Suggested Solutions to Midterm Problems

1. 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. □

2. Consider the following two-player game: given a positive integer N, player A and player B take turns counting to N. In his turn, a player may advance the count by 1 or 2. For example, player A may start by saying "1, 2", player B follows by saying "3", player A follows by saying "4", etc. The player who eventually has to say the number N loses the game.

   A game is *determined* if one of the two players always has a way to win the game. Prove that the counting game as described is determined for any positive integer N; the winner may differ for different given integers. You must use induction in your proof. (Hint: think about the remainder of the number N divided by 3.)

   Solution. We first prove the following claim:

   > When $N = 3k + 1$ for some $k \geq 0$, player B can always win the game.

   The proof is by induction on k.

   Base case ($k = 0$, i.e., $N = 1$): player A has no other choice but say 1 and hence player B wins.

   Inductive step ($k \geq 1$, i.e., $N = 3k + 1 \geq 4$): player A starts either by "1" or "1, 2". In both cases, player B can always count to 3. At this point we have the situation analogous to where the two players are to play a game with $N = 3(k-1)+1$, in which player B can always win from the induction hypothesis.

   We next prove a second claim:

   > When $N = 3k + 2$ or $N = 3(k + 1)$ for some $k \geq 0$, player A can always win the game.

In the case when $N = 3k + 2$, player A starts by saying "1", while in the case when $N = 3(k + 1)$, he starts by "1, 2". After player A's first turn, we have the situation analogous to that player B is to start a game with $N = 3k + 1$, playing the role of player A (to start first in the remaining game). From the first claim, player A (playing the role of player B in the remaining game) will win the game.

Now we see that, for every positive integer N, there is always a player that can win the counting game and hence the game is determined. □

3. Summations whose exact values are hard to compute may be easily and tightly bounded by integrals. For example, if $f(x)$ is monotonically *decreasing*, then

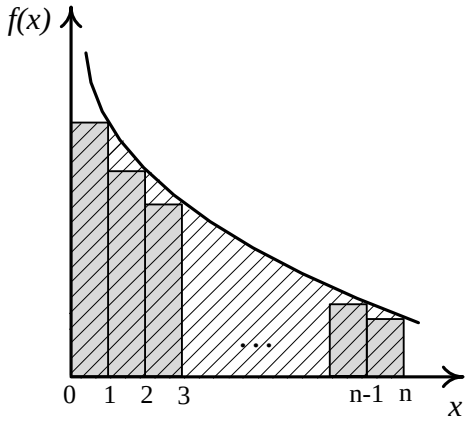
$$\int_1^{n+1} f(x)dx \le \sum_{i=1}^{n} f(i) \le f(1) + \int_1^n f(x)dx.$$

(a) Show that the bounds for $\sum_{i=1}^n f(i)$ are indeed correct.

Solution. Given that $f(x)$ is monotonically decreasing, we have

$$\begin{array}{ccccc}
\int_1^2 f(x)dx & \le & f(1) & \le & f(1) \\
\int_2^3 f(x)dx & \le & f(2) & \le & \int_1^2 f(x)dx \\
\int_3^4 f(x)dx & \le & f(3) & \le & \int_2^3 f(x)dx \\
 & & \cdots & & \\
\int_{n-1}^n f(x)dx & \le & f(n-1) & \le & \int_{n-2}^{n-1} f(x)dx \\
\int_n^{n+1} f(x)dx & \le & f(n) & \le & \int_{n-1}^n f(x)dx \\
\hline
\int_1^{n+1} f(x)dx & \le & \sum_{i=1}^n f(i) & \le & f(1) + \int_1^n f(x)dx
\end{array}$$

So, the bounds for the summation $\sum_{i=1}^n f(i)$ are correct. The following illustrates how the summation is related to the upper bound.



□

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

Solution. We know that $\int \frac{1}{x}dx = \ln x$ (plus some constant, which may be ignored).
$\sum_{i=1}^n \frac{1}{i} \ge \int_1^{n+1} \frac{1}{x}dx = \ln(n+1) - \ln 1 = \ln(n+1) \ge \frac{1}{\log e} \log n$.
So, $\sum_{i=1}^n \frac{1}{i} = \Omega(\log n)$.
$\sum_{i=1}^n \frac{1}{i} \le \frac{1}{1} + \int_1^n \frac{1}{x}dx = 1 + \ln n - \ln 1 \le 1 + \ln n \le \frac{2}{\log e} \log n$ (for $n \ge 3$).
So, $\sum_{i=1}^n \frac{1}{i} = O(\log n)$.
It follows that $\sum_{i=1}^n \frac{1}{i} = \Theta(\log n)$. □

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 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.

