Name _____

Quiz - Module 6: Graphs: Prim's and Dijkstra's

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

Prim's algorithm and *Dijkstra's algorithm* both rely on that fact their problems have the optimal substructure property.	**True**	**False**
Prim's MST algorithm could be altered so that a priority queue is not needed but instead information about edges is sorted once at the start of the algorithm.	**True**	**False**
We know that both *Dijkstra's* and *Prim's* algorithms store info about a node in a priority queue. True or false: One significant difference between them is that for a given node Dijkstra's stores the weight of a single edge in the priority queue, while Prim's stores the sum of more than one edge.	**True**	**False**
For *Prim's algorithm* the algorithm may choose any node as the first node in the tree, but it will find the same spanning tree as long as no edges in the graph have the same weight.	**True**	**False**
For *Dijkstra's algorithm* the algorithm will produce the same tree no matter which node is chosen as the starting node, assuming that no edges in the graph have the same weight.	**True**	**False**
An important goal of using *indirect heaps* is to make *decreaseKey()* faster than $\Theta(n)$.	**True**	**False**
When using *indirect heaps*, indices in the indirect heap must always be updated whenever a value in the heap is swapped with its parent or one of its children.	**True**	**False**
Indirect Heaps provide a way to locate an element in the heap that is not the first or last item in $\Theta(1)$ time.	**True**	**False**

2. [1 points] If we asked you to use an exchange argument to prove the correctness of *Prims MST algorithm*, which of the following best summarizes the approach you would use to do this proof?
   - We prove we reach an optimal result by choosing an edge and repeatedly exchanging it with the next smaller-weight edge until we find one that does not introduce a cycle.
   - We show that if the greedy choice is not correct, then some other edge was chosen that is not the edge that Prim's chose, but that using the Prim's edge would produce a result that's better.
   - We argue that if a spanning tree with smaller total weight than the one found by Prim's does exist, it must have "exchanged" at least one edge in Prim's tree with another edge that introduces a cycle.
3. [4 points] For each algorithm below, list the runtime of the algorithm under the various conditions in each column. List your runtimes in *Big-Theta* notation.

Algorithm	**Indirect-Heap**	**Min-Heap** (*find() is linear time*)
Dijkstra's Algorithm		
Prim's Algorithm		

4. [5 points] Complete the implementation of *Prim's algorithm* below by filling in the blanks with appropriate code.

```
MST–Prims(G, wt) {
    /*Omitted... initialize PQ and start node value*/
    while (PQ not empty) {
        v = PQ.ExtractMin();
        for each w adj to v {
            if (w is unseen) {
                cost[w] = ____________________________

                PQ.________________________(w, cost[w] ); // function name?
                parent[w] = v;
            }
            else if (w is fringe && ____________________________ ) {

                cost[w] = ____________________________

                PQ.________________________(w, cost[w]); // function name?
                parent[w] = v;
            }
        }
    }
}
```

In class, we saw a proof of correctness for Dijkstra's algorithm. Answer the following questions about that proof.

5. [1 points] In this proof by induction, state the inductive hypothesis in terms of what is true for the first k vertices chosen by the algorithm.

6. [1 points] During our inductive step, we came to this expression: $opt(v_i) + wt(e) > opt(v_i') + wt(e') + \delta$. Briefly explain what v_i' and e' represents in this formula.

7. [1 points] Briefly explain what δ represents in the formula given in the last question. (Assume that e is the edge that connects v to a non-tree vertex w.)

8. [1 points] Briefly explain why the proof would no longer work if δ can be negative. What problem arises?

Name _____

Quiz - Module 7: Greedy Algorithms

1. [7 points] Answer the following True/False.

Optimal substructure is a property of an optimization problem (e.g., coin changing problem) and not a property of a specific algorithm.	**True**	**False**
When proving correctness of the greedy choice for coin changing, we used an exchange argument, assuming that a quarter was *not* optimal for $A \geq 25$ but determined that fewer coins could be used if we did include a quarter.	**True**	**False**
If only given dimes and nickels in the *coin changing problem*, issuing the dime first still leads to an optimal solution if one exists (though maybe the solution won't exist).	**True**	**False**
For the *fractional knapsack problem*, if all the items have the same value-to-weight ratio, then any combination of items that fills the knapsack as much as possible is optimal.	**True**	**False**
The *fractional knapsack problem* has *optimal substructure*	**True**	**False**
In the Daycare homework problem, a feasible solution for the problem is some proper subset of the rooms to be remodeled.	**True**	**False**
A *feasible solution* for the *unweighted interval scheduling probelm* only requires the intervals selected do not conflict with one another.	**True**	**False**

2. [5 points] For each of the following *Greedy algorithms* studied in class, read the new proposed *Greedy Choice Property* for that problem and select whether or not this new greedy choice property will still lead to an *optimal solution*. **Put a checkmark in the corresponding box next to each entry.**

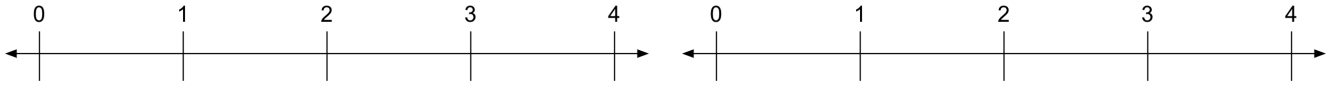
Problem	**Greedy Choice**	**Optimal?**	**Not Optimal?**
Coin Change	Issue the smallest coin first up to largest coin last		
Unweighted Interval Scheduling	Use same approach as in class, but sort by decreasing start time. Take the interval with latest start time first and proceed.		
Unweighted Interval Scheduling	Use same approach as in class, but sort by increasing start time. Take the interval with earliest start time first and proceed.		
Unweighted Interval Scheduling	Take interval with fewest conflicts and repeat		
Fractional Knapsack Problem	Start with smallest value-to-weight ratio, but leave enough room for all other items. *For example, if $C = 10$ and every item except the smallest ratio one has total sum weight of 7, then fill as much of $C - 7 = 10 - 7 = 3$ of the knapsack with the lowest ratio item and repeat.*		

