6-1 (page 142)

a) The two procedures do not always create the same heap.

b) Since MAX-HEAP-INSERT runs in O(log*n*) time and we are calling it *n* times, BUILD-MAX-HEAP' will run in O(*n*log*n*).

7-4 (page 161)

a) QUICKSORT' sorts the array correctly because it is simply a modified version of QUICKSORT, instead of calling QUICKSORT again, it loops back to the beginning of the method with a different value of p (for the part of the array to the right of the pivet).

b) This scenario will happen when the size of the array called to QUICKSORT' is always one less than the previous call, therefore requiring n calls. If the list is in sorted order than the stack depth will be $\Theta(n)$.

c) If we modify the code so that the call to QUICKSORT' is on the smaller of the two half arrays divided by the pivot, rather than automatically choosing the left sub-array we guarantee that the size will be <= *n*/2. This check can be done in constant time before the call to QUICKSORT' by using an if statement, and will ensure that the stack depth is $\Theta(\log n)$.

Design an algorithm that, given both an input array A containing n distinct positive integers and a positive integer target t, prints the pairs (A[i], A[j]) of array values with product t (i.e., with A[i] × A[j] = t). Your algorithm should run in O(nlogn) time. You should organize your answer into the following three parts. First, clearly state your algorithm(in English or in pseudocode). Second, prove that your algorithm is correct. And third, provide a proof of its worst-case running time.

Solution:

```
1. Sort the array using Heapsort or Mergesort, worst case O(nlogn).
2. i = 1; j = n
3. while i != j do  O(n)
     if A[i] * A[j] = t then print (A[i], A[j]) and i++ and j--
     if A[i] * A[j] > t then j--
     if A[i] * A[j] < t then i++
```

We sort the array and look at the largest and smallest numbers, if their product is equal to t, we output the two indices. Otherwise, if the product is bigger than t, that means we need smaller numbers, so we decrease j to get to a smaller number in the array. Otherwise, the product is less than t, we need a bigger number and we increase I to get to a bigger number.

It runs in O(*n*log*n* + *n*) = O(*n*log*n*)

3

Suppose you are given an array A with n entries, with each entry holding a distinct number. You are promised that the sequence of values A[1], A[2], . . . , A[n] is unimodal: For some index p between 1 and *n*, the values in the array entries increase up to position p in A, and then decrease the remainder of the way until position *n*. (So if you were to draw a plot with the array position j on the x-axis and the value of the entry A[j] on the y-axis, the plotted points would rise until x-value p, where they would achieve their maximum, and then fall from there on.) You would like to find the "peak entry p without having to read the entire array - in fact, by reading as few entries of A as possible. Give a divide and conquer algorithm that finds the peak entry of A in O(logn) time.

Specifically:

(a) Unambiguously describe your algorithm (in English or pseudocode). Your algorithm can behave arbitrarily on input arrays that are not unimodal, and you can assume that the array length n is a power of 2.
(b) Argue that your algorithm is correct.
(c) Prove that your algorithm runs in O(log*n*) time.

Solution:

Suppose the first and last elements of the array are A[start] and A[end]. Take the middle element of Array A at index p. If A[p] > A[p-1] then can ignore A[start..p-1]. If A[p] > A[p-1] then A[p] is the peak, otherwise we recurse on A[p+1]. If A[p] < A[p-1] then we have passed the peak and we can ignore A[p..end] and we recurse on A[start..p-1]. In the meanwhile we have to check the boundary situations.

The algorithm is similar to a binary search. At each step, we omit half the array, but in the process the peak is never omitted. Eventually we will be left with the peak element which is returned.

Like the binary search, the algorithm runs in O(log*n*).

We are given a sequence of *n* numbers < a_1 , a_2 , . . . , a_n > which we assume are all distinct, and we define an inversion to be a pair i < j such that $a_i > a_j$. It is generally assumed that the number of inversions in a sequence is a good measure of how far the sequence is from the sorted sequence. However, one might feel that this measure is too sensitive. Let us call a pair a significant inversion if i < j and $a_i > 2a_j$. Give an O(*n* log *n*) algorithm to count the number of significant inversions in a sequence.

Solution:

Let A be the array of *n* given integer numbers. Without any loss of generality, we assume that the elements a_i, *i* = 1, 2, …, *n* belong to the set {1, 2, …, *n*}. The algorithm for finding all the inversions (the question asks a slightly different type of inversions) can be stated as follows:

- Divide the array A into two halves: A[1..*n*/2] and A[*n*/2 + 1..*n*]
- Recursively find the inversions in A[1..*n*/2] and A[*n/*2+1..*n*]
- Find all the inversions between A[1..*n*/2] and A[*n*/2 + 1..*n*].
    - Sort the elements of A[*n*/2+1 .. *n*] and store it in B
    - Consider each element *i* of A[1*..n*/2]. Determine the number of elements of B less than *i*. This number is the number of inversions the element *i* has with the array A[*n*/2+1..*n*]. Clearly, we can find this in O(log*n*) time. Therefore the merging step requires *n*/2*O(log*n*) time. Hence the divide and conquer algorithm is O(*n*log^2*n*). Why?? Notice that for two elements *i* and *j* in A[1..*n*/2], i<j , the number of inversions i with A[*n*/2+1..*n*] is no more than that of *j*. Therefore, we can reduce the merge step to be linear by considering the elements of A[1..*n*/2] in increasing order of their values and determine the inversions in A[*n*/2+1..*n*] using linear search (the search for some *j* can start from where the previous inversion search has stopped.

5

Suppose you are given a list of n integers with many duplications, so that there are only O(log n) distinct numbers in the list.

(a) Show how to sort the numbers in O(n log log n) time.
(b) Why is this result not a violation of the comparison based sorting lower bound?

Solution:

a) If you use the mergesort algorithm, the process can subdivide a list at most O(loglog$n$) time after which all the elements of any list will have the same value. (log$n$ elements can be divided into halves O(loglog$n$) time.
b) The Ω($n$log$n$) bound for sorting assumes that all possible array of n values are possible inputs to the algorithm. Here the inputs are restricted to be arrays with only O(log$n$) different values. As an extreme case of the same restriction, if the inputs are restricted to be arrays with only a single value, the sorting can be done in linear time.