Answer 1:

<u>Logic –</u>

Algorithm will apply Bellman-Ford strategy, to calculate chokepoint of every vertex. Algorithm will make use of array (HashMap can be used to improve efficiency) to save chokepoint of every vertex. In every iteration, the value of chokepoint will be updated with weight of the maximum edge taken to reach to that vertex.

Here for simplicity, we will consider the given weights and vertex names are in integer.

(We can improve the efficiency of overall algorithm by breaking the final loop when value of chokepoint remains same after one iteration.)

<u>Analysis –</u>

As algorithm is making use of Bellman-Ford strategy, in the worst case run time complexity of the algorithm will be $O(\#e * \#v)$.

<u>Algorithm –</u>

```
class Vertex {
	int vertexName;
	//...
}

class SourceVertex extends Vertex {
	//...
}

class Edge {
	Vertex fromVertex;
	Vertex toVertex;
	//...
}

class Graph {
	Vertex[] vertexs;
	Edge[] edges;
	//...
}

class Weights {
	int [][] weights;
	//...
}


public int[] findChokePoints(Graph G, Weights w, SourceVertex s) {
	// Returns a array that gives chokepoint of every vertex

	int minWeight                 = 0;
	int source                    = s.vertexName;
	int n                         = G.vertexs.length;

	// Create a matrix chokepoint to save chokepoint of every vertex
	int[] chokePoints             = new int[n];

	// Create a matrix currentWeights to save minimum weight of every vertex
	int[] currentWeights          = new int[n];
```

```
        // Initialize both the arrays
        for (Vertex vertex : G.vertexs)     {
               currentWeights[vertex.vertexName]   = Integer.MAX_VALUE;
               chokePoints[vertex.vertexName]      = Integer.MIN_VALUE;
               }

        currentWeights[source] = 0;

        for (int i=1; i<=n-1; i++)  {
               for (Edge e : G.edges)       {
                      int u = e.fromVertex.vertexName;
                      int v = e.toVertex.vertexName;
                      if (u == source)      {
                             minWeight = w.weights[u][v];
                      }
                      else   {
                             minWeight = Math.min(currentWeights[u], w.weights[u][v]);
                      }
                      if (currentWeights[v] > minWeight && currentWeights[u] != Integer.MAX_VALUE)
                             currentWeights[v] = minWeight;

                      if (chokePoints[v] < minWeight)
                             chokePoints[v]  = minWeight;
               }
        }
        // Return array containing chokepoint details
        return chokePoints;
}
```

Answer 2:

Logic –

Algorithm makes use of Bellman-Ford and instead of 2D, it will create a 3D array/matrix where the last term Z would indicate the number of tolls taken to reach to that vertex. For simplicity let`s consider that all vertex names are in integer format.

Analysis –

As we are making use of the Bellman-Ford which is also taking number of tolls into consideration, the time complexity of the algorithm will be $O(\#e * \#v * \#k)$. Thus in the worst case, where number of tolls and number of edges is equal to number of vertices the complexity will become $O(\#v^3)$.

Recurrence –

$$DP(v,i,k) = min \begin{cases} DP(v,i,k-1), & check\ distance\ with\ one\ less\ toll \\ \min\limits_{(u,v)\in E} \begin{cases} DP(u,i-1,k-1), & if\ v \in T \\ DP(u,i-1,k), & if\ v\ !\in T \end{cases} & check\ distance\ coming\ from\ its\ parent \end{cases}$$

Algorithm –

```
public static int[] findChokePoints(Graph G, Weights w, Tolls T, SourceVertex s, DestinationVertex t,
int k) {
       int n = G.vertexs.length;
       int distanceTravelled = 0;

       int [] parent = new int[n];
       int[][][] DP = new int[n][n][k];

       for (Vertex v : G.vertexs)  {
              if (v.vertexName == s.vertexName)  {
                     DP[s.vertexName][0][0] = 0;
                     DP[s.vertexName][0][k] = 0;
              }
              else   {
                     DP[v.vertexName][0][0] = Integer.MAX_VALUE;
              }
       }

       //Starting bellman-Ford
       //#vertices-1 iterations
       for (int i=1; i < n; i++)   {
              for (int j=1; j < k; j++)   {
                     for (Vertex v : G.vertexs)  {
                            DP[v.vertexName][i][k] = Integer.MAX_VALUE;
                            for (Edge e : G.edges && e.toVertex.equals(v))   {
                                   if (T.contains(v.vertexName))      {
                                           distanceTravelled= DP[e.fromVertex.vertexName][i-1][k-1];
                                   }
                                   else if(!T.contains(v.vertexName)) {
                                           distanceTravelled= DP[e.fromVertex.vertexName][i-1][k];
                                   }
                                   if (DP[v.vertexName][i][k] > distanceTravelled)  {
                                           DP[v.vertexName][i][k] = distanceTravelled;
                                           parent[v.vertexName] = e.fromVertex.vertexName;
                                   }
                            }
                     }
              }
       }

       // If destination is not reachable in given number of tolls then return null

       if (DP[t.vertexName][n-1][k] != Integer.MAX_VALUE)      {
              return null;
       }

       Stack stk = new Stack<Integer>();
       ArrayList<Integer> paths = LinkedList<Integer>();

       //Calcularte path taken to reach to destination
       stk.push(t.vertexName);
       //Add destination in the path traveled.
       paths.append(t.vertexName);
       while(!stk.isEmpty()){
              int m = s.pop();
              stk.push(m.vertexName);
              paths.append(m);
              if(m.vertexName == s.vertexName)   {
                     break;
              }
       }

       return paths;
}
```

