Dynamic Programming

Not Easy But Build Logic

i) Tabulation : Bottom up (from leaf)

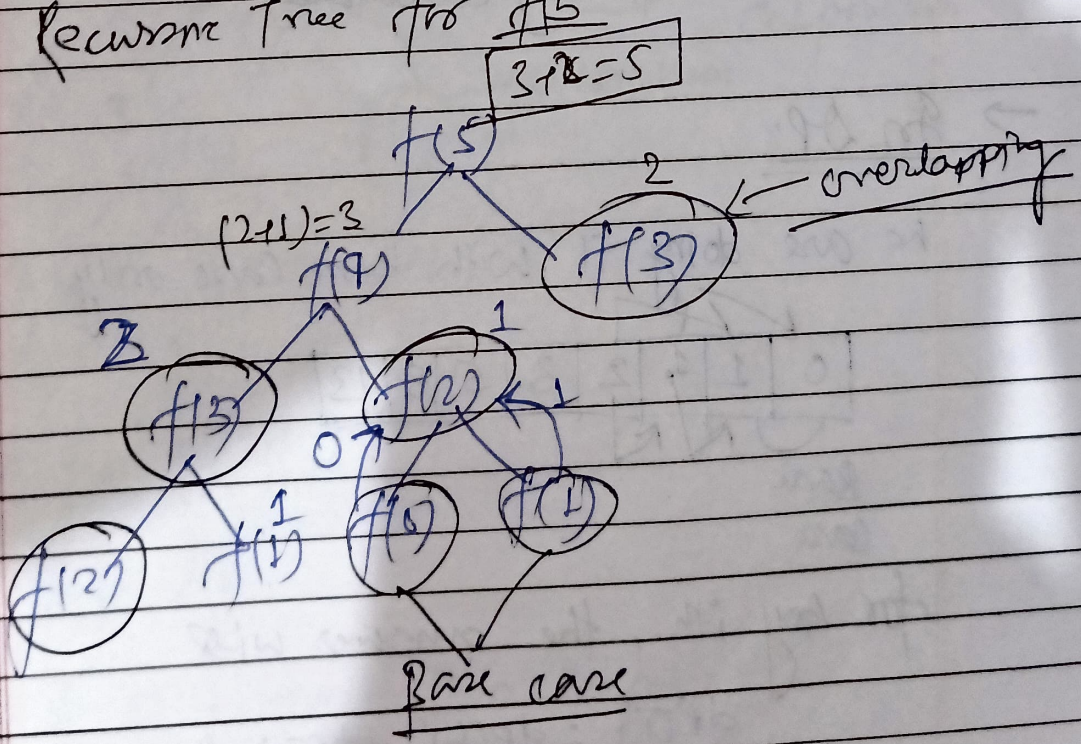
ii) Memoization :- Top-Down

1. **Fibonacci No**

0 1 1 2 3 5 8 13 21 .....

$f(n) = f(n-1) + f(n-2)$ (Recursive Relation)

Recursive Tree for fib 5

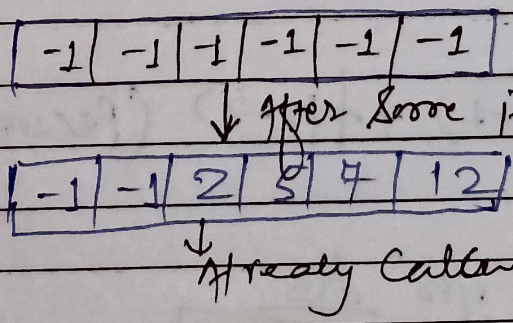


Tabulation

Try to go from Base Case to the required Answer

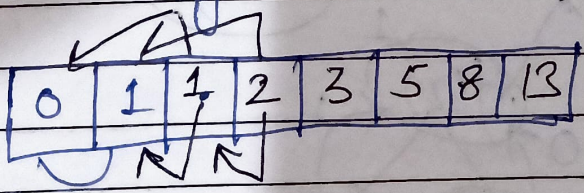
In Memoization, What we are doing

→ Initially filled the Array with -1 And during iteration if Array is filled with some other value then we have already calculated for that index and simple return that.



