Suggested Solutions to Midterm Problems

1. Consider the geometric series: 1, 2, 4, 8, 16, ... Prove *by induction* that any positive integer can be written as a sum of distinct numbers from this series.

   Solution. The proof is by *strong induction* on n that represents an arbitrary positive integer.

   Base case ($n = 1$): the statement is obviously true, as 1 is itself in the geometric series.

   Inductive step ($n > 1$): we consider two cases separately: when n is even and when n is odd.

   Case 1: n is even. Let $n = 2 \times k$, where $k \geq 1$. By the induction hypothesis, let k be the sum of the series $s_1, s_2, \ldots, s_j$, which are distinct numbers taken from the geometric series. Then, n is the sum of $2 \times s_1, 2 \times s_2, \ldots, 2 \times s_j$, which are also distinct numbers from the geometric series.

   Case 2: n is odd. Let $n = 2 \times k + 1$, where $k \geq 1$. By the induction hypothesis, let k be the sum of the series $s_1, s_2, \ldots, s_j$, which are distinct numbers from the geometric series. Then, n is the sum of $1, 2 \times s_1, 2 \times s_2, \ldots, 2 \times s_j$, which are also distinct numbers from the geometric series. □

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. In the so-called implicit representation of a binary tree, the tree nodes are stored in an array, say A, such that

   (a) the root is stored in $A[1]$ and

   (b) the left child of (the node stored in) $A[i]$ is stored in $A[2i]$ and the right child in $A[2i+1]$. (Note: a nonexistent child may be indicated by a special mark/value in the corresponding cell for storing the child.)

   Prove *by induction* that, for complete binary trees, the implicit representation is compact in the sense that, if we label the tree nodes from top to bottom and left to right with the numbers 1 through n (where n is the number of nodes in the tree), then the node labeled i is stored in $A[i]$ for $1 \leq i \leq n$.

   Solution. We first observe that a complete binary tree of n nodes is obtained from another of $n-1$ nodes as follows. If node $n-1$ (according to the labeling from top to bottom and left to right) is the left child of a node, node n is simply added as the right child of the same node. If node $n-1$ is the right child of a node i ($i < n-1$), then node n is added as the left child of node $i+1$.

   The proof of compactness is by induction on n.

   Base case ($n = 1, 2$): when $n = 1$, the only element is the root which is labeled 1 and, according to the implicit representation, is stored in $A[1]$. When $n = 2$, the node labeled 1 is stored in $A[1]$ as in the case of $n = 1$. The node labeled 2 is the left child of node 1, i.e., $A[1]$, and according to the implicit representation, is stored in $A[2 \times 1]$, i.e., $A[2]$.

   Inductive step ($n > 2$): from the induction hypothesis and the observation stated in the beginning, the part of nodes 1 through $n-1$ are stored in $A[1..n-1]$ in the right order. We need to show that the last node, labeled n, will indeed be stored in $A[n]$ according to the implicit representation. There are two cases:

   When $n-1$ is even ($n - 1 \geq 2$), node $n-1$, stored in $A[n-1]$ (from the induction hypothesis), is the left child of $A[\frac{n-1}{2}]$ (according to the implicit representation), i.e., node $\frac{n-1}{2}$ (from the induction hypothesis). Hence, node n should be the right child of node $\frac{n-1}{2}$ (or $A[\frac{n-1}{2}]$) and stored in $A[2 \times \frac{n-1}{2} + 1]$, i.e., $A[n]$.

   When $n-1$ is odd ($n - 1 \geq 3$), node $n-1$, stored in $A[n-1]$, is the right child of $A[\frac{n-1-1}{2}]$, i.e., node $\frac{n-1-1}{2}$. Hence, node n should be the left child of node $\frac{n-1-1}{2} + 1$ (or $A[\frac{n-1-1}{2} + 1]$) and stored in $A[2(\frac{n-1-1}{2} + 1)]$, i.e., $A[n]$.

