1. Which of the following best characterizes the relationship between the two functions (2 pts)

$f(n)$ is the closed form of the function $T(n) = 3T(n/4) + n$
$g(n)$ is the closed form of the function $T(n) = 2T(n/2) + 1$

[✓ A)] $f(n)$ grows asymptotically faster than $g(n)$
B) $g(n)$ grows asymptotically faster than $f(n)$
C) $f(n)$ and $g(n)$ grow at the same rate

2. Which of the following best describes the runtime complexity of the recurrence relation: (2 pts)

$$T(n) = 3T(n/3) + \sqrt{n}$$

[✓ a.] Q(n)
b. Q$(n \log n)$
c. Q(n^2)
d. Q$(n^2 \log n)$
e. None of the above

3. Which of the following best characterizes the two statements about the two functions: (2 pts)

$f(n) = n \lg \sqrt{n}$ and $g(n) = n \lg n$

A) $f(n)$ Î $O(g(n))$
B) $f(n)$ Î $o(g(n))$

a. Only A is TRUE
b. Only B is TRUE
[✓ c.] Both A and B are TRUE
d. Both A and B are False

4. Given the recursively-defined function

$$T(n) = \begin{cases} 2T\left(\left\lfloor \sqrt{n} \right\rfloor\right) + \lg n & n > 1 \\ 200 & n = 1 \end{cases}$$

we know that we cannot simply analyze the runtime using Master Theorem. But we can gain some understanding of the running time using an induction argument as our motivation. Prove that $T(n)$ Î $O(n\lg n)$ using the **definition** of big-O. As part of this process, fill in the following information: (4 pts)

What inequality will you specifically be proving? $T(n)$ £ ____________________________________

What are the constants that you will be determining? ____

What is a lower bound of one of your constants in terms of the other? __________________________

Give specific values for your constants: _____________________________

You will need a base case. To help you determine the necessary constants, fill in as many rows in the following table as necessary in order to ensure that your choices are appropriate (I may have included more than you will need):

n	T(n)	Your function (based on your inequality above)

THIS IS BACKGROUND INFORMATION FOR PROBLEMS 5 – 8:

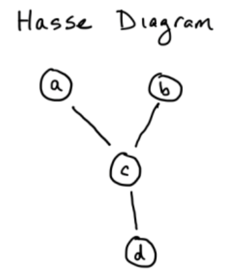
Recall that a Hasse diagram is a graphical rendering of a partially ordered set. Suppose that R is a partially ordered set, or a *poset*, and is therefore a relation on a set A, and the (**undirected**) graph, G = (V, E) is a Hasse diagram representing R, so that each vertex, v, in V is an element of A (in other words, A = V), and each edge in G is an edge in the **Hasse diagram representation** of R. This is exactly how I presented Hasse diagrams in class. I am just trying to remind you, and make sure that you understand this framework.

In addition, suppose that each vertex has a level associated with it, which is denoted by level(v), where level(v) is a whole number. If u and v are vertices in V, and (u, v) is an edge in E, then level(v) ≠ level(u). You should NOT assume that | level(v) – level(u) | = 1, however. An example of a Hasse diagram, G, is shown below:

A = { a, b, c, d}

G = (V, E) where V = { a, b, c, d } and E = { (a, c), (b, c), (c, d) }

level(a) = 5, level(b) = 4, level(c)=3, level(d) = 2



Suppose that we are interested in the following algorithms describing a process on a relation, R, represented by an adjacency matrix, M:

ReflexiveClosure(M)

Input: An n x n adjacency matrix M, representing a relation, R, on a set A, where | A | = n

Output: An n x n matrix MRC, representing the reflexive closure of R

SymmetricClosure(M)

Input: An n x n adjacency matrix M, representing a relation, R, on a set A, where | A | = n

Output: An n x n matrix MSC, representing the symmetric closure of R

TransitiveClosure(M)

Input: An n x n adjacency matrix M, representing a relation, R, on a set A, where | A | = n

Output: An n x n matrix MTC, representing the transitive closure of R

CreateMatrix(n, k):

Input: a whole number n, and a real number k

Output: an n x n matrix whose elements are all equal to k

You will be able to assume these algorithms work to convert the input to the output, as described, **UNLESS** a problem specifically requires you to develop them, of course! You may not change the data structure for the input or the output, however. You may want to use these functions in Problem 7, which is why I'm providing their names for reference here.

