Assignment #3

Student #: [redacted]

Name: Zaid Albirawi

UWO email: zalbiraw@uwo.ca

1. Give pseudocode to reconstruct an LCS from the completed c table and original sequences $X = (x_1, x_2, \dots, x_m)$ and $Y = (y_1, y_2, \dots, y_n)$ in $O(m+n)$ time, without using the b table.

   Algorithm: Print-LCS (c, X, Y, i, j)
   Input: the c table, the sequence X, the sequence Y, and the i and j values of the last character of the LCS in the c table.
   Output: prints the LCS.

   ```
   Print-LCS (c, X, Y, i, j)
   if i = 0 or j = 0 then
           return
   if X[i]=Y[j] then
           Print-LCS(c, X, Y, i-1, j-1)
           print Xi
   else if c[i, j] = c[i-1, j] then
           Print-LCS(c, X, Y,  i-1, j)
   else Print-LCS(c, X, Y,  i, j-1)
   ```

   At worst case, the time complexity of the algorithm will be $O(m+n)$. In the worst case scenario, the algorithm will execute $m+n-1$ times, before either i or j reach 0.

2. Suppose that in a 0-1 knapsack problem, the order of the items when sorted by increasing weight is the same as their order when sorted by decreasing value. Give an efficient algorithm to find an optimal solution to this variant of knapsack problem and argue that your algorithm is correct.

   Let $i_1, i_2, \dots, i_n$ represent the item list, while the weight list, $w_1, w_2, \dots, w_n$, and the value list $v_1, v_2, \dots, v_n$, represent the items' and weights and values respectively. Since the order of the items when sorted by increasing weight is the same as their order when sorted by a decreasing values we conclude that,

$$w_1 \leq w_2 \leq \dots \leq w_n$$
$$v \geq v_2 \geq \dots \geq v_n$$

   Algorithm,

   Algorithm: knapsack (I, W)
   Input: I is the item list, where W is the value of the maximum weight.
   Output: a list of item that will generate the optimal solution.

   knapsack (I, W)
   j ← 0
   w ← 0
   list ← Null
   while w + w_j <= W **do**
   w ← w + w_j
   list ← I_j
   j ← j + 1
   return list

   Proof by cases:
   Since $w_1 \leq w_2 \leq \dots \leq w_n$ and $v \geq v_2 \geq \dots \geq v_n$ then as the weight increases the value decreases. Assume that after adding k items we are left with w weight remaining to be filled up, when then have 3 cases. Case one, if $w_{k+1} > w$ then we have reached the optimal solution because $w_{k+1} \leq w_{k+2} \leq \dots \leq w_n$ therefore, we can't add anymore items. Case 2, $w_{k+1} = w$, to obtain the optimal solution we add the item I_{k+1} to the list because even if $w_{k+1} = w_{k+2} = \cdots = w_n$, the value of $v_{k+1} \geq v_{k+1} \geq \cdots \geq v_n$. Case 3, $w_{k+2} \leq w - w_{k+1}$ then add I_{k+1} update the value of w and refer to cases 1 or 2.

   Algorithm complexity $O(n)$, worst case is when all the items (n) fit into the knapsack.

3. Describe an efficient algorithm that, given a set $\{x_1, x_2, \ldots, x_n\}$ of points on the real line, determines the smallest set of unit-length closed intervals that contains all of the given points. Argue that your algorithm is correct.

   Firstly, sort the set $\{x_1, x_2, \ldots, x_n\}$ so that $x_1 \leq x_2 \leq \ \ldots, \leq x_n$ by using a minimum heap sort. Secondly, while the heap is not empty, pull the root, $x_1 = min$, and construct a closed interval so that the interval length is equal to 1 unit-length, $[x_1, x_1 + 1]$. Remove all the points contained in the closed interval from the heap and add that interval to the solution set.

   Algorithm: Unit_Length_Set(A, n)
   Input: A an array of points [0 … n]
   Output: A set of closed intervals that will contain all the points in A.

   Unit_Length_Set(A, n)
   min_heap ← A
   point ← min_heap.getMinimum
   while min_heap.size > 0 **do**
   interval ← [point, point + 1]
   while interval.contains(point) **do**
   min_heap.remove(point)
   point ← min_heap.getMinimum
   set ← interval
   return set

   The algorithm is correct because, creating a closed interval that starts at the smallest value in the heap ensures that the point belongs to that interval. The algorithm then removes that point from the heap and checks if the next smallest point and checks if it belongs to that interval, if it is then the point is removed from the heap else it start another interval. This step ensures that the number of sets of unit-length are minimal and that these intervals hold all the points of A.

4. Show that no compression scheme can expect to compress a file of randomly chosen 8-bit characters by even a single bit. (Hint: Compare the number of possible files with the number of possible encoded files.)

   Let S represent the source file, while C will represent the compressed file. Any compression algorithm requires the elements of S and C to be distinct. Therefore, the number of possible source files using n bits and compressed files using n bits is $2^{n+1} - 1$, where n = 8, hence $2^{9+1} - 1 = 511$. To conclude, no compression scheme can compress a file of randomly chosen 8-bit characters by even a single bit because the number of possible source files is equal to the number of possible compressed files.

