2D- DYNAMIC PROGRAMMING

—

INGENUITY WORKSHOP#5

Sai Surya
Naga Jaswanth

PROBLEMS

1. UNIQUE PATHS (https://leetcode.com/problems/unique-paths/description)
2. UNIQUE PATHS WITH OBSTACLES (https://leetcode.com/problems/unique-paths-ii/description/)
3. BOOK SHOP -CSES (https://cses.fi/problemset/task/1158)
4. KNAPSACK VARIANT (SPECIAL MENTION)

   (https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=1071)

What is 2D-DP ?

Simply put ,its just Dynamic Programming on 2 variables .

Ex : check the dp relation in fibonacci series :
fib(n)=fib(n-1)+fib(n-2)

Which depends only on single variable n , hence it is a 1D - dp .

A relation like :

dp[i][j]=dp[i-1][j]+dp[i][j-1]

Depends on two variables i and j , hence it is a 2D-dp.

What is 2D-DP ?

$$dp[i][j] = 1 + min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1])$$

3D - DP

dp[m][n][o]=min(dp[m-1][n][o],min(dp[m][n-1][o],dp[m][n][o-1]))

Generalization :
‘n’D-DP is simply DP on n variables

Problem -1 : UNIQUE PATHS

62. Unique Paths

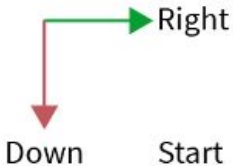
Medium | Topics | Companies

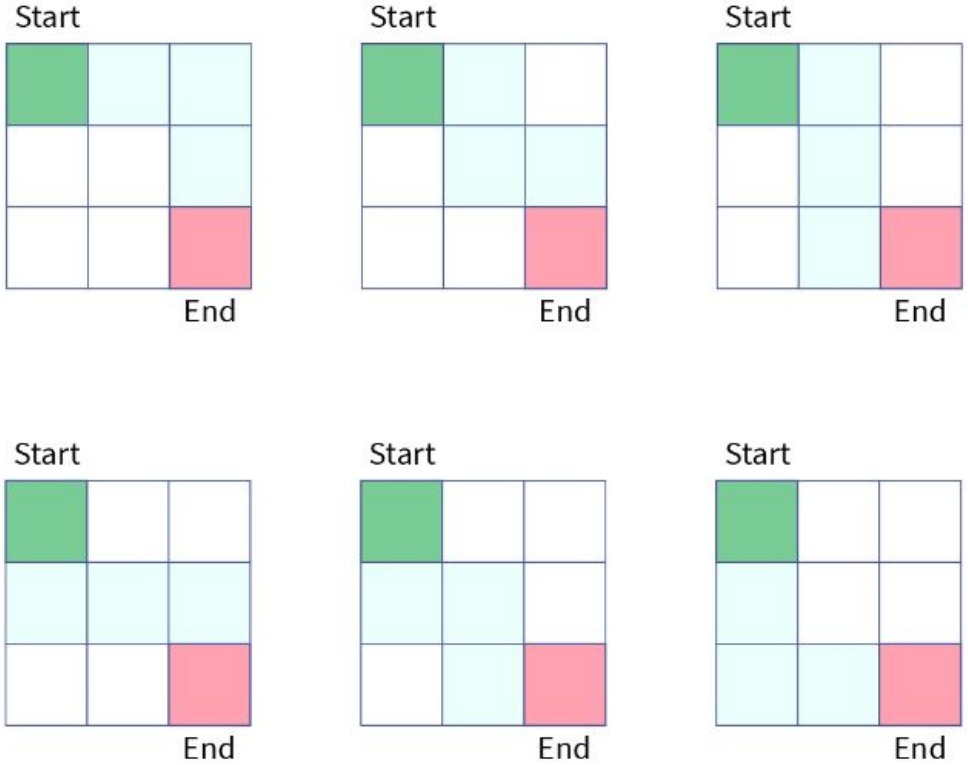
There is a robot on an `m x n` grid. The robot is initially located at the **top-left corner** (i.e., `grid[0][0]`). The robot tries to move to the **bottom-right corner** (i.e., `grid[m - 1][n - 1]`). The robot can only move either down or right at any point in time.

Given the two integers `m` and `n`, return *the number of possible unique paths that the robot can take to reach the bottom-right corner*.

The test cases are generated so that the answer will be less than or equal to `2 * 10^9`.

