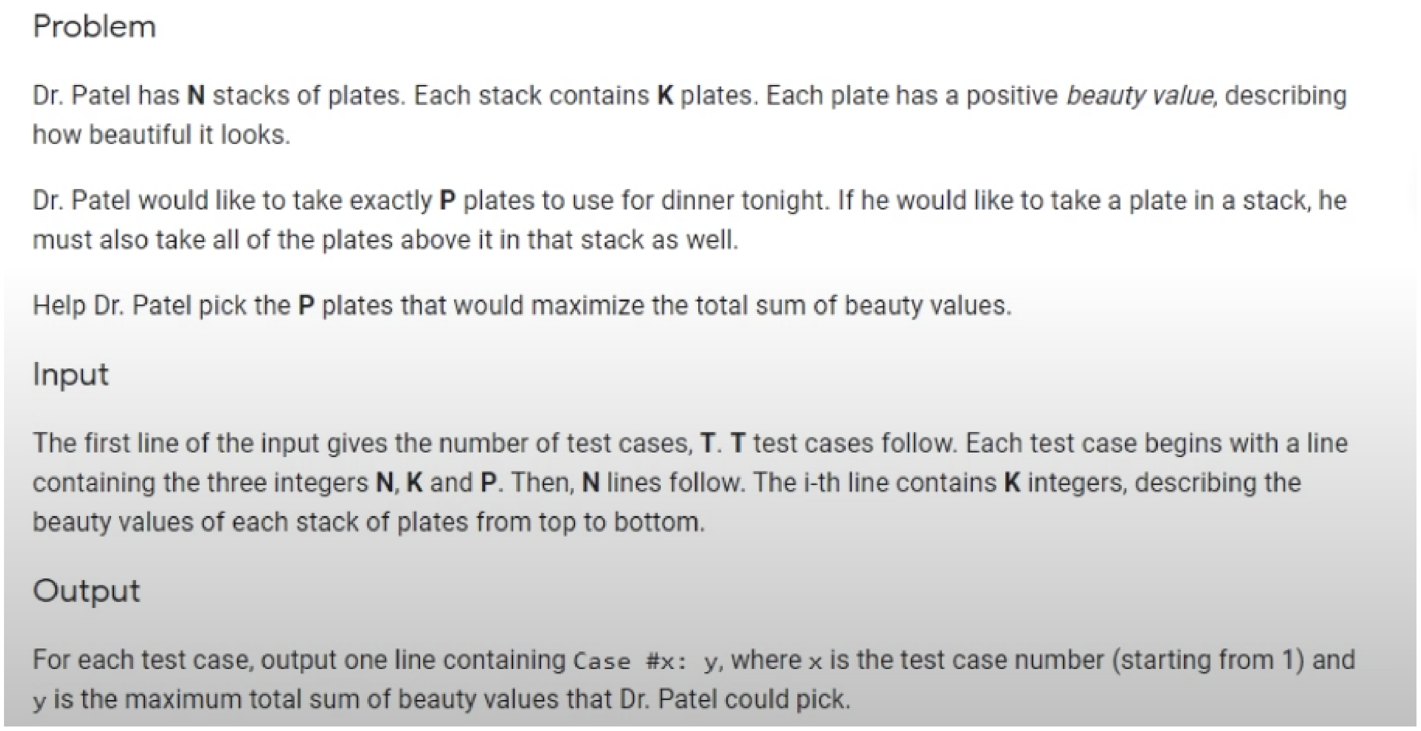

Problem

Dr. Patel has **N** stacks of plates. Each stack contains **K** plates. Each plate has a positive *beauty value*, describing how beautiful it looks.

Dr. Patel would like to take exactly **P** plates to use for dinner tonight. If he would like to take a plate in a stack, he must also take all of the plates above it in that stack as well.

Help Dr. Patel pick the **P** plates that would maximize the total sum of beauty values.

Input

The first line of the input gives the number of test cases, **T**. **T** test cases follow. Each test case begins with a line containing the three integers **N**, **K** and **P**. Then, **N** lines follow. The i-th line contains **K** integers, describing the beauty values of each stack of plates from top to bottom.

Output

For each test case, output one line containing `Case #x: y`, where `x` is the test case number (starting from 1) and `y` is the maximum total sum of beauty values that Dr. Patel could pick.

Algorithm Plan

This problem can be solved by using **dynamic programming**. Let's understand why and arrive at the definition of the subproblems that will allow us to solve this problem.

The two aspects of dynamic programming include:

1. Optimal substructure: If the optimal solution to a problem P, of size n, can be calculated by looking at the optimal solutions to subproblems [p1,p2,…] with size less than n, then this problem P is considered to have an optimal substructure. Here, we can see the optimal solutions to the subproblems defined by the stack index and the number of plates picked up so far can help us arrive at the final answer.

2. Memoization (overlapping subproblems): Memoization dictates the storing of previously calculated results of the subproblem and uses the stored result for the same subproblem. This removes the extra effort to calculate again and again for the same problem. Representing this problem as a decision tree, we can see that memoizing the sub branches can help eliminate redundancies.

Before we specify our subproblems, let's explain how we will think of our dynamic problem solutions. Like most dynamic programming problems, we will choose to use an array (in this case a 2D array or matrix as there are two variable states defining our subproblems) denoted as dp. Each entry in our subproblem $dp[i][l]$ will represent the optimal solution to our subproblem.

Secondly, note that the approach being considered here is an example of bottom-up dynamic programming. We are computing the solutions to the subproblems in a strict order defined by stack index and # of plates.

<u>Let's define the subproblems</u>.

$dp[i][l] \ = \ the\ maximum\ beauty\ sum\ of\ maximum\ l\ plates\ picked\ until\ the\ ith\ stack\ from\ the\ left.$

Let's define the recurrence:

Base cases(s): Immediately, we know that if we use 0 stacks from the left and pick 0 plates, the maximum beauty sum is 0. Hence, $dp[0][0] \ = 0.$

Initialisation: Since this is a maximization problem, we set the values of our dp array to be -∞.
The dp array itself will be a 51 x 1501 matrix as dictated by the limits of *n* and *k* in our problem.

Recursive case:
$dp[i \ + \ 1][l \ + j\,] \ = \ max(dp[i \ + \ 1][l \ + j\,], \ dp[i][l] \ + \ sum\ of\ beauty\ values\ in\ current\ stack$
$when\ picking\ up\ j\ more\ plates\ )$.

Code Solution (Written in C++)

C/C++

```
#include <bits/stdc++.h>
using namespace std;

#define ll long long
#define ar array

int n, k, p, a[50][30];
int dp[51][1501];

void solve(){
      cin >> n >> k >> p;
      memset(dp, 0xc0, sizeof(dp))
      dp[0][0] = 0;
      for (int i = 0; i < n; ++i){
            memcpy(dp[i+1], dp[i], sizeof(dp[0]))
            for (int j = 0; b = 0; j < k; ++j){
                  cin >> a[i][j];
                  b += a[i][j];
                  for (int l = 0; l + j + 1 <= p; ++l){
                        dp[i+1][l + j + 1] = max(dp[i+1][l + j +
                        1], dp[i][l] + b)
                  }
            }
      }
      cout << dp[n][p] << "\n"
}
int main(){
      ios::sync_with_stdio(0)
      cin.tie(0);
      int t, i = 1;
      cin >> t;
      while(t--){
            count << "Case #" << i << ": ";
            solve();
```

```
                ++i;
        }
}
```