E/15/385

Weerasinghe S.P.A.P.E

CO 322-Data Structures and Algorithms

Lab 02

Dynamic Programming

1. Using recursion (or otherwise) implement the *minCost* function.

```
/*
* E/15/385
* Lab 02
* Dynamic Programming
* */
class Train {
    static int [][] cost = {{0, 3, 12, 23, 41}, // cost from 0
              {0, 0,  2,  4, 34}, // cost from 1
              {0, 0,  0,  12, 3}, // cost from 2
              {0, 0,  0,  0, 12}, // cost from 3
              {0, 0,  0,  0,  0}  // cost from 4
    };

    static int iMax = 5;
    static int jMax = 5;

    // Just for testing, min cost from 0 to 4 should be 8.
    static int answer = 8;

    //recursive function to find the minimum cost from i to j

    public static int minCost(int i, int j) {

        // If i is same as j
        // or j is next to i
        if (i == j || i+1 == j)
            return cost[i][j];

        // Initialize min cost as direct ticket from
        // source 'i' to destination 'j'.
        int min = cost[i][j];

        // Try every intermediate vertex to find minimum
        //Travere through the 2D array
        for (int k = i+1; k<j; k++)
        {
            int c = minCost( i, k) + minCost( k, j);
            if (c < min)
                min = c;
        }
        return min;
    }
```

```
    public static void main(String [] args) {
        int start=2;
        int end =4;
   int r = minCost(0,4);
   System.out.println("Minimum cost from "+start+ " from "+end+" = "+ r);
   if(r == answer)
       System.out.println("Your implementation might be correct");
   else
       System.out.println("Too bad. Try again (" + r + ")");
    }
    }
```

2. What is the runtime complexity of your implementation.

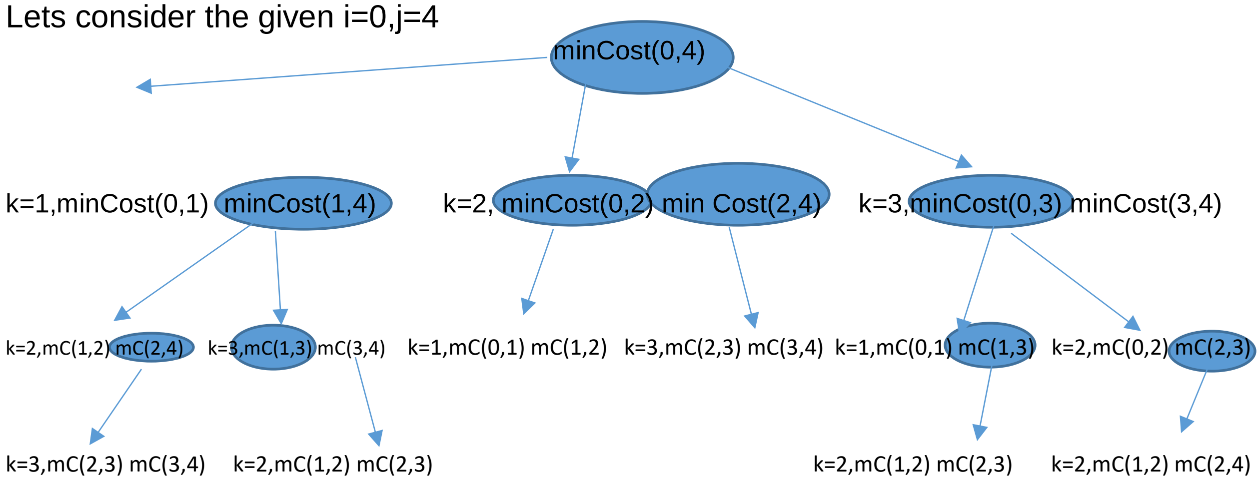
```
public static int minCost(int i, int j) {

    // If i is same as j
    // or j is next to I                                           //cost      //Time
    if (i == j || i+1 == j)                                        //c1         // 1
        return cost[i][j];

    // Initialize min cost as direct ticket from
    // source 'i' to destination 'j'.
    int min = cost[i][j];                                           //c2         //1

    // Try every intermediate vertex to find minimum
    //Travere through the 2D array
    for (int k = i+1; k<j; k++)                                     //c3          //
    {
        int c = minCost( i, k) + minCost( k, j);
        if (c < min)
            min = c;
    }
    return min;
}
```

Lets consider the given i=0,j=4



Time complexity of the above implementation is exponential as it tries every possible path from 0 to N-1.Time complexity is O(n!) The above solution solves same sub problems multiple times,it can be seen from the above recursion tree. More the stations more sub problems are involved when there are more and more stations to calculate in between.

