Name SOLUTION

Quiz - Module 8: Dynamic Programming

1. [6 points] Answer the following True/False.

If a problem has *optimal substructure*, then dynamic programming can be used to solve it.	**True**	**False**
Memoization tables (solution arrays) are effective at reducing how often subproblems are computed / solved.	**True**	**False**
The size of the dynamic programming solution array is always the same as the overall runtime of the algorithm, because you simply need to fill in the array.	**True**	**False**
Our solution to the *log cutting* problem worked by locating the last cut, but it could have worked similarly by locating the first cut instead (and working through the log the other way).	**True**	**False**
When solving the *weighted activity selection* problem, the function $P()$ allows us to quickly look up whether two activities are compatible with one another.	**True**	**False**
When using a top-down approach with memoization (using a table or list) in dynamic programming, we must initialize that data structure with some value for every possible subproblem that might be needed to calculate a solution.	**True**	**False**

2. [4 points] For each of the bottom-up dynamic programming algorithms we studied in class, list the size of the dynamic programming table (the memoization array) and the overall Big-Theta runtime of the algorithm. Present both of these in terms of the variables provided and make sure to include the total runtime (including sorting, reading in input, etc.).

Problem	**Variables**	**Size of Array**	**Runtime**
Log Cutting	n (number of log sections)	n	$\Theta(n^2)$
Weighted Activity Selection	n (number of activities)	n	$\Theta(n \lg n)$
Coin Change	n (number of coins), A (amount of change to make)	$n * A$	$\Theta(n * A)$
Discrete Knapsack Problem	n (number of items), C (capacity of knapsack)	$n * C$	$\Theta(n * C)$

3. [3 points] The code below shows the *backtracking* implementation to the *coin change problem*. Fill in the blanks with the appropriate lines of code.

```
FindSolution(int c, int a){
        if(amount <= 0) return;

        if(denoms[c] > a || sol[c,a]==sol[c+1][a])
                findSolution( c+1 , a );
        else {
                print("Used coin of denomination: " + denoms[c])
                findSolution( c , a - denoms[c] )
        }
}
```

You build houses, and purchased a street with n empty plots of land. On each plot, you can build one of three style houses (Victorian, Cape Cod, and Craftsman), but your customers DO NOT want to have the same as their neighbor. Can you find the optimal way to build houses to maximize profit while ensuring no two neighbors houses have the same style?

The input contains three arrays of size n, specifying the price you can sell each style on each of the n plots of land (divided by $10,000). For example, if $V = \{20, 14, 35\}$, $CC = \{26, 10, 10\}$, and $CR = \{3, 8, 19\}$, then the optimal solution is to build a Cape Code on plot 1, a Craftsman on plot 2, and a Victorian on Plot 3. Your total profit would then be $CC[1] + CR[2] + V[3] = 26 + 8 + 35 = 69 * 10,000 = \$690,000$.

Hint for the following problems: it may help if you draw out the table $P(s, i)$.

4. [2 points] Suppose our sub-problem definition is $P(s, i)$, representing the optimal way to build on the first i plots of land only, while placing a house of style s on the last plot. State the base case(s).

$P(s, \emptyset) = \emptyset$

5. [2 points] Now state which sub-problem is the solution to the overall problem. *Note: This will involve a small number of sub-problems, not just one.*

max (P("V", n), P("CR", n), P("CC", n))

oops!

6. [2 points] State a recursive solution to $P(i, "Craftsman")$ in terms of smaller sub-problems.

P("CR", i) = max (P("V", i-1), P("CC", i-1)) + CR[i]

7. [2 points] Lastly, how many total sub-problems are there to solve?

3 (n+1) or 3 n

Name SOLUTION

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

1. [6 points] Answer the following True/False.

Prim's algorithm and *Dijkstra's algorithm* both solve the *single-source* shortest path problem.	**True**	**False**
Both *Dijkstra* and *Prim's* algorithms store fringe nodes on a priority queue, and choose the fringe node that "looks best" at each step.	**True**	**False**
If all edges in an undirected connected graph have the same edge-weight value k, you can use either BFS or Dijkstra's algorithm to find the shortest path from s to any other node t.	**True**	**False**
An *indirect heap* makes *find()* and *decreaseKey()* faster (among others), but *insert()* becomes asymptotically slower because the indices in the indirect heap must be updated while percolating.	**True**	**False**
Indirect Heaps require some way to reference an element in the heap using an integer index value. Otherwise, we won't know where in the indirect array to look initially.	**True**	**False**
Indirect Heaps use extra space that is asymptotically worse than a *min-heap* (with no indirect array).	**True**	**False**

2. [1 points] If we asked you to use an "exchange argument" to prove the correctness of *Prim's MST algorithm*, which of the following best summarizes the approach you would use to do this proof?

- We look at the tree the algorithm finds at each stage, and then "exchange" some edges in the solution found so far in order to reduce the total weight of the tree.
- We choose a candidate edge to add to the tree using the greedy choice, and repeatedly "exchange" it with smaller-weight edges until we reach the optimal answer.
- We consider the possible existence of an edge that leads to a better result than when we use the edge our algorithm selected, and then we "exchange" our algorithm's choice of edge into that solution to show that such an edge cannot exist.

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	**Min-Heap** (*find() is linear time*)	**Indirect-Heap**
Prim's Algorithm	$\Theta(V \log V + EV)$ or $\Theta(EV)$ or $\Theta(V^3)$	$\Theta(E \log V)$ or $\Theta(E \log E)$ or $\Theta(V \log V + E \log V)$
Dijkstra's Algorithm	same as above	same as above

4. [4 points] Given the implementation of *Dijkstra's algorithm* below, change it into an implementation of *Prim's algorithm* by **only** crossing out bits of code. You can cross out parts of a line or entire lines but you cannot add any new code.

```
dijkstra(G, wt, s){
    /*Omitted... initialize PQ and start node cost*/
    while (PQ not empty){
        v = PQ.ExtractMin();
        for each w adj to v{
            if (w is unseen){
                cost[w] = ~~cost[v] +~~ wt(v,w)
                PQ.Insert(w, cost[w] );
                parent[w] = v;
            }
            else if (w is fringe && ~~cost[v] +~~ wt(v,w) < cost[w] ){
                cost[w] = ~~cost[v] +~~ wt(v,w)
                PQ.decreaseKey(w, cost[w]);
                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] What were we trying to prove? State formally (as in class) or in a few words.

distance calculated to each node is optimal (shortest) distance from start node to that node

6. [1 points] State and argue why the base case holds for claim.

Calculated to be Ø.

Distance from start to start is Ø.

7. [1.5 points] During our inductive step, we came to this expression: $opt(v_i) + wt(e) > opt(v_i') + wt(e') + \delta$. Briefly explain what this formula represents (*assume that* $e = (v, w) \in E$). We are looking for a description of what the left side of the inequality represents and what the right side represents.

The left side is what Dijkstra's calculates.

The right side is a hypothetically better path where e' is the edge chosen instead of e.

8. [1.5 points] Briefly explain why the proof would no longer work if δ can be negative. What problem arises?

δ is the cost of getting to w using that hypothetical better path. If it could be negative, our proof by contradiction fails, and Dijkstra's algorithm is not correct.