   □

4. (15 points) 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) (5 points) 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) (10 points) 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 $h = O(\log n)$. □

5. (15 points) Consider the Knapsack Problem: Given a set S of n items, where the i-th 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 a recursive version of the algorithm for determining whether a solution to the Knapsack Problem exists, where we have ignored the tag values for recording the subset of items that constitute the solution. The algorithm should be invoked with Knapsack(n, K).

```
**Algorithm Knapsack**$(m, k)$;
**begin**
    **if** $k = 0$ **then** return $true$;
    **if** $m = 0$ **then** return $false$;
    **if** Knapsack$(m-1, k)$ **then** return $true$
    **else if** $k - S[m] \geq 0$ **then** return Knapsack$(m-1, k - S[m])$
        **else** return $false$;
**end**
```

(a) (5 points) Given an input, Knapsack(m, k) may be invoked with the same combination of m and k at different points of execution. Why? Please give an example.

Solution. Suppose there are 4 items, with sizes 2, 3, 5, and 5, and we are looking for a subset whose sizes sum to 15. Assuming that recursive function calls are used, below is the two-dimensional table P whose entries are filled with -, O, I, or left blank when the algorithm terminates.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	O		-	-		(-)		(-)	(-)		(-)		-	-		-
$k_1 = 2$			I			-		(-)			(-)		-			-
$k_2 = 3$						I					(-)					-
$k_3 = 5$											I					-
$k_4 = 5$																I

The parenthesized entries are invoked more than once. In particular, Knapsack$(2, 10)$ is invoked via the following two different invocation chains: Knapsack$(4, 15)$ → Knapsack$(3, 15)$ → Knapsack$(2, 10)$ and Knapsack$(4, 15)$ → Knapsack$(3, 10)$ → Knapsack$(2, 10)$. □

(b) (10 points) How will you propose to avoid duplicate invocations? Please revise the code to incorporate your proposal. (Hint: use an array to memorize the result of an invocation.)

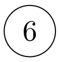
Solution. The following algorithm assumes a global two-dimensional array P, whose entries are three-valued: 0 (result not known), -1 (negative), 1 (positive).

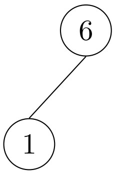
```
Algorithm Knapsack($m, k$);
begin
    for $i := 0$ to $m$ do
        for $j := 0$ to $k$ do
            $P[i, j] := 0$;   // result not known yet
    return Mem_Knapsack($m, k$);
end

Function Mem_Knapsack($m, k$);
begin
    if $P[m, k] \neq 0$ then return $P[m, k]$;
    if $m = 0$ then $P[m, k] := -1$;
    if $k = 0$ then $P[m, k] := 1$;
    if Mem_Knapsack($m - 1, k$) then $P[m, k] := 1$
    else $P[m - 1, k] := -1$;
         if $k - S[m] \geq 0$ then return Mem_Knapsack($m - 1, k - S[m]$)
         else $P[m, k] := -1$;
    return $P[m, k]$;
end
```

□

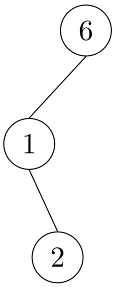
6. Show all intermediate and the final AVL trees formed by inserting the numbers 6, 1, 2, 5, 4, and 3 (in this order) into an empty tree. Please use the following ordering convention: the key of an internal node is larger than that of its left child and smaller than that of its right child. If re-balancing operations are performed, please also show the tree before re-balancing and indicate what type of rotation is used in the re-balancing.

Solution. (Willy Chang)

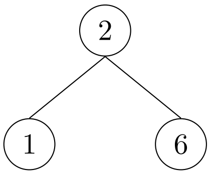
Insert 6: 



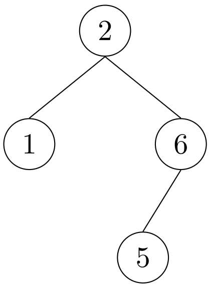
Insert 1:



