## Solution to Exercise 2.3-3

The base case is when $n = 2$, and we have $n \lg n = 2 \lg 2 = 2 \cdot 1 = 2$.

For the inductive step, our inductive hypothesis is that $T(n/2) = (n/2)\lg(n/2)$. Then

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2(n/2)\lg(n/2) + n \\ &= n(\lg n - 1) + n \\ &= n \lg n - n + n \\ &= n \lg n \,, \end{aligned}$$

which completes the inductive proof for exact powers of 2.

## Solution to Exercise 2.3-5

***This solution is also posted publicly***

Procedure BINARY-SEARCH takes a sorted array $A$, a value $v$, and a range $[low\,..\,high]$ 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 $A[low\,..\,high]$ 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 ≤ high
    mid = ⌊(low + high)/2⌋
    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 = ⌊(low + high)/2⌋
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 Exercise 2.3-7

The following algorithm solves the problem:

1. Sort the elements in $S$.
2. Form the set $S' = \{z : z = x - y \text{ for some } y \in S\}$.
3. Sort the elements in $S'$.
4. If any value in $S$ appears more than once, remove all but one instance. Do the same for $S'$.
5. Merge the two sorted sets $S$ and $S'$.
6. There exist two elements in $S$ whose sum is exactly $x$ if and only if the same value appears in consecutive positions in the merged output.

To justify the claim in step 4, first observe that if any value appears twice in the merged output, it must appear in consecutive positions. Thus, we can restate the condition in step 5 as there exist two elements in $S$ whose sum is exactly $x$ if and only if the same value appears twice in the merged output.

Suppose that some value $w$ appears twice. Then $w$ appeared once in $S$ and once in $S'$. Because $w$ appeared in $S'$, there exists some $y \in S$ such that $w = x - y$, or $x = w + y$. Since $w \in S$, the elements $w$ and $y$ are in $S$ and sum to $x$.

Conversely, suppose that there are values $w, y \in S$ such that $w + y = x$. Then, since $x - y = w$, the value $w$ appears in $S'$. Thus, $w$ is in both $S$ and $S'$, and so it will appear twice in the merged output.

Steps 1 and 3 require $O(n \lg n)$ steps. Steps 2, 4, 5, and 6 require $O(n)$ steps. Thus the overall running time is $O(n \lg n)$.

## Solution to Problem 2-4

***This solution is also posted publicly***

*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 $A[p..q]$ to $L$ and $A[q+1..r]$ 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 $L$

```
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] = ∞
R[n_2 + 1] = ∞
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] ≤ 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 $\infty$ in $R$, since no value in $L$ will be greater than $\infty$.

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: $\Theta(n \lg n)$.

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 = ⌊(p + r)/2⌋
    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
```

## Solution to Exercise 3.1-1

First, let's clarify what the function $\max(f(n), g(n))$ is. Let's define the function $h(n) = \max(f(n), g(n))$. Then

$$h(n) = \begin{cases} f(n) & \text{if } f(n) \geq g(n) , \\ g(n) & \text{if } f(n) < g(n) . \end{cases}$$

Since $f(n)$ and $g(n)$ are asymptotically nonnegative, there exists $n_0$ such that $f(n) \geq 0$ and $g(n) \geq 0$ for all $n \geq n_0$. Thus for $n \geq n_0$, $f(n) + g(n) \geq f(n) \geq 0$ and $f(n) + g(n) \geq g(n) \geq 0$. Since for any particular $n$, $h(n)$ is either $f(n)$ or $g(n)$, we have $f(n) + g(n) \geq h(n) \geq 0$, which shows that $h(n) = \max(f(n), g(n)) \leq c_2(f(n) + g(n))$ for all $n \geq n_0$ (with $c_2 = 1$ in the definition of $\Theta$).

Similarly, since for any particular $n$, $h(n)$ is the larger of $f(n)$ and $g(n)$, we have for all $n \geq n_0$, $0 \leq f(n) \leq h(n)$ and $0 \leq g(n) \leq h(n)$. Adding these two inequalities yields $0 \leq f(n) + g(n) \leq 2h(n)$, or equivalently $0 \leq (f(n) + g(n))/2 \leq h(n)$, which shows that $h(n) = \max(f(n), g(n)) \geq c_1(f(n)+g(n))$ for all $n \geq n_0$ (with $c_1 = 1/2$ in the definition of $\Theta$).

## Solution to Exercise 3.1-4

***This solution is also posted publicly***

$2^{n+1} = O(2^n)$, 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

$0 \leq 2^{n+1} \leq c \cdot 2^n$ 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

$0 \leq 2^{2n} \leq c \cdot 2^n$ 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 Problem 4-1

Note: In parts (a), (b), and (d) below, we are applying case 3 of the master theorem, which requires the regularity condition that $af(n/b) \leq cf(n)$ for some constant $c < 1$. In each of these parts, $f(n)$ has the form $n^k$. The regularity condition is satisfied because $af(n/b) = an^k/b^k = (a/b^k)n^k = (a/b^k)f(n)$, and in each of the cases below, $a/b^k$ is a constant strictly less than 1.