5. Develop a reasonably efficient algorithm for the SymmetricClosure(M) process. Your algorithm should be written at a high level (not code, but if you are doing a calculation, it is OK to indicate that particular calculation…as an example, $x \leftarrow 5t + 4$ is fine, but $x \leftarrow 4$ ^ 6 is not, since I am not sure what you mean by the symbol ^). What is the run-time of your algorithm? (5 pts)

   KEY IDEA: The Key idea is to add the inverse edge for every edge that exists in R. So for every (a,b) contained in R, we need to add (b,a) in order to obtain symmetric closure. Given an adjacency matrix, we'll need to iterate through each vertex looking for edges and keeping track of edges we've visited. When we find an edge, we will reverse the order of the edges, and change the value to 1 to indicate there is an edge going both directions.

   RUN-TIME: ______________O(V + E)______________

   Algorithm: SymmetricClosure(M)

   Input: An n x n adjacency matrix M, representing a relation, R, on a set A, where | A | = n

   Output: An n x n matrix MSC, representing the symmetric closure of R

   PROCESS:

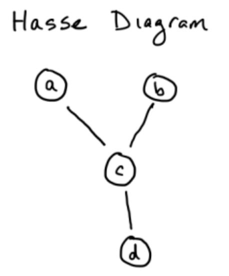
```
    Iterate over adjacency matrix checking for edges
    Once edge is found, reverse the order of the vertices and change value to 1
    Do this until all edges have been visited
    Return n x n matrix MSC
```

6. For this exam question, we will interpret the information conveyed **explicitly** by the following Hasse diagram, G (which also appeared in the background information page), representing a poset R:

A = { a, b, c, d}

G = (V, E) where V = { a, b, c, d } and E = { (a, c), (b, c), (c, d) }

level(a) = 5, level(b) = 4, level(c)=3, level(d) = 2



- Note that in specifying the level of each vertex, the following information regarding the levels of the vertices holds:
  - Based only on the set of vertices and the set of edges for the graph G = (V, E), we may NOT assume anything about how level(a) and level(b) compare to each other in the above example
  - Based on the nature of Hasse diagrams, we know level(a) > level(c), and level(c) > level(d)
  - The level assignment of a vertex in a Hasse diagram has the transitive property: if level(a) > level(b) and level(b) > level(c), then level(a) > level(c)
- The <u>**undirected**</u> edges in G correspond to <u>**directed**</u> edges in R, but since G is undirected, an edge between vertices u and v can be represented as either (u, v) or (v, u) in E, but not in R:
  - (a, c) ∈ R
  - (b, c) ∈ R
  - (c, d) ∈ R
  - NOTE: if (x, y) ∈ G, then either (x, y) ∈ R or (y, x) ∈ R

Given your understanding of a Hasse diagram representation of a partially ordered set, R, as well as the properties of R, list ALL of the elements in the relation R represented by the Hasse diagram shown. Your elements must have the correct structure: (x, y) ∈ R means that there is a **directed** edge FROM x TO y. What is | R |? (2 pts)

| R |= ______(x,y)_____

R =

Give the **adjacency matrix** representation, M, for the relation, R, whose elements you listed above, using our convention that vertices in an adjacency matrix appear in alphabetical order. Is $M = M^T$? (2 pts)

M =

$M = M^T$: **TRUE** [✓ **/ FALSE**]

7. Develop an algorithm that takes, as input, an **undirected connected** graph G = (V, E) representing a Hasse diagram of a relation, R, where each vertex has a level assigned to it, and produces an **adjacency matrix** representation for the relation described by that Hasse diagram. Your algorithm should NOT be code! It should clearly describe the process at a high level, and possibly incorporate previously referenced processes mentioned in the background. Notice that the structure of your input is a set of vertices and a set of edges, and the structure of your output is an adjacency matrix. (5 pts)

   KEY IDEA:

   Algorithm: HasseToMatrix(G = (V, E))

   Input: An **undirected connected** graph, G = (V, E) representing a Hasse diagram for a poset R on a set V, where V is a set of vertices, E is a set of edges, | V | = n, and each vertex has a level assigned to it, denoted level(v)

   Output: An n x n adjacency matrix M that represents the poset R

   PROCESS:

8. Develop a relatively efficient algorithm that takes, as input, an undirected connected graph G = (V, E) where | V | = n and | E | = m, representing a Hasse diagram of a relation, R, where each vertex has a level assigned to it, and produces a set of all **MAXIMAL** elements in the relation described by that Hasse diagram. You may not work with a different data structure (like an adjacency list or an adjacency matrix for R or G in this problem). You must work with G, as specified by two sets of objects: vertices and edges. Your algorithm should NOT be code, but should describe any iterative processes clearly (especially for the run-time analysis)! Your KEY IDEA should clearly describe your process. What is the run-time of your algorithm? (8 pts)

KEY IDEA:

RUN-TIME:______________________________________

Algorithm: MaximalElements(G = (V, E))

Input: An **undirected connected** graph, G = (V, E) representing a Hasse diagram for a poset R on a set V, where | V | = n, and each vertex has a level assigned to it, denoted level(v)

Output: A set of all maximal elements in R.

