Assignment 3 - CS 3340B

Name: Riley Emma Gavigan

Student Number: 251150776

Student ID: rgavigan

Question 1

Modified KMP Algorithm [Largest Matching Prefix of P]

```
Input: Text T[1...n] and Pattern P[1...m]
Output: String of largest matching prefix of P matching substring of T
begin
  i := 1
  q := 0
  largestMatch := 0
  currentMatch := 0


  while i <= n do
    // When there is a match between the next pattern element and T[i]
    if T[i] == P[q + 1] then
      // Increment current match
      currentMatch := currentMatch + 1
      // Check against largestMatch and update if necessary
      if currentMatch >= largestMatch then
        largestMatch := currentMatch
      i := i + 1
      q := q + 1
    // When there is not a match between next pattern element and T[i]
    else
      if q == 0 then
        i := i + 1
      else
        q := next[q]
      // Reset current match back to 0
      currentMatch := 0
  return P[1...largestMatch] // Return the largest matching prefix of P
end



// Test Case For Modification Testing:
T = abcdedegf
P = def
The result we want is "de", the largest matching prefix of P
Running Through Algorithm:
i = 1 -> 1 = 3: Simply increment i as q == 0 each time.
i = 4: Match b/w d and d, currentMatch = 1. largestMatch = 1. q = 1, i = 5.
i = 5: Match b/w e and e, currentMatch = largestMatch = 2. q = 2, i = 6.
i = 6: No match. q = next[q] = 1 then q = 0. Now match d and d, q = 1, i = 7.
i = 7: Match b/w e and e, currentMatch = 2 = largestMatch. Set i = 8, q = 2.
i = 8 and i = 9: No matches, final result is P[1...2] = "de". Correctly works.
```

Question 2

Suffix Array for **hippityhoppity**:

Map h → 1, i → 2, o → 3, p → 4, t → 5, y → 6

i	suffix T[i:]	Mapping	rank[i]
1	hippityhoppity	12442561344256	1
2	ippityhoppity	24425613442560	3
3	ppityhoppity	44256134425600	10
4	pityhoppity	42561344256000	8
5	ityhoppity	25613442560000	5
6	tyhoppity	56134425600000	12
7	yhoppity	61344256000000	14
8	hoppity	13442560000000	2
9	oppity	34425600000000	6
10	ppity	44256000000000	9
11	pity	42560000000000	7
12	ity	25600000000000	4
13	ty	56000000000000	11
14	y	60000000000000	13

Sorted Suffix Array:

i	suffix T[i:]	Mapping	rank[i]
1	hippityhoppity	12442561344256	1
8	hoppity	13442560000000	2
2	ippityhoppity	24425613442560	3
12	ity	25600000000000	4
5	ityhoppity	25613442560000	5
9	oppity	34425600000000	6
11	pity	42560000000000	7
4	pityhoppity	42561344256000	8
10	ppity	44256000000000	9
3	ppityhoppity	44256134425600	10
13	ty	56000000000000	11
6	tyhoppity	56134425600000	12
14	y	60000000000000	13
7	yhoppity	61344256000000	14

Sorted Indices: [1, 8, 2, 12, 5, 9, 11, 4, 10, 3, 13, 6, 14, 7]

Question 3

14.4-2: Give pseudocode to reconstruct an LCS from the completed c table and the original sequences $X = [x_1, x_2, ..., x_m]$ and $Y = [y_1, y_2, ..., y_n]$ in $O(m + n)$ time, without using the b table.

```
Reconstruct_LCS(c, X, Y, i j)
Input: Table c, Sequence X, Sequence Y, Index i, Index j
Output: Printed reconstruction of the LCS
begin
  if c[i, j] == 0:
    return
  if X[i] == Y[j]:
    Reconstruct_LCS(c, X, Y, i - 1, j - 1) // decrement i and j
    print X[i] // print next value in LCS
  else if c[i, j - 1] > c[i - 1, j]:
    Reconstruct_LCS(c, X, Y, i, j - 1) // decrement j
  else:
    Reconstruct_LCS(c, X, Y, i - 1, j) // decrement i


// If we wanted to also return a string of the LCS, we could use a global variable or
// a parameter of the current string of the LCS, and we could add X[i] to it after
// the print X[i] statement, wihch could be returned at the very end of the algorithm
```

Question 4

15.2-3: Supposed 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 the knapsack problem, and argue that your algorithm is correct.