Answer 3:

<u>Logic –</u>

Algorithm will first calculate values of deficit of every value in the original array R and will save it in another matrix R`(deficits). Now interval of largest deficit can be found out by finding the range of largest contiguous sum of the new array R`.

<u>Analysis –</u>

In worst case as we our using two nested loops the time complexity would be at maximum will be $O(n^2)$. In the best case, algorithm uses only one loop and goes ahead till it doesn`t find out the sum less than zero, the time complexity in worst case will be $O(n)$.

<u>Algorithm –</u>

Worst Case : (order of $n^2$ )

```
class Interval{
	int intrvlLeft;
	int intrvlright;

	public Interval(int a, int b)	{
		intrvlLeft = a;
		intrvlright= b;
	}
}

public static Interval maxDeficitInterval (int[] R)	{

	double sum		= 0;
	double avg		= 0;
	double maxSoFar	= 0;

	//variable to save the lower bound of the maximum deficit interval
	int intrvlLeft= 0;
	//variable to save the upper bound of the maximum deficit interval
	int intrvlright	= 0;

	//variable to save the deficits of individual elements of given array.
	double[] deficits = new double[R.length];

	//calculate average of given array
	for(int i = 0; i < R.length; i++)	{
		sum+=R[i];
	}
	avg = sum/R.length;

	//calculate deficit of every index in given array
	for(int i = 0; i < R.length; i++)	{
		deficits[i] = avg - R[i];
	}

	//calculate the interval values
	for(int l = 0; l < deficits.length; l++)	{
		sum = 0;
		for(int r = 0; r < deficits.length; r++)	{
			sum += deficits[r];
			if (maxSoFar < sum)	{
```

```
                              maxSoFar      = sum;
                              intrvlLeft    = l;
                              intr vlright  = r;
                       }
                }
         }
         //Return array containing interval details
         return new Interval(intrvlLeft, intrvlright);
}
```

Algorithm Case: (order of n )

```
public static Interval getMaxDeficitInterval(int[] R)    {

         double sum            = 0;
         double avg            = 0;
         double maxSoFar = 0;
         double maxForWholeSol = 0;

         //variable to save the lower bound of the maximum deficit interval
         int intrvlLeft= 0;
         int intrvlLeftForSol  = 0;
         //variable to save the upper bound of the maximum deficit interval
         int intrvlright       = 0;

         //variable to save the deficits of individual elements of given array.
         double[] deficits = new double[R.length];

         //calculate average of given array
         for(int i = 0; i < R.length; i++)   {
                sum+=R[i];
         }
         avg = sum/R.length;

         //calculate deficit of every index in given array
         for(int i = 0; i < R.length; i++)   {
                deficits[i] = avg - R[i];
         }

         //calculate the interval values
         for(int l = 0; l < deficits.length; l++)   {
                maxSoFar = maxSoFar + deficits[l];
                if (maxSoFar > maxForWholeSol)     {
                       maxForWholeSol = maxSoFar;
                       intrvlLeft = intrvlLeftForSol;
                       intrvlright = l;
                }

                if(maxSoFar < 0)     {
                       intrvlLeftForSol = l+1;
                       maxSoFar = 0;
                }
         }
         //Return array containing interval details
         return new Interval(intrvlLeft, intrvlright);
}
```

Answer 4:

<u>Logic –</u>

Modify dp for finding longest common subsequence (LCS) of two strings to find LCS of three strings. {let X be the first string with length l, Y be second string with length m and Z be third string with length n}

LCS3(X[:l], Y[:m], Z[:n])

Start comparing characters from the end of strings.

