Suggested Solutions to Midterm 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.

   Solution. The proof is by induction on n.

   Base case ($n = 1$): There is only one subset of 2 ($= n+1$) numbers from $\{1, 2\}$, which is the set $\{1, 2\}$ itself. 1 divides 2.
   Inductive step ($n = k+1 > 1$): We need to show that any selection (subset) of $k+2$ numbers from $\{1, 2, \cdots, 2k, 2k+1, 2k+2\}$ contains two numbers, one of which divides the other. If the selection includes $k+1$ numbers from $\{1, 2, \cdots, 2k\}$, then by the induction hypothesis we are done. Otherwise, the selection must contain both $2k+1$ and $2k+2$ and also include a selection S of other k numbers from $\{1, 2, \cdots, 2k\}$.
   Case one ($k+1 \in S$): $k+1$ divides $2k+2$.
   Case two ($k+1 \notin S$): If S happens to contain two numbers one of which divides the other, then we are done. Otherwise, from the induction hypothesis, S must contain a number that divides $k+1$. This is so, because (a) $k+1$ does not divide any number in $\{1, 2, \cdots, 2k\}$ and (b) $S \cup \{k+1\}$ is a selection of $k+1$ numbers from $\{1, 2, \cdots, 2k\}$ and by the induction hypothesis must contain two numbers one of which divides the other. The number that divides $k+1$ also divides $2k+2$. □

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.)

   Solution. Let $G(n)$ denote the sum of the heights of all nodes in a complete binary tree with n nodes. For a full binary tree (a special case of complete binary trees) with $n = 2^{h+1} - 1$ nodes where h is the height of the tree, we already know that $G(n) = 2^{h+1} - (h+2) = n - (h+1) \leq n - 1$. With this as a basis, we prove the general case of arbitrary complete binary trees by induction on the number n (≥ 1) of nodes.

   Base case ($n = 1$ or $n = 2$): When $n = 1$, the tree is the smallest full binary tree with one single node whose height is 0. So, $G(n) = 0 \leq 1 - 1 = n - 1$. When $n = 2$, the tree has one additional node as the left child of the root. The height of the root is 1, while that of its left child is 0. So, $G(n) = 1 \leq 2 - 1 = n - 1$.

   Inductive step ($n > 2$): If n happens to be equal to $2^{h+1} - 1$ for some $h \geq 1$, i.e., the tree is full, then we are done; note that this covers the case of $n = 3 = 2^{1+1} - 1$. Otherwise, suppose $2^{h+1} - 1 < n < 2^{h+2} - 1$ ($h \geq 1$), i.e., the tree is a "proper" complete binary tree with height $h+1 \geq 2$. We observe that at least one of the two subtrees of the root is full, while the other is complete (possibly full). There are three cases to consider:

   Case 1: The left subtree is full with n_1 nodes and the right one is complete but not full with n_2 nodes (such that $n_1 + n_2 + 1 = n$). In this case, both subtrees much be of height h and $n_1 = 2^{h+1} - 1$. From the special case of full binary trees and the induction hypothesis, $G(n_1) = 2^{h+1} - (h+2) = n_1 - (h+1)$ and $G(n_2) \leq n_2 - 1$. $G(n) = G(n_1) + G(n_2) + (h+1) \leq (n_1 - (h+1)) + (n_2 - 1) + (h+1) = (n_1 + n_2 + 1) - 2 \leq n - 1$.

Case 2: The left subtree is full with n_1 nodes and the right one is also full with n_2 nodes. In this case, the left subtree much be of height h and $n_1 = 2^{h+1} - 1$, while the right subtree much be of height $h-1$ and $n_2 = 2^h - 1$. ¿From the special case of full binary trees, $G(n_1) = 2^{h+1} - (h+2) = n_1 - (h+1)$ and $G(n_2) = 2^h - (h+1) = n_2 - h$. $G(n) = G(n_1) + G(n_2) + (h+1) \leq (n_1 - (h+1)) + (n_2 - h) + (h+1) = (n_1 + n_2 + 1) - (h+1) \leq n-1$.

