Midterm

Note

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

Problems

1. Consider a round-robin tournament among n players. In the tournament, each player plays once against all other $n-1$ players. There are no draws, i.e., for a match between A and B, the result is either A beat B or B beat A. Prove *by induction* that, after a round-robin tournament, it is always possible to arrange the n players in an order $p_1, p_2, \cdots, p_n$ such that p_1 beat p_2, p_2 beat p_3, $\cdots$, and p_{n-1} beat p_n. (Note: the "beat" relation, unlike "$\geq$", is not transitive.)

2. Find the error in the following proof that all horses are the same color.

   CLAIM: In any set of h horses, all horses are the same color.

   PROOF: By induction on h.

   Basis ($h = 1$): In any set containing just one horse, all horses clearly are the same color.

   Inductive step ($h > 1$): We assume that the claim is true for $h = k$ ($k \geq 1$) and prove that it is true for $h = k+1$. Take any set H of $k+1$ horses. We show that all the horses in this set are the same color. Remove one horse from this set to obtain the set H_1 with just k horses. By the induction hypothesis, all the horses in H_1 are the same color. Now replace the removed horse and remove a different one to obtain the set H_2. By the same argument, all the horses in H_2 are the same color. Therefore all the horses in H must be the same color, and the proof is complete.

3. Let $G(h)$ denote the least possible number of nodes contained in an AVL tree of height h. Let us assume that the empty tree has height -1 and a single-node tree has height 0.

   (a) Please give a recurrence relation that characterizes (fully defines) G.

   (b) Based on the recurrence relation, prove that the height of an AVL tree with n nodes is $O(\log n)$.

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. Let $x_1, x_2, \cdots, x_{2n-1}, x_{2n}$ be a sequence of $2n$ real numbers. Design an algorithm to partition the numbers into n pairs such that the maximum of the n sums of pair is minimized. It may be intuitively easy to get a correct solution. You must explain how the algorithm can be designed using induction.

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

7. 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 (except some constant space for exchanging two elements) . Give an analysis of its time complexity. The more efficient your algorithm is, the more points you will get for this problem.

8. Below is a variant of the partition algorithm for quicksort.

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

Draw a decision tree of the algorithm for the case of **Partition**$(A, 1, 3)$. 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 partition 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 A.

9. Construct a Huffman code tree for a text composed from seven characters A, B, C, D, E, F, and G with frquencies 15, 4, 2, 7, 21, 3, and 10 respectively.

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	1	2	3	4

(b)

1	2	3	4	5	6	7	8	9
−1	0	1	2	3	4	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
```

- 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
```