Suggested Solutions to Midterm Problems

Problems

1. Let $a_1, a_2, \cdots, a_n$ be positive real numbers such that $a_1 a_2 \cdots a_n = 1$. Prove *by induction* that $(1+a_1)(1+a_2)\cdots(1+a_n) \geq 2^n$. (Hint: In the inductive step, try introducing a new variable that replaces two chosen numbers from the sequence.)

   Solution. The proof is by induction on n.

   Base case ($n = 1$): $a_1 = 1$. So, $(1 + a_1) = 2 \geq 2^1$.

   Induction step ($n > 1$): In any sequence $a_1, a_2, \cdots, a_n$ ($n > 1$) of positive real numbers where $a_1 a_2 \cdots a_n = 1$, there must exist two numbers a_i and a_j such that $a_i \geq 1$ and $a_j \leq 1$. Without loss of generality, we assume that the two numbers are a_{n-1} and a_n (this can always be achieved by swapping numbers in the sequence). As $(1 - a_{n-1})(1 - a_n) \leq 0$, it follows that $a_{n-1} + a_n \geq 1 + a_{n-1}a_n$. Let a'_{n-1} be the number equal to $a_{n-1}a_n$ (which is also a positive real number) so that $a_1 a_2 \cdots a_{n-2} a'_{n-1} = a_1 a_2 \cdots a_{n-2} a_{n-1} a_n = 1$.

   $(1 + a_1)(1 + a_2)\cdots(1 + a_{n-2})(1 + a_{n-1})(1 + a_n) = (1 + a_1)(1 + a_2)\cdots(1 + a_{n-2})(1 + a_{n-1} + a_n + a_{n-1}a_n) \geq (1 + a_1)(1 + a_2)\cdots(1 + a_{n-2})((1 + a_{n-1}a_n) + (1 + a_{n-1}a_n)) = 2(1 + a_1)(1 + a_2)\cdots(1 + a_{n-2})(1 + a_{n-1}a_n) = 2(1 + a_1)(1 + a_2)\cdots(1 + a_{n-2})(1 + a'_{n-1})$, which from the induction hypothesis $\geq 2 \times 2^{n-1} = 2^n$. □

2. 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}. □

3. Below is an algorithm for solving a variant of the Towers of Hanoi puzzle with an additional fourth peg D; `Towers_Hanoi` is an algorithm for the original puzzle.

```
Algorithm Four_Towers_Hanoi(A,B,C,D,n);
begin
   if n<=2 then
      Towers_Hanoi(A,B,C,n);
   else
      Four_Towers_Hanoi(A,D,B,C,n-2);
      Towers_Hanoi(A,B,C,2);
      Four_Towers_Hanoi(D,B,C,A,n-2);
end;
```

Consider alternatives of first moving $n-k$ disks (for some value of k, not necessarily 2) to D. Let $T(n)$ denote the number of moves needed for n disks. Write a recurrence relation for $T(n)$ with k as a parameter. Can you tell which value of k will be the best, i.e., resulting in a smaller asymptotic upper bound for $T(n)$? Why?

Solution. The base case `Towers_Hanoi(A,B,C,n)`, where $k \geq n \geq 1$, takes $2^n - 1$ moves. A recurrence relation for $T(n)$ is obtained as follows:

$$\begin{cases} T(n) = 2^n - 1 & \text{, for } k \geq n \geq 1 \\ T(n) = 2T(n-k) + T(k) & \text{, for } n > k \end{cases}$$

For $n > k$,

$$\begin{array}{rl} T(n) = & 2T(n-k) + 2^k - 1 \\ 2T(n-k) = & 2(2T(n-2k) + 2^k - 1) = 2^2T(n-2k) + 2(2^k-1) \\ 2^2T(n-2k) = & 2^2(2T(n-3k) + 2^k - 1) = 2^3T(n-3k) + 2^2(2^k-1) \\ \cdots & \cdots \\ 2^{i-1}T(n-(i-1)k) = & 2^iT(n-ik) + 2^{i-1}(2^k-1) = 2^i(2^{n-ik}-1) + 2^{i-1}(2^k-1) \\ \hline T(n) = & 2^i(2^{n-ik}-1) + (2^{i-1} + 2^{i-2} + \cdots + 1)(2^k-1) \end{array}$$

where $k \geq (n-ik) \geq 1$.

