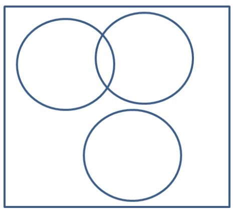
Suggested Solutions to Midterm Problems

1. Prove *by induction* that the regions formed by n circles in the plane can be colored with two colors such that any neighboring regions (which share an arc, not just a point) are colored differently.

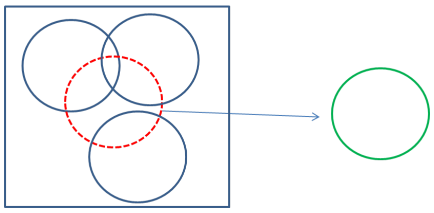
   Solution. (Wei-Hsien Chang)

   The proof is by induction on the number (n) of circles.

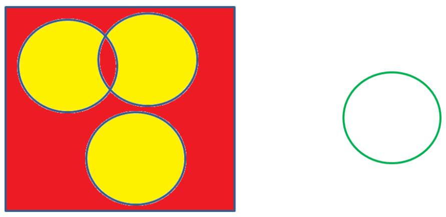
   - Base case: n=1, trivial.
   - Induction Hypothesis: We assume that for all n, $1 \leq n < k$, the regions formed by n circles can be colored as required.
   - Inductive Step: $n = k$ $(k > 1)$.
     We call the graph with k circles G.

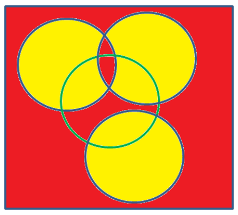
     Now we randomly remove 1 circle from G to get a graph with the remaining $(k-1)$ circles, called G'.

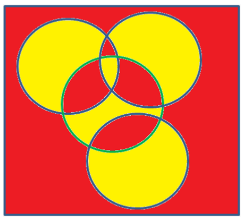
     Because G' has less than k circles, we can color it due to the induction hypothesis.

     Now we put the removed circle back and observe that several parts of colored G' fall inside the circle.

To meet the requirement, we flip the colors of the regions inside the circle.



Both the regions of G' (and hence of G) outside the circle and the regions of G inside the circle meet the coloring requirement.
The regions that share an arc on the circle also have different colors, because they are of the same color in G', and we have flipped one of the colors.
So, the coloring of all regions in G meets the requirement. This completes the inductive proof.

□

2. Prove *by induction* that, for complete binary trees with three nodes or more, one of the two subtrees under the root is a full binary tree and the other is a complete binary tree. (Note: full binary trees are special cases of complete binary trees.)

   Solution. For a complete binary tree with n nodes, its nodes can be numbered 1 through n compactly such that the root is numbered 1 and, for a node numbered i (≥ 1), its left child (if existent) is numbered $2i$ and its right child (if existent) is numbered $2i+1$. Conversely, a binary tree whose nodes can be compactly numbered as above must be a complete binary tree. With this compact numbering, a node can be uniquely identified by a number, and the parent of a non-root node i (> 1) is identified by $\lfloor \frac{i}{2} \rfloor$. The *last/youngest* parent is the node with the largest number that has a child. Clearly, it is identified by $\lfloor \frac{n}{2} \rfloor$ and its immediate right sibling is identified by $\lfloor \frac{n}{2} \rfloor + 1$. When n is even, to obtain a complete binary tree with $n+1$ nodes, the new leaf (numbered $n+1$) should be added as the right child of the youngest parent, namely node $\lfloor \frac{n}{2} \rfloor$, and when n is odd, it should be added as the left child of node $\lfloor \frac{n}{2} \rfloor + 1$. Let us refer to this node where the next new leaf should be added as the *prospective* parent.

   The height (or depth) of a complete binary tree is the number of levels (or parent-child edges) one needs to go through from the root to the last node. The number of nodes of a full binary can be calculated as $2^{h+1} - 1$, where h is the height of the tree. We say that a complete binary tree is *proper* if it is not a full binary tree. To facilitate the inductive proof, we refine/strengthen the proposition in the problem statement as follows:

   > For a complete binary tree with n (≥ 3) nodes, the two subtrees of the root satisfy exactly one of the following conditions:
   >
   > (a) The left subtree is a proper complete binary tree and is one-level taller than the right subtree, which is a full binary tree. In this case, the prospective parent is in the left subtree and it is also the prospective parent of the subtree.
   >
   > (b) The left subtree is a full binary tree and is as tall as as the right subtree, which is a proper complete binary tree. In this case, the prospective parent is in the right subtree and it is also the prospective parent of the subtree.
   >
   > (c) The two subtrees are both full binary trees of the same height. In this case, the prospective parent is in the left subtree and it is also the prospective parent of the subtree.