Then depending on this outcome, problem will be divided into 2 cases.

i) Characters from all strings matches

In this case, algorithm will append character in the list of common subsequences (CS) and then run LCS3 on the remaining string

```
if (X[l] == Y[m] == Z[n])
    cs.append(X[l])
    cs.append(LCS3(X[:l-1], Y[:m-1], Z[:n-1]))
```

ii) Characters from all or two strings do not match.

In this, again three situations arise,

1. Decrement length of string one by removing last character from input string and run LCS3
2. Decrement length of string two by removing last character from input string and run LCS3
3. Decrement length of string three by removing last character from input string and run LCS3
4. Return the list with maximum length out of above 3 and append it in the original CS list

```
if ( X[l] != Y[m]  ||  Y[m] != Z[n]  || X[l] != Y[n] )
    getListWithMaximumLength (
            LCS3(X[:l-1], Y[:m], Z[:n]),
            LCS3(X[:l], Y[:m-1], Z[:n]) ,
            LCS3(X[:l], Y[:m], Z[:n-1])
            )
```

Base cases for this algorithm, they will be as follows –

i) If length of any one or all string becomes less than 0 then return the CS.

```
if ( l < 0  ||  m < 0  ||  n < 0 )
    return cs
```

<u>Recurrence –</u>

$$\mathrm{LCS3}(X[0{:}l],\ Y[0{:}m],\ Z[0{:}n]) = \begin{cases} \mathrm{cs.append}(X[l]).\mathrm{append}\big(\mathrm{LCS3}(X[0{:}l], Y[0{:}m-1], Z[0{:}n])\big) & \text{if last character of all string matches} \\ , & \\ \mathrm{getStringWithMaximumLength}\begin{cases} \mathrm{LCS3}(X[0{:}l], Y[0{:}m], Z[0{:}n-1]) \\ \mathrm{LCS3}(X[0{:}l], Y[0{:}m-1], Z[0{:}n]) \\ \mathrm{LCS3}(X[0{:}l-1], Y[0{:}m], Z[0{:}n]) \end{cases} & \text{if } \textit{last character of all strings doesn`t match} \end{cases}$$

<u>Correctness –</u>

Recurrence that we have taken into consideration is correct.

<u>Proof –</u>

Proof by induction.

Base Case :

Let, l, m and n are length of string one, two and three respectively.

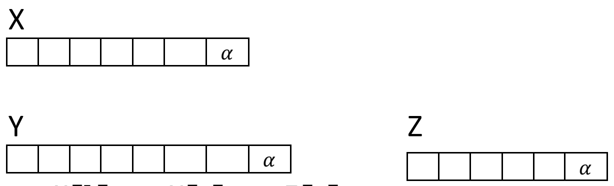
Either l < 0  or  m < 0 or n < 0

This is trivial case, if l < 0; then value of X[l] is undefined thus, common subsequence for all 3 strings at that time will be an empty string.

Similar case will arise if any of the ‘m’ or ‘n’ is less than 0.

Induction Step :

Suppose the recurrence holds true till indices l + m + n – 1;
Let us consider we have to compare strings X[0:l]; Y[0:m] & Z[0:n].
Now there will be two conditions either last character of all strings matches or they don't.



In case `X[l] == Y[m] == Z[n] ==` $\alpha$, then according to our algorithm the longest common subsequence of X, Y and Z will be: – '$\alpha$' + longest common subsequence of X[0:l-1], Y[0:m-1], Z[0:n-1].

We will try to prove this by contradiction. Suppose the longest common subsequence does not include $\alpha$ then this means that there exists a common subsequence that does not end with $\alpha$ however, this is not possible as the last character of every string is $\alpha$. Thus, the longest common subsequence must have $\alpha$ in it. Hence the recurrence that we got from our algorithm is correct.

Now suppose, last characters of all strings are not matching.

Let`s say `Z[n] !=` $\alpha$, then it might be the case that $\alpha$ is second last character in Z, thus we need to check longest common subsequence of X[0:l], Y[0:m], Z[0:n-1]. Similarly for the case where `Y[n] !=` $\alpha$ longest common subsequence can be found by analyzing X[0:l], Y[0:m-1], Z[0:n]. Also, we can write same argument for X. Thus longest common subsequence will be string with maximum length out of LCS3(X[0:l], Y[0:m], Z[0:n-1]) and LCS3(X[0:l], Y[0:m-1], Z[0:n]) and LCS3(X[0:l-1], Y[0:m], Z[0:n]). This is exactly same recurrence returned by algorithm.