→ In DP.

we are doing it with base case only



for any ith, the answer will be

$$dp[i] = dp[i-1] + dp[i-2]$$

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



int recursiveFib(int n,vector<int>&memo)
{
    if(n<=1)
    {
        return 1;
    }
    if(memo[n]!=-1)
    {
        return memo[n];
    }

    return recursiveFib(n-1,memo)+recursiveFib(n-2,memo);
}
/*

In tabulation approach

we need to calculate our answer through give base case

*/
int dpFib(int n)
{
    int dp[n+1];
    dp[0]=1;
    dp[1]=1;

    for(int i=2;i<=n;i++)
    {
        dp[i]=dp[i-1]+dp[i-2];
    }
    return dp[n];
}


int main()
{
    int n;
    cin>>n;
     vector<int>memo(n+1,-1);
    int ans1=recursiveFib(n,memo);
    int ans2=dpFib(n);
    cout<<ans1<<" "<<ans2<<endl;
}


/*

In recusive + memoization


time complexity: O(n)
space complexity: O(n) for array  + O(n) because of recursion stack


```



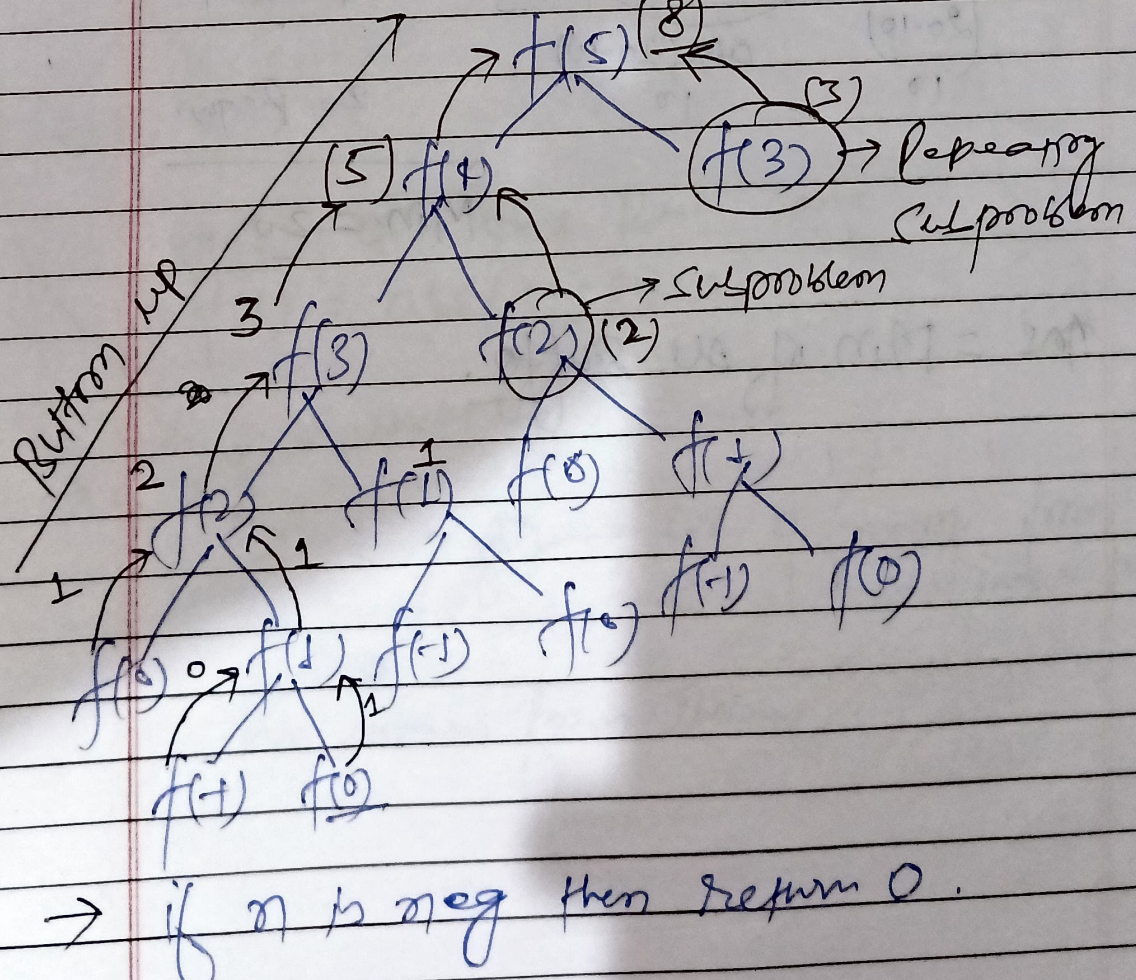
```cpp
In dp


time complexity: O(n)
space complexity: O(n)