For an asymptotic analysis, let us look at the case when n is a multiple of k, in which case $n - ik$ must equal k.

$$\begin{aligned} T(n) = & \; 2^i(2^{n-ik}-1) + (2^{i-1} + 2^{i-2} + \cdots + 1)(2^k-1) \\ = & \; 2^i(2^k-1) + (2^{i-1} + 2^{i-2} + \cdots + 1)(2^k-1) \\ = & \; (2^i + 2^{i-1} + \cdots + 1)(2^k-1) \\ = & \; (2^{\frac{n-k}{k}+1} - 1)(2^k-1) \\ = & \; 2^{\frac{n-k}{k}+k+1} - 2^{\frac{n-k}{k}+1} - 2^k + 1 \end{aligned}$$

A value of k that minimizes $\frac{n-k}{k} + k$ will also minimize $T(n)$. The rest is left as an exercise.
□

4. In the implementation of an AVL tree, a rebalancing process using rotation operations may be needed after an `insert` or `delete`. Design the first part of a procedure for `insert`, up to the point when the node where a rotation is needed (i,e., the *critical* node) is determined (when rebalancing is needed). You are not required to design the part for rotation. Please present your procedure in an adequate pseudo code and make assumptions wherever necessary.

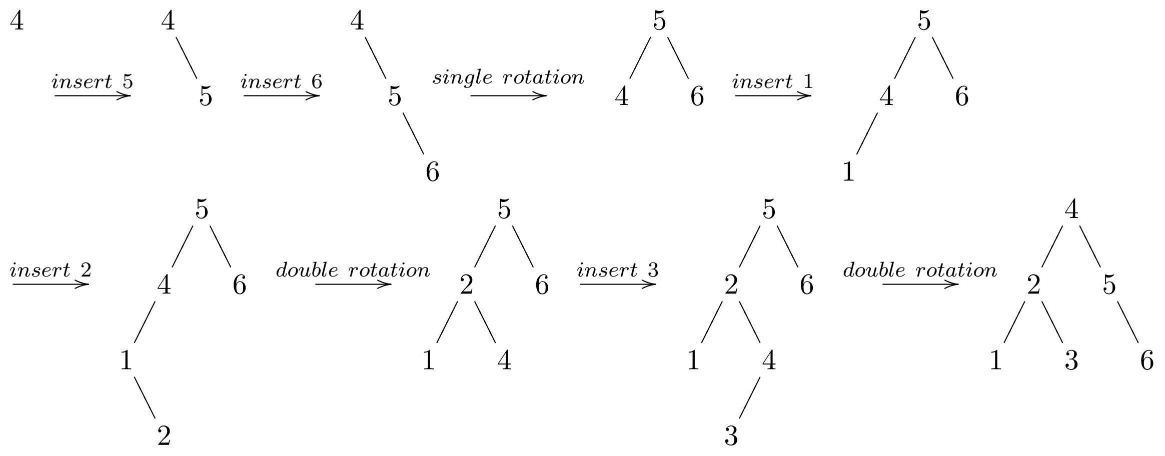
   Solution. (Wen-Chin Chan)

```
Algorithm insert($node$, $x$)
    begin
        if $node = null$ then
            $node$ := new $Node(x)$
            $node.balance := 0$
            $node.height := 0$
        else if $x < node.value$ then
            insert($node.left$, $x$)
            if $node.height < node.left.height + 1$ then
                $node.height := node.left.height + 1$
            $node.balance := node.left.height - node.right.height$
            if $node.balance = 2$ then
                if $node.left.balance = 1$ then
                    print "Single Rotate"
                else print "Double Rotate"
        else /* $x > node.value$
            insert($node.right$, $x$)
            if $node.height < node.right.height + 1$ then
                $node.height := node.right.height + 1$
            $node.balance := node.left.height - node.right.height$
            if $node.balance = -2$ then
                if $node.right.balance = -1$ then
                        print "Single Rotate"
                else print "Double Rotate"
    end
```