Hence from base case and induction step we can say that our recurrence is correct.

<u>Analysis –</u>

As we are filling up the 3D array matrix while solving the sub solution, the complexity of this algorithm will be $O(l * m * n)$. Where l, m and n stands for the lengths of the strings that are being compared. In the worst case, when all the strings are of same size 'n', the complexity will become $O(n^3)$.

<u>Algorithm –</u>

```
ArrayList<char> cs = new LinkedList<char>(max (l, m, n));

ArrayList<char> LCS3(String X, String Y, String Z)      {
        int l = X.length() - 1;
        int m = Y.length() - 1;
        int n = Z.length() - 1;
        /* Following steps build cs[l+1][m+1][n+1] in bottom up fashion.
           Note that L[i][j][k] contains length of LCS of X[0..i-1] and Y[0..j-1]  and Z[0.....k-1]
        */
        String[][][] cs = new String[l+1][m+1][n+1];

        for (int i=0; i<=l; i++)  {
            for (int j=0; j<=m; j++)  {
                for (int k=0; k<=n; k++) {
                    if (i == 0 || j == 0|| k == 0)
                         cs[i][j][k] = "";
                    else if (X.charAt(i) == Y.charAt(j)  &&  X.charAt(i)==Z.charAt(k))
                         cs[i][j][k] = cs[i-1][j-1][k-1].append(X.charAt(i));
                    else
                         cs.[i][j][k] =
                            getListWithMaximumLength (
                                   cs[i-1][j][k],
                                   cs[i][j-1][k],
                                   cs[i][j][k-1]);
                }
            }
        }

        /* L[l][m][n] contains length of LCS for X[0..l-1] and Y[0..m-1] and Z[0..n-1]*/
        return cs[l][m][n];

}

ArrayList<char> getListWithMaximumLength (ArrayList<char>A, ArrayList<char>B, ArrayList<char>C) {
              if (A.length() > B.length()  &&  A.length() > C.length())
              return A;
              if (B.length() > A.length()  &&  B.length() > C.length())
              return B;
              if (C.length() > A.length()  &&  C.length() > B.length())
              return c;
}
```

Answer 5:

Logic –

Let i and j represents the lower and upper bound on the number of coins remaining and
DP(i, j) represents the maximum value user can collect from coins between i and j and
$V_i$ represents the value of the coin at the $i^{th}$ position.

Logic for this algorithm is to maximize the value of all coins collected by first user and minimize the values collected by second. As there are two ways for user to collect the coins, the function will have two conditions and it will return the maximum value out of them.

Analysis –

As we are filling up the 2D array matrix while solving the sub solution, the complexity of this algorithm will be $O(m * n)$. where m and n stands for the number of rows and columns in the matrix. Since the matrix is of size *n*n*, the complexity in the worst case will be $O(n^2)$.

Recurrence –

$$\text{DP(i, j)} = max \begin{cases} \text{Vi} + \min \begin{cases} DP(i+2, j) \;\; If\ second\ user\ chose\ first\ coin \\ , \\ DP(i+1, j-1)\ if\ second\ user\ chose\ last \end{cases} ,\ \boldsymbol{If\ first\ user\ chose\ the\ first\ coin} \\ , \\ Vj + \min \begin{cases} DP(i+1, j-1) \;\; if\ second\ user\ chose\ first \\ , \\ DP(i, j-2) \;\; if\ second\ user\ chose\ last\ coin \end{cases} ,\ \boldsymbol{If\ first\ user\ chose\ the\ last\ coin} \end{cases}$$

## Algorithm –

```
public int[][] maximizeProfit(int[] arr)  {
       // Returns optimal maximum cash first player can collect from given 'n' coins
       // let us assume that 'n' is even, so that every user gets a fair chance

           // Create a table to store solutions of subproblems
          int n = arr.length;
          int[][] val = new int[n][n];

           for (int interval = 0; interval < n; ++interval)
              // Fill matrix val using above recursive formula.
              // map is filled from diagonal to map[0][n-1] which is the result.
           {
               for (int i = 0, j = interval; j < n; ++i, ++j)
               {
                   int firstChoseFirst                       =  arr[i];
                   int firstChoseLast                        =  arr[j];
                   int bothUsersChoseFirst                   =  0;
                   int bothUsersChoseLast                    =  0
                   int firstChoseFirstSecondChoseLast        =  0;
                   int firstChoseLastSecondChoseFirst        =  0;

                   if ((i+2) <= j)
                     bothUsersChoseFirst = val[i+2][j];

                   if ((i+1) <= (j-1))
                          firstChoseFirstSecondChoseLast     =  val[i+1][j-1;

                   firstChoseLastSecondChoseFirst      = firstChoseFirstSecondChoseLast;

                   if (i <= (j-2))
                     bothUsersChoseLast    =   val[i][j-2];

                   val[i][j] = Math.max(
                                       firstChoseFirst
                                       + Math.min( bothUsersChoseFirst,
                                                   firstChoseFirstSecondChoseLast),
                                       firstChoseLast
                                       + Math.min( firstChoseLastSecondChoseFirst,
                                                   bothUsersChoseLast)
                                       );

               }
           }
           // Return solution to original problem
           return val[0][n-1];
       }
```

