**1. Flood fill Algorithm**

Given a 2D screen, location of a pixel in the screen ie(x,y) and a color(K), your task is to replace color of the given pixel and all adjacent(excluding diagonally adjacent) same colored pixels with the given color K.

Example:

{{1, 1, 1, 1, 1, 1, 1, 1},  
{1, 1, 1, 1, 1, 1, 0, 0},  
{1, 0, 0, 1, 1, 0, 1, 1},  
{1, **2, 2, 2, 2,** 0, 1, 0},  
{1, 1, 1,**2, 2**, 0, 1, 0},  
{1, 1, 1, **2, 2, 2, 2**, 0},  
{1, 1, 1, 1, 1, **2**, 1, 1},  
{1, 1, 1, 1, 1, **2, 2,** 1},  
 };

 x=4, y=4, color=3

{{1, 1, 1, 1, 1, 1, 1, 1},  
{1, 1, 1, 1, 1, 1, 0, 0},  
{1, 0, 0, 1, 1, 0, 1, 1},   
{1, **3, 3, 3, 3**, 0, 1, 0},  
{1, 1, 1, **3, 3**, 0, 1, 0},  
{1, 1, 1, **3, 3, 3, 3,** 0},  
{1, 1, 1, 1, 1, **3**, 1, 1},  
{1, 1, 1, 1, 1, **3, 3**, 1}, };

**Note:**Use zero based indexing.

**Input:**  
The first line of input contains an integer T denoting the no of test cases. Then T test cases follow. The first line of each test case contains Two integers N and M denoting the size of the matrix. Then in the next line are N\*M space separated values of the matrix. Then in the next line are three values x, y and K.

**Output:**  
For each test case print the space separated values of the new matrix.

**Constraints:**  
1 <= T <= 100  
1 <= M[][] <= 100

**Example:  
Input:**  
3  
3 4  
0 1 1 0 1 1 1 1 0 1 2 3  
0 1 5  
2 2  
1 1 1 1  
0 1 8  
4 4   
1 2 3 4 1 2 3 4 1 2 3 4 1 3 2 4  
0 2 9

**Output:**  
0 5 5 0 5 5 5 5 0 5 2 3  
8 8 8 8  
1 2 9 4 1 2 9 4 1 2 9 4 1 3 2 4

CODE:

using namespace std;

void paint(int n,int m,int arr[100][100],int x,int y,int prevc,int newc)

{

if(x>=n || y>=m || x<0 || y<0)

return;

if(arr[x][y]==newc)

return;

if(arr[x][y]!=prevc)

return;

arr[x][y]=newc;

paint(n,m,arr,x+1,y,prevc,newc);

paint(n,m,arr,x-1,y,prevc,newc);

paint(n,m,arr,x,y+1,prevc,newc);

paint(n,m,arr,x,y-1,prevc,newc);

return;

}

int main()

{

int t;

cin>>t;

while(t--)

{

int n,m;

cin>>n>>m;

int arr[100][100];

for(int i=0;i<n;i++)

for(int j=0;j<m;j++)

cin>>arr[i][j];

int x,y,k;

cin>>x>>y>>k;

// for(int i=0;i<n;i++)

// {

// for(int j=0;j<m;j++)

// cout<<arr[i][j]<<" ";

// cout<<endl;

// }

// cout<<"------------------------------------------------"<<endl;

int prevc=arr[x][y];

int newc=k;

paint(n,m,arr,x,y,prevc,newc);

for(int i=0;i<n;i++)

for(int j=0;j<m;j++)

cout<<arr[i][j]<<" ";

cout<<endl;

}

return 0;

}

**2. Number of paths**

The problem is to count all the possible paths from top left to bottom right of a **MxN** matrix with the constraints that from each cell you can either move to **right** or **down**.

**Input:**  
The first line of input contains an integer **T,** denoting the number of test cases. The first line of each test case is **M** and **N**, M is number of rows and N is number of columns.

**Output:**  
For each test case, print the number of paths.

**Constraints:**  
1 ≤ T ≤ 30  
1 ≤ M,N ≤ 10

**Example:**  
**Input**  
2  
3 3  
2 8

**Output**  
6  
8

CODE:

using namespace std;

int count(int i,int j,int n,int m)

{

if(i==n-1 || j==m-1)

return 1;

if(i>=n || j>=m)

return 0;

return (count(i+1,j,n,m) + count(i,j+1,n,m));

}

int main()

{

int t;

cin>>t;

while(t--)

{

int n,m;

cin>>n>>m;

cout<<count(0,0,n,m)<<endl;

}

return 0;

}

**3. Combination Sum - Part 2**

Given an array of integers **A[]** of size **N** and a sum **B**, find all unique combinations in A where the sum is equal to B. Each number in A may only be used once in the combination.

**Note:**  
   All numbers will be positive integers.  
   Elements in a combination (a1, a2, … , ak) must be in non-descending order. (ie, a1 ≤ a2 ≤ … ≤ ak).  
   The combinations themselves must be sorted in ascending order.

**Example 1:**

**Input:**

N = 7

A = {9, 1, 2, 7, 6, 1, 5}

B = 8

**Output:** (1 1 6)(1 2 5)(1 7)(2 6)

**Explaination:** These are the only possible

combinations for getting sum 8.

**Example 2:**

**Input:**

N = 5

A = {8, 1, 8, 6, 8}

B = 12

**Output:** Empty

**Explainatioin:** We cannot obtain sum 12

from the given elements.

**Your Task:**  
You don't need to read input r print anything. Your task is to complete the function combinationSum() which takes the array **A[]**, the length of the array **N** and the sum **B** as input parameters and returns all the combinations, otherwise, if there is no combination present it returns an empty list.

