CS 3100 Quiz Day 4 (Retakes)

This packet contains the quizzes for this quiz day. This **cover sheet** is here to provide instructions, and to cover the questions until the quiz begins. **do not remove this cover sheet** until your proctor instructs you to do so.

You will have the entire class period to complete these quizzes. Each quiz is two pages (front and back of one sheet of paper) worth of questions. Make sure to **write your name and computing id at the top of each individual quiz**.

When you are done, you will come to the front of the room and cut off the staple to this quiz booklet. Afterward, you will discard this cover sheet and submit each quiz separately in a different pile. The proctors will be available at the front of the room to clarify this if you have any questions.

This quiz is CLOSED text book, closed-notes, closed-calculator, closed-cell phone, closed-computer, closed-neighbor, etc. Questions are worth different amounts, so be sure to look over all the questions and plan your time accordingly. Please sign the honor pledge below.

In theory, there is no difference between theory and practice.
But, in practice, there is.

THIS COVER SHEET WILL NOT BE SUBMITTED. DO NOT PUT WORK YOU WANT GRADED ON THIS PAGE

Quiz - Module 1: Basic Graphs

Name ______________________________

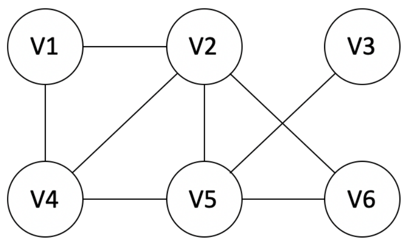
1. [8 points] Answer the following True/False questions regarding *graphs and their basic algorithms*.

Once *BFS* sets the distance to a node, this value will never be changed	**True**	**False**
While executing *BFS*, every reachable node will be added to, and removed from, the queue exactly once	**True**	**False**
In *undirected, connected graphs* some nodes can be unreachable from others	**True**	**False**
When representing a graph using an *adjacency matrix*, $\Theta(\lvert E\rvert)$ space is always consumed	**True**	**False**
When representing a graph using an *adjacency list*, looking up the existence of an edge could take $\Theta(V)$ time	**True**	**False**
If *DFS* is implemented recursively, and nodes are *NOT* marked as visited, then an infinite loop could occur.	**True**	**False**
DFS uses less memory than *BFS* but is asymptotically slower	**True**	**False**
DFS-Sweep will always visit every node in the graph	**True**	**False**

2. [6 points] Consider the following implementation of the recursive portion of *Depth-First Search*. Fill in the blanks in the provided implementation

```
def dfs_recurse(graph, curnode):
    print("Currently visiting node: " + curNode.name)

    curNode.color = "___________"  //White, Gray, or Black

    alist = graph.get_adjlist(curnode)  //get neighbors

    for v in alist:

        if v.color == "__________":

            dfs_recurse(_________, _________)

    curNode.color = "___________"

    return
```

This question will ask you to step through the execution of *Breadth-First Search*. Consider the following Graph and state of the nodes / queue:



Name	***Distance***	***Path***
V1	0	NULL
V2	1	V1
V3		
V4	1	V1
V5		
V6		

Queue
V2
V4

**Front of Queue*

3. [3 points] For each node below, list its updated distance and path value after **just one** iteration of *BFS* is complete.

Node	**Distance**	**Path**
v3		
v5		
v6		

4. [2 points] Now, list the state of the Queue after this one iteration of *BFS*, in order from front to back.

5. [1 points] Now, list the shortest path to node *v5* according to this execution of *BFS*. List the nodes in order in the box here:

NOTHING BELOW THIS POINT WILL BE GRADED

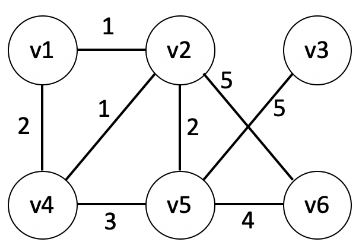
Quiz - Module 2: Advanced Graphs

Name ______________________________

1. [8 points] Answer the following True/False questions regarding *graphs and their algorithms*.

