Analysis of Algorithms Homework 2

Question 1.

```
dynamicKnapsackSol(int n, int W, int weight[], int value[]){
     Let K be table of [n+1][W+1]
     for(i = 1; i < n; i++)
          K[i][0] = 0
     for(j = 1; j < W; j++)
          K[0][j] = 0

     for(i = 1; i < n; i++)
          for(j = 1 j < W; w++)
               if(j < i[weight])
                    K[i][j] = K[i-1][j]
               else
                    K[i][j] =
                    max(K[i-1][j],K[i-1][j-weight[i]]+value[i]
     //K[n][W] is the final answer
}
```

The algorithm takes as inputs the maximum weight W, the number of items n and two sequences $v = \{v_1, v_2, \ldots, v_n\}$ and $w = \{w_1, w_2, \ldots, w_n\}$. The $c[i][j]$ values are stored inside a table $c[0\ldots n][0\ldots W]$ whose entries are computed in the following order.The first row of C is filled from left to right then second third and so fourth. At the end of the algorithm, $c[n][W]$contains the maximum value the thief can take

Question 2.

A greedy algorithm would solve this problem optimally. If we maximize the distance that can be covered from some particular point, there is a place to get gas if we run out. Our first stop should be at the longest point from the initial starting position, which would be <= to m miles away. The problem at hand also exhibits optimal substructure because once we have chosen our first stopping point p, we solve the sub-problem with the assumption we are starting at p. Adding these two plans together yields an optimal solution. It's important to show that the greedy approach yields a first stopping point contained in some optimal solution. Let R be an optimal solution with the professor stopping at positions $r_1, r_2, \ldots, r_k$. Let f_1 denote the farthest stopping we can reach from the starting point. Then we can replace r_1 by f_2 to create a generate a modified solution F since $r_2 - r_1 < r_2 - f_1$. In essence, we can make it to the positions contained in G without running out of gas. Since G has the same amount of stops, we can safely conclude that f_1 is contained inside some optimal solution. Therefore a greedy approach is valid.

Question 3.

Part A

Where **T** is initial tour

Generate permutation generates another possible combination of cities

Score calculates the total tour cost

```
bruteForceSolver(T){
     best_tour = T
     best_score = score(T)
     while(permutations of T)
          S = score(T)
          generatePermutation()
          if(s < best_score)
               Best_score = s
               Best_tour = tour
}
```

Run time is (n-1)! Which is incredibly slow

Part B

Observation: Consider any shortest path in our TSP graph G. Then any subpath of this shortest path is also a shortest path (between its own endpoints). So, the above observation suggests that TSP exhibits the optimal substructure that will allow us to use DP

```
FindMinRoute(Tsp[][]){
     let sum = 0
     let  increment = 0
     let j = 0
     let i = 0
     let minVal = infinity
     let visitedRouteList be a empty list
     visitedRouteList.add(0)
     Let route be table of [tsp.size()]

     while (i < tsp.size and j < tsp[i].size) {
           /* This is the corner of the 2x2 array*/
            if (increment >= tsp[i].size - 1) {
                break
            }
            /* If this path has not been visited then
            and if cost is less then, update the cost */
            if (j != i and !(visitedRouteList.contains(j))) {
                if (tsp[i][j] < minVal) {
                    minVal = tsp[i][j]
                    route[increment] = j + 1
                }
            }
            j++

            /* Check all paths from the
            ith city */
            if (j == tsp[i].size) {
                sum += minVal
                minVal = infinity
                visitedRouteList.add(route[increment] - 1)
                j = 0
                i = route[increment] - 1
```

```
            increment++
        }
    }
    /* Update the ending city in array
     from city which was last visited */
    i = route[increment - 1] - 1
    for (j = 0; j < tsp.size; j++) {
        if ((j != i) and tsp[i][j] < minVal) {
            minVal = tsp[i][j]
            route[increment] = j + 1            }
    }
    sum += minVal
```

Run Time: $O(N^2 * \log_2 N)$

Assume n = 4

Assume we have a graph whose edges cost as follows
$c(AB) = 100$
$c(AC) = 101$ $c(BC) = 100$
$c(AD) = 200$ $c(BD) = 101$ $c(CD) = 150$
If the algorithm starts from A it will generate a tour of ABCDA with a cost of 100+100+150+200 which is 550 which is a very inefficient solution

Part C.

/* Recursive Solution*/

```
def TSPGetMinDistance(mainSource, source, cities)
   if len(cities) == 1
       minDis = GetCostVal(source, cities[0], mainSource) +
GetCostVal(cities[0], 0, mainSource)
       return minDis
   else
       Dist = [] is a list
       for city in cities
           cities = cities[city]
           dcities.remove(city)
           Dist.add(GetCostVal(source, city, source) +
TSPGetMinDistance(mainSource, city, dcities))
       iterative_process.add(Dist)
       return min(Dist)
```

Worst Case Run Time: $O(2^n n^2)$

Space complexity: $O(2^n n)$

This is our cost matrix. The diagonals are 0 because the distance from a city to itself is just 0.

$$E = \begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

g(2, ∅) = c_{21} = 5
g(3, ∅) = c_{31} = 6
g(4, ∅) = c_{41} = 8

```
F = g(1,{2,3,4}) = min( {c_12 + c_23 + c_34 + c_41} ,
{c_13+c_32+c_24+c_41} , {c_14 + c_42 + c_23 + c_31} )

= min ( {10+9+12+8} , {15+13+10+8} , {8+8+9+6} )
= min ({39},{46},{31})

= 31 is the shortest distance

City (1->2, 2->3, 3->4)
     10+9+12 = 31
```

/* Garbage Attempt*/

```
A set of locations V nd a cost function c that calculates
distance
Assume Dtsp is infinity
DynamicTSP(N,s)
     Visited[s] = 1
     Cost  = 0
     if(N =2 and k!=s)
          Cost(N,k) = dist(s,k)
          Return cost
     Else
          for(j = 0; j < N; j++)
               for( i = 0; i < N and visited[i] = 0; i++)
                    if(i!=s and j!=s)
                         Cost(N,j) =
```

/*Garbage Attempt code did not work */