**Expected Time Complexity:** O(2N)  
**Expected Auxiliary Space:** O(2N)

**Constraints:**  
0 < N < 13  
0 <= A[i] <= 9  
0 < B <= 30

CODE:

// Initial Template for C++

#include <bits/stdc++.h>

using namespace std;

// } Driver Code Ends

// User function Template for C++

class Solution{

public:

void combination(vector<int>& candidates, int target, vector<int> curr, vector<vector<int>>& result, int idx) {

if(target < 0)

return;

if(target == 0) {

result.push\_back(curr);

return;

}

for(int i = idx; i<candidates.size(); i++) {

if(i > idx && candidates[i] == candidates[i-1])

continue; //ignore duplicate elements

curr.push\_back(candidates[i]);

combination(candidates, target-candidates[i], curr, result, i+1);

curr.pop\_back();

}

}

vector<vector<int>> combinationSum(vector<int> &candidates, int N, int B){

vector<vector<int>> result;

vector<int> curr;

sort(candidates.begin(), candidates.end()); //because we will ignore duplicate elements

combination(candidates, B, curr, result, 0);

return result;

}

};

// { Driver Code Starts.

int main()

{

int t;

cin>>t;

while(t--)

{

int N, x, B;

cin>>N;

vector<int> A;

for(int i = 0;i < N;i++)

{

cin>>x;

A.push\_back(x);

}

cin>>B;

Solution ob;

vector<vector<int>> result;

result = ob.combinationSum(A, N, B);

if(result.size() == 0)

cout<<"Empty"<<endl;

else{

for(int i = 0;i < result.size(); i++){

cout<<"(";

for(int j = 0; j < result[i].size();j++){

cout<<result[i][j];

if(j < result[i].size() - 1)

cout<<" ";

}

cout<<")";

}

cout<<endl;

}

}

return 0;

}

**4. Special Keyboard**

Imagine you have a special keyboard with the following keys:

Key 1:  Prints 'A' on screen  
Key 2: (Ctrl-A): Select screen  
Key 3: (Ctrl-C): Copy selection to buffer  
Key 4: (Ctrl-V): Print buffer on screen appending it after what has already been printed.

Find maximum numbers of A's that can be produced by pressing keys on the special keyboard N times.

**Example 1:**

**Input:** N = 3

**Output:** 3

**Explaination:** Press key 1 three times.

**Example 2:**

**Input:** N = 7

**Output:** 9

**Explaination:** The best key sequence is

key 1, key 1, key 1, key 2, key 3,

key4, key 4.

**Your Task:**  
You do not need to read input or print anything. Your task is to complete the function **optimalKeys()** which takes N as input parameter and returns the maximum number of A's that can be on the screen after performing N operations.

**Expected Time Complexity:** O(N2)  
**Expected Auxiliary Space:** O(N)

**Constraints:**  
1 < N < 75

CODE:

#include <bits/stdc++.h>

using namespace std;

// } Driver Code Ends

// User function Template for C++

class Solution{

public:

unsigned long long int optimalKeys(int N){

if (N <= 6)

return N;

// An array to store result of subproblems

int screen[N];

int b; // To pick a breakpoint

// Initializing the optimal lengths array for uptil 6 input

// strokes.

int n;

for (n = 1; n <= 6; n++)

screen[n - 1] = n;

// Solve all subproblems in bottom manner

for (n = 7; n <= N; n++) {

// Initialize length of optimal string for n keystrokes

screen[n - 1] = 0;

// For any keystroke n, we need to loop from n-3 keystrokes

// back to 1 keystroke to find a breakpoint 'b' after which we

// will have ctrl-a, ctrl-c and then only ctrl-v all the way.

for (b = n - 3; b >= 1; b--) {

// if the breakpoint is at b'th keystroke then

// the optimal string would have length

// (n-b-1)\*screen[b-1];

int curr = (n - b - 1) \* screen[b - 1];

if (curr > screen[n - 1])

screen[n - 1] = curr;

}

}

return screen[N - 1];

}

};

// { Driver Code Starts.

int main(){

int t;

cin>>t;

while(t--){

int N;

cin>>N;

Solution ob;

cout<<ob.optimalKeys(N)<<"\n";

}

return 0;

}

**5. Josephus problem**

Given the total number of persons **n** and a number **k** which indicates that**k-1**persons are skipped and **kth** person is killed in circle in a fixed direction.â€‹  
The task is to choose the **safe place in the circle** so that when you perform these operations starting from **1stplace** in the circle, you are the last one remaining and survive.

**Example 1:**

**Input:**

n = 3 k = 2

**Output:** 3

**Explanation:** There are 3 persons so

skipping 1 person i.e 1st person 2nd

person will be killed. Thus the safe

position is 3.

**Example 2:**

**Input:**

n = 5 k = 3

**Output:** 4

**Explanation:** There are 5 persons so

skipping 2 person i.e 3rd person will

be killed. Thus the safe position is 4.

**Your Task:**  
You don't need to read input or print anything. You are required to complete the **function josephus ()** that takes**two parameters n and k** and **returns**an integer denoting**safe position**.

**Expected Time Complexity:**O(N).  
**Expected Auxiliary Space:**O(N).

**Constraints:**  
1 <= k, n <= 20

CODE:

#include <bits/stdc++.h>

using namespace std;

int josephus(int n, int k);

int main() {

int t;

cin>>t;//testcases

while(t--)

{

int n,k;

cin>>n>>k;//taking input n and k

//calling josephus() function

cout<<josephus(n,k)<<endl;

}

return 0;

}// } Driver Code Ends

/\*You are required to complete this method \*/

int josephus(int n, int k)

{

if(n==1)

return 1;

return ((josephus(n-1,k)+k-1)%n+1);

//Your code here

}