Running *Dijkstra's Algorithm* on an unweighted graph will still return the correct shortest paths	**True**	**False**
Dijkstra's algorithm is slower than *Breadth-First Search*	**True**	**False**
In the *Water Jugs* problem from class, each edge represented filling up or dumping out one jug (only these two options)	**True**	**False**
Prim's Algorithm and *Dijkstra's Algorithm* have the same runtime	**True**	**False**
Kruskal's Algorithm is faster than *Prim's Algorithm*	**True**	**False**
Some graphs have multiple *Minimum Spanning Trees*	**True**	**False**
Indirect Heaps can be used to update the priority of a node in a min-heap in $\Theta(\log n)$ time	**True**	**False**
When using *path compression*, a find-union's find() method might be $\Theta(n)$ time, but the compression **ensures** that **the next** call to find() is $\Theta(1)$	**True**	**False**

2. [3 points] Consider the graph below. In the table, list the *order in which the nodes become known* if *Prim's Algorithm* is executed starting at V_1 (i.e., put a 1 in the box if that node become known first, a 2 in the box of the node that becomes known second, etc.)



Order Nodes Added:	**V1**	**V2**	**V3**	**V4**	**V5**	**V6**

3. [3 points] Consider the same graph again. In the table below, list the *order in which the edges are added* if *Kruskal's Algorithm* is executed. Make sure to write edges as a tuple with the smaller index node listed first (e.g., (V_1, V_4))

Order Edges Added:	**Edge 1**	**Edge 2**	**Edge 3**	**Edge 4**	**Edge 5**

4. [6 points] Complete the implementation of the find-union's *union* method using *union by rank* below. Fill in the code in each of the blanks to accomplish this.

```
Union(x,y):
  Link(find(x), find(y))

Link(x,y):
  if x.rank > y.rank:
    ___.parent = ___                       //x or y
  else
    ___.parent = ___                       //x or y
    if x.rank ___ y.rank:                  //<, >, or ==
      ___.rank = ___.rank + 1              //x or y
```

NOTHING BELOW THIS POINT WILL BE GRADED

Quiz - Module 3: Divide and Conquer

Name ______________________________

1. [8 points] Answer the following True/False questions.

When applying the *Master Theorem*, it is possible that none of the three cases apply	**True**	**False**
Quickselect with *Median of Medians* finds the *i'th order statistic* in $\Theta(n)$ time	**True**	**False**
Quickselect can be used to make *Quicksort* run in $\Theta(n \log n)$ and this approach is faster in practice than *Mergesort*	**True**	**False**
Quickselect invokes *Median of Medians*, and *Median of Medians* also invokes *Quickselect*	**True**	**False**
If our *Closest Pair of Points* algorithm compares every point in the strip to every other point in the strip, then the final running time will be $\Theta(n^3)$	**True**	**False**
When implementing *Closest pair of points*, we could compare each point in the strip to the next *11 points* and the algorithm would still work (and still be fast).	**True**	**False**
In *Closest Pair of Points*, it is possible to process the points in the strip BEFORE making the two recursive calls (if we want to)	**True**	**False**
The *combining* step of our *Maximum SubArray* problem from class ran in $\Theta(n)$ time	**True**	**False**

2. [6 points] Use the master theorem to establish the runtime of the following three recurrence relations. You MUST list the case that you used (or NONE if impossible) as well as the big-theta runtime (or N/A if impossible). The master theorem is listed on the back of this quiz.