*/
```

2. Count No of ways to reach N by taking 1 or 2 steps.

→ Exactly same as fibonacci number Calculation.

→ if n=0, that means you can not move anywhere so number of moves will be 1.

Recursion Tree for n=5



→ if n is neg then return 0.

```cpp
/*
you have to take only 1 or 2 steps

then count number of ways to reach N  and return it ;

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

int recursiveCount(int n,vector<int>&memo)
{
    if(n==0)
    {
        return 1;
    }
    if(n<0)
    {
        return 0;
    }
    if(memo[n]!=-1)
    {
        return memo[n];
    }

    return recursiveCount(n-1,memo)+recursiveCount(n-2,memo);
}

int dpCount(int n )
{
    int dp[n+1];
    dp[0]=1;
    for(int i=1;i<=n;i++)
    {
        dp[i]=dp[i-1]+ (i-2>=0?dp[i-2]:0);
    }
    return dp[n];
}


int main()
{
    int n;
    cin>>n;
  vector<int>memo(n+1,-1);
  int ans1= recursiveCount(n,memo);
  int ans2= dpCount(n);

  cout<<ans1<<" "<<ans2<<endl;
}

/*

In recusive + memoization


time complexity: O(n)
space complexity: O(n) for array  + O(n) because of recursion stack


In dp


time complexity: O(n)
space complexity: O(n)