(d) The two subtrees are both full binary trees and the left subtree is one-level taller than the right subtree. In this case, the prospective parent is in the right subtree and it is also the prospective parent of the subtree.

Now, a proof by induction on the number of nodes should be easier to construct and is left as an exercise. □

3. Consider the following variant of Euclid's algorithm for computing the greatest common divisor of two positive integers.

```
**Algorithm Euclid_Simplified** $(m, n)$;
**begin**
    // assume that $m > 0 \wedge n > 0$
    $x := m$;
    $y := n$;
    **while** $x \neq 0 \wedge y \neq 0$ **do**
        **if** $x < y$ **then** swap($x$,$y$);
        $x := x - y$;
    **od**
    ...
**end**
```

where swap(x,y) exchanges the values of x and y.

(a) (5 points) To speak about the values of a variable at different times during an execution, let m', n', x', and y' denote respectively the new values of m, n, x, and y after the next iteration of the while loop (m, n, x, and y themselves denote the current values of these variables at the start of the next iteration). Please give a precise relation between m', n', x', and y' and m, n, x, and y.

Solution. From the loop body, we deduce the following relationship (assuming that the loop condition holds):

$$\begin{array}{ll} & ((x < y) \to (x' = y - x) \wedge (y' = x)) \\ \wedge & ((x \geq y) \to (x' = x - y) \wedge (y' = y)) \\ \wedge & m' = m \\ \wedge & n' = n \end{array}$$

□

(b) (10 points) Prove *by induction* that the following is a loop invariant of the while loop:

$$x \geq 0 \wedge y \geq 0 \wedge (x \neq 0 \vee y \neq 0) \wedge \gcd(x, y) = \gcd(m, n).$$

(Note: by convention, $\gcd(0, z) = \gcd(z, 0) = z$ for $z > 0$.)

Solution. Let $Inv(m, n, x, y)$ denote the assertion to be proven a loop invariant.

(1) When the flow of control reaches the loop for the first time, $x = m$ and $y = n$, with $m > 0$ and $n > 0$. Obviously, $x \geq 0$, $y \geq 0$, $x \neq 0 \vee y \neq 0$, and $\gcd(x, y) = \gcd(m, n)$ and therefore the assertion $Inv(m, n, x, y)$ holds.

(2) Assume that $Inv(m, n, x, y)$ is true at the start of the next iteration and the loop condition $(x \neq 0 \wedge y \neq 0)$ holds. We need to show that $Inv(m', n', x', y')$ also holds.

There are two cases to consider: when $x < y$ and when $x \geq y$. We prove the first case; the second case can be proven similarly.

Suppose $x < y$. $x' = y - x > 0$ and hence $x' \geq 0$; also, $y' = x \geq 0$ (from the induction hypothesis). These also imply that $x' \neq 0 \vee y' \neq 0$. $\gcd(x', y') = \gcd(y - x, x) = \gcd(y, x) = \gcd(x, y)$, which from the induction hypothesis equals $\gcd(m, n) = \gcd(m', n')$, and therefore $\gcd(x', y') = \gcd(m', n')$. Therefore, $Inv(m', n', x', y')$ holds and this concludes the proof. $\square$

4. 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.

   We have discussed in class two approaches to implementing a solution that we designed by induction: one uses dynamic programming (see the Appendix), while the other uses recursive function calls.

   Suppose there are 5 items, with sizes $2, 3, 5, 7, 8$, and we are looking for a subset whose sizes sum to 15. Assuming recursive function calls are used, please give the two-dimension table P whose entries are filled with -, O, I, or left blank when the algorithm terminates. Which entries of $P[n, K]$ are visited/computed more than once? Please mark those entries in the table.

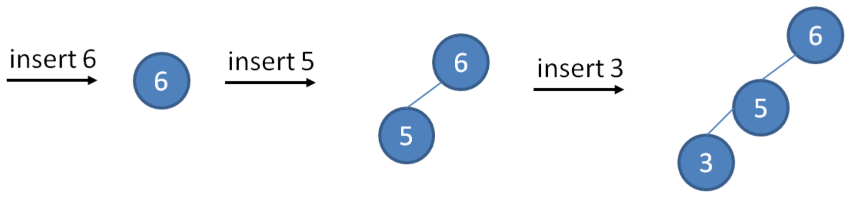
   Solution.

