your PARTITION$'$ procedure should take $\Theta(r-p)$ time.

c. Modify the RANDOMIZED-PARTITION procedure to call PARTITION$'$, and name the new procedure RANDOMIZED-PARTITION$'$. Then modify the QUICKSORT procedure to produce a procedure QUICKSORT$'(A, p, r)$ that calls RANDOMIZED-PARTITION$'$ and recurses only on partitions of elements not known to be equal to each other.

d. Using QUICKSORT$'$, how would you adjust the analysis in Section 7.4.2 to avoid the assumption that all elements are distinct?

sol)

a. If all elements are equal, then when PARTITION returns, $q = r$ and all elements in $A[p..q-1]$ are equal. We get the recurrence $T(n) = T(n-1)+T(0)+\Theta(n)$ for the running time, and so $T(n) = \Theta(n^2)$.

b. The PARTITION$'$ procedure:

```
PARTITION'(A, p, r)
  x = A[p]
  i = h = p
  for j = p + 1 to r
      // Invariant: A[p..i-1] < x, A[i..h] = x,
      //            A[h+1..j-1] > x, A[j..r] unknown.
      if A[j] < x
          y = A[j]
          A[j] = A[h+1]
          A[h+1] = A[i]
          A[i] = y
          i = i + 1
          h = h + 1
      elseif A[j] == x
          exchange A[h+1] with A[j]
          h = h + 1
  return (i, h)
```

c. RANDOMIZED-PARTITION$'$ is the same as RANDOMIZED-PARTITION, but with the call to PARTITION replaced by a call to PARTITION$'$.

```
QUICKSORT'(A, p, r)
  if p < r
      (q, t) = RANDOMIZED-PARTITION'(A, p, r)
      QUICKSORT'(A, p, q - 1)
      QUICKSORT'(A, t + 1, r)
```

d. Putting elements equal to the pivot in the same partition as the pivot can only help us, because we do not recurse on elements equal to the pivot. Thus, the subproblem sizes with QUICKSORT$'$, even with equal elements, are no larger than the subproblem sizes with QUICKSORT when all elements are distinct.

Problem 7-4

7-4 Stack depth for quicksort

The QUICKSORT algorithm of Section 7.1 contains two recursive calls to itself. After QUICKSORT calls PARTITION, it recursively sorts the left subarray and then it recursively sorts the right subarray. The second recursive call in QUICKSORT is not really necessary; we can avoid it by using an iterative control structure. This technique, called ***tail recursion***, is provided automatically by good compilers. Consider the following version of quicksort, which simulates tail recursion:

```
TAIL-RECURSIVE-QUICKSORT(A, p, r)
1  while p < r
2      // Partition and sort left subarray.
3      q = PARTITION(A, p, r)
4      TAIL-RECURSIVE-QUICKSORT(A, p, q − 1)
5      p = q + 1
```

a. Argue that TAIL-RECURSIVE-QUICKSORT$(A, 1, A.length)$ correctly sorts the array A.

Compilers usually execute recursive procedures by using a ***stack*** that contains pertinent information, including the parameter values, for each recursive call. The information for the most recent call is at the top of the stack, and the information for the initial call is at the bottom. Upon calling a procedure, its information is ***pushed*** onto the stack; when it terminates, its information is ***popped***. Since we assume that array parameters are represented by pointers, the information for each procedure call on the stack requires $O(1)$ stack space. The ***stack depth*** is the maximum amount of stack space used at any time during a computation.

b. Describe a scenario in which TAIL-RECURSIVE-QUICKSORT's stack depth is $\Theta(n)$ on an n-element input array.

c. Modify the code for TAIL-RECURSIVE-QUICKSORT so that the worst-case stack depth is $\Theta(\lg n)$. Maintain the $O(n \lg n)$ expected running time of the algorithm.

sol)

a. QUICKSORT′ does exactly what QUICKSORT does; hence it sorts correctly. QUICKSORT and QUICKSORT′ do the same partitioning, and then each calls itself with arguments $A, p, q - 1$. QUICKSORT then calls itself again, with arguments $A, q + 1, r$. QUICKSORT′ instead sets $p = q + 1$ and performs another iteration of its **while** loop. This executes the same operations as calling itself with $A, q + 1, r$, because in both cases, the first and third arguments (A and r) have the same values as before, and p has the old value of $q + 1$.

b. The stack depth of QUICKSORT′ will be $\Theta(n)$ on an n-element input array if there are $\Theta(n)$ recursive calls to QUICKSORT′. This happens if every call to PARTITION(A, p, r) returns $q = r$. The sequence of recursive calls in this scenario is

QUICKSORT′$(A, 1, n)$,
QUICKSORT′$(A, 1, n-1)$,
QUICKSORT′$(A, 1, n-2)$,
$\vdots$
QUICKSORT′$(A, 1, 1)$.

Any array that is already sorted in increasing order will cause QUICKSORT′ to behave this way.

c. The problem demonstrated by the scenario in part (b) is that each invocation of QUICKSORT′ calls QUICKSORT′ again with almost the same range. To avoid such behavior, we must change QUICKSORT′ so that the recursive call is on a smaller interval of the array. The following variation of QUICKSORT′ checks which of the two subarrays returned from PARTITION is smaller and recurses on the smaller subarray, which is at most half the size of the current array. Since the array size is reduced by at least half on each recursive call, the number of recursive calls, and hence the stack depth, is $\Theta(\lg n)$ in the worst case. Note that this method works no matter how partitioning is performed (as long as the PARTITION procedure has the same functionality as the procedure given in Section 7.1).

```
QUICKSORT″(A, p, r)
  while p < r
      // Partition and sort the small subarray first.
      q = PARTITION(A, p, r)
      if q − p < r − q
          QUICKSORT″(A, p, q − 1)
          p = q + 1
      else QUICKSORT″(A, q + 1, r)
          r = q − 1
```

The expected running time is not affected, because exactly the same work is done as before: the same partitions are produced, and the same subarrays are sorted.