*/
```


6/28/22, 4:59 PM countWaysToReachN.cpp

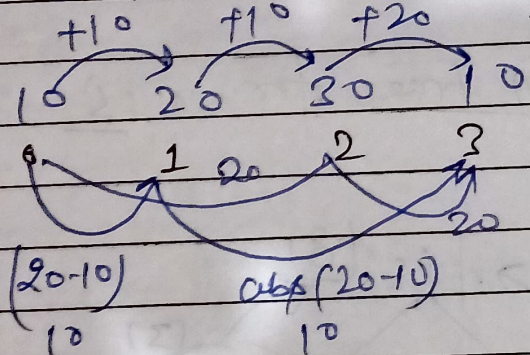
3. Frog jump

a_i = Energy

You have to find min energy required to reach from 0 to n-1 by taking 1 or 2 steps ahead

If you are at a_i → then you can move a_{i+1} or a_{i+2}.

here n = 4



Reach 0 → 3

40 Req

40 Req

20 Requir

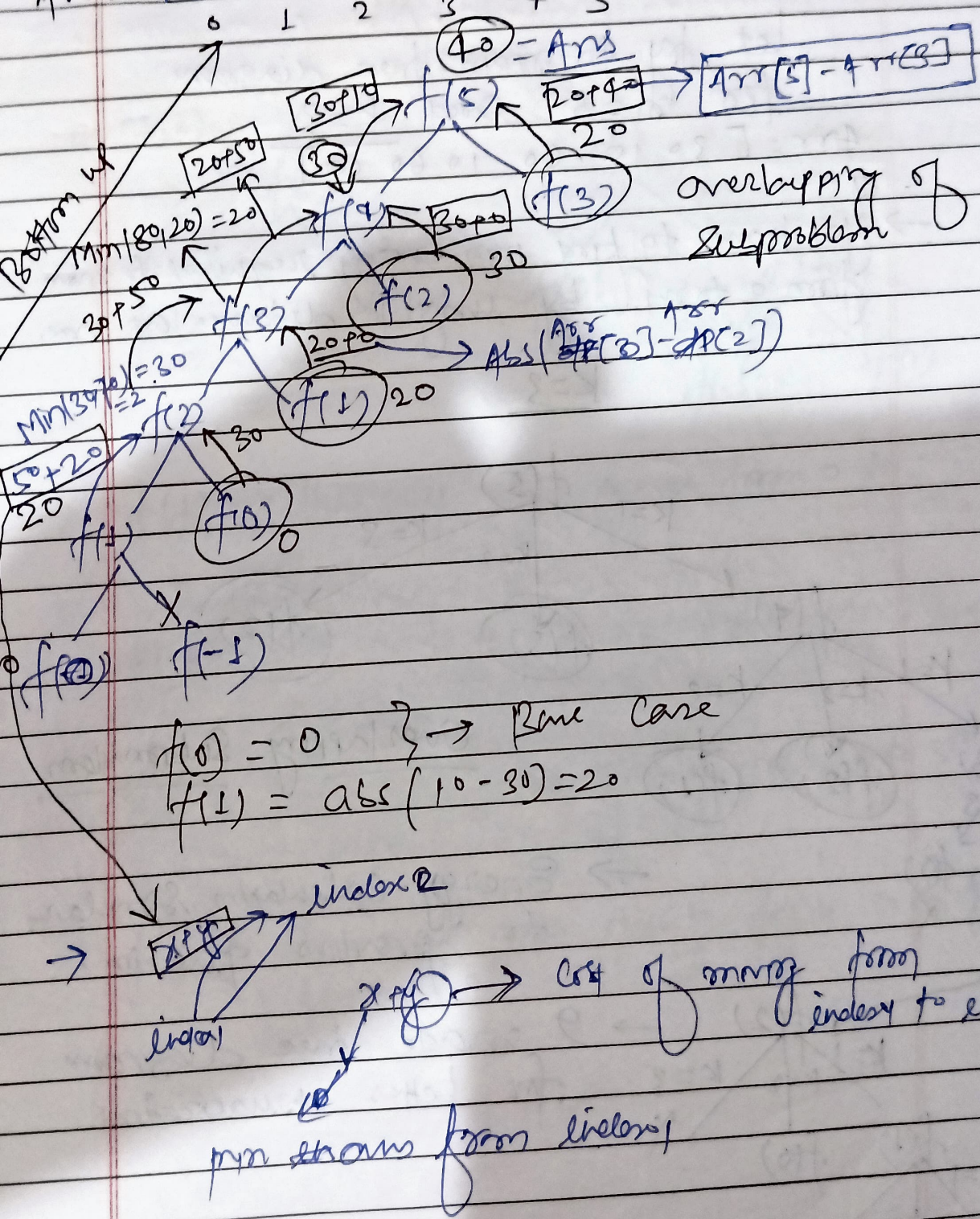
Min = 20

This = Min of all steps.

Recursive Tree Diagram

Arr = [30, 10, 60, 10, 60, 50] n = 6

0 1 2 3 4 5



```cpp
/*
Ways to write recursive code

1. try to represent the problem in a terms of index

2.Do all possible stuffs on that index and accrodingly to the problem statement

3. if question asking  COUNT ALL WAYS ==> Sum of all the stuffs
   if question asking MIN OR MAX      ==>  Then we will take max or min from the calculated stuffs
*/

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

int recursiveJump(vector<int>&input,vector<int>&memo,int n)
{
    if(n==0)
    {
        return 0;
    }
    if(n<0)
    {
        return INT_MAX;
    }

    if(memo[n]!=-1)
    {
        return memo[n];
    }

    int oneStep= recursiveJump(input,memo,n-1)+ abs(input[n]-input[n-1]);

    int twoStep= n-2>=0?recursiveJump(input,memo,n-2)+abs(input[n]-input[n-2]):INT_MAX;

    return min(oneStep,twoStep);
}

int dpJump(vector<int>&input)
{
    int dp[input.size()];
    dp[0]=0;
    for(int i=1;i<input.size();i++)
    {
        dp[i]=min(dp[i-1]+abs(input[i]-input[i-1]),i-2>=0?dp[i-2]+abs(input[i]-input[i-2]):INT_MAX);
    }
    return dp[input.size()-1];
}

int main()
{
    int n;
    cin>>n;
/*
6
30 10 60 10 60 50
```



```cpp
output=40
*/
    vector<int>v(n,0);
    for(int i=0;i<n;i++)
    {
        cin>>v[i];
    }
    vector<int>memo(n+1,-1);
    int ans1= recursiveJump(v,memo,n-1);
    int ans2= dpJump(v);
    cout<<ans1<<" "<<ans2<<endl;
}