PROCESS:

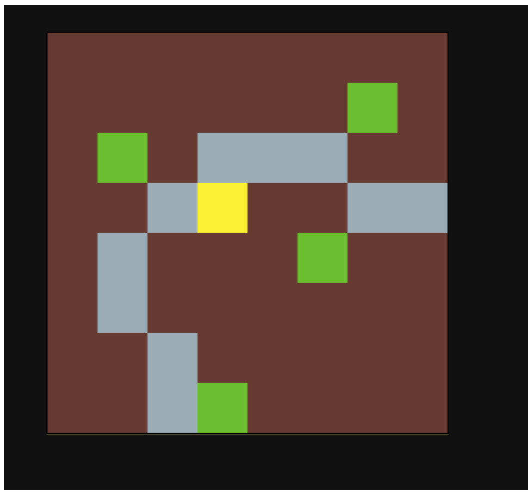
THIS IS BACKGROUND INFORMATION FOR PROBLEMS 9 – 11:

In the latest *Animal Crossing* game, you decorate and customize an island to your satisfaction. One of the mechanics involves rocks; up to six rocks will appear in random locations, unless they are destroyed (which causes them to re-appear, again in random locations). Thus, one strategy to efficiently control rock locations is to make all other possible locations invalid via placing mannequins throughout your island. Mannequins and other solid objects (such as trees) prevent rocks from appearing in any of the 8 adjacent cells on the game's grid. Some surfaces, like stone paths, merely prevent rocks from appearing on the path; they may still appear in any adjacent cell, however. Finally, rocks will *never* appear on the very edge of the map. In this problem, you face a slight simplification: You are given an n x n grid. Each cell in the grid has an x- and y-coordinate for its location, which can be referenced by cell[x] and cell[y], and a cell type, which is referenced by cell[type]. There are **five** types of cells (which are color-coded for illustration purposes):

- TREE (green): a rock CANNOT be placed in a TREE cell OR in any of the eight cells adjacent to it, regardless of the adjacent cell type
- MANNEQUIN (red): a rock CANNOT be placed in a MANNEQUIN cell OR in any of the eight cells adjacent to it, regardless of the adjacent cell type (in other words, a MANNEQUIN cell behaves like a TREE cell)
- PATH (gray): a rock CANNOT be placed in a PATH cell, but can be placed on any DIRT cell adjacent to it, unless that adjacent DIRT cell is blocked because of a TREE or a MANNEQUIN cell
- DIRT (brown): a mannequin can be placed in any DIRT cell; placing a mannequin in a DIRT cell converts that DIRT cell into a MANNEQUIN cell. A rock can be placed in any DIRT cell that is not on the edge of the grid, UNLESS it is blocked by a TREE or a MANNEQUIN cell
- GOAL (yellow): this is the target DIRT cell into which we would like to place a rock

In this problem, you are given an n x n grid complete with a random arrangement of TREE, PATH and DIRT cells, along with a GOAL cell that is NOT an edge cell OR adjacent to any TREE cells.

Your objective is to place mannequins in available DIRT cells on the grid so that the ONLY DIRT cell available (i.e. unblocked) for a rock is the GOAL cell. You would like to incorporate what you've learned in this Algorithms class into your strategy. You decide that the cells should also have a status that reflects their rock availability status (referenced by cell[status]). TREE, MANNEQUIN and PATH cells all have a status "blocked", DIRT cells that are blocked by a TREE or MANNEQUIN cell also have a status of "blocked". All other DIRT cells have a status of "unblocked". You would also like to know if a cell is in a CORNER, EDGE, or INTERIOR position (referenced by cell[position]). Here is a sample 8 x 8 grid with TREE, PATH, DIRT and GOAL cells (there are no MANNEQUIN cells in this grid).



9. Describe an algorithm that modifies a grid based on the ATTEMPT to place a mannequin in a cell. If a mannequin cannot be placed in the cell, return the grid unchanged. As usual, your algorithm should be developed at a high level, and make use of the properties of a cell discussed in the background information. Your algorithm should return the grid reflecting any changes, as well as the number of interior cells adjacent to the given cell that were modified. NOTE: you may decide that you want to slightly modify the output from this process to serve your purposes for problem 10, which is fine, as long as your changes are clear. (5 pts)

KEY IDEA:
The key idea is to check that the cell is unblocked. If the cell is unblocked, we will place the mannequin, otherwise we will return the grid unchanged. If we place a mannequin, we will iterate over the neighbors to see the status of the surrounding cells. If they are unblocked, we will modify them to be blocked and adjust the number of modified cells to represent the change.
Algorithm: PlaceMannequin(grid, cell)