   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].belong + 1
end
```

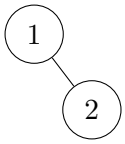
□

5. Show all intermediate and the final AVL trees formed by inserting the numbers 1, 2, 5, 7, 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.

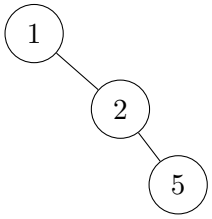
   Solution.

Insert 1:

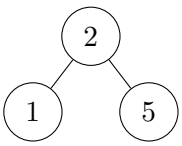
Insert 2:



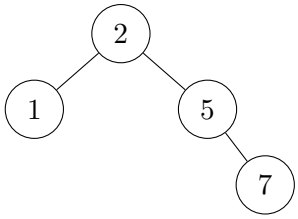
Insert 5:



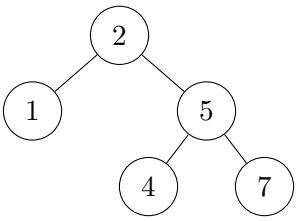
Single rotation at 1:



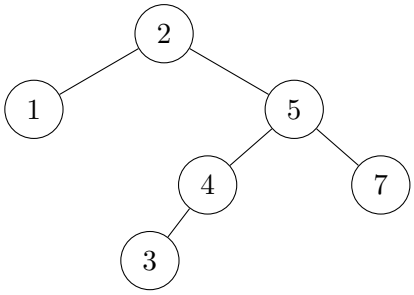
Insert 7:



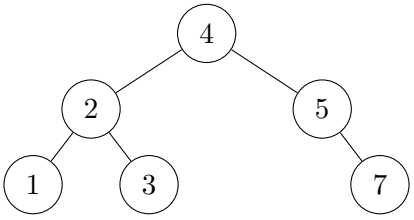
Insert 4:



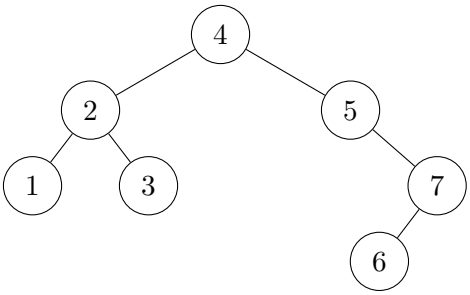
Insert 3:



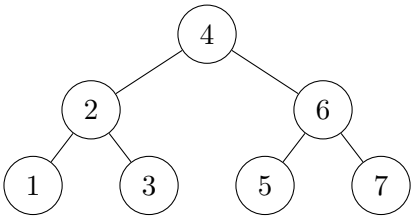
Double rotation at 2:



Insert 6:



Double rotation at 5:



□

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] ≤ 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	8	3	6	5	9	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	7	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. 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.

   Solution.

```
Partition3(X, Left, Right);
begin
  if X[Left] > X[Right] then swap(X[Left], X[Right]);
  pivot1 := X[Left];
  pivot2 := X[Right];
  i := Left;
  k := Right;
  j := Left + 1;
```