This variant of the 0-1 knapsack problem can be solved optimally by **taking the items with the greatest values**.

The reasoning behind this is because if items sorted by their increasing weight matches sorting by decreasing value, this is what sample data would look like:

W: 1 2 3

V: 40 30 20

V/W: 40, 15, 6.66

Where in this case, the items are as follows $[w_n, v_n] = [1, 40], [2, 30], [3, 20]$. This makes it very evident that the items are perfectly sorted in respect to value/weight, where the element with the smallest weight has value 40/weight, the second element has value 15/weight, and the last has value 6.66/weight.

Algorithm:

Consider an optimal solution where we take an item with weight w_1, value v_1, and drop an item with weight w_2, value v_2, if $w_1 > w_2$ and $v_1 < v_2$, we can substitute item 1 with item 2 for a better, more optimal solution. This case will always hold as the higher weight item will have a lower value.

Question 5

Modify the minimum spanning tree algorithm to find the maximum spanning tree:

```
// Modified Prim's Algorithm to find Maximum Spanning Tree
MST_Prim(G, w, r)
  for each u in V[G] do
    key[u] := -infinity // set to negative infinity for max spanning tree
    pi[u] := NIL
  key[r] := 0
  Q := V[G]
  while Q != 0 do
    u := Extract_Max(Q) // Get the maximum vertex
    for each v in Adj[u] do
      if v in Q and w(u, v) > key[v] then // if weight greater than max in current MST
        key[v] := w(u, v)
        pi[v] := u
        update key[v] in Q
```

Modified Kruskal's Algorithm to find Maximum Spanning Tree

Idea: Choose most expensive edge in a graph

Algorithm:

```
    put all edges in a max heap
    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
```

Question 6

Find a counter example with three vertices that show Dijkstra's algorithm does not work when there is a negative weight edge.

Consider a graph with vertices A, B, and C, where there is an edge from A to B with weight 2, an edge from C to B with weight -10, and an edge from A to C with weight 4.

Run Dijkstra's algorithm on this graph to find the shortest path from A to C:

1. Start at vertex A, set distance to 0 and all others to infinity.
2. Visit A and update neighbour's distances. Now the distance to A is 0, to B is 2, and to C is 4.
3. Visit vertex B since it's the next smallest. No relaxation can be performed because there is no outgoing edge from B.
4. Visit vertex C, which relaxes B with a new weight of 4 - 10 = -6. This is not allowed, we are relaxing a node that has already been visited and was deemed 'minimal' by Disjkstra's SSSP algorithm.

Table Progress:

A	B	C
0	infinity	infinity
0	**2**	4
0	-6	**4**

As we can see, there is an error because the shortest path from A to B has a distance of -6, but Dijkstra's algorithm failed to find this path. In Dijkstra's algorithm, since B was already visited we deemed it finished and that the shortest path to B was already found. However in reality, the path could be further reduced by this negative edge. This is a counter-example that successfully shows that Dijkstra's algorithm does not work for negative edges.

Question 7

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

The Floyd-Warshall All-Pair Shortest Paths algorithm can be modified to solve this problem to find a cycle in G with the minimum weight.

Algorithm Design:

The Floyd-Warshall algorithm is used to find the shortest distance between any two vertices in the graph G. A variable will be used to store the minimum weight cycle found in the graph, initially set to infinity. Afterwards, for every vertex v in V, the algorithm will check if there is a path from a vertex v to itself (a cycle), and if the cycle has a weight lower than the current minimum weight, this cycle will be stored and the minimum weight will be updated. After all vertices have been checked, the algorithm is complete and the minimum weight cycle is returned.

Completeness:

The algorithm is correct because it first uses the Floyd-Warshall algorithm to create and fill the distance matrix. Furthermore, it checks every single vertex as a possible cycle start/endpoint and will correctly update the minimum weight cycle every time a cycle is found to have a minimum weight compared to previously checked cycles. It then returns the minimum weight cycle after every vertex has been checked, meaning there is no other possibility for a smaller cycle. Because we are guaranteed that there is no negative cycle in the graph, there is no need to check for negative cycles.