Input: an n x n grid, and a cell in which to attempt to place a MANNEQUIN

Output: an ordered pair consisting of the n x n grid with the given cell converted to a MANNEQUIN type, if possible, and the number of interior adjacent cells that were modified (a whole number between 0 and 8)

PROCESS:
```
For cell in grid;
    If cell[position] = interior, corner or edge;
        Return grid;

    if cell[status].blocked = false:
        int modified <- 0
        int goal < - []
        cell[type] < - MANNEQUIN
            x <-cell.x
            y <-cell.y
            for x – 1 and x + 1;
                check if cell[type] is gold
                    if true, goal[] <- cell x, cell y
                if y or y-1 or y + 1 = unblocked
                change cell[status] <- blocked
                increment modified by # changed
            for y – 1 and y + 1;
                check if cell[type] is gold
                    if true, goal[] <- cell x, cell y

                if x or x – 1 or x + 1 = unblocked
                change cell[status]<- blocked
                increment modified by # changed
        return grid, modified, goal

    if grid[x][y].blocked = true:
        return grid;
```

10. You want practice working with recursion. Describe a RECURSIVE approach to placing the mannequins on the grid so that the only unblocked cell is the GOAL cell. You may use previously defined functions, but you must be consistent. You may want to use Problem 9 as a helper function, and if you want additional output, be sure to explain that in your key idea. For this problem, be sure to describe the key idea, the base case, and the recursive step clearly. You may modify the input for this algorithm to tie in with your key idea; if you do modify the input, be sure to make those modifications clear. Again, no code! (5 pts)
KEY IDEA:

The Key Idea is to call place mannequin recursively until we run out of spaces while checking for the gold space. We'll need to add a condition in our helper function that checks for a gold space and return the coordinate cell of that space. Since we can't place a mannequin adjacent to our goal, we'll have to retroactively remove the mannequin if it overlaps.

Algorithm: PlaceMannequinsRecursive(grid, _______________cell_____________)
Input: an n x n grid, with TREE, DIRT, and PATH cells, as well as ONE GOAL cell

Output: the n x n grid with mannequins placed so that only the GOAL cell (and possibly edge cells, since it is not necessary to block those cells) are unblocked
PROCESS:

```
    Recursively iterate over our grid using PlaceMannequin(grid, cell);
    When goal coordinates are returned,
    Iterate over the neighbors of goal to find blocked cells
        If blocked cells status = MANNEQUIN or If blocked cells status = TREE
        Remove mannequin and place it two cells away from goal cell
        Change status of cells goal cell to unblocked
        Call PlaceMannequin(grid, cell) on new unblocked cells
    Return grid
```

11. You want to impress your friends and decide that you will develop an algorithm that MINIMIZES the number of mannequins on the grid while achieving the desired result (that only the GOAL cell, and possibly edge cells, are unblocked). Throwing caution to the wind you decide to use a GREEDY approach when placing the mannequins on the grid. Describe the KEY IDEA in the development of your greedy algorithm to place the minimum number of mannequins on the grid. For this problem, you will not actually develop the process. You will just explain how you would take a greedy approach to solve this problem. If I don't understand your approach, or how it is a greedy algorithm, you will not get credit for this problem, so be clear! (3 pts)

KEY IDEA:

The key idea here is to check that each cell around a proposed cell is empty, that way when we place the mannequin we can see that the area around the mannequin is completely covered by one. Once we've iterated over the entire grid, we can go back to the unblocked cells to determine if the GOAL cell is a neighbor, if it is, we can't place a mannequin in that spot. So we need to move to a neighboring spot away from the GOAL cell, and check for any surrounding unblocked cells and add a mannequin to cover the cell we just left. This should result in a minimum amount of mannequins while leaving only the GOAL and EDGES behind.

12. Is the following statement TRUE or FALSE? If it is TRUE, provide a proof. If it is FALSE, give a specific counter-example and explain why your counter-example is relevant by clearly indicating that the conditions in the hypothesis are met but the conclusion fails to hold. (5 pts)

Let G = (V, E) be a **connected**, **directed** graph. The statement is "IF HYPOTHESIS, THEN CONCLUSION"

HYPOTHESIS: There is a path from u to v consisting of edges in E, and d[u] < d[v] in a **depth-first search** of G

CONCLUSION: v must be a descendant of u in the depth-first forest produced.

This statement is (circle one): [✓ **TRUE (with proof)**] / **FALSE (with counter-example)**

True because if the discovery time of u comes before the discovery time of v, then f[v] > f[u] meaning v is a descendent in a DFS of G. This may not prove true if we used a BFS of G.

SCORE:______________________