PW SKILLS

Lesson Plan

DP-6

PW SKILLS

Today's Checklist:

- SPOJ Tourist
- Edit distance

SPOJ TOURIST - Tourist

A lazy tourist wants to visit as many interesting locations in a city as possible without going one step further than necessary. Starting from his hotel, located in the north-west corner of city, he intends to take a walk to the south-east corner of the city and then walk back. When walking to the south-east corner, he will only walk east or south, and when walking back to the north-west corner, he will only walk north or west. After studying the city map he realizes that the task is not so simple because some areas are blocked. Therefore he has kindly asked you to write a program to solve his problem.

Given the city map (a 2D grid) where the interesting locations and blocked areas are marked, determine the maximum number of interesting locations he can visit. Locations visited twice are only counted once.

Input
The first line in the input contains the number of test cases (at most 20). Then follow the cases. Each case starts with a line containing two integers, W and H (2 ≤ W , H ≤ 100), the width and the height of the city map. Then follow H lines, each containing a string with W characters with the following meaning:

. Walkable area
* Interesting location (also walkable area)
# Blocked area

You may assume that the upper-left corner (start and end point) and lower-right corner (turning point) are walkable, and that a walkable path of length H + W - 2 exists between them.

Output
For each test case, output a line containing a single integer: the maximum number of interesting locations the lazy tourist can visit.

Example

Input:
2
9 7
*.....
.....**#.
..**...#*
..####*#.
.*.#*.*#.
...#**...
*.....
5 5
.*.*.
*###.
..*
.###*
.*.*.

Output:
7
8

Code:

```
# include <iostream>
# include <cstdio>
# include <cstring>
# include <algorithm>
using namespace std;
# define MAX 104
# define inf 1000000000
int n,m;
char road[MAX][MAX];
int dp[MAX][MAX][MAX];
int cost( int row1 , int col1 , int row2,  int col2)
{
    if(row1==row2 && col1 == col2)
    {
        if(road[row1][col1]=='*')
            return  1 ;
        return 0;
    }
    int ans=0;

 if(road[row1][col1]=='*')
        ans++;
    if(road[row2][col2]=='*')
        ans++;
    return ans;
}
int solve(int row1,int col1,int row2)
{
 int col2=(row1+col1)-(row2);
 if(row1==n-1 and col1==m-1 and row2==n-1 and col2==m-1)
  return 0;
 if(row1>=n or col1>=m or row2>=n or col2>=m)
  return -1*inf;
 if(dp[row1][col1][row2]!=-1)return dp[row1][col1][row2];
 int ch1=-1*inf,ch2=-1*inf,ch3=-1*inf,ch4=-1*inf;
 if(road[row1][col1+1]!='#' and road[row2+1][col2]!='#')
  ch1=cost(row1,col1+1,row2+1,col2)+solve(row1,col1+1,row2+1);
 if(road[row1][col1+1]!='#' and road[row2][col2+1]!='#')
  ch2=cost(row1,col1+1,row2,col2+1)+solve(row1,col1+1,row2);
 if(road[row1+1][col1]!='#' and road[row2][col2+1]!='#')
  ch3=cost(row1+1,col1,row2,col2+1)+solve(row1+1,col1,row2);
 if(road[row1+1][col1]!='#' and road[row2+1][col2]!='#')
  ch4=cost(row1+1,col1,row2+1,col2)+solve(row1+1,col1,row2+1);
 return dp[row1][col1][row2]=max(ch1,max(ch2,max(ch3,ch4)));

}
int main()
{
 //freopen("in","r",stdin);
    int t;
    scanf("%d",&t);
    while(t--)
    {
```

```
  scanf("%d %d",&m,&n);
        memset(dp,-1,sizeof dp);
         int res=0;
         for (int i=0; i<n; i++)
            scanf("%s",road[i]);
         if(road[n-1][m-1]=='#' || road[0][0]=='#'  )
             res=-1*inf;
         if(road[0][0]=='*')
             res++;
         road[0][0]='.';
         if(road[n-1][m-1]=='*')
             res++;
  road[n-1][m-1]='.';
         res+=solve(0,0,0);
         res=max(res,0);
         printf("%d\n",res);
     }
     return 0;
}
```

Leetcode 72. Edit Distance

Given two strings word1 and word2, return the minimum number of operations required to convert word1 to word2.

You have the following three operations permitted on a word:

- Insert a character
- Delete a character
- Replace a character

Example 1:

Input: word1 = "horse", word2 = "ros"
Output: 3

Explanation:
horse -> rorse (replace 'h' with 'r')
rorse -> rose (remove 'r')
rose -> ros (remove 'e')

Example 2:

Input: word1 = "intention", word2 = "execution"
Output: 5

Explanation:
intention -> inention (remove 't')
inention -> enention (replace 'i' with 'e')
enention -> exention (replace 'n' with 'x')
exention -> exection (replace 'n' with 'c')
exection -> execution (insert 'u')

Constraints:

- 0 <= word1.length, word2.length <= 500
- word1 and word2 consist of lowercase English letters.

Approach:

To apply DP, we define the state dp[i][j] to be the minimum number of operations to convert word1[0..i) to word2[0..j).

For the base case, that is, to convert a string to an empty string, the mininum number of operations (deletions) is just the length of the string. So we have dp[i][0] = i and dp[0][j] = j.

For the general case to convert word1[0..i) to word2[0..j), we break this problem down into sub-problems.

Suppose we have already known how to convert word1[0..i - 1) to word2[0..j - 1) (dp[i - 1][j - 1]), if word1[i - 1] == word2[j - 1], then no more operation is needed and dp[i][j] = dp[i - 1][j - 1].

If word1[i - 1] != word2[j - 1], we need to consider three cases.

1. Replace word1[i - 1] by word2[j - 1] (dp[i][j] = dp[i - 1][j - 1] + 1);
2. If word1[0..i - 1) = word2[0..j) then delete word1[i - 1] (dp[i][j] = dp[i - 1][j] + 1);
3. If word1[0..i) + word2[j - 1] = word2[0..j) then insert word2[j - 1] to word1[0..i) (dp[i][j] = dp[i][j - 1] + 1).

So when word1[i - 1] != word2[j - 1], dp[i][j] will just be the minimum of the above three cases.

Code:

```
class Solution {
public:
    int minDistance(string word1, string word2) {
        int m = word1.size(), n = word2.size();
        vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
        for (int i = 1; i ⩽ m; i++) {
            dp[i][0] = i;
        }
        for (int j = 1; j ⩽ n; j++) {
            dp[0][j] = j;
        }
        for (int i = 1; i ⩽ m; i++) {
            for (int j = 1; j ⩽ n; j++) {
                if (word1[i - 1] == word2[j - 1]) {
                    dp[i][j] = dp[i - 1][j - 1];
                } else {

  dp[i][j] = min(dp[i - 1][j - 1], min(dp[i][j - 1], dp[i - 1]
[j])) + 1;
                }
            }
        }
        return dp[m][n];
    }
};
```