CS3340 Assignment 3

By Bradley Assaly-Nesrallah (#250779140)

1. We compute the next[] function for the pattern P = babbabbabbababbabb , lets define s[i] as the string, Next[i] as the prefix function, where the following table represents the next pattern for the given function :

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Next[i]	0	0	1	1	2	3	4	5	6	7	8	9	2	3	4	5	6	7

2. We modify the KMP string matching algorithm to find the largest prefix of P that matches a substring of T. Since this is designing an algorithm, the answer must contain three parts, firstly a description of the algorithm in English, secondly show that the algorithm is correct and finally an analysis of the time complexity of the algorithm.

First a description of the modified KMP algorithm, this algorithm will consist of a modified KMP where for each substring it matches with P, starting from p1, it stores a max value for the matching of P with a substring of T, and if this is greater than the current max it stores the value, then once the algorithm has checked all of T, it returns the largest prefix of P that matches a substring of T, corresponding to the max size obtained during the algorithm.

Secondly we prove correctness, the base case is that the maximum is 0, or that P is not matching at all with a substring of T, then as the algorithm runs if a substring matches with P, then the max is updated if the length of the prefix of P that matches is greater than the max, so the output will always be correct, so we are done. Finally we compute the time complexity of the algorithm, we know that KMP is O(n + k) since the portions are O(k) and O(n) and keeping track of the maximum can be done in constant time during the second portion in each iteration so the complexity is still linear O(n + k), so we are done.

3. 15.4-2 We must give the pseudocode to reconstruct an LCS from the completed c table and the original sequences X = <x1,x2,…,xm> and Y=<y1,y2,…,yn> in O(m+n) time without using the b table, we let c be the c table, X and Y the sequences and i and j be their sizes respectively so the pseudocode is:

```
print_LCS(c, X, Y, i, j)
        if c[I,j] == 0 then return
        if X[i] == Y[j] do {
                print_LCS(c, X, Y, i-1, j-1)
                print X[i] }
        else if c[ i-1, j ] > c[ i, j-1 ] then print_LCS(c, X, Y, i-1,j)
        else do print_LCS(c, X, Y, I, j-1)
```

that is the completed algorithm in pseudocode so we are done.

4. 16.2-3 We have the 0-1 knapsack problem, where the order of items when sorted by increasing weight is the same as their order when sorted by decreasing value, we must give an efficient algorithm to find an optimal solution to this problem and prove its correctness. An optimal solution to this problem is to pick the lightest and most valuable item that you can pick at any point. Now we must prove the correctness of the algorithm, suppose that for some item x that we included but also a smaller more valuable item y that we didn't include, then we could replace x in the knapsack with y, and we know that it could fit because j is lighter and will thus increase the total value because y is more valuable. Hence all moves will be correct so the algorithm is correct.

5. We have a variant of the 0-1 knapsack problem, with identical assumption except that there is an unlimited supply of each item. The solution to this problem where we are given a knapsack with weight W and a set of n items with a val v and weight w, the solution to this algorithm is to find the most efficient items if given a finite time or most efficient item if there is an infinite time and to pack the knapsack in with that, in other worst to find the item that has the most value per weight that can fit into the knapsack and to fill the backpack with that item. So the theif will evaluate the possible packing and use this evaluation to find an optimal solution, once an optimal set of items is found since there is an unlimited supply the thief must find the optimal supply of items to maximize the value of the items in the backpack, this is the optimal solution to maximize value in this modification of the 0-1 knapsack problem so we are done.

6. We must modify the minimum spanning tree algorithm to find the maximum spanning tree, since we are designing an algorithm the answer must contain a description in English, a proof of correctness and an analysis of the time complexity. Firstly a modified MST algorithm to find the maximum spanning tree is we take all of the edge weights and negate them, or multiply their weight by -1, then we apply the MST algorithm (Kruskal's) to compute the spanning tree, the result is the maximum spanning tree in the graph.