/*
In recusive + memoization
 time complexity: O(n)
 space complexity: O(n) for array  + O(n) because of recursion stack

In dp
    time complexity: O(n)
    space complexity: O(n)


*/
```

4. Frog jump with K distance

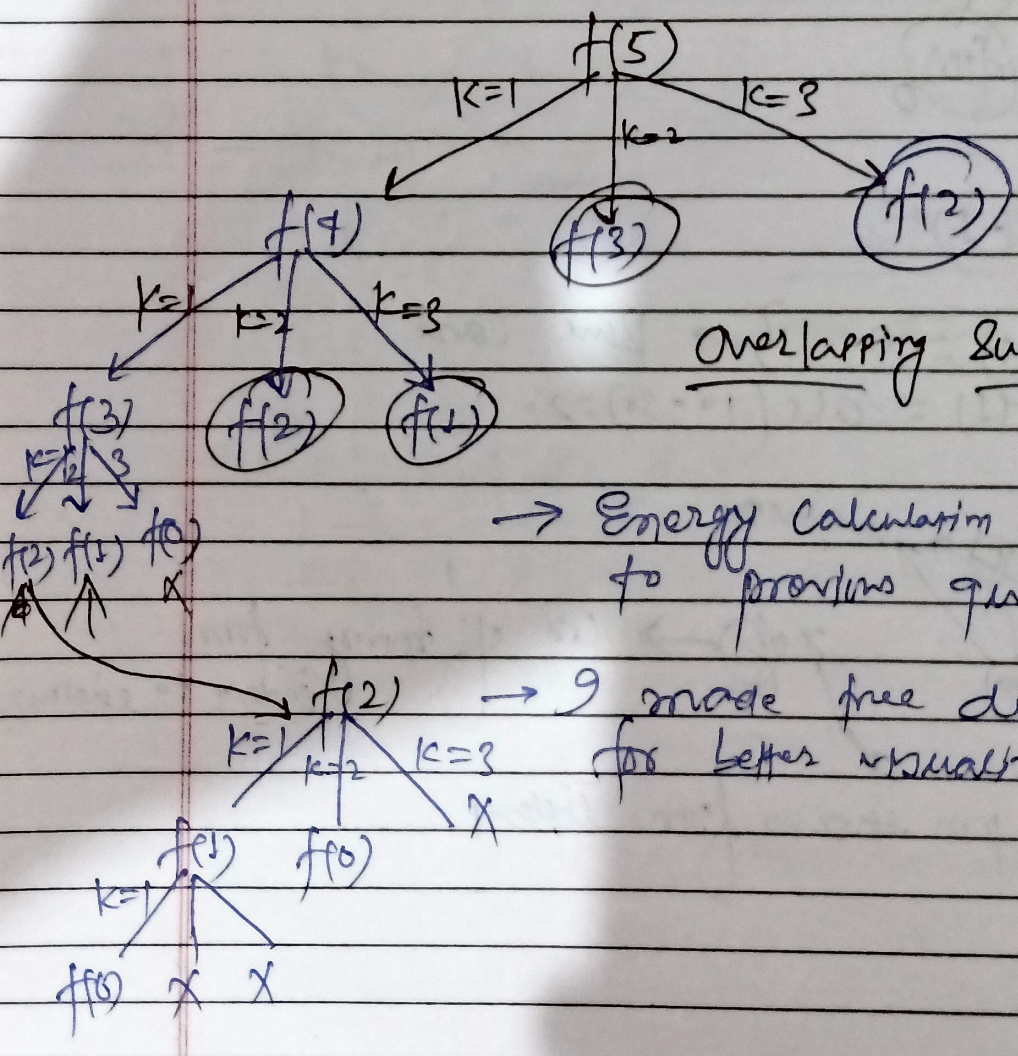
→ This question is similar to frog jump

let try to make tree diagram for n=6 and K=3

Arr = [30 10 60 10 60 50]

→ You have to find min energy required to move from 0 to n-1 by using k distance at max.

K=3



Overlapping Subproblem

→ Energy calculation similar to previous question.

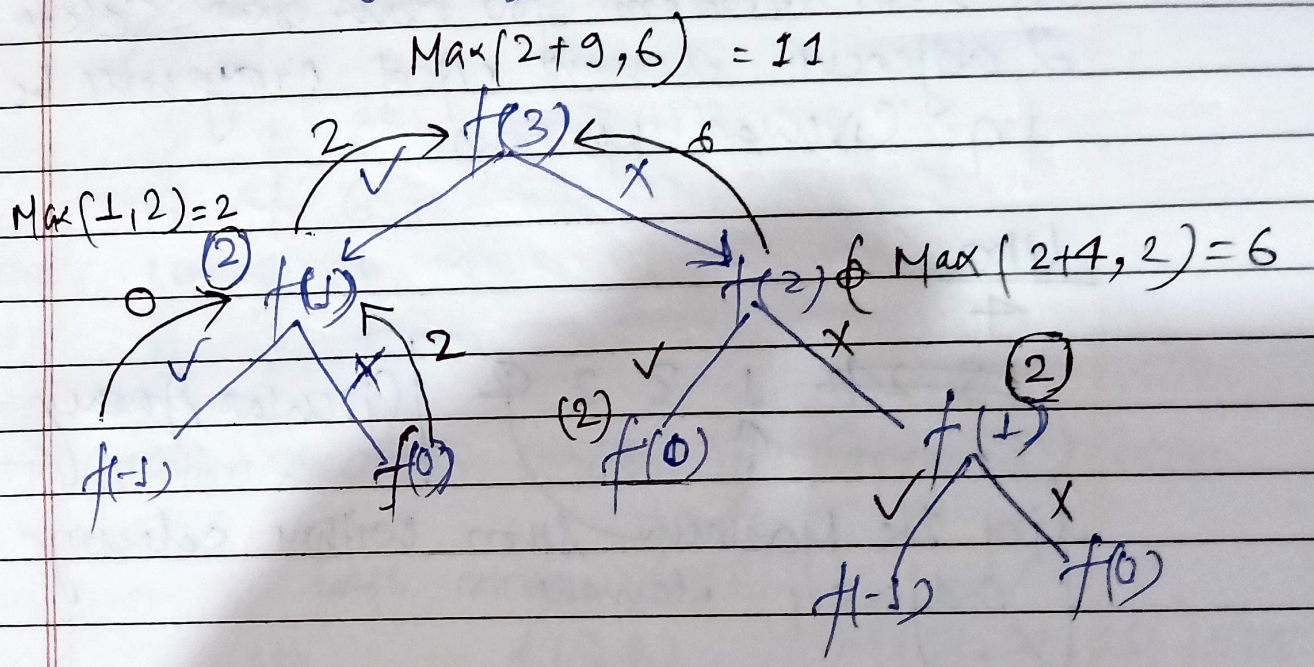
→ I made tree diagram for better visualization

