DAY 13:

SESSION 1:

Toll Gate Collection

ID:11135 Solved By 693 Users

There is a national highway which is of length **N** kilometers. **K** toll gates are currently present in the highway. But due to a recent court order, the government can collect fee only at toll gates which are separated by more than D kilometers. The distance of these K toll gates from the starting point and the fee collected in each of these K toll gates are passed as input. The program must print the maximum revenue that the government can collect in a one way trip (from start to ending point).

Boundary Condition(s):
5 <= N <= 1000
1 <= D <= 1000
2 <= K <= 100
1 <= Fee collected at each toll gate<= 500.

Input Format:
The first line contains N and D separated by a space.
The second line contains K.
The third line contains the distance in kilometers from the starting point for the K toll gates, with the values separated by a space.
The fourth line contains the fee collected at the K toll gates, with the values separated by a space.

Output Format:
The first line will contain the the maximum revenue that can be collected from start to end.

Example Input/Output 1:
Input:
200 50
5
60 70 120 130 140
50 70 50 30 20

Output:
100

Explanation:
There are 5 tollgates present at 60, 70, 120, 130 and 140 kms respectively.
As the tollgates should be separated by more than 50kms, the maximum revenue will be obtained when the tollgates at **60^{th}** km and **120^{th}** km are selected as the total revenue is 50+50 = **Rs.100**.
The tollgates at 70^{th} and 120^{th} kms cannot be chosen to give 70+50 = Rs.120 revenue as they are not separated by more than **50** kms.

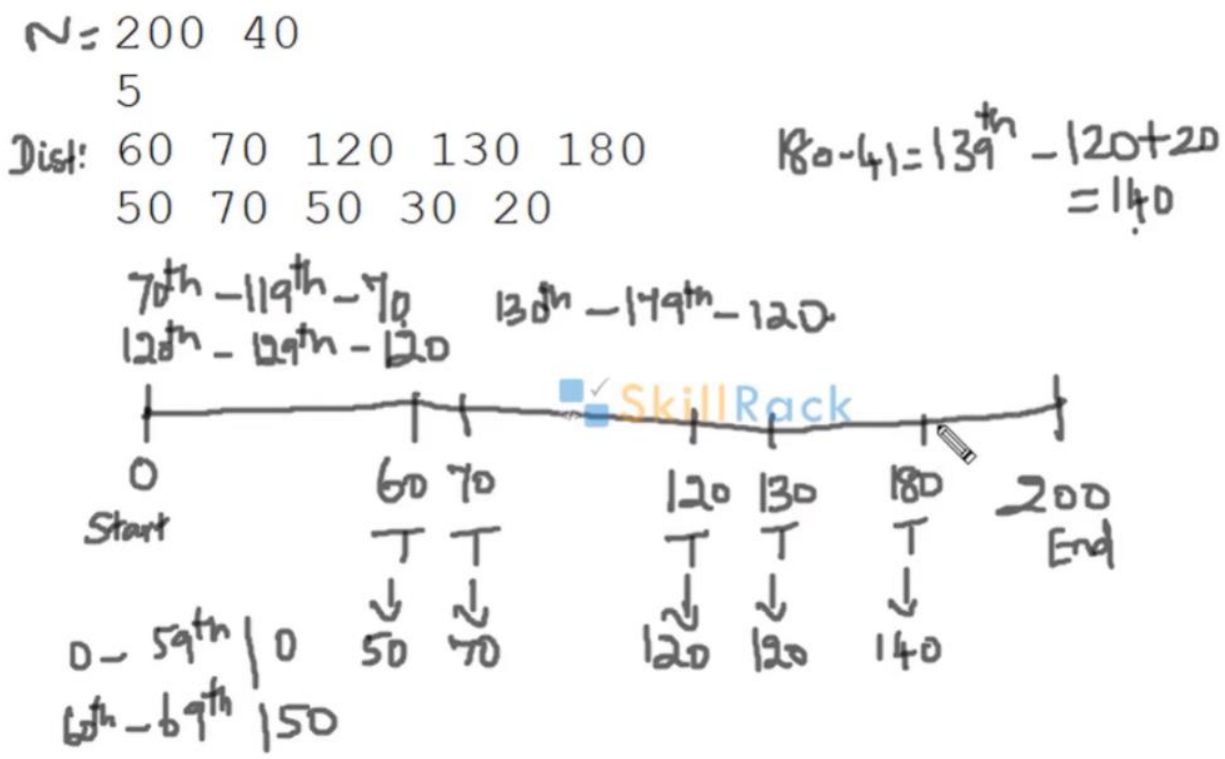
Example Input/Output 2:
Input:
200 40
5
60 70 120 130 180
50 70 50 30 20