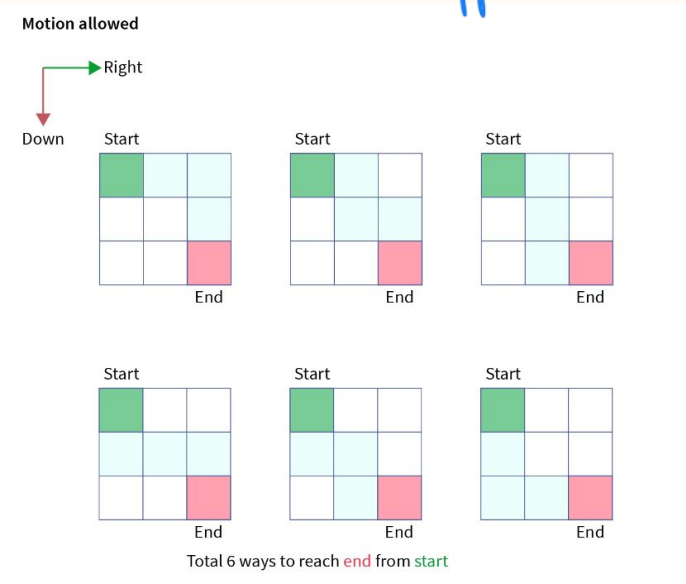
Motion allowed





Total 6 ways to reach end from start

Recursion Approach:



$$countWays(i,j) = countWays(i-1,j) + countWays(i,j-1)$$

```
f(i,j) {
if (i==0 && j==0) return 1
if (i<0 || j<0) return 0
top = f(i-1, j);
left = f(i, j-1);
return top + left;
}
```

T.C → $(2^{N \times m})$

S.C → O(path length)

Let convert it into Dynamic Programming:

$dp[m][n] = \{-1\}$

↳ to store the values

```
f(i,j) {
    if (i==0 && j==0) return 1
    if (i<0 || j<0)  return 0
    if (dp[i][j] != -1)  return dp[i][j]
    top = f(i-1, j)
    left = f(i, j-1)
    return  dp[i][j] = top + left
}
```

T.C → $O(N \times M)$

S.C → $O((N-1)+(M-1)) + O(N \times M)$

The space can be further optimized by eliminating the recursive stack space in Tabulation approach (bottom up)

```
dp[m,n]
for(int i=0 to m-1){
    for(int j=0 to n-1){
        if (i==0 && j==0) dp[0][0]=1
        dp[i][j]= dp[i-1][j] + dp[i][j-1]
        }
    }
```

T.C → O(N×m)

S.C → O(N×m)

Trick to identify a DP problem :

If there is a line something like “out of all possible ways” then its a Recursion application.

We have to compute all possible ways for the given problem , and either count the ways or find minimum or maximum .

And recursion can be modified to DP , to optimize it .

DP = recursion + memoization .

Steps to conquer a DP application :

Try to represent the problem in terms of index.

Do all possible ways on that index , according to the problem statement.

If the question says :

- Count all possible ways : find the sum of all possible ways.
- Minimum : find the minimum of all possible ways
- Maximum : find the maximum of all possible ways

Now in this recursion , use either memoization (top-down) or tabulation (bottom up) approach to convert into DP .

Problem -2 : UNIQUE PATHS-2

63. Unique Paths II

Medium | Topics | Companies | Hint

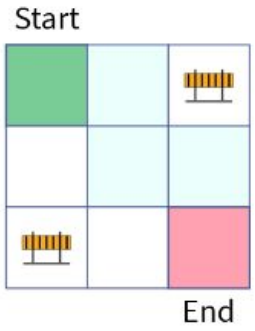
You are given an `m x n` integer array `grid`. There is a robot initially located at the **top-left corner** (i.e., `grid[0][0]`). The robot tries to move to the **bottom-right corner** (i.e., `grid[m - 1][n - 1]`). The robot can only move either down or right at any point in time.

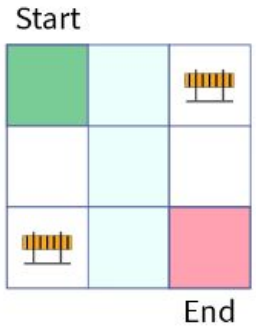
An obstacle and space are marked as `1` or `0` respectively in `grid`. A path that the robot takes cannot include **any** square that is an obstacle.

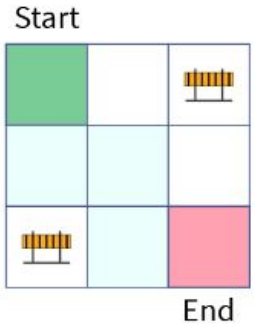
Return *the number of possible unique paths that the robot can take to reach the bottom-right corner*.

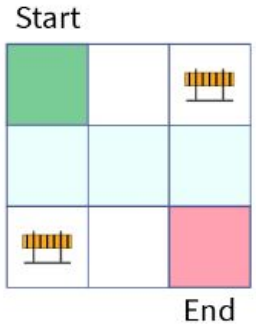
The testcases are generated so that the answer will be less than or equal to $2 * 10^9$.

