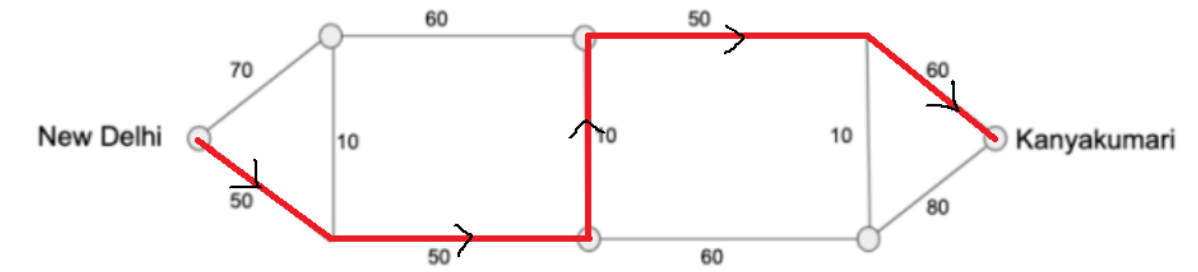
End Sem Assignment

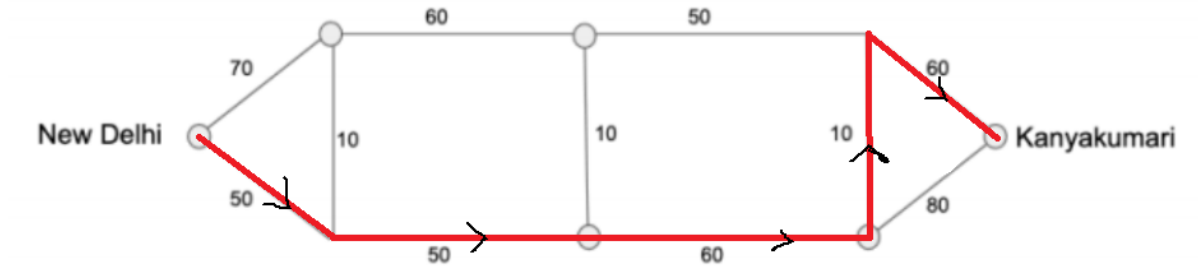
Algorithms Design and Analysis

Name: Shaunak Pal
Roll No.: 2018098

Ans 1) a)



b)



c) **PSEUDOCODE:**

(Note: djikstra (g, s, d) denotes the usual djikstra algorithm run on graph g, from source vertex s, and distances stored in d.)

```
function find_route (graph g, vertex source, vertex end, int r):
    /* Initialize a new graph where we store the new modified graph which contains
    shortest distance from each fuel station to each fuel station and a dist array to store
    distances */
    graph newGraph = g
    dist[n] <- Initialize an array with length as number of fuel vertices
    /* Run djikstra on all vertices which have fuel station */
    for i in newGraph.vertices do:
        if (vertex is fuel station) then:
            djikstra (newGraph,  i, dist)
    /* Remove the edges which have length greater than r */
    for edge e in newGraph.edges do:
        if (e.weight > r) then:
            newGraph.removeEdge(e)
    /* Run final djikstra from given source to get the shortest path to end stored in dist
    array*/
    djikstra (newGraph, source, dist)
    return dist[end]
```

JUSTIFICATION:

If our path is not the most optimal path, then:
Case 1: The path doesn't pass through one of the refuel stations we expected, but this is not possible since the final djikstra gives the shortest path from source to every vertex including the end.
Case 2: The distance between the fuel stations itself is shorter than expected. This is not possible since we again used djikstra to calculate the e shortest path between each fuel station and ran it for each refueling vertex which gives the closest fuel station from 1 fuel station to other.
Since both the cases contradict, our algorithm does give the optimal path.

