Midterm

Note

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

Problems

1. Given a set of $n+1$ numbers out of the first $2n$ (starting from 1) natural numbers 1, 2, 3, . . ., $2n$, prove *by induction* that there are two numbers in the set, one of which divides the other.

2. Prove *by induction* that the sum of the heights of all nodes in a complete binary tree with n nodes is at most $n-1$. You may assume it is known that the sum of the heights of all nodes in a *full* binary tree of height h is $2^{h+1}-h-2$. (Note: a single-node tree has height 0.)

3. Consider bounding summations by integrals.

   (a) If $f(x)$ is monotonically *increasing*, then
   $$\int_0^n f(x)dx \leq \sum_{i=1}^{n} f(i).$$
   Show that this is indeed the case.

   (b) If $f(x)$ is monotonically *decreasing*, then
   $$\sum_{i=1}^{n} f(i) \leq f(1) + \int_1^n f(x)dx.$$
   Show that this is indeed the case.

4. Consider a variant of the Knapsack Problem 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 adequate pseudocode 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.

5. Show all intermediate and the final AVL trees formed by inserting the numbers 7, 5, 2, 1, 4, 3, and 6 (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. Below is the Mergesort algorithm in pseudocode:

```
Algorithm Mergesort $(X, n)$;
begin  $M\_Sort(1, n)$  end

procedure M_Sort $(Left, Right)$;
begin
    if $Right - Left = 1$ then
        if $X[Left] > X[Right]$ then $swap(X[Left], X[Right])$
    else if $Left \neq Right$ then
            $Middle := \lceil \frac{1}{2}(Left + Right) \rceil$;
            $M\_Sort(Left, Middle - 1)$;
            $M\_Sort(Middle, Right)$;
            // the merge part
            $i := Left$;  $j := Middle$;  $k := 0$;
            while $(i \leq Middle - 1)$ and $(j \leq Right)$ do
                $k := k + 1$;
                if $X[i] \leq X[j]$ then
                    $TEMP[k] := X[i]$;  $i := i + 1$
                else $TEMP[k] := X[j]$;  $j := j + 1$;
            if $j > Right$ then
                for $t := 0$ to $Middle - 1 - i$ do
                    $X[Right - t] := X[Middle - 1 - t]$
            for $t := 0$ to $k - 1$ do
                $X[Left + t] := TEMP[t]$
end
```

Given the array below as input, what are the contents of array *TEMP* after the merge part is executed for the first time and what are the contents of *TEMP* when the algorithm terminates? Assume that each entry of *TEMP* has been initialized to 0 when the algorithm starts.

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

7. The partition procedure in the Quicksort algorithm chooses an element as the pivot and divide the input array $A[1..n]$ into two parts such that, when the pivot is properly placed in $A[i]$, the entries in $A[1..(i-1)]$ are less than or equal to $A[i]$ and the entries in $A[(i+1)..n]$ are greater than or equal to $A[i]$. Please design an extension of the partition procedure so that it chooses two pivots and divides the input array into three parts. Assuming the two pivots are eventually placed in $A[i]$ and $A[j]$ $(i < j)$ respectively, the entries in $A[1..(i-1)]$ are less than or equal to

$A[i]$, the entries in $A[(i+1)..(j-1)]$ are greater than or equal to $A[i]$ and less than or equal to $A[j]$, and the entries in $A[(j+1)..n]$ are greater than or equal to $A[j]$.

Please present your extension in adequate pseudocode and make assumptions wherever necessary. Give an analysis of its time complexity. The more efficient your algorithm is, the more points you will be credited for this problem.

8. Consider rearranging the following array into a max heap using the *bottom-up* approach.

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

Please show the result (i.e., the contents of the array) after a new element is added to the current collection of heaps (at the bottom) until the entire array has become a heap.

9. 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 pseudocode. 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.

10. Below is a variant of the bubble sort algorithm in pseudocode.

```
**Algorithm Bubble_Sort** $(A, n)$;
**begin**
    **for** $i := 1$ **to** $n-1$ **do**
        **for** $j := 1$ **to** $n-i$ **do**
            **if** $A[j] > A[j+1]$ **then**
                swap($A[j]$, $A[j+1]$);
        **end for**
    **end for**
**end**
```

Draw a decision tree of the algorithm for the case of $A[1..3]$, i.e., $n = 3$. In the decision tree, you must indicate (1) which two elements of the original input array are compared in each internal node and (2) the sorting result in each leaf. Please use X_1, X_2, X_3 (not $A[1]$, $A[2]$, $A[3]$) to refer to the elements (in this order) of the original input array.

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. It does not attempt to maximize the number of items in the solution.

```
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
```

- Below is 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
```