Case 3: The left subtree is complete but not full with n_1 nodes and the right one is full with n_2 nodes. In this case, the left subtree much be of height h, while the right subtree much be of height $h-1$ and $n_2 = 2^h - 1$. ¿From the induction hypothesis and the special case of full binary trees, $G(n_1) \leq n_1 - 1$ and $G(n_2) = 2^h - (h+1) = n_2 - h$. $G(n) = G(n_1) + G(n_2) + (h+1) \leq (n_1 - 1) + (n_2 - h) + (h+1) = (n_1 + n_2 + 1) - 1 = n - 1$. □

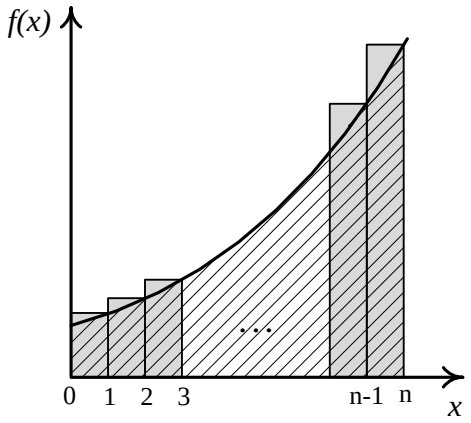
3. Consider bounding summations by integrals.

   (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.

   Solution. (Jing-Jie Lin)
   This is easily seen by comparing the areas (on the $R \times R$ plane) defined by the formulae on the two sides. As shown in the following diagram, the integral $\int_0^n f(x)dx$ equals the area under the curve that is shaded with thin parallel lines. The area is apparently no larger than the total area of the vertical bars which represents $\sum_{i=1}^{n} f(i)$.



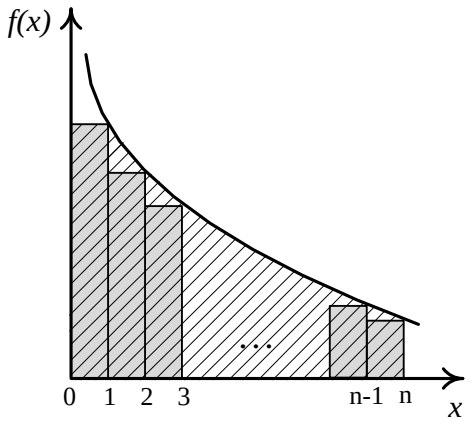
□

   (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.

   Solution. (Jing-Jie Lin)
   Similar to the previous solution.



□

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.

   Solution. (Yi-Wen Chang)

   To find the largest possible subset of items, we modify the *Knapsack* algorithm in the Appendix to obtain *Knapsack_ForMaxSubset*, as shown below. Each element $P[i,k]$ in the result array P now contains a new integer variable *size* which memorizes the size of the current largest subset for k.

```
Algorithm Knapsack_ForMaxSubset(S, K);
begin
    P[0,0].exist := true;
    P[0,0].size := 0;
    for k := 1 to K do
        P[0,k].exist := false;
        P[0,k].size := 0;
    for i := 1 to n do
        for k := 0 to K do
            P[i,k].exist := false;
            P[i,k].size := 0;
            if k − S[i] ≥ 0 and P[i − 1, k − S[i]].exist then
                if P[i − 1, k].exist and P[i − 1, k].size ≥ P[i − 1, k − S[i]].size + 1 then
                    P[i,k].exist := true;
                    P[i,k].belong := false;
                    P[i,k].size := P[i − 1, k].size;
                else
                    P[i,k].exist := true;
                    P[i,k].belong := true
                    P[i,k].size := P[i − 1, k − S[i]].size + 1;
```

```
            else if P[i − 1, k].exist then
                P[i, k].exist := true;
                P[i, k].belong := false;
                P[i, k].size := P[i − 1, k].size;
    if ¬P[n, K].exist then
       print "no solution"
    else i := n;
         k := K;
         while k > 0 do
            if P[i, k].belong then
                print i;
                k := k − S[i];
            i := i − 1;
end
```

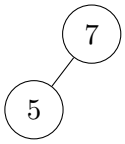
The complexity remains the same, which is $O(nK)$. When a solution (a subset of items whose sizes sum to exactly K) exists, the printed result will be the largest among such subsets. □

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.

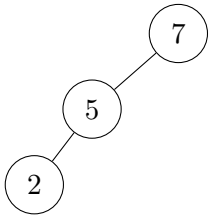
Solution.