Output:
140

Explanation:
There are 5 tollgates present at 60, 70, 120, 130 and 180 kms respectively.
As the tollgates should be separated by more than 40 kms, the maximum revenue will be obtained when the tollgates at **70^{th}** km, **120^{th}** km and **180^{th}** km are selected as the total revenue is 70+50+20 = **Rs.140**.

Max Execution Time Limit: 500 millisecs

Logic:



Code:

```
import java.util.*;

public class tollGateCollection {

    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int N = s.nextInt();
        int D = s.nextInt() + 1; //since we have to check from that km eg:if 50
we have to check from 51th km
        int K = s.nextInt();
        int tollGates[] = new int[K];
        int fee[] = new int[K];
        for (int index = 0; index < K; index++) {
            tollGates[index] = s.nextInt();
        }
        for (int index = 0; index < K; index++) {
            fee[index] = s.nextInt();
        }
        int gateIndex = 0;
        int kilometer[] = new int[N + 1];
```

```
        for (int index = 1; index <= N; index++) {
            if (index == tollGates[gateIndex]) {
                int curr = fee[gateIndex];
                if ((index - D) > 0) {
                    curr += kilometer[index - D];
                }
                kilometer[index] = Math.max(curr, kilometer[index - 1]);
                gateIndex++;
            } else {
                kilometer[index] = kilometer[index - 1];
            }
            if (gateIndex == K) {
                System.out.println(kilometer[index]);
                break;
            }
        }
    }
}
```

Session 2:

Wildcard Pattern Matching

ID:11136 Solved By 672 Users

The program must accept a **text** and a **wildcard pattern** as the input. The program must print "**Matching**" if the wildcard is matched with text. Else the program must print "**Not matching**" as the output.

The wildcard pattern can include the characters '**?**' and '*****'
'**?**' – Matches any single character
'*****' – Matches any sequence of characters (including the empty sequence)

Boundary Condition(s):
1 <= Length of text <= 100
1 <= Length of wildcard pattern <= 50

Input Format:
The first line contains the text.
The second line contains the wildcard pattern.

Output Format:
The first line contains either "Matching" or "Not matching".

Example Input/Output 1:
Input:
abbbbbbbbccbbbbbed
a*b?d

Output:
Matching

Explanation:
The wildcard pattern is "a*****b?d".
'*****' can be replaced by "bbbbbbbbccbbbb".
'**?**' can be replaced by "e".
Hence the output is Matching