Recurrence	Case Used	Θ Runtime
$T(n) = 4 * T(\frac{n}{4}) + n \log n$		
$T(n) = 4 * T(\frac{n}{2}) + n^2$		
$T(n) = T(\frac{n}{2}) + n$		

3. [6 points] Consider the following problem, and the code that attempts to solve it. Fill in the missing blanks to complete the *Divide and Conquer* solution. **Problem Statement:** You are given a list of numbers that is sorted (non-decreasing) but has a long (1 or more) string of 8's. Return the starting and ending indices of the 8's. *EXAMPLE:* if given the array $[1, 3, 6, 8, 8, 8, 8, 8, 11, 14]$ you should return $(3, 7)$.

```
/* This is pseudo-code */
function findStartAndEnd(int[] a, int start, int end):
        int mid = (start + end) / 2
        if(a[mid] < 8):

                return findStartAndEnd(a, _______, _______)
        else if(a[mid] > 8):

                return findStartAndEnd(a, _______, _______)
        else if(a[mid] == 8):

                start = findStart(a, _______, _______)

                end = findEnd(a, _______, _______)
                return (start, end)


/* Find the START index of the 8's only */
function findStart(int[] a, int start, int end):
        if(start == end)

                return start //or end

        int mid = (start + end) / 2
        if(a[mid] < 8):

                return findStart(_________,_________)
        else if(a[mid] >= 8)

                return findStart(_________,_________)

function findEnd(int[] a, int start, int end):
        /* LEFT BLANK, BUT ANALOGOUS TO FINDSTART */
```

NOTHING BELOW THIS POINT WILL BE GRADED

MASTER THEOREM: FOR YOUR REFERENCE

For a recurrence of form $T(n) = aT(\frac{n}{b}) + f(n)$, let $k = \log_b a$

- Case 1: if $f(n) = O(n^{k-\varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^k)$
- Case 2: if $f(n) = \Theta(n^k)$, then $T(n) = \Theta(n^k \log n)$
- Case 3: if $f(n) = \Omega(n^{k+\varepsilon})$ for some constant $\varepsilon > 0$, and if $af(\frac{n}{b}) \le cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$

Quiz - Module 4: Greedy Algorithms

Name ____________________

1. [8 points] Answer the following True/False questions.

Greedy Algorithms never backtrack (i.e., they never undo their decisions and consider a different option instead).	**True**	**False**
Our greedy algorithm for *making change* could have progressed from the smallest denomination (e.g., 1-cent coin) up to the highest and would still have worked.	**True**	**False**
Our greedy algorithm for *interval scheduling* could have handled the last interval first (using latest start time) and would have still worked.	**True**	**False**
Our greedy algorithm for *fractional knapsack* can add the items to the knapsack in any order and it will still work.	**True**	**False**
For the knapsack problem, if the sum of the weights of the items is less than the capacity of the knapsack, then it is always optimal to steal all the items	**True**	**False**
For the *bridge crossing problem*, once you start using the fastest walker to escort the two slowest walkers across, you will never change strategy again.	**True**	**False**
The *Minimum-Spanning Tree* problem has *Optimal Substructure*	**True**	**False**
Dijkstra's Algorithm is a *greedy algorithm* because it chooses the next known node greedily and never looks back (the node with the shortest distance right now).	**True**	**False**

2. [6 points] In this module, we studied many *Greedy Algorithms*. For each one below, list the runtime of the brute-force algorithm AND the runtime of our greedy algorithm.

Problem	**Variables**	**Brute-Force Notes**	**Brute-Force Runtime**	**Greedy Runtime**
Making Change	**A** (change to make), **C** (number of coin denominations)	Use up to A of each coin denomination at most		
Interval Scheduling	**n** (num. intervals)	Try every subset of intervals		
Fractional Knapsack	**n** (num. items to steal), **W** (capacity of sack)	Try every unique order of adding the items to the knapsack		

Consider the following problem: You are driving across the country, and your gas tank has a capacity of C liters. You are also given a list of gas station locations $G = \{g_1, g_2, ..., g_n\}$ along your route, your rate F of fuel-consumption (in liters/kilometer), and the rate R at which you can fill up your tank (in liters/minute). Your goal is to minimize the **number of minutes you spend filling your tank with gas** along the way.

3. [2 points] Which of the following best describes WHY this problem has optimal substructure (color in the box next to your choice). Consider a solution of the form $\{o_1, o_2, ..., o_n\}$ where each o_i is the number of mins spent refueling at each station (0 means you didn't stop at that gas station).