□

5. Show all intermediate and the final AVL trees formed by inserting the numbers 4, 5, 6, 1, 2, and 3 (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 a rotation is performed during an insertion, please also show the tree before the rotation.

Solution. (Chi-Jian Luo)



□

6. Design an efficient algorithm that, given an array A of n integers and an integer x, determine whether A contains two integers whose sum is exactly x. Please present your algorithm in an adequate pseudo code 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. The straightforward solution of trying every pair in A would take $O(n^2)$ time, as there are $\frac{n(n-1)}{2}$ possible pairs. When A is sorted (in increasing order), finding the pair (if it exists) can be done much more efficiently as follows: If $A[1] + A[n] < x$, then $A[1]$ cannot be one of the pair we are looking for, as $A[1] + A[j]$ will be smaller than x for any $j \leq n$. On the other hand, if $A[1] + A[n] > x$, then $A[n]$ cannot be one of the pair we are looking for, as $A[i] + A[n]$ will be greater than x for any $i \geq 1$. In either case, we can eliminate one element. So, we sort A first with, for example, the heapsort algorithm and then invoke the procedure below.

```
procedure Find_Pair ($A, n, x$);
begin
    $i := 1$;
    $j := n$;
    while $i < j$ do
        if $A[i] + A[j] = x$ then
            break;
        if $A[i] + A[j] < x$ then
            $i := i + 1$;
        else $j := j - 1$;
    if $i < j$ then
        print $i, j$;
    else print “no solution”
end
```

The while loop will be executed at most $n-1$ times, hence the running time of the procedure is $O(n)$. Together with the sorting part, the whole algorithm will run in $O(n \log n)$ time. $\square$

7. The Knapsack Problem 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.

   Below is an algorithm for determining whether a solution to the 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 $k - S[i] \geq 0$ then
                    if $P[i-1,k-S[i]].exist$ then
                      $P[i,k].exist := true$;
                      $P[i,k].belong := true$
end
```

   (a) Design an algorithm to recover the solution recorded in the array P. (5 points)

   Solution.

```
Procedure Print_Solution $(S, P, n, K)$;
begin
    if $\neg P[n,K].exist$ then
        print "no solution"
    else $i := n$;
         $k := K$;
         while $k > 0$ do
             if $P[i,k].belong = true$ then
               print $i$;
               $k := k - S[i]$;
             $i := i - 1$
end
```

□

(b) Modify the given algorithm to solve a variation of the knapsack problem where each item has an unlimited supply. (10 points)

Solution. Insert "$P[0,0].belong := 0$;" after "$P[0,0].exist := true$;" and modify the last five lines before "end" as follows:

```
        $P[i,k].belong := 0$
    **else if** $k - S[i] \geq 0$ **then**
        **if** $P[i, k - S[i]].exist$ **then**
            $P[i,k].exist := true$;
            $P[i,k].belong := P[i, k - S[i]].belong + 1$
```

□

8. Let $x_1, x_2, \cdots, x_n$ be a sequence of real numbers (not necessarily positive). Design an $O(n)$ algorithm to find the subsequence $x_i, x_{i+1}, \cdots, x_j$ (of consecutive elements) such that the product of the numbers in it is maximum over all consecutive subsequences. The product of the empty subsequence is defined to be 1. Please present your algorithm in an adequate pseudo code and make assumptions wherever necessary.

Solution. (Wen-Chin Chan)