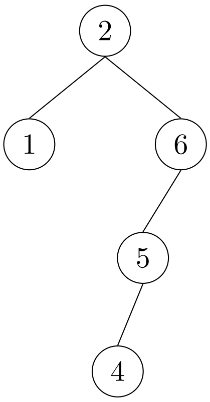
Insert 2: →



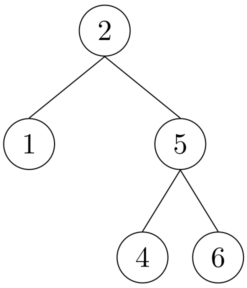
([LR] double right rotation)



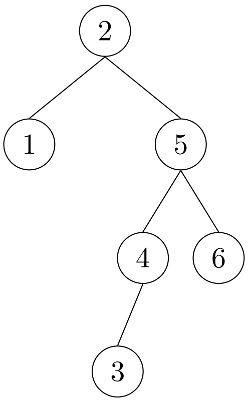
Insert 5:



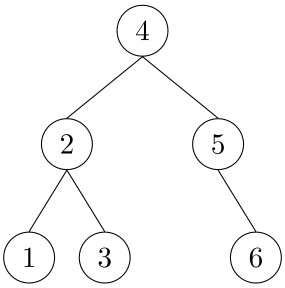
Insert 4: →



([LL]/[R] single right rotation on chain 6-5-4)



Insert 3: →



([RL] double left rotation on chain 2-5-4) □

7. Apply the Quicksort algorithm to the following array. Show the contents of the array after each partition operation. If you use a different partition algorithm (from the one discussed in class), please describe it.

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. (Wei-Shao Tang)

1	2	3	4	5	6	7	8	9	10	11	12
9	10	4	6	11	7	8	2	1	12	3	5
1	**5**	4	6	**3**	7	8	2	$\mathbf{\underline{9}}$	12	**11**	**10**
$\mathbf{\underline{1}}$	5	4	6	3	7	8	2	$\underline{9}$	12	11	10
$\underline{1}$	**3**	4	**2**	$\mathbf{\underline{5}}$	7	8	**6**	$\underline{9}$	12	11	10
$\underline{1}$	**2**	$\mathbf{\underline{3}}$	**4**	$\underline{5}$	7	8	6	$\underline{9}$	12	11	10
$\underline{1}$	2	$\underline{3}$	4	$\underline{5}$	**6**	$\mathbf{\underline{7}}$	**8**	$\underline{9}$	12	11	10
$\underline{1}$	2	$\underline{3}$	4	$\underline{5}$	6	$\underline{7}$	8	$\underline{9}$	**10**	**11**	$\mathbf{\underline{12}}$
$\underline{1}$	2	$\underline{3}$	4	$\underline{5}$	6	$\underline{7}$	8	$\underline{9}$	$\mathbf{\underline{10}}$	11	$\underline{12}$

□

8. Please present in suitable pseudocode the algorithm (discussed in class) for rearranging an array $A[1..n]$ of n integers into a max heap using the *bottom-up* approach.

   Solution.

```
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;
```

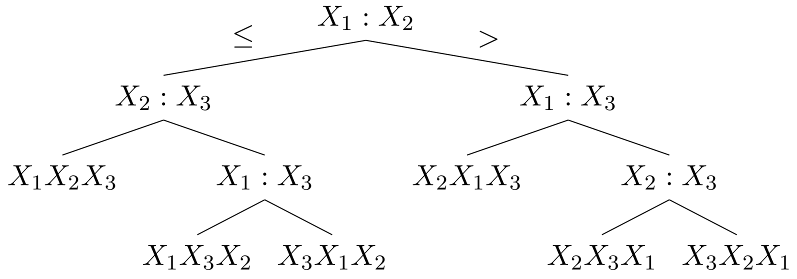
□

9. Below is a variant of the insertion sort algorithm.

```
Algorithm Insertion_Sort ($A, n$);
begin
    for $i := 2$ to $n$ do
        $x := A[i]$;
        $j := i$;
        while $j > 1$ and $A[j-1] > x$ do
            $A[j] := A[j-1]$;
            $j := j - 1$;
        end while
        $A[j] := x$;
    **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 to refer to the elements (in this order) of the original input array.

Solution.



□

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