3. [2 points] Show that you understand how the *Unweighted Activity Selection Problem* has **optimal substructure** by explaining it using the following example: *Because the optimal set of intervals (start and end times shown) is* $\{(1,3),(4,5),(5,8)\}$ *for intervals that fall between time* 1 *and* 8*, it must be the case that...*

For this problem, we are given a set of points on a 1-dimensional number line (e.g., $P = \{-0.8, 2, 2.5\}$). Our goal is to *cover* each point with any number of *unit-length intervals*. A unit-length interval is an interval that has a width of exactly 1.0. For example, one solution to the example above would be to use two intervals, one from $(-1, 0)$ to cover the -0.8 and another from $(1.8, 2.8)$ to cover both the 2 and 2.5 (note that there is wiggle room with the intervals, and thus many possible unique solutions).

These questions are about finding an optimal greedy algorithm for covering all of the points given in P with as few unit-length intervals as possible.

4. [2 points] Consider this greedy choice: We select an interval to cover as many points at once as we can. Show this does not always work by drawing pictures showing the greedy's selected intervals and a separate picture showing the optimal set of intervals. *(Hint: We used six points and had this algorithm cover 4, 1, and 1 points respectively when the optimal is to cover 3, and 3 points respectively)*



5. [1 points] Describe a better *Greedy Choice Property* for this problem. *Note: You HAVE to cover the first (minimum) point. Think about how to cover that one while minimizing the work left on remaining points.*

6. [2 points] In a few sentences, describe why this property works. *No need for a proof here, just a short bit of intuition behind why it will work.*

Name _____

Quiz - Module 8: Dynamic Programming

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

In the *Drainage* homework assignment, if we use top-down dynamic programming then we don't need to worry about the order we process the cells (no need to sort).	**True**	**False**
Our solution to the *Discrete Knapsack Problem* ran in time $\Theta(W * n^2)$.	**True**	**False**
The dynamic programming solution to the *Discrete Knapsack Problem* considers all options for a particular item: The case where that item is included in an optimal solution and the case where it is not in some optimal solution.	**True**	**False**
If a problem is recursive, but does NOT have *overlapping subproblems*, then dynamic programming will not improve the runtime.	**True**	**False**
Even if the set of coins is such that our greedy algorithm (using the largest coin) will work, the *dynamic programming* solution will still compute the correct result but will have a worse runtime compared to the greedy algorithm.	**True**	**False**
The dynamic programming solution to *Log Cutting* had a runtime of n^2, even though the memoization array was of size n	**True**	**False**
When solving the *weighted activity selection* problem, we needed to precompute the values for $P()$. Computing this function has better time complexity than any other part of the algorithm.	**True**	**False**
When solving the *discrete knapsack problem*, if an item completely fills the knapsack then that item is definitely in the optimal solution	**True**	**False**

2. [3 points] For this question, you will need to state whether or not the alteration to each dynamic programming algorithm will still produce the correct result. Read the alteration to the algorithm and then place a **checkmark** in the corresponding box in the table to note if the new approach still produces the optimal solution or not.

Problem	**Variant**	**Optimal?**	**Not Optimal?**
Log Cutting	Same as in class, but work from the end of the log to the front instead (i.e., identify the location of the first cut instead of the last cut).		
Weighted Activity Selection	Same as in class but sort intervals by descending start times. $P(i)$ function returns first interval AFTER interval i that does not conflict. Work from last interval down to first.		
Discrete Knapsack Problem	Same as in class, but sort the items according to their value-to-weight ratio first. Consider items in this particular order when building the DP table.		

3. [3 points] Take a look at the following *top-down dynamic programming* implementation for the *coin change problem*. Add in the missing lines of code to complete the implementation.

```
//A is amount of change to make, C is which coin in denom we are considering.
CoinChange (denom, A, C) {

        if(memory[A][C] != -1) ____________________________

        n = denom.last //n is index of penny
        if(A == 0){ memory[A][C] = 0; return 0; }
        else if(C == n){ memory[A][n] = A; return A; }

        best = infinity
        if( denom[C] <= A) best = ____________________________
        best = Min(best, CoinChange(denom, A, C+1))


        ____________________________________
        return best
}
```

For this problem, you are given a set of n boxes $B = \{b_1, b_2, ..., b_n\}$ which are given to you in increasing volume. You are also given a function $f(b_i, b_j)$ which returns true if and only if box b_i fits fully inside of box b_j (*Note that box b_i can be smaller than box b_j in volume but still not fit inside)*. You are planning to give your friend a birthday present, and want to create a large set of nested presents. Your friend will have to open a present, only to find ANOTHER present. These will get smaller and smaller until they finally find the real present. Given the list of boxes, can you figure out the *most number of boxes that can be nested together*?

4. [1 points] Suppose our sub-problem is $P(i)$: the most number of boxes that can be nested using *only boxes 1 through i* AND such that *box i is the outermost box (box i MUST be included)*. State the base case for $P(1)$.

5. [2 points] State a recursive solution to $P(i)$ in terms of smaller sub-problems. *Hint, you might need to check multiple sub-problems, and you will need to use the function $f()$ here.*

6. [1 points] Now state which sub-problem(s) is the solution to the **overall problem**. *Note, this will be chosen among a set of possible options, so think carefully. Is it necessarily the case that you have to use the largest box?*

7. [1 points] Lastly, how many total sub-problems are there to solve?