Complexity:

The algorithm runs correctly in $O(|V|^3)$. The time complexity of the Floyd-Warshall algorithm is known and it is $O(|V|^3)$. Furthermore, we perform another pass afterward on every single vertex to update the minimum weight cycle, which will take $O(|V|^2)$ time because each vertex V has to then go through every other vertex to return to itself as a cycle. This only adds on to the complexity of Floyd-Warshall, giving a total time complexity of $O(|V|^3 + |V|^2) = O(|V|^3)$.

Question 8

```
'''
Name: Riley Emma Gavigan
Student Number: 251150776
Student ID: rgavigan
CS 3340B - Assignment 3 - Single Source Shortest Path Algorithm
'''
import sys

# Class to representa  node in the graph
class Node:
    # Instance variables
    u = None
    d = None

    # Constructor that initializes the node and the distance
    def __init__(self, u, d):
        self.u = u
        self.d = d

    # Method to get the node number
    def get_node(self):
        return self.u

    # Method to get the distance
    def get_distance(self):
        return self.d

    # Method to set the distance
    def set_distance(self, d):
        self.d = d

    # Method to print out the node in string format
    def __str__(self):
        return f"    Node: {self.u} <Distance: {self.d}>"

# Class to represent an edge in the graph
class Edge:
    # Instance variables
    u = None
    v = None
    weight = None

    # Constructor that initializes the endpoint nodes and the weight of the edge
    def __init__(self, u, v, weight):
        self.u = u
        self.v = v
        self.weight = weight

    # Method to get the first endpoint node of the edge
    def get_u(self):
        return self.u

    # Method to get the second endpoint node of the edge
    def get_v(self):
        return self.v

    # Method to get the weight of the edge
    def get_weight(self):
        return self.weight

    # Method to print out the edge in string format
    def __str__(self):
        return f"    Edge: {self.u} -> {self.v} <Weight: {self.weight}>"

# Class to represent a minimum heap that will be used in SSSP Algorithm
class Heap:
    # Instance variables
    A = [] # array of the keys [sorted by Node.distance]
    H = [] # heap that holds the indices for A
    max_val = None
    heap_max = None

    # Initialize a heap with the array of keys and the number of keys
    def heap(self, keys, n):
        # Set edge_array to the keys array [will contain all distances of nodes]
        self.A = keys

        # Add an element at start of edge array to make indexing by 1
        self.A.insert(0, None)

        # Set the max to n (nodes are from 1 -> n)
        self.max_val = n

        # Set the heap to an array of 2 * max elements
        self.H = [None] * (2 * self.max_val)

        # Set the heap max to 2 * max
        self.heap_max = 2 * self.max_val

        # Call heapify on the heap
        self.heapify()

    # Method to heapify the heap with bottom-up-approach
    def heapify(self):
        # Loop from 1 to n
        for i in range (1, self.max_val + 1):
            # Set heap[i + n - 1] to i
            self.H[i + self.max_val - 1] = i

        # Loop from n - 1 to 1
        for i in range (self.max_val - 1, 0, -1):
            # if A[H[2i]] < A[H[2i + 1]]
            if self.A[self.H[2 * i]].get_distance() < self.A[self.H[2 * i + 1]].get_distance():
                self.H[i] = self.H[2 * i]
            # if A[H[2i]] >= A[H[2i + 1]]
            else:
                self.H[i] = self.H[2 * i + 1]

    # Method that returns true if element with id is in heap, false otherwise
    def in_heap(self, id):
        # Comparison edge
        compare_edge = Edge(0, 0, 0)

        # Loop through the edge array
        for i in range(1, len(self.A)):
            # If the id of the node is equal to passed in id
            if self.A[i].get_node() == id:
                return True
        # Return false if id edge is not found in edge array
        return False

    # Method that returns true if heap is empty, false otherwise
    def is_empty(self):
        # If the heap is empty
        if self.H[1] == 0:
            return True
        # If the heap is not empty
        else:
            return False

    # Method that returns the minimum key in the heap (weight)
    def min_key(self):
        return self.A[self.H[1]].get_distance()

    # Method that returns the minimum id of the heap
    def min_id(self):
        return self.H[1]
```

```