```cpp
/*
   Ways to write recursive code

        1. try to represent the problem in a terms of index

        2.Do all possible stuffs on that index and accrodingly to the problem statement

        3. if question asking COUNT ALL WAYS == > Sum of all the stuffs if question asking MIN OR MAX ==
        > Then we will take max or
    min from the calculated stuffs
 */



#include<bits/stdc++.h>
using namespace std;
/*
this problem is extension of frogJump
*/

int recursiveJump(vector<int> &input, vector<int> &memo, int n, int k)
{
   if (n == 0)
   {
       return 0;
   }
   if(memo[n]!=-1)
   {
         return memo[n];
   }
   int minAns=INT_MAX;
//    we need to check for each k distance from the current index
   for(int i=1;i<=k;i++)
   {
         if(n-i>=0)
         {
              int ans=recursiveJump(input,memo,n-i,k)+abs(input[n]-input[n-i]);
              minAns=min(minAns,ans);
         }
   }
    return memo[n]= minAns;
}


int dpJump(vector<int>&input,int k)
{
    int dp[input.size()];
    dp[0]=0;
    for(int i=1;i<input.size();i++)
    {
        dp[i]=INT_MAX;
        for(int j=1;j<=k;j++)
        {
            if(i-j>=0)
            {
                dp[i]=min(dp[i],dp[i-j]+abs(input[i]-input[i-j]));
            }
        }
    }
    return dp[input.size()-1];
}

int main()
{
    int n;
    cin>>n;
    int k;
    cin>>k;
    vector<int>v(n,0);
    for(int i=0;i<n;i++)
    {
        cin>>v[i];
    }
    vector<int>memo(n+1,-1);
    int ans1= recursiveJump(v,memo,n-1,k);
    int ans2= dpJump(v,k);
    cout<<ans1<<" "<<ans2<<endl;