```
Algorithm Maximum_Consecutive_Subsequence(X,n)
begin
  $Global\_Max := 1$;
  $Suffix\_Max := 1$;
  $Suffix\_Min := 1$;
  for i := 1 to n do
    if $X[i] > 0$ then
      if $Suffix\_Max \times X[i] > Global\_Max$ then
        $Suffix\_Max := Suffix\_Max \times X[i]$;
        $Global\_Max := Suffix\_Max$;
        $Suffix\_Min := Suffix\_Min \times X[i]$;
        if $Suffix\_Min \geq 0$ then
          $Suffix\_Min := 1$;
      else
        $Suffix\_Max := Suffix\_Max \times X[i]$;
        $Suffix\_Min := Suffix\_Min \times X[i]$;
        if $Suffix\_Max < 1$ then
          $Suffix\_Max := 1$;
        if $Suffix\_Min \geq 0$ then
```

```
                $Suffix\_Min := 1$;
        else if $X[i] < 0$ then
            if $Suffix\_Min \times X[i] > Global\_Max$ then
                $Global\_Max := Suffix\_Min \times X[i]$;
                $Suffix\_Min := Suffix\_Max \times X[i]$;
                $Suffix\_Max := Global\_Max$;
            else
                $Suffix\_Max := Suffix\_Max \times X[i]$;
                $Suffix\_Min := Suffix\_Min \times X[i]$;
                $swap(Suffix\_Max, Suffix\_Min)$;
                if $Suffix\_Max < 1$ then
                    $Suffix\_Max := 1$;
                if $Suffix\_Min \geq 0$ then
                    $Suffix\_Min := 1$;
        else /* $X[i] = 0$ */
            $Suffix\_Max := 1$;
            $Suffix\_Min := 1$;
end
```

□

9. Consider rearranging the following array into a max heap.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
8	2	3	5	12	14	7	1	6	4	10	15	13	9	11

(a) Design a systematic procedure for performing this task. Please present your procedure in an adequate pseudo code and make assumptions wherever necessary. (10 points)

Solution. There are two approaches: top-down and bottom-up. We have discussed both in class. Below is an algorithm using the bottom-up approach.

```
Algorithm Build_Heap(A,n);
begin
   for i := n DIV 2 downto 1 do
      parent := i;
      child1 := 2*parent;
      child2 := 2*parent + 1;
      if child2 > n then child2 := child1;
      if A[child1]>A[child2] then maxchild := child1
      else maxchild := child2;
      while maxchild<=n and A[parent]<A[maxchild] do
         swap(A[parent],A[maxchild]);
```

```
            parent := maxchild;
            child1 := 2*parent;
            child2 := 2*parent + 1;
            if child2 > n then child2 := child1;
            if A[child1]>A[child2] then maxchild := child1
            else maxchild := child2;
            end;
      end;
end;
```

□

(b) The procedure above most likely will consist of a number of rounds. Given the above input, please show the result (i.e., the contents of the array) after each round until it becomes a heap. (5 points)

Solution. (Wen-Chin Chan)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
8	2	3	5	12	14	7	1	6	4	10	15	13	9	11
8	2	3	5	12	14	$\underline{11}$	1	6	4	10	15	13	9	11
8	2	3	5	12	$\underline{15}$	11	1	6	4	10	$\underline{14}$	13	9	7
8	2	3	5	12	15	11	1	6	4	10	14	13	9	7
8	2	3	$\underline{6}$	12	15	11	1	$\underline{5}$	4	10	14	13	9	7
8	2	$\underline{15}$	6	12	$\underline{14}$	11	1	5	4	10	$\underline{3}$	13	9	7
8	$\underline{12}$	15	6	$\underline{10}$	14	11	1	5	4	$\underline{2}$	3	13	9	7
$\underline{15}$	12	$\underline{14}$	6	10	$\underline{13}$	11	1	5	4	2	3	$\underline{8}$	9	7

□