Secondly we must prove the correctness of the algorithm, since we take the graph and negate all the edge weights, let G' be the graph without edge set, we know that edge set has the property if any edges are added onto the graph G' then the graph must form the cycle with that edge because all edges have a positive weight. Thus the graph G contains the set of the number of edges with weights in order to get the minimum total edge weight for the edge set ie the minimum spanning tree, and thus the graph G' contains the maximum total edge weights, thus by inverting the edge weights and computing the minimum spanning tree on the negated graph we produce the maximum spanning tree, hence the output of the algorithm is correct in all cases.

Finally we compute the time complexity of the algorithm, negating the edges takes time O(E) where E is the number of edges, we know the use of MST (Kruskals) takes O(ElogV), and determining the edge weights not in the minimum spanning tree takes O(E) so the total running time is O(ElogV). We have found an algorithm modification of MST algorithm to compute the maximum spanning tree, have proved its correctness, and determined its time complexity so we are done.

7. We must prove that Dijkstra's algorithm does not work when there is a negative weight edge, a counter example will suffice, so we give an example with three vertices to prove this, suppose there is G with three vertexes, 1,2,3 and three edges, so we have V={1,2,3} and three edges expressed in a tuple of (v1,v2,w) where v1 and v2 are vertices and w is the weight, E= {(1,3,2), (1,2,5), (2,3,-10)}. When you use Dijkstra on this graph it will go from A to C, and then after the relaxation phase of the algorithm, it will

never find the shortest path from 1 to 2 then 3. Therefore since we have a counterexample when Dijsktras does not work with negative weights we have proved that it is not correct, that is why one of the assumptions of the algorithm is only positive weights, so we are done.

8. We have a weighted directed graph G = (V,E) where there are negative weights in G, but there is no negative cycle in G, we must determine if the all pair shortest path algorithm is still correct. We claim that this algorithm is correct, if there are negative weights, but no negative cycles, so suppose we have a graph with negative edges but no negative cycle: the algorithm will first add a new node q to the graph connected by zero weight edges to the other nodes, next the bellman ford algorithm is used, starting from the new vertex q to find for each vertex the minimum weight h(v) for a path from the new node q to v, the algorithm checks for negative cycles, however since our graph G does not contain a negative cycle it continues to run, next the edges of the original graph G are reweighted using the values computed by the Bellman ford algorithm, where each edge from u to v have a weight w(u,v), is given a new weight w + h(u)-h(v), so the new graph is now reweighed using bellman ford, so we now may remove the vertices   q, and now have a reweighed graph with no negative edges and can now perform dijkstras algorithm to find the shortest path, which is known to have a correct output, hence we have proven that if a weighted graph G has negative weights but no negative cycle then the all pair shortest path algorithm is correct, so we are done.

9. Let G-= (V,E) be a weighted directed graph with no negative cycle, we must design an algorithm to find a cycle in G with minimum weight, since we are designing an algorithm the answer must consist of a description in English, a proof of correctness, and an analysis of the time complexity. Firstly an algorithm to solve this problem for a directed graph G, with vertices V and Edges E, the algorithm will loop over all pairs (u,v) of two vertices u and v, and find the pair that minimized distance(u,v) + distance(v,u), so it will compare the distance of u,v and v,u to a minimum, and if less it will save the pair, the algorithm will check every single vertices and loop through each pair so it will find the directed cycle that starts and ends with the same vertex and contains at least one edge.

Now we must prove the correctness of the algorithm, we start at a given vertex, and since we go over all pairs (u,v) and compare the distance with the minimized distance, so by definition path(u,v) + path(v,u) is by definition a cycle and since we have comparted each possible pair by distance we know that the output must be the minimized shortest path cycle, so the output of the algorithm must be correct, so we are done. Finally we must analyze the time complexity of the algorithm, note that for each cycle of the algorithm it takes O(V^2) to check all of the vertices for each pair, and since we have V different vertices, we will be performing the loop V^2 V times so the final time complexity of the algorithm is O(V^3) which is the desired time complexity of the runtime so we are done.