Example Input/Output 2:
Input:
abbbbbbbbccbhd
a*b??b?d

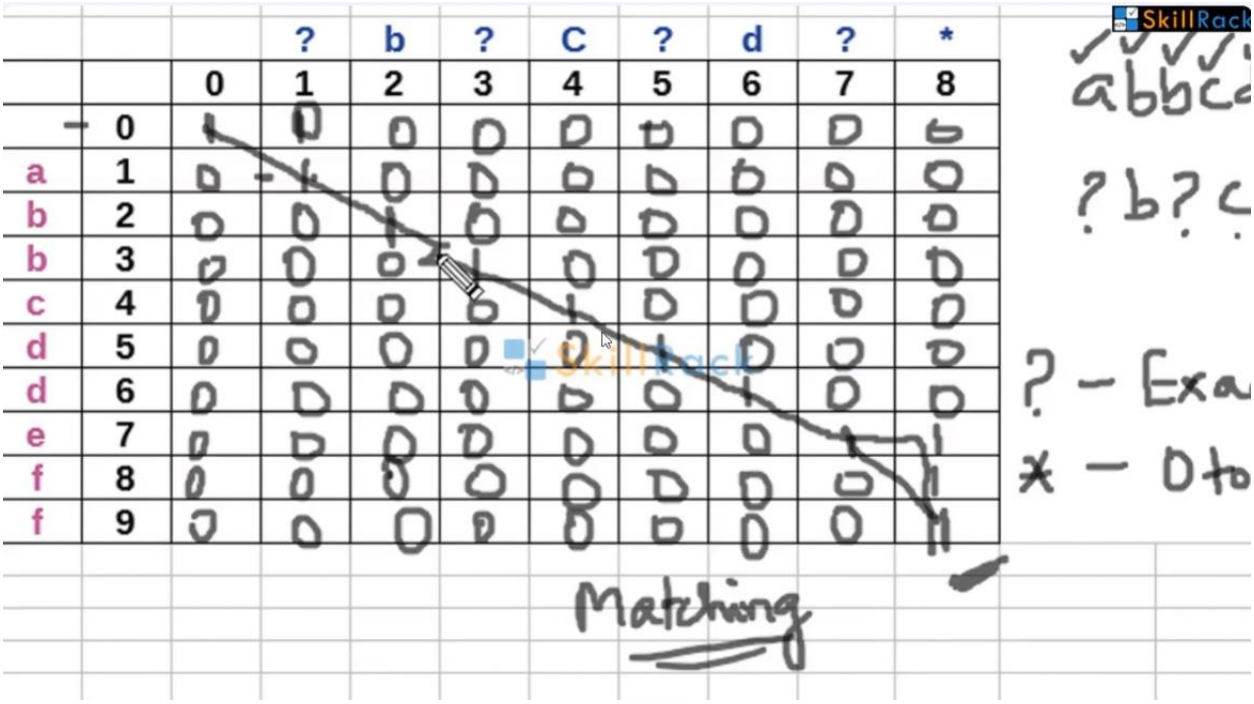
Output:
Matching

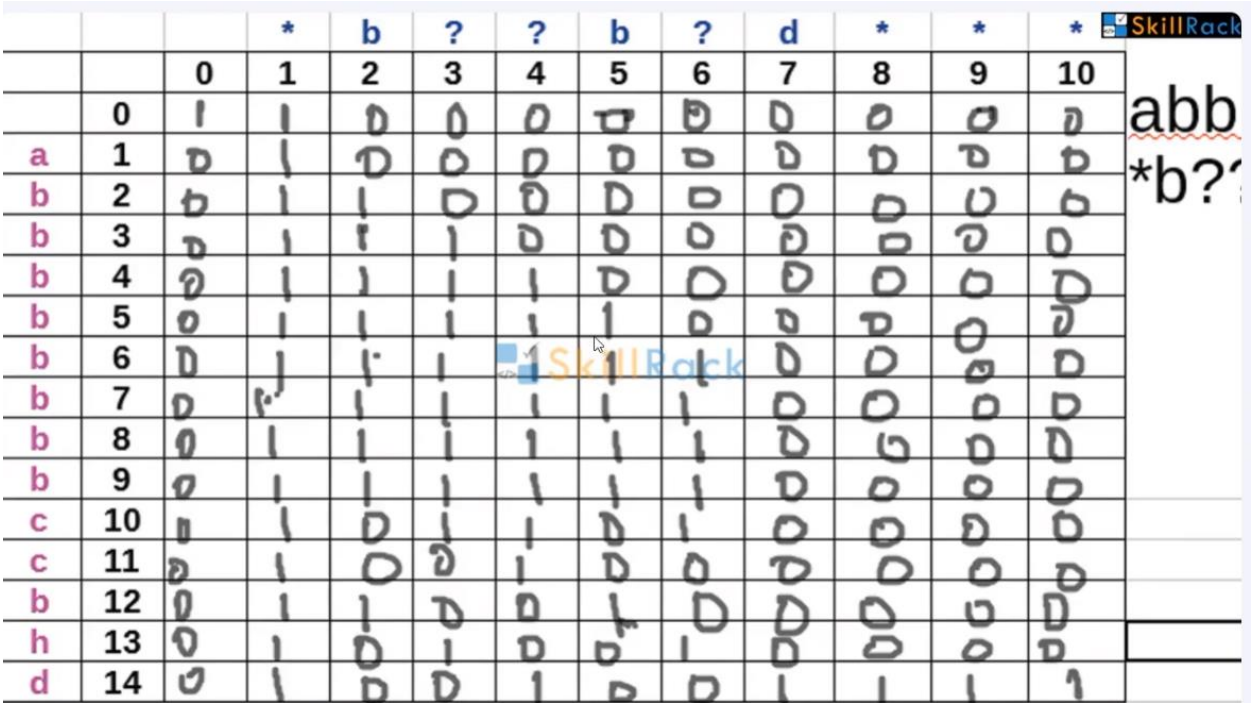
Example Input/Output 3:
Input:
abbbbbbbbccbhd
*c??b?d