- ☐ The optimal solution definitely involves stopping at station 1 for o_1 minutes, but the rest of the steps might be different if we had started our journey at station 1.
- ☐ The optimal solution can be cut in half, and each half may not be optimal itself, but together the whole trip is optimal
- ☐ The optimal solution has two parts: You drive to station 1 as fast as you can, and then drive the rest of the way as quickly as possible
- ☐ The optimal solution has two parts: You drive to station and stop for o_1 mins and then you drive the rest of the way, minimizing your time refueling.

4. [4 points] Consider the two greedy choices below. Color in the box next to the choice that will minimize the amount of time spent filling the gas tank.

- ☐ Drive as far as you can (i.e., until you reach a gas station and cannot make it to the next one). Stop and fill up all the way. Repeat.
- ☐ At each gas station, stop and fill up the tank just enough to make it to the next gas station. Repeat.

NOTHING BELOW THIS POINT WILL BE GRADED

Quiz - Module 5: Dynamic Programming

Name ______________________________

1. [8 points] Answer the following True/False questions.

DP works best with *overlapping subproblems* (subproblems solved many times)	**True**	**False**
Our DP algorithm for $RodCutting$ used $\Theta(n)$ space, but $\Theta(n^2)$ time	**True**	**False**
In order to *backtrack* efficiently for *Rod Cutting*, we created a second array that stored the size of the last cut at each step	**True**	**False**
Our DP solution to *fibonacci* can be improved to use $\Theta(1)$ space	**True**	**False**
memoization is a term we use for the general approach of storing solutions to subproblems, and looking them up later	**True**	**False**
Our DP *knapsack* algorithm had $n * W$ array cells to compute, and each took $\Theta(n)$ to compute, for a total runtime of $\Theta(n^2 * W)$	**True**	**False**
Our making change algorithm filled out each cell of the DP array by checking the values of at most two other cells.	**True**	**False**
for the *Weighted Interval Scheduling* problem, we were able to compute the $P()$ function for all intervals in $\Theta(n \log n)$ time	**True**	**False**

For this problem, step through the *Discrete Knapsack* dynamic programming solution from class. The recurrence for this is $F(k, w) = Max(F(k-1, w), V[k] + F(k-1, w-W[k]))$. The input includes three items with values $V = [3, 5, 6]$ and weights $W = [1, 2, 3]$. The capacity of the knapsack is $C = 4$

	$w = 0$	1	2	3	4
$k = 0$	0	0	0	0	0
1	0	3	3	3	3
2	0	3	5	8	8
3	0	3			

2. [3 points] First, fill out the last three cells of the array above.

3. [3 points] Now, color in the box for each statement below that is *true* about the execution of the algorithm above.

- ☐ For cell $k = 3, w = 2$ only one option was possible because item 3 is too heavy
- ☐ The overall solution includes the third item
- ☐ The $k = 0, w = 4$ cell is 0 because there is no room in the knapsack to fill

For these questions, you will solve the *House Painting Problem (this is a real Google interview question)*. Assume we are given a straight row of n houses (indexed from 1) and each can be painted one of three colors (Red, Green, or Blue). We are also given three arrays ($C_R[]$, $C_G[]$, and $C_B[]$). Each array stores the cost of painting the ith house that respective color. For example, $C_R[2] = 5$ represents that it costs 5 dollars to paint the second house red. Your task: Find the cheapest way to paint all n houses such that no two adjacent houses are painted the same color.

We are going to solve this problem using three separate (similar) recurrences. Let $R(i)$ be the minimum cost to paint the first i houses such that the house at index i is painted Red. $G(i)$ and $B(i)$ are analogous, with the house at index i being painted Green and Blue respectively.