# Method that returns the key of the element with id in the heap
def key(self, id):
    return self.A[id].get_distance()

# Method that deletes the element with the minimum key from the heap
def delete_min(self):
    # Create an edge to hold the minimum edge
    removed_edge = Node(0, float("inf"))

    # Set A[1] to be removed edge
    self.A[0] = removed_edge

    # Set heap[heap[1] + max - 1] to 0
    self.H[self.H[1] + self.max_val - 1] = 0

    # Create edge v to reference A[H[1]]
    v = self.A[self.H[1]]

    # Set i to floor((H[1] + n - 1) / 2)
    i = (self.H[1] + self.max_val - 1) // 2

    # Reheapify
    while i >= 1:
        # If A[H[2i]] < A[H[2i + 1]]
        if self.A[self.H[2 * i]].get_distance() < self.A[self.H[2 * i + 1]].get_distance():
            self.H[i] = self.H[2 * i]
        # If A[H[2i]] >= A[H[2i + 1]]
        else:
            self.H[i] = self.H[2 * i + 1]

        # Set i to floor(i / 2)
        i = i // 2

    # Return removed edge
    return v

# Method that decreases the key of the element with id to new_key if its current key is greater than new_key
def decrease_key(self, id, new_key):
    # Get the new key
    self.A[id].set_distance(new_key)

    # Set i to floor((id + max - 1) / 2)
    i = (id + self.max_val - 1) // 2

    # Reheapify
    while i >= 1:
        # If A[H[2i]] < A[H[2i + 1]]
        if self.A[self.H[2 * i]].get_distance() < self.A[self.H[2 * i + 1]].get_distance():
            self.H[i] = self.H[2 * i]
        # If A[H[2i]] >= A[H[2i + 1]]
        else:
            self.H[i] = self.H[2 * i + 1]

        # Set i to floor(i / 2)
        i = i // 2

# Method to print out the heap contents
def print_heap(self):
    # Print out the heap
    for i in range(1, len(self.H)):
        print(self.H[i], end = " ")
    print()

    # Print out the nodes in A
    for i in range(1, len(self.A)):
        print(self.A[i].get_node(), self.A[i].get_distance(), end = " ")
    print()
```

```
# Method to perform Dijkstra's Algorithm on the graph using a heap data structure
def dijkstra(keys, source):
    # Fill edges array with edges from the adjacency list
    edges = []
    for i in range(1, len(keys)):
        for j in range(1, len(keys[i])):
            edges.append(Edge(keys[i][j][0], i, keys[i][j][1]))

    # Create array with source set to 0 and the rest set to infinity
    nodes = []
    for i in range(1, len(keys)):
        if i == source:
            nodes.append(Node(i, 0))
        else:
            nodes.append(Node(i, float("inf")))

    # Create min heap of the nodes
    heap = Heap()
    heap.heap(nodes, len(nodes))

    # Create array to hold the shortest path
    shortest_path = []

    # While the heap is not empty
    while len(shortest_path) < len(keys) - 1:
        # Delete the minimum element from the heap
        u = heap.delete_min()

        # Add the node to the shortest path
        shortest_path.append(u)

        # For each edge adjacent to node u
        for i in range(0, len(keys[u.get_node()])):
            # Get the node v
            v = keys[u.get_node()][i][0]

            # If v is in the heap
            if heap.in_heap(v):
                # If the distance of u plus the weight of the edge is less than the distance of v
                if u.get_distance() + keys[u.get_node()][i][1] < heap.key(v):
                    # Decrease the key of v to the new distance
                    heap.decrease_key(v, u.get_distance() + keys[u.get_node()][i][1])

    # Print the shortest path
    for i in range(1, len(shortest_path)):
        print(f"({shortest_path[i - 1].get_node()},{shortest_path[i].get_node()}) : {shortest_path[i].get_distance()}")



# Main Method to run the program with input file name
def main():
    # Get the input file
    input_lines = sys.stdin.readlines()

    # Read an integer from the first line and store as # of vertices
    num_vertices = int(input_lines[0])

    # Create an empty adjacency list
    adjacency_list = []

    # Create an empty list for each vertex
    for i in range(num_vertices + 1):
        adjacency_list.append([])

    # Read the remaining lines of the file and fill in the adjacency list
    for i in range(1, len(input_lines)):
        # Split the line by spaces
        line = input_lines[i].split()
        # Add the edge to the adjacency list representation of the graph
        adjacency_list[int(line[0])].append([int(line[1]), int(line[2])]) # source: [destination, weight]

    # Run Dijkstra's algorithm
    dijkstra(adjacency_list, 1)

main()
```