*a.* $T(n) = 2T(n/2) + n^3 = \Theta(n^3)$. This is a divide-and-conquer recurrence with $a = 2$, $b = 2$, $f(n) = n^3$, and $n^{\log_b a} = n^{\log_2 2} = n$. Since $n^3 = \Omega(n^{\log_2 2+2})$ and $a/b^k = 2/2^3 = 1/4 < 1$, case 3 of the master theorem applies, and $T(n) = \Theta(n^3)$.

*b.* $T(n) = T(9n/10) + n = \Theta(n)$. This is a divide-and-conquer recurrence with $a = 1$, $b = 10/9$, $f(n) = n$, and $n^{\log_b a} = n^{\log_{10/9} 1} = n^0 = 1$. Since $n = \Omega(n^{\log_{10/9} 1+1})$ and $a/b^k = 1/(10/9)^1 = 9/10 < 1$, case 3 of the master theorem applies, and $T(n) = \Theta(n)$.

*c.* $T(n) = 16T(n/4) + n^2 = \Theta(n^2 \lg n)$. This is another divide-and-conquer recurrence with $a = 16$, $b = 4$, $f(n) = n^2$, and $n^{\log_b a} = n^{\log_4 16} = n^2$. Since $n^2 = \Theta(n^{\log_4 16})$, case 2 of the master theorem applies, and $T(n) = \Theta(n^2 \lg n)$.

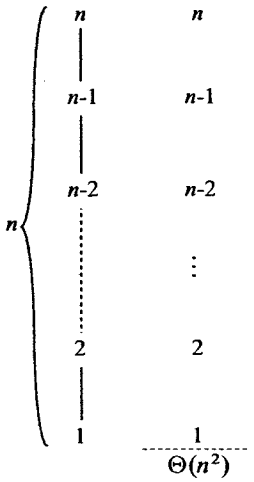
*d.* $T(n) = 7T(n/3) + n^2 = \Theta(n^2)$. This is a divide-and-conquer recurrence with $a = 7$, $b = 3$, $f(n) = n^2$, and $n^{\log_b a} = n^{\log_3 7}$. Since $1 < \log_3 7 < 2$, we have that $n^2 = \Omega(n^{\log_3 7 + \epsilon})$ for some constant $\epsilon > 0$. We also have $a/b^k = 7/3^2 = 7/9 < 1$, so that case 3 of the master theorem applies, and $T(n) = \Theta(n^2)$.

*e.* $T(n) = 7T(n/2) + n^2 = O(n^{\lg 7})$. This is a divide-and-conquer recurrence with $a = 7$, $b = 2$, $f(n) = n^2$, and $n^{\log_b a} = n^{\log_2 7}$. Since $2 < \lg 7 < 3$, we have that $n^2 = O(n^{\log_2 7 - \epsilon})$ for some constant $\epsilon > 0$. Thus, case 1 of the master theorem applies, and $T(n) = \Theta(n^{\lg 7})$.

*f.* $T(n) = 2T(n/4) + \sqrt{n} = \Theta(\sqrt{n} \lg n)$. This is another divide-and-conquer recurrence with $a = 2$, $b = 4$, $f(n) = \sqrt{n}$, and $n^{\log_b a} = n^{\log_4 2} = \sqrt{n}$. Since $\sqrt{n} = \Theta(n^{\log_4 2})$, case 2 of the master theorem applies, and $T(n) = \Theta(\sqrt{n} \lg n)$.

*g.* $T(n) = T(n-1) + n$

Using the recursion tree shown below, we get a guess of $T(n) = \Theta(n^2)$.




First, we prove the $T(n) = \Omega(n^2)$ part by induction. The inductive hypothesis is $T(n) \geq cn^2$ for some constant $c > 0$.

$$\begin{aligned} T(n) &= T(n-1) + n \\ &\geq c(n-1)^2 + n \\ &= cn^2 - 2cn + c + n \\ &\geq cn^2 \end{aligned}$$

if $-2cn + n + c \geq 0$ or, equivalently, $n(1 - 2c) + c \geq 0$. This condition holds when $n \geq 0$ and $0 < c \leq 1/2$.

For the upper bound, $T(n) = O(n^2)$, we use the inductive hypothesis that $T(n) \leq cn^2$ for some constant $c > 0$. By a similar derivation, we get that $T(n) \leq cn^2$ if $-2cn + n + c \leq 0$ or, equivalently, $n(1 - 2c) + c \leq 0$. This condition holds for $c = 1$ and $n \geq 1$.

Thus, $T(n) = \Omega(n^2)$ and $T(n) = O(n^2)$, so we conclude that $T(n) = \Theta(n^2)$.

*h.* $T(n) = T(\sqrt{n}) + 1$

The easy way to do this is with a change of variables, as on page 86 of the text. Let $m = \lg n$ and $S(m) = T(2^m)$. $T(2^m) = T(2^{m/2}) + 1$, so $S(m) = S(m/2) + 1$. Using the master theorem, $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$ and $f(n) = 1$. Since $1 = \Theta(1)$, case 2 applies and $S(m) = \Theta(\lg m)$. Therefore, $T(n) = \Theta(\lg \lg n)$.

## Solution to Exercise 5.2-1

***This solution is also posted publicly***

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

***This solution is also posted publicly***

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 $\pi$ 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.