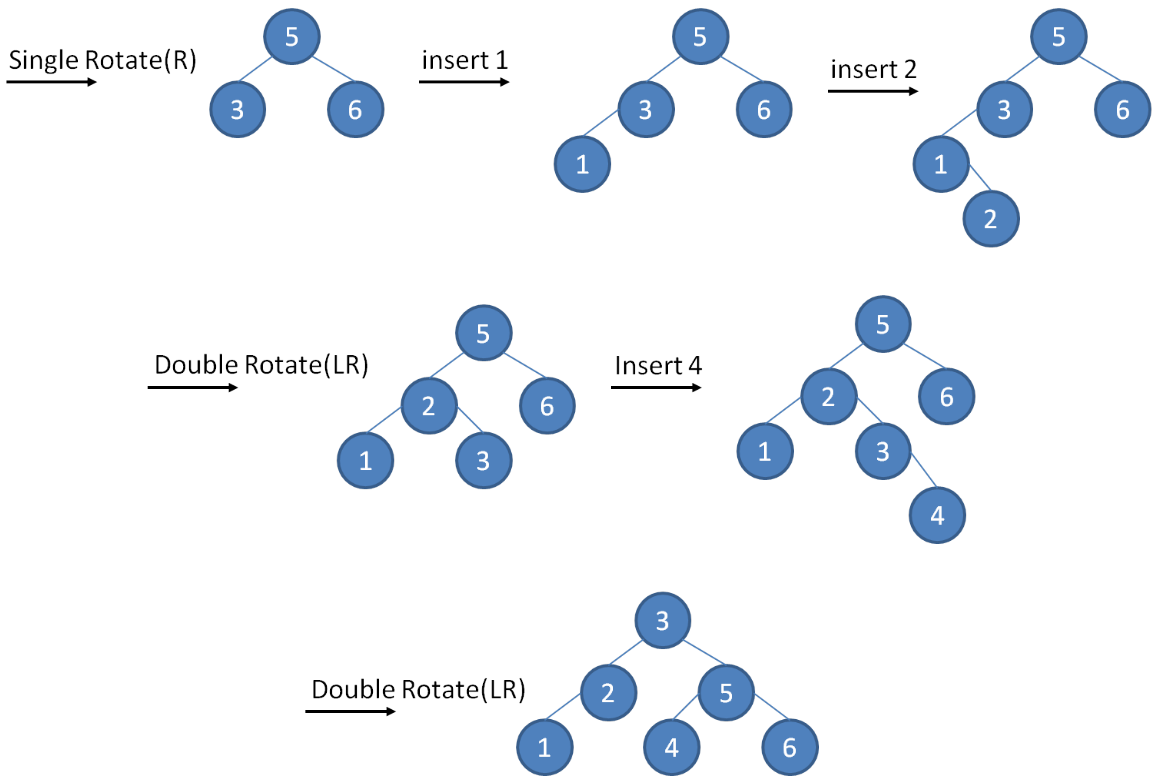
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	O	-		(-)		(-)	-	-	(-)		(-)		-	-		-
$k_1 = 2$	O			-		-		-	-		-		-			-
$k_2 = 3$				I					-		-					-
$k_3 = 5$									I							-
$k_4 = 7$																I
$k_5 = 8$																O

   $P[0, 3]$, $P[0, 5]$, $P[0, 8]$, and $P[0, 10]$ are visited more than once. $\square$

5. Show all intermediate and the final AVL trees formed by inserting the numbers 6, 5, 3, 1, 2, and 4 (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. (Wei-Hsien Chang)





□

6. The input is a set S with n real numbers. Design an $O(n)$ time algorithm to find a number that is *not* in the set. Prove that $\Omega(n)$ is a lower bound on the number of steps required to solve this problem.

   Solution. When there is just one number (i.e., $n = 1$) in S, it is trivial to find a real number different from the number in S, e.g., by adding one to or subtracting one from the existing number. When there are exactly two numbers, we simply take their average which will be different from both numbers, since by the definition of a set, all numbers in S are distinct. When there are more than two numbers, we proceed as follows.

   Store the first two numbers as a pair and read the remaining numbers one by one. When the next number read falls between the pair, we replace the smaller of the pair by the number just read. At the end we will have a pair of real numbers and none of the numbers in S falls between the pair. We take the average of the pair which will be different any number in S.

