Suggested Solutions to Midterm 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.)

   Solution. The proof is by induction on the number n of players.

   Base case ($n = 2$): There are exactly two players, say A and B. Either A beat B, in which case we order them as A, B, or B beat A, in which case we order them as B, A.

   Induction step ($n > 2$): Pick any of the n players, say A. From the induction hypothesis, the other $n-1$ players can be ordered as $p_1, p_2, \cdots, p_{n-1}$ such that p_1 beat p_2, p_2 beat p_3, $\cdots$, and p_{n-2} beat p_{n-1}. We now exam the result of the match played between A and p_1. If A beat p_1, then we get a satisfying order $A, p_1, p_2, \cdots, p_{n-1}$. Otherwise ($p_1$ beat A), we continue to exam the result of the match played between A and p_2. If A beat p_2, then we get a satisfying order $p_1, A, p_2, \cdots, p_{n-1}$. Otherwise ($p_2$ beat A), we continue as before. We end up either with $p_1, p_2, \cdots, p_{i-1}, A, p_i, \cdots, p_{n-1}$ for some $i \leq n-1$ or eventually with $p_1, p_2, \cdots, p_{n-1}, A$ if A is beaten by every other player, in particular p_{n-1}. □

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.

   Solution. The inductive step is erroneous, as one cannot prove the claim for the case of $h = 2$ assuming it holds for $h = 1$. For $h = 2$, the two sets H_1 and H_2 (resulted from removing one of the hourses) are both of size 1 and do not have any common member. The horses in each of the two sets are indeed the same color, since there is just one horse in each set. However, the two horses from the two sets (which does not overlap) may have different colors. □

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.

   Solution. The recurrence relation can be defined as follows:

$$\begin{cases} G(-1) & = 0 \\ G(0) & = 1 \\ G(h) & = G(h-1) + G(h-2) + 1, \quad h \geq 1 \end{cases}$$

□

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

Solution. A precise solution to $G(h)$ may be derived by establishing the relation $G(h) = F(h+3) - 1$, where $F(n)$ is the n-th Fibonacci number (as defined in Chapter 3.5 of Manber's book) for which we already know the closed form; the proof is in fact quite simple by induction. However, we will prove directly a lower bound for $G(h)$, namely $\Omega((\frac{3}{2})^h)$, which is good enough to show its exponential growth. The proof is by induction on h, showing that $G(h) \geq \frac{2}{3}(\frac{3}{2})^h$, for $h \geq 0$.

Base case ($h = 0$ or $h = 1$): When $h = 0$, $\frac{2}{3}(\frac{3}{2})^0 = \frac{2}{3} \leq 1 = G(0)$. When $h = 1$, $\frac{2}{3}(\frac{3}{2})^1 = 1 \leq 2 = G(1)$.

Inductive step ($h > 1$): $G(h) = G(h-1)+G(h-2)+1$, which from the induction hypothesis $\geq \frac{2}{3}(\frac{3}{2})^{h-1} + \frac{2}{3}(\frac{3}{2})^{h-2} + 1 \geq (1 + \frac{2}{3})(\frac{3}{2})^{h-2} = (1 + \frac{2}{3})(\frac{3}{2})^{-2}(\frac{3}{2})^h = \frac{20}{27}(\frac{3}{2})^h \geq \frac{2}{3}(\frac{3}{2})^h$.

Therefore, for an AVL tree of size n, its height h must be such that $\frac{2}{3}(\frac{3}{2})^h \leq G(h) \leq n$. It follows that $h \leq \frac{1}{\log 1.5} \log n + 1$ (base 2 logarithm), implying $h = O(\log n)$. □

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.

   Solution.

```
Algorithm Knapsack_Unlimited (S, K);
begin
    P[0,0].exist := true;
    P[0,0].belong := 0;
    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 := 0
            else if k - S[i] ≥ 0 then
                    if P[i,k-S[i]].exist then
```

$P[i,k].exist := true;$
$P[i,k].belong := P[i,k-S[i]].belong + 1$

end

From the main nested for-loops, we see that the complexity is $O(nK)$. (Note: the bound should be understood as $O(n2^{\log K})$, where $\log K$ represents the input size of the number K.) □

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.