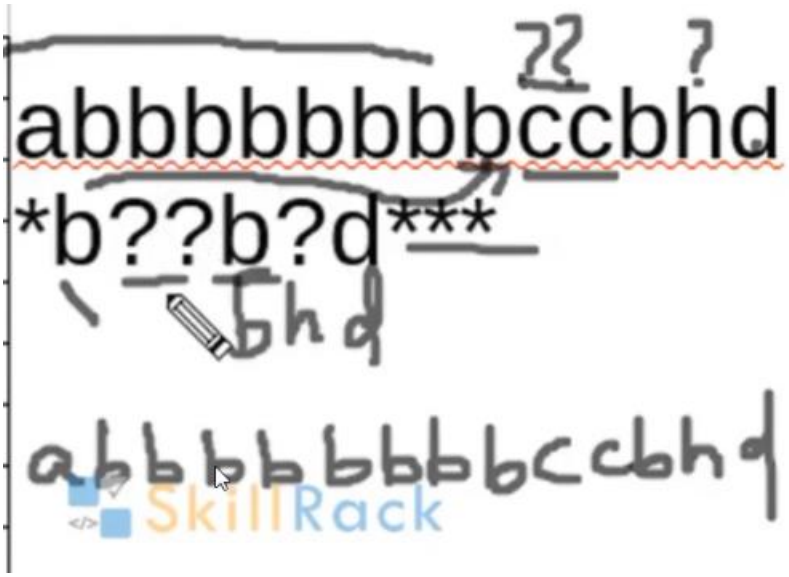
Output:
Not matching

Max Execution Time Limit: 500 millisecs

Logic:


? b ? C ? d ? *
0 1 2 3 4 5 6 7 8
0 a b b c d d e f f
1 2 3 4 5 6 7 8 9
abbc
?b?c
? - Exa
* - 0 to
Matching
SkillRack


* b ? ? b ? d * * *
0 1 2 3 4 5 6 7 8 9 10
0 a b b b b b b b b b c c b h d
1 2 3 4 5 6 7 8 9 10 11 12 13 14
abb
*b?
SkillRack



Code:

```
import java.util.*;

public class wildCardPatternMatching {

    public static void main(String[] args) {
        // Your Code Here
        Scanner in = new Scanner(System.in);
        String text = in.nextLine();
        String pattern = in.nextLine();
        int R = text.length(); //in row we have the text
        int C = pattern.length(); //in column we have pattern
        boolean matrix[][] = new boolean[R + 1][C + 1]; //boolean matrix to find
the matchin //R+1 and C+1 as we consider index from 1 to R,C as in every dynamic
programming
        matrix[0][0] = true; //the first character will always be true
        if (pattern.charAt(0) == '*') { //corner case - when * comes it can match
a series of words
            matrix[0][1] = true;
        }
        for (int row = 1; row <= R; row++) {
            for (int col = 1; col <= C; col++) {
                if (pattern.charAt(col - 1) == '?' || pattern.charAt(col - 1) ==
text.charAt(row - 1)) { //col-1 as index starts from 1 // after || for character
match
                    matrix[row][col] = matrix[row - 1][col - 1]; //value depens
on previous diagonal value
```

```
                } else if (pattern.charAt(col - 1) == '*') {
                    matrix[row][col] = matrix[row - 1][col] || matrix[row][col -
1]; //top cell value || left cell value
                }
            }
        }
        System.out.println(matrix[R][C] ? "Matching" : "Not matching");
    }
}
```