Answer 6:

Logic –

Let DP($r_i$, $c_j$, k) represents the maximum sum of the items picked from given matrix, starting from row $r_i$, column $c_j$ and by choosing 'k' items.

The algorithm can be solved in 2 ways either by choosing few items from same row and choosing remaining items from subsequent rows or by choosing all the items from the subsequent rows. Thus the algorithm takes the maximum out of those two values.

In this algorithm, for simplicity (for m*n matrix) we number the rows from 1 to m (instead of 0 to m-1) and Z access index from 1 to k (instead of 0 to k-1). The solution for problem to find maximum pick up sum from 3*3 matrix by choosing 2 items will be found by solving DP(1, 0, 2).

Base case for this algorithm will be when value of i or k will be 0, at that time the value of DP will be 0.

Recurrence –

A is the given 2D array (matrix)
m is total number of rows
n is total number of columns
$r_i$ is $i^{th}$ row from the input matrix
$c_j$ is $j^{th}$ column from the input matrix
k is total items to be chosen

$$DP(r_i, c_j, k) = \begin{cases} \begin{cases} 0 & If\ i == 0\ OR\ k == 0 \ldots base\ case, \\ A[i][j] + DP(ri, cj+1, k-1) & If\ (m-i) * n < k \ldots\ elements\ to\ choose\ are\ very\ large, \end{cases} \\ MAX \begin{cases} A[i][j] + DP(ri, cj+1, k-1) & If\ we\ choose\ an\ element\ from\ current\ row, \\ DP(ri+1, 0, k) & If\ we\ don`t\ choose\ any\ element\ from\ the\ current\ row, \end{cases} \end{cases}$$

Analysis –

As we are filling up the 3D array matrix while solving the sub solution, the complexity of this algorithm will be $O(m * n * k)$. where m, n and k stands for the number of rows, columns and number of items to be chosen respectively. In the worst case when all of the values are equal to 'n', the time complexity will be $O(n^3)$.

Algorithm –

```
public int maximizeSumInMatrix (int[][] arr, int picks) {
// Returns maximum sum of 'picks' items from given 2D array
// let us assume that 'n' is even, so that every user gets a fair chance

       int sum       = 0;
       int count     = 0;
       int rows      = arr.length;
       int cols      = arr[0].length;
       int [] items  = new int[picks];

    // Create a matrix maxSum to store solutions of subproblems
    // the very first bottom row and very first column along Z-axis (picks) will contain 0
       int [][][] maxSum= new int[rows+1][cols][picks+1];

    // Update matrix to store solutions of base level subproblems
       for (int j = 1; j <= cols; j++)     {
              sum                    += arr[rows][j];
```

```
                maxSum[rows][0][j]    = sum;
            }

        // Update matrix to store solutions of intermediate level subproblems
            for (int i = 1; i <= rows; i++)    {
                for (int j = 1; j <= cols; j++)    {
                    for (int k = 1; k <= picks; k++)   {

                        // Base case
                        if (k == 0 || i == 0){
                            maxSum[i][j][k]      = 0;
                        }
                        else   {
                            if ( (rows - i + 1)*j < k ) {
                                maxSum[i][j][k] = arr[i][j] + maxSum[i][j+1][k-1];
                                //if (i == 1 && (j + k == picks))
                                //     items[count++]  = arr[i][j];
                            }
                            else   {
                                maxSum[i][j][k] =
                                        Math.max(
                                            arr[i][j] + maxSum[i][j+1][k-1],
                                            maxSum[i+1][0][k]);
                            }
                        }
                    }
                }
            }
            // Return solution to original problem
            return a[1][0][picks];
        }
```

Collaborators :

Jhalak Maheshwari
Manish Yadav
Prasad Tajane
Shikhar Parsoon
Sourabh Punja