RUNTIME:

Dijkstra's algo is run for all refuel vertices so O(V*(E+VlogV)) and edges which are greater than r are removed so O(E) and then final djikstra is run O(E+VlogV). Therefore, the overall run time is, **O (V(E+VlogV))**

Ans 2)

a) i) **BASE CASE:**

dp[0] = 1 (for a sequence of length 1)

RECURRENCE:

dp[i] = max(dp[i], dp[j] + 1) where $0 < j < i$ if $A_j > A_i$ when j is odd and $A_j < A_i$ when j is even
dp[i] = 1 if no such j exists

ii) **DP TABLE:**

dp[i] = { 0, if i = 0
{ max(dp[i], dp[j] + 1), where $0 <= j < i$ if $A_j > A_i$ when j is odd and $A_j < A_i$ when j is even
{ 1, if no such j exists

PSEUDOCODE:

```
function LSS (list sequence, list sub_sequence):
        n = sizeof (sequence)
        /* dp array of 1 dimension */
        dp = [ 1 for i from 1 to n ]
        for i from 1 to n:
                for j from 1 to i:
                        if ((dp%2==0 and sequence[i] < sequence[j]) or (dp%2!=0 and
                        sequence[i]>sequence[j])) then:
                                dp[i] = max (dp[i], dp[j] + 1)
        sub_sequence = dp
        return sub_sequence
```

iii) **RUNTIME:** The outer runs from start 1 to end n once where for each i in the outer loop maximum possible number of switching sub sequences is calculated from 1 to i, i.e. the inner loop runs i times which is n at max.
Therefor the runtime is n * n times, **$O(n^2)$.**

b) **PSEUDOCODE:**

```
function LSS_nonDP (list sequence [1: N], list sub):
    /* a variable which is 1 if we are looking for a decreasing sub sequence and 0 when
    looking for an increasing sequence */
    low = 1
    for i from 2 to N do:
        /* find the lowest consecutive number or highest consecutive number
        according to need */
        if ((low and sequence[i] > sequence[i-1]) or (!low and sequence[i] <
        sequence[i-1])) then:
            sub.append(sequence[i-1])
            low = 1 – low
    /*  To check if last element in sequence belong to sub sequence */
    if ((low and sequence[N] > sub[sizeof(sub)-1)] or (! low and sequence[N] <
    sub[sizeof(sub)-1])) then:
        sub.append(sequence[n-1])
    return sub
```

RUNTIME:

Since we run through the whole sequence just once its **O(n).**

JUSTIFICATION:

Suppose our algorithm does not give the most optimal solution. This means that there's an optimal solution that has more switching sequences than the ones we have, i.e. there exists an A_i, A_j, A_k such that $A_i < A_j > A_k$ or $A_i > A_j < A_k$.
But according to our if condition whenever we search for a decreasing sub sequence and we find an increase or when we are finding an increasing sequence and we find a decrease in the sequence; we append the elements. Thus, it is not possible that any switching has been skipped. Therefore, there cannot be a more optimized solution. Hence our algorithm is correct.

Ans 4) **CONSTRUCTION:**

This problem can be converted to a max flow problem. We create a new graph G'(V', E'), from the given graph G(V, E) where each vertex in G is extended to k vertices in G', i.e. we create k new vertices for each vertex in G and connect them i.e. total number of vertices V' = (k+1) * V.
So now our vertex V in graph G looks like {V'_0, V'_1, V'_2, ……. V'_k in graph G'}.

Now, for each vertex V in graph G we connect V'_i to V'_{i+1} in the graph G' each with capacity r so that at most r rats can pass through the edges and the rats can spend at most k nights at a vertex V. now the equivalent of connecting edge u-v in graph G is to connect u'_i to v'_{i+1} in graph G' (for all $1 < i < k$)

Therefore, now our new start vertex becomes s'_0 and end vertex becomes t'_k. After this we can use the usual Ford Fulkerson's algorithm to find the max flow in the graph G'. Now if the max flow exceeds or is equal to r then all the rats can exit the maze in k nights.

JUSTIFICATION:

Let us suppose our algorithm gives the wrong answer i.e. there is a way for the rats to exit but our algorithm returns they cannot. Since we used Ford Fulkerson's algorithm and it is a known fact about its correctness, the only problem can be with our graph. This means there is some edge missing in our graph. But for every edge u-v in graph G we have connected u'_i to v'_{i+1} and we cannot connect u'_l to v'_l since they cannot change vertices in 0 night and in order to the rats to stay at a vertex we have provided k vertices in graph G' to each vertex v in G. Therefore, no extra edges are possible, hence our algorithm is bound to be true.

RUNTIME:

The runtime is of the basic Ford Fulkerson's algorithm on graph G' and since V' = k * V and E' = k * E our overall runtime is **O (rk(V+E))**.