```
  while (j < k) do
    if X[j] < pivot1 then
      i := i + 1;
      swap(X[i], X[j]);
      j := j + 1;
    else
      if X[j] > pivot2 then
        k := k - 1;
        swap(X[j], X[k]);
      end if;
    end if;
  end while;
  swap(X[Left], X[i]);
  swap(X[Right], X[k]);
end
```

The algorithm contains one layer of for-loop, where the bound condition ($j < k$) can only grow stricter (k may decrease but not increase) and each iteration takes a constant amount of time, so it is clearly linear-time. □

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

Solution.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	3	7	5	1	9	13	6	4	11	10	12	15	14	8
2	3	7	5	1	9	<u>14</u>	6	4	11	10	12	15	<u>13</u>	8
2	3	7	5	1	<u>15</u>	14	6	4	11	10	12	<u>9</u>	13	8
2	3	7	5	<u>11</u>	15	14	6	4	<u>1</u>	10	12	9	13	8
2	3	7	<u>6</u>	11	15	14	<u>5</u>	4	1	10	12	9	13	8
2	3	<u>15</u>	6	11	<u>12</u>	14	5	4	1	10	<u>7</u>	9	13	8
2	<u>11</u>	15	6	<u>10</u>	12	14	5	4	1	<u>3</u>	7	9	13	8
<u>15</u>	11	<u>14</u>	6	10	12	<u>13</u>	5	4	1	3	7	9	<u>2</u>	8

□

9. Two computers, each with a set of n integers, try to collaboratively find the n-th smallest element of the union of the two sets. The two computers can communicate by sending messages and they can perform any kind of local computation. A message can contain one element or one integer; a message with two numbers should be counted as two messages. Design an algorithm for the search task so that the number of messages exchanged is minimized. You can assume, for simplicity, that all the elements are distinct.

   Please present your algorithm in an adequate pseudo code and make assumptions wherever necessary. Give an analysis of its message complexity (the number of messages exchanged). The more efficient your algorithm is, the more points you will get for this problem.

   Solution. Let us refer to the two computers as Computer 0 and Computer 1. Computer 0 holds its set of n integers in array A and Computer 1 in array B. For convenience, we will

also treat arrays as sets so that set operations may be applied to arrays. The numbers assumed to be distinct, $A \cup B$ holds totally $2n$ integers. For a number a in A, if it is ranked j-th in A and k-th in $B \cup \{a\}$ such that $j+k-1 = n$, then it is the number ranked n-th in $A \cup B$; analogously, for a number in B.

To find the n-th smallest number (the number ranked n-th) in $A \cup B$, the two computers run the same procedure **Search** as below with different arguments, by invoking **Search**$(0, A, n)$ and **Search**$(1, B, n)$ respectively. The basic idea of the procedure is that Computer 0 first selects a candidate and sends it to Computer 1 for validation (checking whether $j + k - 1 = n$). If Computer 1 found the candidate to be too small, it suggests a larger candidate; and if too large, it suggests a smaller candidate. When a candidate is found to be ranked n-th, the discover sends the same number back to the proposing computer and stops; the proposing computer also stops when it receives the number and finds it to be the same as the last number it sent.

Algorithm Search (i, X, n);
begin
    sort X in ascending order;
    if $i = 0$ **then**
        $L := 1$;
        $R := n$;
        $j := \lceil \frac{L+R}{2} \rceil$;
        $a := X[j]$;
        send **message**(a, j) to Computer $(1 - i)$;
    else // $i = 1$
        receive **message**(b, j) from Computer $(1 - i)$;
        $k :=$ the rank of b in $X \cup \{b\}$;
        if $j + k - 1 = n$ **then**
            send **message**(b, k) to Computer $(1 - i)$;
            stop;
        else
            if $j + k - 1 < n$ **then**
                $L := k$;
                $R := n$;
            else // $j + k - 1 > n$
                $L := 1$;
                $R := k - 1$;
            end if;
            $j := \lceil \frac{L+R}{2} \rceil$;
            $a := X[j]$;
            send **message**(a, j) to Computer $(1 - i)$;
    end if;
    done := *false*;
    repeat
        receive **message**(b, j) from Computer $(1 - i)$;
        if $b = a$ **then**
            done := *true*;
        else
            $k :=$ the rank of b in $X \cup \{b\}$;
            if $j + k - 1 = n$ **then**

```
                done := true;
                send message(b, k) to Computer (1 − i);
            else
                if j + k − 1 < n then
                    L := k;
                else
                    R := k − 1;
                end if;
                j := ⌈(L+R)/2⌉;
                a := X[j];
                send message(a, j) to Computer (1 − i);
            end if;
        end if;
    until done
end
```

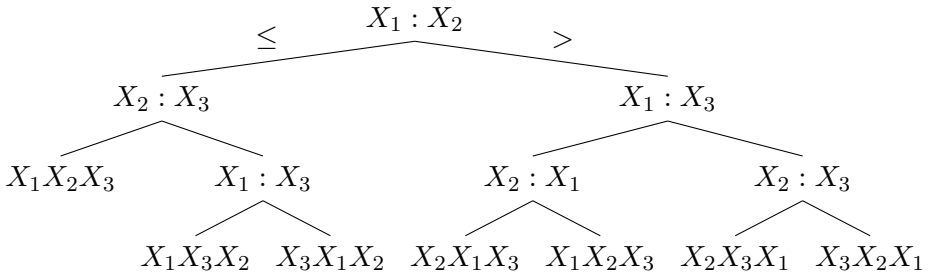
Every message (except the first and the last) received helps the receiving computer to cut the search space by at least one half, each computer initially having a search space of n integers. So, at most $2 \log n$ messages were sent and received, plus the first and the last messages the latter of which was used to signal discovery. Every message contains just two numbers and is counted as two messages. It follows that the message complexity is $O(\log n)$. □

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

```
Algorithm Bubble_Sort (A, n);
begin
    i := n;
    repeat
        swapped := false;
        for j := 1 to i − 1 do
            if A[j] > A[j + 1] then
                swap(A[j], A[j + 1]);
                swapped := true;
            end if
        end for
        i := i − 1;
    until (not swapped)
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.

Solution.



Note: the 3rd leaf from the right contains an impossible outcome and the corresponding decision $(X_2 : X_1)$ is not necessary. However, the algorithm makes it anyway, as it does not memorize X_1 and X_2 have been compared earlier. □

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
```