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 $A[1 \,..\, j-1]$ consists of the $j - 1$ smallest elements in the array $A[1 \,..\, n]$, and this subarray is in sorted order. After the first $n - 1$ elements, the subarray $A[1 \,..\, n-1]$ 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 $[low \,..\, high]$ 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 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 $A[p\,..\,q]$ 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 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
```

```
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 ∞ 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: $\Theta(n \lg n)$.