   We next argue for the lower bound $\Omega(n)$. This is quite straightforward, since every number in S must be read and there are n numbers. Any algorithm that skips a number may return a wrong result, as the result may happen to be equal to the number that is skipped.
   □

7. 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. $\square$

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

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. (Jui-Shun Lai)

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

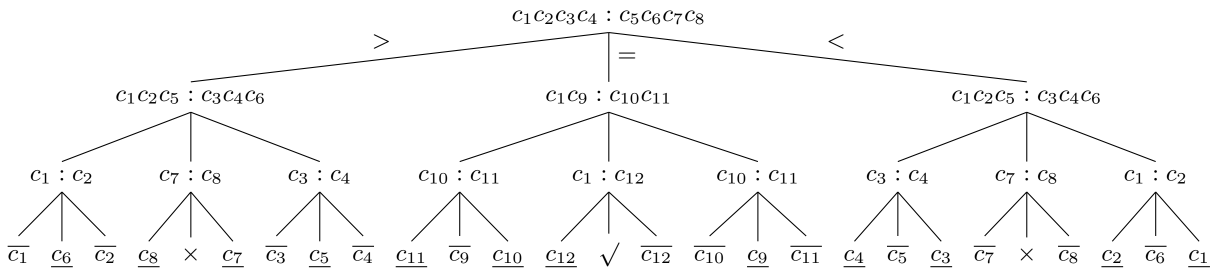
$\square$

9. (15 points) Below is a variation of the n-coins problem.

You are given a set of n coins $\{c_1, c_2, \ldots, c_n\}$, among which at least $n-1$ are identical "true" coins and at most one coin is "false". A false coin is either *lighter* or *heavier* than a true coin. Also, you are given a balance scale, which you may use to compare the total weight of any m coins with that of any other m coins. The problem is to find the "false" coin, or show that there is no such coin, by making some sequence of comparisons using the balance scale.

(a) For the case of $n = 12$, design a scheme to find the false coin (if there is one) with only three comparisons using the balance. Please use c_1, c_2, …, c_{12} to identify the coins in your scheme.

Solution. Below is a scheme, presented as a decision tree, for finding the false coin (if there is any) among the given 12 coins. As the label of a leaf, $\overline{c_i}$ ($\underline{c_i}$) means that coin c_i is a heavier (lighter) false coin. Two of the leaves are impossible outcomes and are marked with $\times$, while $\surd$ marks the outcome that there is no false coin.


$c_1c_2c_3c_4 : c_5c_6c_7c_8$
$>$
$=$
$<$
$c_1c_2c_5 : c_3c_4c_6$
$c_1c_9 : c_{10}c_{11}$
$c_1c_2c_5 : c_3c_4c_6$
$c_1 : c_2$
$c_7 : c_8$
$c_3 : c_4$
$c_{10} : c_{11}$
$c_1 : c_{12}$
$c_{10} : c_{11}$
$c_3 : c_4$
$c_7 : c_8$
$c_1 : c_2$
$\overline{c_1}$ $\underline{c_6}$ $\overline{c_2}$ $\underline{c_8}$ $\times$ $\underline{c_7}$ $\overline{c_3}$ $\underline{c_5}$ $\overline{c_4}$ $\underline{c_{11}}$ $\overline{c_9}$ $\underline{c_{10}}$ $\underline{c_{12}}$ $\surd$ $\overline{c_{12}}$ $\overline{c_{10}}$ $\underline{c_9}$ $\overline{c_{11}}$ $\underline{c_4}$ $\overline{c_5}$ $\underline{c_3}$ $\overline{c_7}$ $\times$ $\overline{c_8}$ $\underline{c_2}$ $\overline{c_6}$ $\underline{c_1}$

□

(b) Prove that, when $n = 12$, it is not possible to find the false coin with just two comparisons, implying that using just three comparisons is optimal. (Hint: think about decision trees and how many possible outcomes there can be.)

Solution. One of the 12 coins may be a false coin and it can be either lighter or heavier, which represents 24 possible outcomes. Together with the case of no false coin, there are totally 25 possible outcomes. One comparison with the balance may produce three possible results, and two comparisons produces at most nine results. This is insufficient for covering the 25 possible outcomes. □

(c) To generalize the preceding result, prove that in the worst case it is impossible to solve the n-coins problem with k comparisons (for any n and k) if $n > \frac{(3^k-1)}{2}$.

Solution. As we have noted in the previous subproblem, each use of the balance scale may produce three possible results. Solutions to the n-coins problem fall within the model of decision trees where each internal node has three branches. Any such tree of height k can contain at most 3^k leaves, representing at most 3^k different outcomes. For n coins, there are $2n+1$ possible outcomes: (a) one of the n coins is lighter (n possibilities), (b) one of the n coins is heavier (another n possibilities), and (c) none of the coins is false (one possibility). Therefore, 3^k must be greater than or equal to $2n+1$ for a solution with k comparisons to exist. In other words, no solution with k comparisons exists if $2n+1 > 3^k$, i.e., if $n > \frac{(3^k-1)}{2}$. □

Appendix

- Below is an 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**
```