Motion allowed



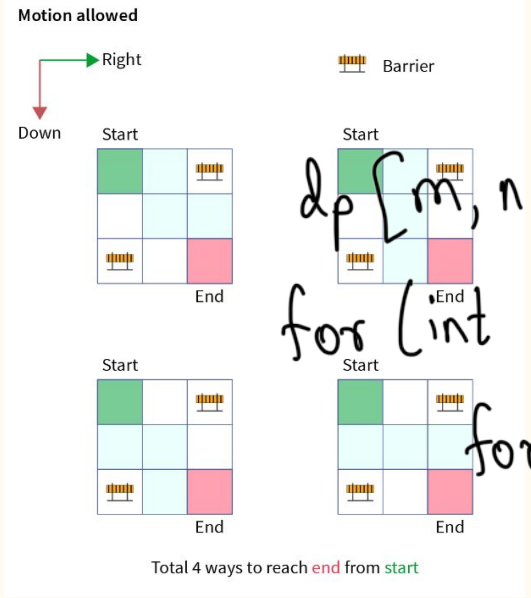






Total 4 ways to reach end from start

if barrier is found, we skip it.
i.e that path cannot be taken.




```
dp[m, n]

for (int i=0 to m-1){
    for(int j=0 to n-1){
        if (i==0 && j==0) dp[0][0] =1
        if (dp[i][j] = -1) continue; //skip if barriers found.
        dp[i][j] = dp[i-1][j] + dp[i][j-1]
    }
}
```

T.C → O(N×m)

S.C → O(N×m)

Another variant : Each cell is given a cost

Allowed motion in the matrix

$$minCost(i,j) = min(minCost(i-1,j) + minCost(i,j-1)) + cost[i][j]$$

Start

1	7	9	2
8	6	3	2
1	6	7	8
2	9	8	2

End

Min-cost path is traced cost = 29

Book Shop

TASK | STATISTICS

Time limit: 1.00 s **Memory limit:** 512 MB

You are in a book shop which sells n different books. You know the price and number of pages of each book.

You have decided that the total price of your purchases will be at most x. What is the maximum number of pages you can buy? You can buy each book at most once.

Input

The first input line contains two integers n and x: the number of books and the maximum total price.

The next line contains n integers $h_1, h_2, \ldots, h_n$: the price of each book.

The last line contains n integers $s_1, s_2, \ldots, s_n$: the number of pages of each book.

Output

Print one integer: the maximum number of pages.

Constraints

- $1 \leq n \leq 1000$
- $1 \leq x \leq 10^5$
- $1 \leq h_i, s_i \leq 1000$

PROBLEM-3 BOOK SHOP

example: n=4, x=10

prices:	4	8	5	3
pages:	5	12	8	1

optimal o/p = books 1 & 3

5+8 = 13 ← max. pages

I) Greedy

choosing books with max. pages until we can't buy any further

⇒ book 2 and ans = 12

but optimal is 13

II) We need to compute all possible ways of buying books and find the maximum possible pages.

⇒ Recursion ⇒ DP.

Recursive approach:

I) Convert into indexes: (i, weight)

II) Try all possible stuff: 1. Buy
2. don't Buy

III) Find the maximum of all.

```
f(i, w){
    if (i == 0) {
        if (price[i] ≤ w) return price[0]
        return 0
    }
    no buy = 0 + f(i-1, w)
    buy = INT_MIN
    if (price(i) ≤ w)  buy = price(i) + f(i-1, w - cost(i))
    return max(buy, no buy)
}
```

10130 SuperSale

SPECIAL MENTION

There is a SuperSale in a SuperHiperMarket. Every person can take only one object of each kind, i.e. one TV, one carrot, but for extra low price. We are going with a whole family to that SuperHiperMarket. Every person can take as many objects, as he/she can carry out from the SuperSale. We have given list of objects with prices and their weight. We also know, what is the maximum weight that every person can stand. What is the maximal value of objects we can buy at SuperSale?

Input

The input consists of T test cases. The number of them ($1 \leq T \leq 1000$) is given on the first line of the input file. Each test case begins with a line containing a single integer number N that indicates the number of objects ($1 \leq N \leq 1000$). Then follows N lines, each containing two integers: P and W. The first integer ($1 \leq P \leq 100$) corresponds to the price of object. The second integer ($1 \leq W \leq 30$) corresponds to the weight of object. Next line contains one integer ($1 \leq G \leq 100$) its the number of people in our group. Next G lines contains maximal weight ($1 \leq MW \leq 30$) that can stand this i-th person from our family ($1 \leq i \leq G$).

Output

For every test case your program has to determine one integer. Print out the maximal value of goods which we can buy with that family.

Idea for SuperSale:
In case of book shop problem, or in general knapsack problems, we can see that building an entire grid seems wasteful as only last cell is queried, but in this problem, multiple queries are passed. So instead of one person, there are G people each with their own knapsack limit. So each person’s optimum will be obtained from final row I.e. dp[n-1][g[i]]