Question 1: (30 points, 2 points each)

Indicate whether the following statements are true or false. *Give a short justification for each*. Credits will be given only if the justification is correct.

a) If $f(n)$ is $\Theta(h(n))$, then $2^{f(n)}$ is $\Theta(2^{h(n)})$.

False. Counter-example: f(n) = log(n), h(n) = 2 log(n). Then clearly f(n) is $\Theta(h(n))$, but
$2^{f(n)} = 2^{log(n)} = n$
$2^{h(n)} = 2^{2*log(n)} = (2^{log(n)})^{2)} = n^2$
so $2^{f(n)}$ is not $\Theta(2^{h(n)})$.

b) $n^{0.9}\log(n)$ is $O(n)$.

True. We use the limit of the ratio:

Lim $n^{0.9}\log(n)$ / n = lim log(n) / $n^{0.1}$ = lim d/dn (log(n)) / d/dn($n^{0.1}$) = lim (1/n) / (0.1 $n^{-0.1}$) = lim(10/ $n^{1.1}$) = 0, so $n^{0.9}\log(n)$ is $O(n)$.

c) Let $f(n)$ and $g(n)$ be two non-negative functions. If there exists a number n_0 such that $f(n_0) < g(n_0)$, then $f(n)$ is O($g(n)$).

False. For example, for f(n) = n^2 and g(n) = n + 2, and n_0 = 1, we have
$f(n_0) = 1 < g(n_0) = 3$, but $f(n)$ is not O($g(n)$).

d) Suppose that an algorithm A has worst-case running time $O(n \log(n))$ and an algo. B for the same problem has worst-case running time $O(n^2)$. Then it is possible that for some value $n_0 > 0$, the algorithm B runs faster than algorithm A on all inputs of size n_0.

True. For example, if the best-case running times of algorithms A is
$T_{A,BEST}(n) = n \log(n) + 10$,
and the worst-case running time of algoirhtm M is
$T_{B,WORST}(n) = n^2 + 1$,
then algorithm B will run faster than algorithm A on inputs of size 1.

e) It is possible to write a sorting algorithm for which the best-case running time on an array of n integers is $O(n)$.

True. Simply add to any existing sorting algorithm a loops that first checks if the array is already sorted, and stops if it is. The loop will run in time O(n), so the best case running time of the algorithm will be O(n).

f) If one defines $T(n) = \begin{cases} 2\,T(n-1) + n & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$
then the explicit formula for $T(n)$ is $T(n) = 2n^2 - 3n + 2$

Based on the recurrence, we get
$T(1) = 1, T(2) = 4, T(3) = 11, T(4) = 26$

From the explicit formula, we get
$T(1) = 1, T(2) = 4, T(3) = 11, T(4) = 22$
So the explicit formula is incorrect.

g) The only important thing to consider when designing a hash function is that the function needs to be easy to compute.

False. A good hash function should distribute elements as evenly as possible among buckets.

h) In a country where coins have values 1 ¢, 3 ¢, 5 ¢, and 8 ¢, the greedy algorithm for making change is not optimal.

True. To produce a total of 15 ¢, the greedy algorithm will need 8 + 5 + 1 + 1 = 4 coins but it is possible to do better: 5+5+5 = 3 coins.

i) Suppose that the graph below represents a miniature web on which the Google page-rank algorithm is executed. Which of page A or B would obtain the highest page-rank score?

SKIPPED.

j) NP is the class of decision problems that cannot be solved by any polynomial-time algorithm.

False. NP is the class of problems for which positive instances can be checked in polynomial time. In particular, in includes all problems in P.

k) Nobody knows if the *k*-CLIQUE problem (the problem of determining if a graph contains a clique of size at least *k*) is decidable.

False. It is decidable, e.g. by simply trying all possible (n choose k) subsets of vertices and checking if they are connected.

l) If an algorithm uses random numbers during its execution, then there is always a small probability that its output will be incorrect.

False. Las vegas algorithms, such as randomized QuickSort, will always produce the correct answer.

m) This question was encrypted using Ceasar's cypher:
JO DPNQ-361, XF VTFE UIF KBWB QSPHSBNNJOH MBOHVBHF.

Shift every letter/number by minus one:
IN COMP-250, WE USED THE JAVA PROGRAMMING LANGUAGE
→ True

n) In a game between two players *A* and *B* where it is *A*'s turn to play, a position is a loss for *A* if and only if no moves available to *A* lead to a losing position for *B*.

False. That position could be a tie for A, if there exists a moves that leads to a tie.

o) Given enough time to run, the hill-climbing heuristic will always find the optimal solution to any optimization problem.

False, it may get stuck in a local optimum.

Question 2. (12 points)

Give the worst-case running time of the following algorithms, using the simplest Θ() notation (big-Theta notation) possible. The running time may not necessarily be expressed as a function *n*. No justification needed.

