Midterm

Note

This is a closed-book exam. Each problem accounts for 10 points, unless otherwise marked.

Problems

1. Prove *by induction* that the sum of the heights of all nodes in a *full* binary tree of height h is $2^{h+1} - h - 2$ and that the sum equals $n - (h + 1)$, where n is the total number of nodes in the tree. (Note: a single-node tree has height 0.)

2. The *Partition* procedure for the Quicksort algorithm discussed in class is as follows, where *Middle* is a global variable.

```
Partition ($X, Left, Right$);
begin
    $pivot := X[left]$;
    $L := Left$;  $R := Right$;
    while $L < R$ do
        while $X[L] \leq pivot$ and $L \leq Right$ do $L := L + 1$;
        while $X[R] > pivot$ and $R \geq Left$ do $R := R - 1$;
        if $L < R$ then $swap(X[L], X[R])$;
    $Middle := R$;
    $swap(X[Left], X[Middle])$
end
```

   Find an adequate loop invariant for the main while loop, which is sufficient to show that after the execution of the last two assignment statements the array is properly partitioned by $X[Middle]$. Please express the loop invariant as precisely as possible, using mathematical notation.

3. Find the asymptotic behavior of the function $T(n)$ defined as follows:

$$\begin{cases} T(1) = 1 \\ T(n) = T(n/2) + \sqrt{n}, \quad n = 2^i \ (i \geq 1) \end{cases}$$

You should try to solve this problem without resorting to the general theorem for divide-and-conquer relations (see the Appendix) discussed in class. The asymptotic bound should be as tight as possible. (Hint: an effective way is to guess and verify by induction. You may need to try a few choices.)

4. Consider a max heap represented as the following array which may store a maximum of 15 elements.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
15	12	13	11	10	9	7	6	8	1	2	4	3	5	

   (a) Show the resulting heap after *Insert*(14).

   (b) Show the resulting heap after a *Remove*() operation (on the original heap).

5. Show all intermediate and the final AVL trees formed by inserting the numbers 3, 2, 1, 6, 5, and 4 (in this order) into an empty tree. Please use the following ordering convention: the key of an internal node is larger than that of its left child and smaller than that of its right child. If re-balancing operations are performed, please also show the tree before re-balancing and indicate what type of rotation is used in the re-balancing.

6. Let $x_1, x_2, \cdots, x_n$ be a sequence of real numbers (not necessarily positive). Design an $O(n)$ algorithm to find the subsequence $x_i, x_{i+1}, \cdots, x_j$ (of consecutive elements) such that the product of the numbers in it is maximum over all consecutive subsequences. The product of the empty subsequence is defined to be 1.

   Please present your algorithm in an adequate pseudo code and make assumptions wherever necessary. Explain the intuition behind your algorithm so that its correctness becomes clear.

7. The Knapsack Problem is defined as follows: Given a set S of n items, where the i-th item has an integer size $S[i]$, and an integer K, find a subset of the items whose sizes sum to exactly K or determine that no such subset exists.

   Now consider a variant where we want the subset to be as large as possible (i.e., to be with as many items as possible). How will you adapt the algorithm (see the Appendix) that we have studied in class? Your algorithm should collect at the end the items in one of the best solutions if they exist. Please present your algorithm in an adequate pseudo code and make assumptions wherever necessary (you may reuse the code for the original Knapsack Problem). Give an analysis of its time complexity. The more efficient your algorithm is, the more points you will get for this problem.

8. Consider an alternative algorithm for partition in the Quicksort algorithm:

```
**Partition** $(X, Left, Right)$;
**begin**
    $pivot := X[left]$;
    $i := Left$;
    **for** $j := Left + 1$ **to** $Right$ **do**
        **if** $X[j] < pivot$ **then** $i := i + 1$;
                                  $swap(X[i], X[j])$;
    $Middle := i$;
    $swap(X[Left], X[Middle])$
**end**
```

How does this algorithm compare to the algorithm we discussed in class? Please point out the advantages and the disadvantages of this alternative with adequate justification.

9. Apply the Quicksort algorithm to the following array. Show the contents of the array after each partition operation. If you use a different partition algorithm (from the one discussed in class), please describe it.

1	2	3	4	5	6	7	8	9	10	11	12
4	7	10	9	11	6	8	1	5	12	3	2

10. Your task is to design an *in-place* algorithm that sorts an array of numbers according to a prescribed order. The input is a sequence of n numbers x_1, x_2, $\cdots$, x_n and another sequence a_1, a_2, $\cdots$, a_n of n distinct numbers between 1 and n (i.e., a_1, a_2, $\cdots$, a_n is a permutation of 1, 2, $\cdots$, n), both represented as arrays. Your algorithm should sort the first sequence according to the order imposed by the permutation as prescribed by the second sequence. For each i, x_i should appear in position a_i in the output array. As an example, if $x = 23, 9, 5, 17$ and $a = 4, 1, 3, 2$, then the output should be $x = 9, 17, 5, 23$.

Please describe your algorithm as clearly as possible; it is not necessary to give the pseudo code. Remember that the algorithm must be in-place, without using any additional storage for the numbers to be sorted. Give an analysis of its time complexity. The more efficient your algorithm is, the more points you will get for this problem.

Appendix

- The solution of the recurrence relation $T(n) = aT(n/b) + cn^k$, where a and b are integer constants, $a \geq 1$, $b \geq 2$, and c and k are positive constants, is as follows.

$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{if } a > b^k \\ O(n^k \log n) & \text{if } a = b^k \\ O(n^k) & \text{if } a < b^k \end{cases}$$

- Below is an algorithm for determining whether a solution to the Knapsack Problem exists.

```
**Algorithm Knapsack** $(S, K)$;
**begin**
    $P[0,0].exist := true$;
    **for** $k := 1$ **to** $K$ **do**
        $P[0,k].exist := false$;
    **for** $i := 1$ **to** $n$ **do**
        **for** $k := 0$ **to** $K$ **do**
            $P[i,k].exist := false$;
            **if** $P[i-1,k].exist$ **then**
                $P[i,k].exist := true$;
                $P[i,k].belong := false$
            **else if** $k - S[i] \geq 0$ **then**
                    **if** $P[i-1, k-S[i]].exist$ **then**
                        $P[i,k].exist := true$;
                        $P[i,k].belong := true$
**end**
```