Insert 7:

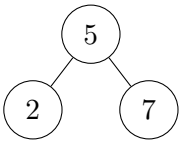
Insert 5:



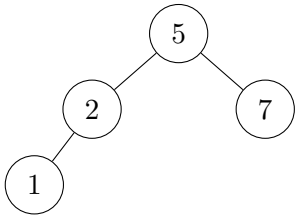
Insert 2:



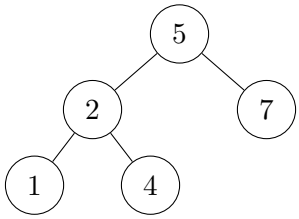
Single rotation at 7:



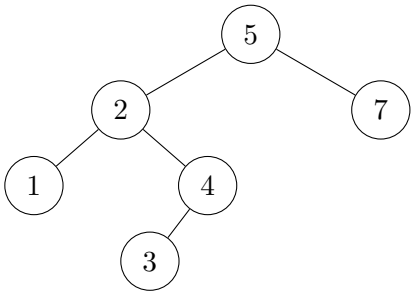
Insert 1:



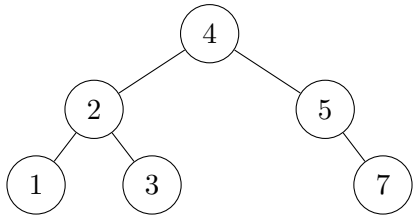
Insert 4:



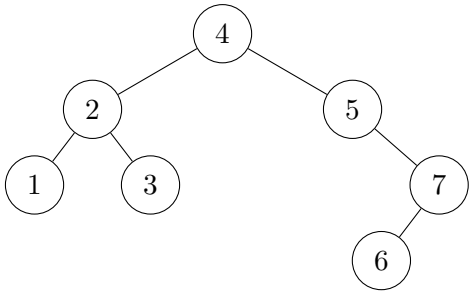
Insert 3:



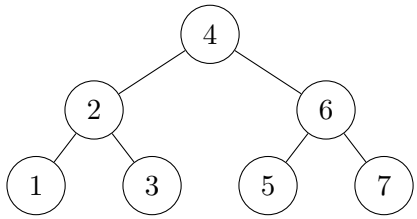
Double rotation at 5:



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 $X[i] \leq X[j]$ **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

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
4	9	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	10	11	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;
  for j := Left+1 to Right-1 do
    if X[j] < pivot1 then
      i := i + 1;
      swap(X[i], X[j]);
    else
      if X[j] > pivot2 then
        k := k - 1;
        swap(X[j], X[k]);
        if X[j] < pivot1 then
          i := i + 1;
          swap(X[i], X[j]);
```

```
  swap(X[Left], X[i]);
  swap(X[Right], X[k]);
end
```

The algorithm contains one simple for-loop, with each iteration taking a constant amount of time, and hence 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
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.

Solution.

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

□

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.

   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 $O(n \log n)$.

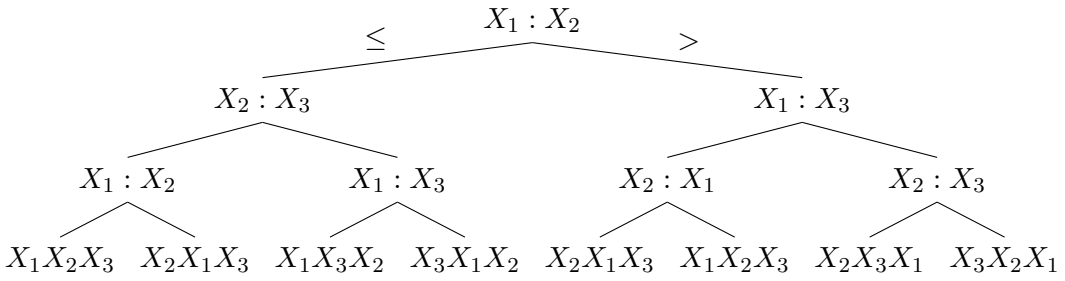
   In fact, there is a much simpler and faster (linear-time) algorithm. In this algorithm, we scan array A form left to right. Whenever $A[i] \neq i$, we swap x_i and $x_{A[i]}$ and also $A[i]$ and $A[A[i]]$. This is repeated until $A[i] = i$ and we then proceed to the next element in array A. Each swap 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. □

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.

Solution.



Note: two leaves (2nd and 6th from the left) contain impossible outcomes and the corresponding decisions are not necessary. However, the algorithm makes them anyway, which shows the inefficiency of the algorithm. □

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] ≥ 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
```