CS3340 Assignment 3

Name: Michael Song

Student Number: 251101048

1. Input: text T[1 . . . n] and pattern P[1 . . . m].
   Output: largest prefix positions.

```
begin
        i := 1;
        q := 0;
        prefixPos := 0
        prefixSize := 0;
        count : = 0
        while i ⩽ n do
                if T[i] == P[q + 1] then
                        i := i + 1;
                        q := q + 1;
                        count++;

                else
                        if count > prefixSize then
                                prefixSize = count;
                                prefixPos = i;
                        count = 0;


                        if q == 0 then
                                i := i + 1;

                        else
                                q := next[q];



print ”The largest prefix can be found beginning at ”, prefixPos - prefixSize;


end
```

2. PrintLowestCommon(c, X, Y, i, j)

```
  if c[i, j] == 0
    return
```

```
if X[i] == Y[j]
   PrintLowestCommon(c, X, Y, i - 1, j - 1)
   print X[i]
else if c[i - 1, j] > c[i, j - 1]
   PrintLowestCommon(c, X, Y, i - 1, j)
else
   PrintLowestCommon(c, X, Y, i, j - 1)
```

3. We will use greedy solution to solve this problem:

   The greedy solution solves this problem in the most efficient way possible as we maximize distance we can cover from a particular point such that there still exists a place to get water before we run out. The first stop is indicated by the longest point away from the starting location which is less than or equal to mm miles away. The problem displays the principle of optimal substructure, since once the first stopping point p is chosen, the subproblem is solved assuming that it us starting at p. Combining these two plans proves to yield the optimal solution. To demonstrate that this greedy approach in fact yields a first stopping point which is contained in some optimal solution, the following will be done.

   Let O be any optimal solution which has the professor stop at positions o_1, $o_{2, \dots}$ $o_{n.}$

   Let g_1 denote the furthest stopping point that we can reach from the starting point.

   Then o_1 may be replaced by g_2 to create a modified solution G since $o_2 – o_1 < o_2 – g_2$. In other words, the positions in G can be reached without running out of water. Since G has the same number of stops, we conclude g_1 is contained in some optimal solution. Therefore, the greedy solution is proven. It can be implemented with O(n).

4. We know that any full binary tree has exactly 2n - 1 nodes. We can encode the structure of our full binary tree by performing a preorder traversal of tree T. For each node that we record in the traversal, we mark a 0 if it is an internal node and a 1 if it is a leaf node. Since we know the tree to be full, this uniquely determines its structure. Next, note that we can encode any character of C in ⌈lgn⌉ bits. Since there are n characters, we can encode them in order of appearance in our preorder traversal using n ⌈lgn⌉ bits.

5. We will use a dynamic programming in order to solve this problem:

   We initially compute all possible costs a line can have in LineCost[][]. LineCost[i][j] will be used to store the cost of putting words from i to j with i and j being the index value of the words in order. If line cost is negative, this proves that there are too many words to be put into a single line and set LineCost[i][j] for that specific instance as infinity. After the LineCost table is completely filled out with possibilities, recursion can be utilized. We can create an OptimizedTotalCost[j] that stores the optimized total cost from 1 to whatever word j proves to be the index of. We know OptimizedTotalCost[j] is 0 if j = 0 and OptimizedTotalCost[j] is min(OptimizedTotalCost[i-1] + LineCost[i, j]) where i >= 1 and i <= j assuming that j > 0. From this, we minimize the potential for empty space to show up and take up cost.

   The running time of this algorithm is $O(n^2)$.

6. We will utilize Kruskal:

   Sort the edges of G into decreasing order by weight. Let T be the set of edges comprising the maximum weight spanning tree. Set T = ∅.
   Add the first edge to T.
   Add the next edge to T if and only if it does not form a cycle in T. If there are no remaining edges exit and report G to be disconnected.
   If T has n−1 edges (where n is the number of vertices in G) stop and output T. Otherwise go to step 3.

   Algorithm:

```
put all edges in a heap; put each vertex in a set by itself;
put each vertex in a set by itself; (make set(v) for each vertex v)
        while not found a MST yet do begin
                delete max edge, {u, v}, from the heap;
        if u and v are not in the same set (find(u) != find(v))
                mark {u, v} as tree edge;
                union sets containing u and v; (union(u, v)))
        if u and v are in the same set
                do nothing;
end
```

7.
```
      A                           Assume A is directed to B and A is directed to C as well
    /     \
   /        \
      /              \

     5                 2

   /                     \

  B   --   (-10)   -->   C
```

Vertices = {A,B,C} , Edges = {(A,C,2), (A,B,5), (B,C,-10)}

This counterexample provides a situation where the algorithm could potentially not be able to find A->B->C.

8. Yes, the all-pair-shortest-path algorithm still proves to be correct. With no negative cycles, but also having negative weight edges, the time complexity is that of O(|V|^2|E|). By attempting to prove this, we can use an example graph of: 1 -> 0 (distance: 4), 0 -> 2 (distance: -2), 1 -> 2 (distance: 3), 2 -> 3 (distance: 2), 3 -> 1 (distance: -1). From this graph, we can get the shortest paths of:

   vertex 0 to vertex 1 is [0 —> 2 —> 3 —> 1]
   vertex 0 to vertex 2 is [0 —> 2]
   vertex 0 to vertex 3 is [0 —> 2 —> 3]
   vertex 1 to vertex 0 is [1 —> 0]
   vertex 1 to vertex 2 is [1 —> 0 —> 2]
   vertex 1 to vertex 3 is [1 —> 0 —> 2 —> 3]
   vertex 2 to vertex 0 is [2 —> 3 —> 1 —> 0]
   vertex 2 to vertex 1 is [2 —> 3 —> 1]
   vertex 2 to vertex 3 is [2 —> 3]
   vertex 3 to vertex 0 is [3 —> 1 —> 0]
   vertex 3 to vertex 1 is [3 —> 1]
   vertex 3 to vertex 2 is [3 —> 1 —> 0 —> 2]

   This proves that the algorithm works with negative weight edges.

9. The algorithm works by first creating two double arrays of 25 by 25 (based on the read in value at the beginning of the text document) in which the other lines of the text document are read into with the first double array storing the destinations (other nodes) that a source node can travel to and the second double array storing the respective weights of each individual vertex. These can then be printed out to fulfill the requirements of the first required output. For the second output, a heap is required. Using this, we can successfully output the necessary edges to be used to achieve the Minimum Spanning Tree. For the third required output of the total weight of the

minimum spanning tree, we simply add the weights of all the edges travelled through get the total weight off the minimum spanning tree.

The algorithm is correct as we can pinpoint exactly where a specific edge is stored through its weight, source, and destination.

The time complexity is that of: O((|V | + |E|) log |V |). It is like this because we iterate through all the vertex and edges by log(vertex).