CS 3340 Assignment 3 - Nicholas Gardi – 250868721 – 4/2/19

1. P = babbabbabbababbabb
   π = 001123456789234567
   this code was used to compute

```java
public class Next {

    public static void main(String args[]) {
    String pattern = "babbabbabbababbabb";
        int i = 1;
        int j = 0;
        int[] prefixArray = new int[18];

        while (i < 18) {

            while (pattern.charAt(i) != pattern.charAt(j) && j > 0) {
                j = prefixArray[j - 1];

            }
            if (pattern.charAt(i) == pattern.charAt(j)) {
                prefixArray[i] = j + 1;
                i++;
                j++;

            } else {
                prefixArray[i] = j;
                i++;
            }
        }

        for (int k = 0; k < prefixArray.length; ++k) {
            System.out.println(prefixArray[k]);
        }
    }
}
```

2. We can modify the KMP Algorithm to find the longest prefix string between two. When finding each matching prefix, we keep track of the length and store the length and location of the longest prefix at each turn of the next table used in the KMP algorithm or update the length and location if there is a new longest prefix with a longer length than the one stored. However, at the end, if we find that both strings are the same or that the first string contains the second string, we just return the found string as this is the longest prefix.

3. ReconstructLCS(table, firstString, secString, row, col):
   if table[row][col] == 0
   return
   if firstString[i] == secString[j]
   ReconstructLCS(table, firstString, secString, row - 1, col - 1)
   print(firstString[i])
   elseif table[row - 1][col] > table[row][col - 1]
   ReconstructLCS(table, firstString, secString, row - 1, col)
   else
   ReconstructLCS(table, firstString, secString, row, col - 1)

4. At each step just pick the lightest (and most valuable) item that you can pick. To see this solution is optimal, suppose that there was some item j that we included but some smaller, more valuable item i that we didn't. Then, we could replace the item j in our knapsack with the item i. it will definitely fit because i is lighter, and it will also increase the total value because i is more valuable. Algorithm:

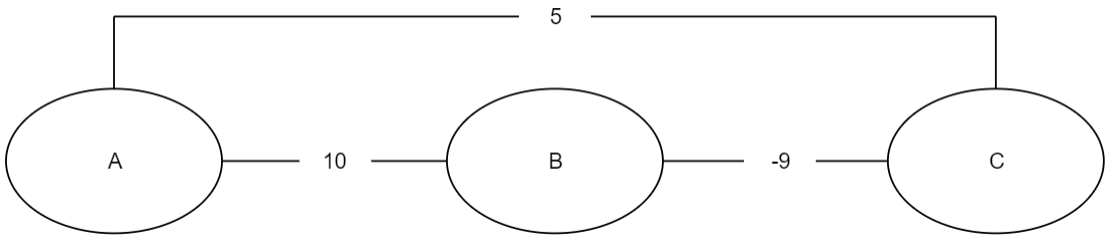
```
Knapsack(n, W):
Initialize n+1 by W+1 table K
For counter = 1 to W do:
        K[0,counter] = 0
For counter = 1 to n do:
        K[counter,0] = 0
For x = 1 to n do
        For y = 1 to W do
                If y < x.weight then K[x,y] = K[x-1,y]
                K[x, y] = max(K[x − 1, y], K[x − 1, y − x.weight] + x.value)
```

5. For the unbounded knapsack problem (unlimited amount of items), you can use a simple array of size W+1 (the weight that the backpack can carry + 1) instead of a 2d array. Array[i] would store the maximum value that the robber can take using all items and a backpack capacity of i. The following algorithm would be used to fill the array and solve this problem.

```
Inputs: valueArray, weightArray, W(capacity of knapsack), n (number of items)
Initialize array A of size W+1
For int i = 0 to W-1
        For int j = 0 to n
                If  weightArray[j] <= i
                        A[i] = max (A[i], A[i – weightArray[j]] + valueArray[j]
Return A[W]
```

6. We can modify Kruskal's algorithm for finding a minimum spanning tree to look for a maximum spanning tree while also using a Union-find algorithm to check for cycles in the graph. Normally you would use a min-heap to sort the edges, but for this case you would use a max-heap to sort the edges into the right order and in this case also the max edge is taken out every time of the graph into the max-heap

7.



Consider the above graph which consists of vertices A,B and C with edges AB, BC, and AC, with edge weights of (AB) = 10, (BC) = -9, (AC) = 5. Dijkstra's algorithm is greedy so it would take path AC as 5 is less than 10 so AC has a lower edge weight than AB. The true shortest path is AB then BC as that would just be length 1. Dijkstra's algorithm does not work with negative edge weights

8. Given a weighted directed graph G = (V,E) with negative weights but no negative cycle, the all-pair shortest-path algorithm is still correct as it takes in all the paths between a pair of vertices in G. The weights for all possibilities of nodes will be considered. If there is any negative edge weight it will be included in the algorithm unlike in Dijkstra's algorithm.

9. To find a cycle in G with a minimum weight you can use the Floyd-Warshall algorithm. Use this algorithm to figure out all the shortest paths where you record in a matrix all the paths from node 1 to the last node n in graph G and then go through those shortest paths to find a cycle with the smallest weight. After you run the FW algorithm and get the paths, you assign one variable called minimum which is set to infinity and for each pair of vertices recorded, if the distance indicated on the matrix of row u and column v (where u and v are numbers that represent nodes) and the distance of row v and column u is less than the minimum, then assign those two distances summed together to find the new minimum and remember the pair of u and v. When you're done running through all the shortest-paths with the above algorithm you would have found the two nodes with a cycle with minimum weight. This is $O(V^3)$ because the FW algorithm is $O(n^3)$ and is the most inefficient part of the solution.