Session 3:

Max Coins - Bottom Row Cannot Pick

ID:11143 Solved By 638 Users

In a room, the **R*C** boxes are arranged as a matrix where each box contains gold coins. A person is allowed to take gold coins from the room with the following conditions.
- He must pick only one box from a row.
- If he has picked a particular box then he cannot pick up the box in the bottom row of the same colomn.
The program must accept the number of gold coins in each box for **N** such rooms. For each room, the program must print the maximum number of gold coins that can be collected by the person as the output.

Boundary Condition(s):
1 <= N <= 100
2 <= R, C <= 100

Input Format:
The first line contains N.
The following lines containing the integers representing the N matrices.

Output Format:
The first N lines, each containing an integer representing the maximum number of gold coins that can be collected by the person.

Example Input/Output 1:
Input:
1
4 4
20 50 100 120
200 100 60 400
60 50 70 900
500 100 90 200

Output:
1720

Explanation:
The maximum number of gold coins that can be collected by the person is **1720**.
1st Row - **120**
2nd Row - **200**
3rd Row - **900**
4th Row - **500**

Example Input/Output 2:

Input:

```
2
5 5
25 98 74 11 89
53 68 36 48 23
4 14 99 48 41
40 22 97 72 1
29 67 61 92 49
2 6
45 10 12 78 66 90
9 1 3 15 12 95
```

Output:

```
395
173
```

Example Input/Output 3:

Input:

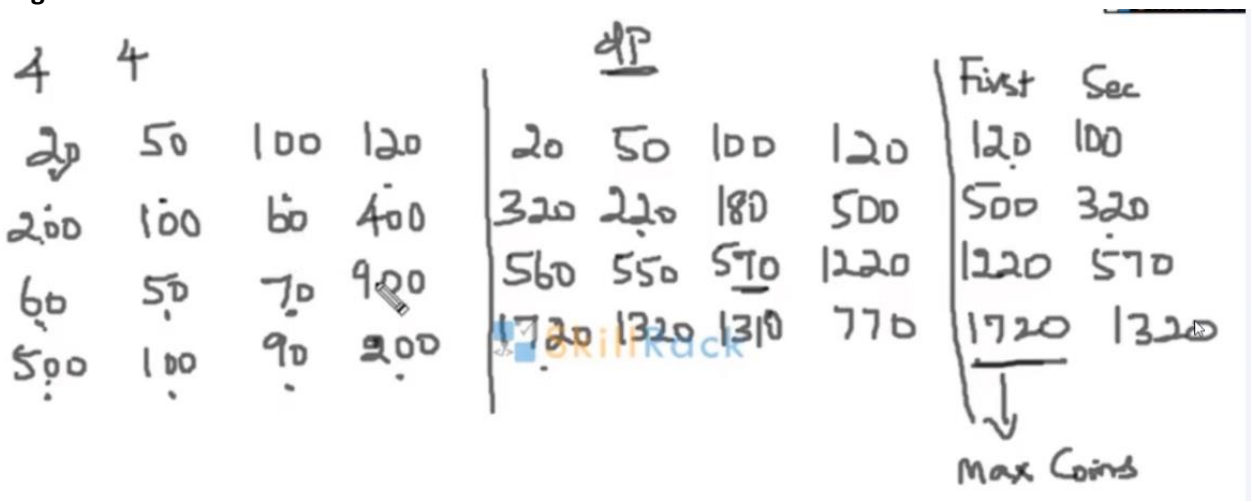
```
3
4 2
30 69
95 7
57 28
80 79
3 4
44 3 16 56
2 88 81 51
18 87 26 59
10 2
55 57
87 32
93 28
26 9
13 87
44 63
84 97
26 63
60 91
41 97
```