4. [2 points] What are the three base cases? Fill in the blanks below

$R(1) = \_\_\_\_\_\_$ $G(1) = \_\_\_\_\_\_$ $B(1) = \_\_\_\_\_\_$

5. [3 points] Now solve the general form of the recurrence in terms of smaller sub-problems (we will do $R(i)$ only, but the other two are similar). Fill in the blanks.

$$R(i) = Min(G(\_\_\_\_\_\_), B(\_\_\_\_\_\_)) + \_\_\_\_\_\_$$

6. [1 points] Which of the following subproblems stores the *overall solution to the problem*. Choose one by filling in the box.

- ☐ $R(n)$
- ☐ $Max(R(n), G(n), B(n))$
- ☐ $Min(R(n), G(n), B(n))$

NOTHING BELOW THIS POINT WILL BE GRADED

Quiz - Module 6: Network Flow

Name ______________________________

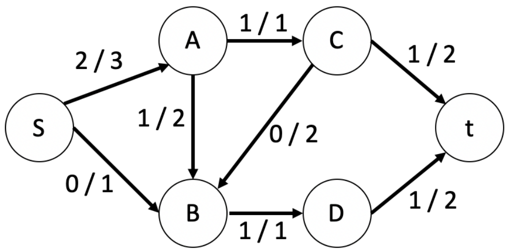
1. [8 points] Answer the following True/False questions.

Ford-Fulkerson has a runtime of $\Theta(\|E\| * f)$	**True**	**False**
Ford-Fulkerson works by repeatedly finding augmenting paths, and pushing **one** unit of flow through that path	**True**	**False**
After reducing *Bi-Partite Matching* to *Network Flow*, we can find the actual matches by looking at edges with **zero** flow going through them	**True**	**False**
In order to reduce *Bi-Partite Matching* to *Network Flow*, we added 2 extra nodes and $\Theta(\|V\|)$ extra edges to the graph	**True**	**False**
A cut is at *full capacity* if all edges across it (both ways) are at full capacity	**True**	**False**
We proved the correctness of the *Flow-Value Lemma* by doing an induction in which we (at each step) moved one node across the cut and showed that the net change in flow was always a positive number	**True**	**False**
Ford-Fulkerson does not work if a network has *multiple sources*, but the graph can be reduced to an equivalent one with a single source	**True**	**False**
It is possible for a *flow network* to have a max-flow of **zero**	**True**	**False**

2. [6 points] The following code is an implementation of *DFS* used for *Ford-Fulkerson* that finds a path in a residual graph. Complete the implementation below.

```
//Returns a path from source (index 0) to sink (index 1) in residual graph
global int[] visited
function DFS(int[][] Gf, int currentNode, int[] pathSoFar):

        visited.append(currentNode)


        if(__________________): //found the sink
                return pathSoFar

        /* Look at all outgoing nodes and try to traverse them */
        for i from 0 to Gf.length-1:

                if i not in ___________ and Gf[________][________] > 0:


                        aPath = DFS(Gf, _______, pathSoFar.append(i))

                        if aPath != NULL:

                                return __________ //found one!

                        pathSoFar.remove(i)
        return NULL
```

The graph G below shows a flow graph after Ford-Fulkerson has found two augmenting paths.



3. [1 points] Are there any augmenting paths in the residual flow graph G_f corresponding to the drawing of G shown above? If so, list the vertices in a valid augmenting path in the box below. If there is not, write "none."
   Vertices in path:

4. [2 points] Look for or find a min-cut in the graph G shown above and list the vertices on the source side of this cut in the box. (Include vertex s in your answer.)

5. [3 points] Which of the following are true statments about the proof of correctness we did in class for the *Max-Flow Min-Cut* theorem? Fill in the box next to each statement that is true.

   ☐ If a flow network (with current flow f) has an augmenting path, then it is still possible that f is maximum if no flow can be pushed through that augmenting path.

   ☐ If a flow network has maximum flow f currently, then a cut that is at full capacity can be found by finding all nodes reachable from the start node.

   ☐ If there exists a cut that is at currently at maximum capacity, then there cannot be an augmenting path because if there were, that path could be used to put the cut over capacity (which is a contradiction).

NOTHING BELOW THIS POINT WILL BE GRADED