Solution. We first fix some notations:

- We represent a partition of a list $x_1, x_2, \cdots, x_{2n-1}, x_{2n}$ into n pairs as a set of sets of two elements $\{\{y_1, y_2\}, \{y_3, y_4\}, \cdots, \{y_{2n-1}, y_{2n}\}\}$, where $y_1, y_2, \cdots, y_{2n-1}, y_{2n}$ is a permutation of $x_1, x_2, \cdots, x_{2n-1}, x_{2n}$.
- For a list A of $2n$ elements, let **MinMaxPair**(A) denote some partition that meets the problem requirement for $2n$ elements.
- $A \setminus B$, where A is a list and B a set, denotes the list of elements in A but not in B. We stipulate that elements in $A \setminus B$ appear in the same order as in A.

We are given a list $X = x_1, x_2, \cdots, x_{2n-1}, x_{2n}$. If $n = 1$, i.e., there are only two elements, the solution is obvious, namely $\{\{x_1, x_2\}\}$. Now consider the cases of $n > 1$. Let y_1 denote the smallest element and y_{2n} the largest element in X. We claim that **MinMaxPair**$(X \setminus \{y_1, y_{2n}\}) \cup \{\{y_1, y_{2n}\}\}$ meets the problem requirement for $2n$ elements, i.e., the pair $\{y_1, y_{2n}\}$ is part of an optimal partition. Suppose $\{y_i, y_j\}$ is the pair with the largest sum in **MinMaxPair**$(X \setminus \{y_1, y_{2n}\})$. Pairing y_{2n} with either y_i or y_j (instead of y_1) would produce a pair whose sum is at least as large as that of $\{y_1, y_{2n}\}$ and that of any pair in **MinMaxPair**$(X - \{y_1, y_{2n}\})$. To compute **MinMaxPair**$(X \setminus \{y_1, y_{2n}\})$, we need to solve the same problem with two elements less and here we invoke the induction hypothesis.

In the above, we select and remove the smallest and the largest elements from the current list in each step. This would incur a complexity of $O(n)$ for each step, making the complexity of the whole algorithm $O(n^2)$. We can improve this to $O(n \log n)$ by sorting the list right in the beginning before pairing up the elements. So, the algorithm can be summarized as follows.

(a) Sort the input list X to get Y.

(b) Suppose the current list $Y = y_1, y_2, \cdots, y_{2i-1}, y_{2i}$ $(i \geq 1)$. Remove and output the pair $\{y_1, y_{2i}\}$ from Y.

(c) Repeat the previpus step until Y is empty.

□

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

Solution.

The contents of array *TEMP* after the merge part is executed for the first time:

1	2	3	4	5	6	7	8	9	10	11	12
3	6	0	0	0	0	0	0	0	0	0	0

The contents of array *TEMP* when the algorithm terminates:

1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	0	0	0

□

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.

Solution. Suppose that array X holds the first sequence and array A the second. Sort A increasingly according to the position values that it stores. Every time when two elements in A, say a_i and a_j, are swapped, we also swap the corresponding elements in X, i.e., x_i and x_j. Once the sorting of A is completed, the elements in X are also sorted as prescribed by A. Any in-place sorting algorithm may be used for sorting A. If we use Heapsort, the complexity is $O(n \log n)$.