5. Compute the $next[\ ]$ function (the prefix function π in the text book) for the pattern $P = babbabbabbababbabb$.

b	a	b	b	a	b	b	a	b	b	a	b	a	b	b	a	b	b
0	0	1	1	2	3	4	5	6	7	8	9	2	3	4	5	6	7

6. Modify the KMP string matching algorithm to find the largest prefix of P that matches a substring of T. In other words, you do not need to match all of P inside T; instead, you want to find the largest match (but it has to start with p_1).

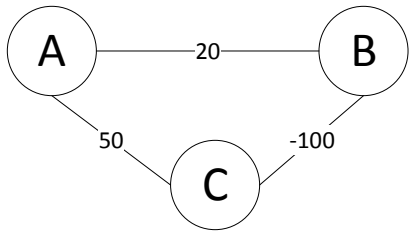
Algorithm: KMP_String_Matching(T, n, P, m)
Input: text T[1 … n] and pattern P[1 … m]
Output: matching positions or the largest prefix of P's position

```
KMP_String_Matching(T, n, P, m)
i ← 1
q ← 0
max ← 0
pos ← 0
pattern ← false
while i <= n do
        if T[i] = P[q + 1] then
                i ← i + 1
                q ← q + 1
        else
                if q = 0 then
                        i ← i + 1
                else q = next [q]
        if q = m then
                print "pattern found at position" i – m
                q = next [q]
                pattern ← true
        if q > max then
                max ← q
                pos ← i - max
if pattern = false then
        if max > 0 then
                print "The largest prefix found" pos – max
        else  print "No prefix found"
```

7. Modify the minimum spanning tree algorithm to find the maximum spanning tree.

   Using Kruskal's algorithm, instead of using a min-heap, use max-heap to sort edges, and instead of removing the min edge, remove the max edge.

8. Find a counter example that shows Dijkstra's algorithm does not work when there is negative weight edge.



   If trying to find the path to B then Dijkstra's algorithm will consider the edges A-B and A-C, since A-B < A-C then algorithm will output A-B as the shortest path with checking the value of A-C + C-B.

9. Given a weighted directed graph $G = (V, E)$, suppose that there are negative weights in G, but there is no negative cycle in G. Is all-pair-shortest-path algorithm still correct? Proven your answer.

   The all-pair-shortest-path algorithm works for negative weighted edges because it computes paths between the graph vertices, pairs, by computing every single path between those pairs and choosing the path with minimal weight. The algorithm computes two types of paths for every pair of vertices, the direct paths, and the indirect paths. Firstly, the algorithm computes each direct path by simply assigning the weights of the edges connecting those pairs to the value of the path, if there exists an edge between those vertices, else it sets the value of that path to infinity. Secondly, the algorithm will find all the indirect paths by finding the path weight from the first vertex to vertex(i), and adding that weight to the weight of the path between that vertex(i) and the second vertex. Since the algorithm ensures that every path between every pair of vertices is calculated then if the graph contains a negatively weighted edge, that edge will be included in the calculation of shortest path as long as it's connected to the graph.

10. Let $G = (V, E)$be weighted directed graph with no negative cycle. Design an algorithm to find a cycle in G with minimum weight. The algorithm should run in the time $O(|V|^3)$.

    To find a cycle in G with minimum weight, use the Floyd Warshall algorithm and modify it so it computes positive cycles. This can be done by changing the for loop that sets d[x, x] to 0, the shortest possible path, to infinity which is the largest possible path, therefore, force the algorithm to calculate the weight of the cycles for that specific vertex, if there exists any. After calculating the weights of all the possible cycles, iterate through the generated d table diagonally at [x, x] and extract the lowest number to find the minimal weight cycle in the graph.

    Algorithm: Floyd_Warshall_Modified(G, V, E)
    Input: V the list of vertices in G, E the list of edges in the same G.
    Output: the cycle with the minimal weight

```
Floyd_Warshall_Modified(G ,V, E)
for x ← 1 to |V| do
        for y ← 1 to |V| do
                if (x, y) ∈ E then
                        d[x, y] ← w (x, y)
                else d[x, y] ← infinity
 for x ← 1 to |V| do
        d[x, x] ← infinity
for m ← 1 to |V| do
        for x ← 1 to |V| do
                for y ← 1 to |V| do
                        if d[x, m] + d[m, y] < d[x, y] then
                                d[x, y] = d[x, m] + d[m, y]
min ← infinity
loc ← Null
for x ← 1 to |V| do
        if min  > d[x, x] then
                min ← d[x, x]
                loc ← x
if loc != Null then
        print "The cycle with the minimal weight is at" loc "with a weight of " min
```

    This algorithm is of $O(|V|^3)$ because of the three nested for loops that compute all the possible paths between ever pair of vertices and that is the most expensive part of the algorithm.