Midterm

Note

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

Problems

1. Prove *by induction* that, for a complete binary tree, one of the two subtrees under the root is a full binary tree and the other is a complete binary tree. An empty tree may be considered a full binary tree and also a complete binary tree. (Note: full binary trees are special cases of complete binary trees.)

2. Consider bounding summations by integrals. We already know that, if $f(x)$ is monotonically *increasing*, then

$$\sum_{i=1}^{n} f(i) \leq \int_{1}^{n+1} f(x)dx.$$

   (a) The sum may also be bounded from below as follows:

$$\int_{0}^{n} f(x)dx \leq \sum_{i=1}^{n} f(i).$$

   Show that this is indeed the case.

   (b) Prove, using this bounding technique, that $\sum_{i=1}^{n} \frac{1}{i} = \Theta(\log n)$. Note that $\frac{1}{i}$ actually decreases when i increases.

3. Consider the problem of merging two skylines, which is a useful building block for computing the skyline of a number of buildings. A skyline is an alternating sequence of x coordinates and y coordinates (heights), ending with an x coordinate (as discussed in class). The sequence of coordinates may be coveniently stored in an array, say A, with $A[0]$ storing the first x coordinate, $A[1]$ the first y coordinate, $A[2]$ the second x coordinate, etc.

   Design a linear-time procedure that prints out the resulting skyline from merging two given skylines. Please present the procedure in suitable pseudocode. The procedure should be named `merge_skylines` and invoked by `merge_skylines(A,m,B,n)`, where `A` and `B` are the two input skylines and `A[m]` and `B[n]` store the final x coordinate of skyline `A` and that of skyline `B` respectively.

4. The Knapsack Problem that we discussed in class is defined as follows: Given a set S of n items, where the ith 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.

We have described in class an algorithm (see the Appendix) to solve the problem. Modify the algorithm to solve a variation of the knapsack problem where each item has an *unlimited* supply. In your algorithm, please change the type of $P[i,k].belong$ into integer and use it to record the number of copies of item i needed. 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 5, 7, 1, 2, 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 ≠ Right then
            Middle := ⌈½(Left + Right)⌉;
            M_Sort(Left, Middle − 1);
            M_Sort(Middle, Right);
            // the merge part
            i := Left;  j := Middle;  k := 0;
            while (i ≤ Middle − 1) and (j ≤ Right) do
                  k := k + 1;
                  if X[i] ≤ 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[1 + 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
7	6	3	8	5	10	11	2	1	12	4	9

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. Below is a variant of the insertion sort algorithm.

```
Algorithm Insertion_Sort (A, n);
begin
    for i := 2 to n do
        x := A[i];
        j := i;
        while j > 1 and A[j − 1] > x do
            A[j] := A[j − 1];
            j := j − 1;
        end while
        A[j] := x;
    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.

9. Consider the text data compression problem we have discussed in class; the problem statement is given below.

   > Given a text (a sequence of characters), find an encoding for the characters that satisfies the prefix constraint and that minimizes the total number of bits needed to encode the text.

   Prove that the two characters with the lowest frequencies must be among the deepest leaves (farthest from the root) in the final code tree.

10. The *next* table is a precomputed table that plays a critical role in the KMP algorithm. For every position j of the second input string $b_1 b_2 \ldots b_m$ (to be matched against the first input string), the value of $next[j]$ tells the length of the longest proper prefix that is equal to a suffix of $b_1 b_2 \ldots b_{j-1}$; the value of $next[0]$ is set to -1 to fit in the KMP algroithm. For each of the following instances of *next*, give a string of letters a and b that gives rise to the table or argue that no string can possibly produce the table.

(a)

1	2	3	4	5	6	7	8	9
−1	0	0	1	1	2	3	4	5

(b)

1	2	3	4	5	6	7	8	9
−1	0	1	2	3	5	1	2	3

Appendix

- Below is an algorithm for determining whether a solution to the (original) 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] ≥ 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
```

- The algorithm for computing the *next* table in the KMP algorithm is as follows.

```
Algorithm Compute_Next (B, m);
begin
    next[1] := −1;  next[2] := 0;
    for i := 3 to m do
        j := next[i − 1] + 1;
        while B[i − 1] ≠ B[j] and j > 0 do
            j := next[j] + 1;
        next[i] := j
end
```