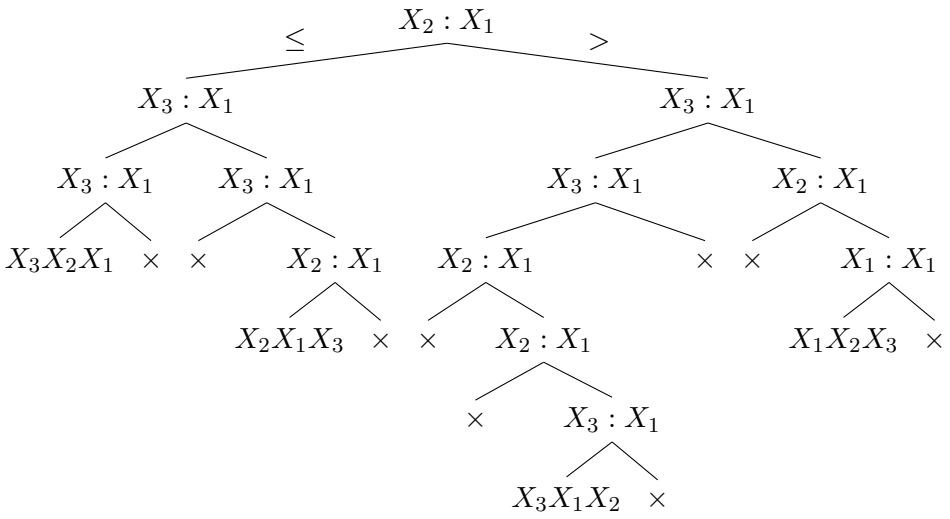
In fact, there is a much simpler and faster (linear-time) algorithm. In this algorithm, we scan array A form left to right. Whenever $A[i] \neq i$, we swap x_i and $x_{A[i]}$ and also $A[i]$ and $A[A[i]]$. This is repeated until $A[i] = i$ and we then proceed to the next element in array A. Each swap of x_i and $x_{A[i]}$ brings one element in X to its final destination. So, we ever need to do such swaps at most n times and the check of whether $A[i] = i$ $(1 \leq i \leq n)$ is done at most $2n$ times in total. The corresponding swaps for A are also performed at most n times. Therefore, this algorithm runs in $O(n)$ time. □

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.

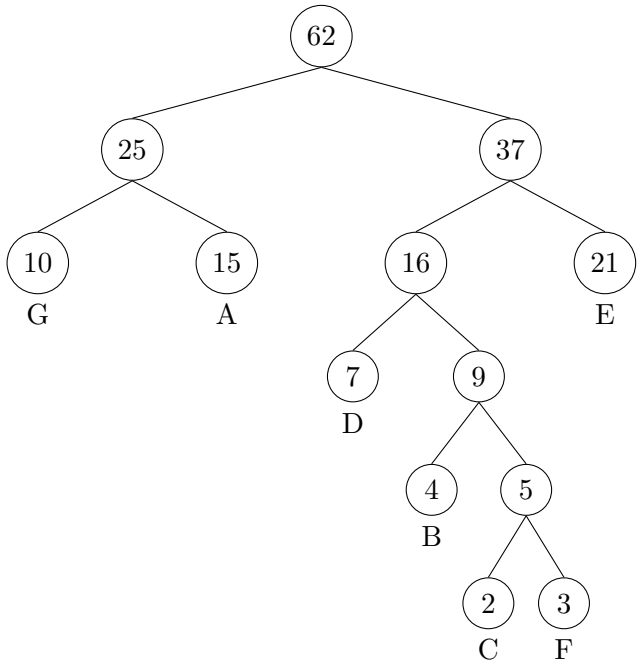
Solution. (Wayne Zeng)



□

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

   Solution. (Hung-Yang Lin)



□

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 $b_1b_2 \ldots b_m$ (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 $b_1b_2 \ldots b_{j-1}$; 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

Solution. There are a few strings that may produce this *next* table, e.g., *abaaabaaa* or *abaaabaab*. □

(b)

1	2	3	4	5	6	7	8	9
−1	0	1	2	3	4	1	2	3

Solution. No string can possibly give arise to this *next* table. Given the values of $next[1..6]$, $next[7]$ cannot be 1.

$next[3] = 1$ implies that $b_1 = b_2$. $next[3] = 2$ implies that $b_1b_2 = b_2b_3$ and hence $b_2 = b_3$. Reasoning along this line, we have $b_1 = b_2 = b_3 = b_4 = b_5$. b_6 is either equal to b_5 or not equal to b_5. If $b_6 = b_5$, then $b_1b_2b_3b_4b_5 = b_2b_3b_4b_5b_6$ and hence $next[7]$ should be 5. If $b_6 \neq b_5$, no prefix of $b_1b_2b_3b_4b_5$ can possibly be equal to any suffix of $b_2b_3b_4b_5b_6$ and hence $next[7]$ should be 0. In either case, $next[7]$ is not equal to 1. □

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
```