3. Argue that dynamic programming can be used to improve the runtime.

The minimum cost to reach N-1 from 0 can be recursively written as following:

```
minCost(0, N-1) = MIN { cost[0][n-1],
                        cost[0][1] + minCost(1, N-1),
                        minCost(0, 2) + minCost(2, N-1),
                        ........,
                        minCost(0, N-2) + cost[N-2][n-1] }
```

The above code is the implementation of above recursive formula.

As I mentined above the time complexity of the above implementation is exponential as it tries every possible path from 0 to N-1
The above solution solves same subrpoblems multiple times (it can be seen by drawing recursion tree as above)

Overlapped sub-problems

Cost 0-1 + cost 1-i
Cost 1-2 + Cost 2-d
Cost 2-3 + Cost 3-d
Cost 3-4 + Cost 4-d

Cost 0-2 + cost 2-d
Cost 2-3 + Cost 3-d
Cost 3-4 + Cost 4-d
Cost 4-5 + Cost 5-d

Cost 0-3 + cost 3-d
Cost 3-4 + Cost 4-d
Cost 4-5 + Cost 5-d
Cost 5-6 + Cost 6-d

re-computations of same sub problems can be avoided by storing the solutions to sub problems and solving problems in bottom up manner.
Here we can use dynamic programming

Let's see how can we use dynamic programming

- We have to Consider the routes as a graph

- We have to perform a topological sort on the vertices of the graph: 0, 1, 2, …, N-1

- Then we Compute the minimum cost to each station and store them in an array minimum[0..N-1]

- Minimum cost for station 0 is 0, then minimum [0] = cost[0][0] = 0

- Minimum cost for station 1 is cost[0][1], then length[1] = cost[0][1]

- [0] + cost[0][1]

- Minmum cost for station 2 is the minimum of:

- minimum[0] + cost[0][2]

- minimum[1] + cost[1][2]

- Minimum cost for station 3 is the minimum of:

- minimum[0] + cost[0][3]

- minimum[1] + cost[1][3]

- minimum[2] + cost[2][3] like wise

Similarly, minimum[4], minimum[5], … minimum[N-1] are calculated.

4. Use dynamic programming to improve the runtime.

```
/*
* E/15/385
* Lab 02
* Dynamic Programming
* */
class Q4 {
    static int [][] cost = {{0, 3, 12, 23, 41}, // cost from 0
            {0, 0,  2,  4, 34}, // cost from 1
            {0, 0,  0,  12, 3}, // cost from 2
            {0, 0,  0,  0, 12}, // cost from 3
            {0, 0,  0,  0,  0}  // cost from 4
    };
```

```java
    static int iMax = 5;
    static int jMax = 5;

    // Just for testing, min cost from 0 to 4 should be 8.
    static int answer = 8;

    //recursive function to find the minimum cost from i to j
    static int LargestNum=Integer.MAX_VALUE;//largest possible integer value

    public static int minCost(int i, int j) {

        {
            // minimum[i] stores minimum cost to reach station i
            // from station 0.
            int [] minimum = new int[j+1-i];//size of the array needed
            for (int k=0; k<j+1-i; k++) {
                minimum[k] = LargestNum;//initialize array with larget possible value
so any other cost is smaller than this
            }
                minimum[0] = 0;//initialize first val with zero


            // traverse through the stations and fing if an intermediate station gives
loswet cost
            for (int m=i; m<j; m++) {
                for (int n=m+1 ; n <j+1; n++) {
                    if (minimum[n - i] > minimum[m - i] + cost[m][n]) {
                        minimum[n - i] = minimum[m - i] + cost[m][n];
                    }
                }

            }

            return minimum[j-i];
        }
    }

    public static void main(String [] args) {
        int r = minCost(0,4);
        //System.out.println("Minimum cost from "+start+ " from "+end+" = "+ r);
        if(r == answer)
            System.out.println("Your implementation might be correct");
        else
            System.out.println("Too bad. Try again (" + r + ")");
    }
}
```

5. Calculate the runtime of your implementation in part 4 above. Assume, hashing is O(1).

We Compute the minimum cost to each station and store them in an array minimum[0..N-1]

- Minimum cost for station 0 is 0, then minimum [0] = cost[0][0] = 0

- Minimum cost for station 1 is cost[0][1], then length[1] = cost[0][1]

- [0] + cost[0][1]

- Minmum cost for station 2 is the minimum of:

- minimum[0] + cost[0][2]

- minimum[1] + cost[1][2]

- Minimum cost for station 3 is the minimum of:

- minimum[0] + cost[0][3]

- minimum[1] + cost[1][3]

- minimum[2] + cost[2][3] like wise

Similarly, minimum[4], minimum[5], … minimum[N-1] are calculated

Above configuration need O(N) extra space
Therefore overall 1+2+3+…..+N=n(n-1)/2

Therefore runtime complexity is O((n^2-n)/2)= O(n^2)

Run time complexity is O(n^2). Which is better than non dynamic programming.
So Dynamic Programming have improved the performance but with use of extra O(N) space