Assignment 3

Nicholas Maisel

```
class Vertex():
    def __init__(self,vid):
        self.vid = vid
        self.d = 0
        self.p = None

class Graph():
    def __init__(self):
        self.verticies = []
        self.wEdges = {}

    def addEdge(self,u,v,w):
        self.wEdges[u,v] = w


    def InitSingleSource(self,src):
        #src = self.verticies[int(src)]
        for vert in self.verticies:
            vert.d = float('Inf')
            vert.p = None
        src.d = 0

    def Relax(self,u,v):
        if v.d > u.d + self.wEdges[u,v]:
            v.d = u.d + self.wEdges[u,v]
            v.p = u

    def ShortestPath(self, G, src, v, emptyList =[]):
        ''' Finds the shortest path by following the predecessor values of
            each vertex '''
        if src == v:
            emptyList.append(src.vid)
        elif v.p == None:
            print("There is no path from:", src.vid, " to: ", v.vid)
        else:
            self.ShortestPath(self,src,v.p,emptyList)
            if v.vid != None:
                emptyList.append(v.vid)
        return(emptyList)



    def BellmanFord(self,src):
        src = self.verticies[int(src)-1]
        self.InitSingleSource(src)
        for vert in range(len(self.verticies)-1):
            for edge in self.wEdges:
                u,v = edge[0], edge[1]
                self.Relax(u,v)
        for edge in self.wEdges:    # check for negative weight cycles
            u,v = edge[0], edge[1]
```

```
52              if v.d > u.d + self.wEdges[u,v]:
53                  print("Negative_weight_cycle_detected.")
54                  return 1
55          for vt in self.verticies:  # Print output
56              print()
57              print(f'Path_from_{src.vid}_to_{vt.vid}_costs:_{vt.d};_', end = '')
58              sPath = self.ShortestPath(self,src,vt,emptyList = [])
59              for i in range(len(sPath)):
60                  if i != len(sPath)-1:
61                      print(f'_{sPath[i]}_âĞŠ', end = '')
62                  else:
63                      print(f'_{sPath[i]}')
```

After reading the data into the program, the Bellman Ford algorithm is called (line 43). The runtime of the Bellman Ford algorithm is fairly simple to understand. On line 45, we call the InitSingleSource function, which iterates over each vertex once, thus θ(V). Then we make |V| - 1 passes over the edges for the loop starting on line 46, to ensure there are no negative weight cycles. Each pass takes θ(E). Therefore the runtime is O(VE).

```
1   def knapsack(itemGen, capacity):
2       sack = {}
3       sackValue = 0
4
5       while capacity > 0:
6
7           try:
8               item = next(itemGen)    # Yield the next value from generator
9               itemName = item[0]
10              itemQty = item[1][1]
11              itemPrice = item[1][0]
12          except:
13              return(sack, sackValue)
14
15          if capacity >= itemQty:
16              sack[itemName] = itemQty #adds all of item[index] qty to sack
17              sackValue += itemQty * itemPrice #adds to sack value
18              capacity -= itemQty #updates cap
19              item[1][1] = 0
20
21          elif capacity < itemQty:
22              sack[itemName] = capacity
23              item[1][1] -= capacity
24              sackValue += capacity * itemPrice
25              capacity = 0
26
27      return(sack, sackValue)
28
29
30  def spice_gen(items):
31      ''' The spice_gen is a generator function
32          used to yeild each spice one at a time'''
33      i = 0
```

```
34      while i < len(items):
35          yield(items[i])
36          i +=1
```

The knapsack algorithm runs at O(n), with some restrictions. It is O(n) because we iterate over the items list (length n). Each one of the lines in the while loop starting on line 5, runs in constant time, thus O(n). However, this does not take into account the time it takes to sort the list. If we are including the time it takes to sort the list using python's built in sorted() function (like I do in the program), then the runtime becomes O(nlogn).