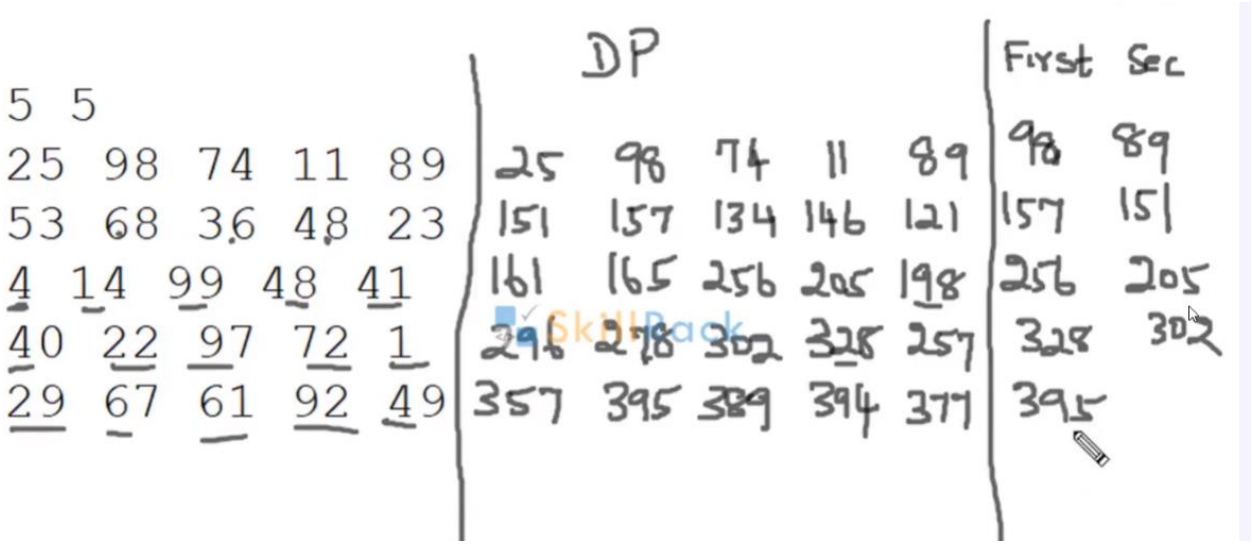
Output:

```
272
224
584
```

Max Execution Time Limit: 500 millisecs

Logic:





Code:

```
import java.util.*;
public class maxCoins {

    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int T = s.nextInt(); //Testcase
        while(T-- > 0){
            int R = s.nextInt(), C = s.nextInt();
            int matrix[][] = new int[R][C];
            for(int row = 0; row < R; row++){
                for(int col = 0; col < C; col++){
                    matrix[row][col] = s.nextInt();
                }
            }
```

```
            int dp[][] = new int[R][C];
            for(int col = 0; col < C; col++){ //filling the first row in dp
matrix
                dp[0][col] = matrix[0][col];
            }

            for(int row = 1; row < R; row++){
                int prevRow[] = Arrays.copyOf(dp[row - 1], C); // C - denotes the
size of the copied matrix //row-1 since index starts from 1
                Arrays.sort(prevRow); //taking a copy of the first row of the
original matrix for sorting and finding the maximum value
                int firstMax = prevRow[C - 1];
                int secondMax = prevRow[C - 2];
                for(int col = 0; col < C; col++){
                    if(dp[row - 1][col] != firstMax){
                        dp[row][col] = firstMax + matrix[row][col];
                    }
                    else{
                        dp[row][col] = secondMax + matrix[row][col];
                    }
                }
            }
            Arrays.sort(dp[R - 1]); //sorting the last row of the dp matrix
            System.out.println(dp[R - 1][C - 1]); //printing the last value of
the dp matrix
        }

    }
}
```