	Θ() Running time

Algorithm1(int n)
    $i \leftarrow n^2$
    while (i > 1) **do**
        $i \leftarrow i / 2$
	$\Theta(\log(n^2)) = \Theta(\log(n))$

Algorithm2(int A[0…n-1], int n)
    // A is an array of n integers
    mergeSort(A, 0, n-1)
    quickSort(A, 0, n-1)
	$\Theta(n \log(n) + n^2) = \Theta(n^2)$

c) (4 points) Give the explicit formula for the following recurrence (no justification needed):

$$T(n) = \begin{cases} T(n-1) + n & \text{if } n > 1 \\ 5 & \text{if } n = 1 \end{cases}$$

Using the substitution method yields:
$T(n) = n\,(n+1)/2 + 4$

d) (4 points) Draw a heap with 7 nodes, containing keys 1, 2, 3, 4, 5, 6, and 7, and such that a post-order traversal would visit nodes in decreasing order. (No justification needed).

```
        1
       / \
    5      2
   / \    / \
  7   6  4    3
```

e) (4 points) What is the result of running the partition algorithm, as defined in class in the context of the QuickSort algorithm, on the following array:

A = [6 2 3 7 2 1 8 5]

Answer: [1 2 3 2 5 6 8 7]

Question 4. (10 points)

Consider the Fibonacci sequence F_0, F_1, F_2,... defined in class:

$F_0 = 0$
$F_1 = 1$
$F_n = F_{n-2} + F_{n-1}$ if $n \geq 2$

The following recursive algorithm computes the *n*-th term of the Fibonacci sequence:

```
Algorithm Fib(int n)
Input: an integer n ≥ 0
Output: Returns Fn
if (n = 0) then
        print " A "
        return 0
if (n = 1) then
        print " B B "
        return 1
for i = 0 to n do                     // note: this means 0 to n inclusively
        print " C "
return Fib(n-2) + Fib(n-1)
```

a) (4 points) What will be printed when Fib(4) is executed?

```
Execution stack:                                Printed
Fib(4)
   for i = 0 to 4 do print "C"                  C C C C C
   Fib(2)
     for i = 0 to 2 do print "C"                C C C
     Fib(0)
          print "A"                             A
     Fib(1)
          Print "B B"                           B B
   Fib(3)
     for i = 0 to 3 do print "C"                C C C C
     Fib(1)
          Print "B B"                           B B
     Fib(2)
          for i = 0 to 2 do print "C"           C C C
          Fib(0)
               print "A"                        A
          Fib(1)
               Print "B B"                      B B
```

Result: C C C C C C C C A B B C C C C B B C C C A B B

b) (6 points)

Let *A(n)* be the total number of letters "A" that will be printed when executing Fib(*n*).

Let *B(n)* be the total number of letters "B" that will be printed when executing Fib(*n*).

Let *C(n)* be the total number of letters "C" that will be printed when executing Fib(*n*).

For example, *A*(3) = 1, *B*(3) = 4, and C(3) = 7.

Write a recurrence for *A(n).*

$$A(n) = \begin{cases} 1 & \text{if } n=0 \\ 0 & \text{if } n=1 \\ A(n-2) + A(n-1) & \text{if } n>1 \end{cases}$$

Write a recurrence for *B(n).*

$$B(n) = \begin{cases} 0 & \text{if } n=0 \\ 2 & \text{if } n=1 \\ B(n-2) + B(n-1) & \text{if } n>1 \end{cases}$$

Write a recurrence for *C(n).*

$$C(n) = \begin{cases} 0 & \text{if } n=0 \text{ or } 1 \\ n + 1 + C(n-2) + C(n-1) & \text{if } n>1 \end{cases}$$

Question 5. (14 points)

Let T be a binary search tree storing distinct integers. Assume that you have a method subtreeSize(treeNode n) that returns the number of nodes in the subtree rooted at n, including n itself. Assume at any call to subtreeSize(treeNode n) takes time O(1).

Problem: Write an algorithm that finds the k-th smallest key contained in the tree (e.g., when k=0, it returns the smallest key. When k=1, it returns the second smallest, etc.). Your algorithm must run in worst-case time O(h), where h is the height of the binary search tree (but you don't need to prove it).

Algorithm findKth(treeNode n, int k)
Input: A treeNode n and an integer k
Output: The k-th smallest key contained in the subtree rooted at n.
/* WRITE YOUR PSEUDOCODE HERE */

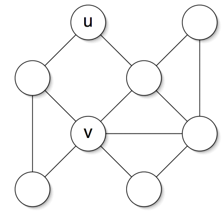
if (n = null) return null

leftSize = subtreeSize(n.leftChild)
if (leftSize = k) return n.key
if (leftSize > k) return findKth(n.leftChild, k)
if (leftSize < k) return findKth(n.rightChild, k – leftSize – 1)

Question 6. (16 points)

In an undirected connected graph $G=(V,E)$, the distance $d(a,b)$ between vertices a and b is the number of edges in the shortest path between a and b. The excentricity of a vertex a is defined as the largest distance between vertex a and any other vertex:

excentricity(a) = max { $d(a,b) : b \in V$ }



For example, in the graph to the right, excentricity(u) = 3 and excentricity(v) = 2.

Problem: Write an algorithm to compute the excentricity of a given vertex in a graph.

Use the following standard graph ADT methods if needed.

- *getNeighbors*(vertex v) returns the list of vertices that are the adjacent to vertex v. It is ok for you to write something like: for each vertex w in *getNeighbors*(v) do ...
- boolean *getVisited*(vertex v) returns TRUE if and only if vertex v has been marked as visited.
- *setVisited*(vertex v, boolean b) sets the visited status of vertex v to b.

You may also want to associate to each vertex an integer called distance, which can be set and accessed through

- int getDistance(vertex v) returns the distance stored in v.
- setDistance(vertex v, int d) sets the distance stored in v to d

Algorithm excentricity(vertex u)
Input: a vertex u from the graph
Output: the excentricity of u
/* WRITE YOUR PSEUDOCODE HERE */

SOLUTION NOT WRITTEN YET…