}

/*
In recusive + memoization

time complexity: O(n*k)
space complexity: O(n) for array + O(n) for recursion stack

In DP

    time complexity: O(n*k)
    space complexity: O(n)


*/
```

5. Maximum Sum of Non-Adjacent Element

Arr = [2, 1, 4, 9]
0 1 2 3

n = 4



→ f(1) → This will give Answer from 0 to 1 index
Ans [0 ... 1] = 2

f(2) → This will give Answer from 0 to 2th index
Ans [0 ... 2] = 6

f(3) → This will give Answer from 0 to 3th index [0 ... 3] = 11

f(4) → This will give Answer from 0 to 9th index [0 ... 9] = 17

```cpp
/*
   Ways to write recursive code

        1. try to represent the problem in a terms of index

        2.Do all possible stuffs on that index and accrodingly to the problem statement

        3. if question asking COUNT ALL WAYS == > Sum of all the stuffs if question asking MIN OR MAX ==
        > Then we will take max or
    min from the calculated stuffs
 */
#include<bits/stdc++.h>
using namespace std;

int recursiveSum(vector<int> &input, vector<int> &memo, int n)
{
   if(n==0)
   {
    //n==0 , only reach if we have not taken n=1 element , if we have taken then this will be -1 ;
    return input[0];
   }
   if(n<0)
   {
    return 0;
   }
    if(memo[n]!=-1)
    {
     return memo[n];
    }
    // taken
    int taken=0;
    int notTaken=0;

    taken= input[n]+recursiveSum(input,memo,n-2);
    notTaken= 0 + recursiveSum(input,memo,n-1);

    return memo[n]=max(taken,notTaken);

}

int countDp(vector<int>&input)
{
    int dp[input.size()];
    dp[0]=input[0];

    for(int i=1;i<input.size();i++)
    {
        dp[i]= max(dp[i-1],(i-2<0?0:dp[i-2])+input[i]);
    }
    return dp[input.size()-1];
}

int main()
{
    int n;
    cin>>n;
```



```cpp
    vector<int>v(n,0);
    for(int i=0;i<n;i++)
    {
        cin>>v[i];
    }
    vector<int>memo(n+1,-1);

    int ans1= recursiveSum(v,memo,n-1);
    int ans2= countDp(v);

    cout<<ans1<<" "<<ans2<<endl;

}

/*
In recusive + memoization

time complexity : O(n)
space complexity : O(n) for array  + O(n) because of recursion stack


In dp

time complexity : O(n)
space complexity : O(n)
*/
```

6. HOUSE Robber

→ In this problem you can not select 2 adjacent element. And array will be in Circular Manner

Like

4

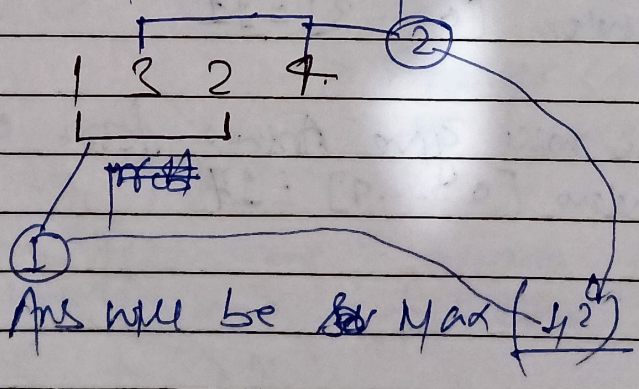
~~3 4~~ 1 3 2 4 | Circular Array

find the Maximum sum without selecting adjacent element

→ Obs

1st element and Last element can not participate in solution because they were adjacent

Now Break the problem into 2 problem



Ans will be Max (1, 2)

→ Apply the last question Approach and you will get the Answer

```cpp

/*
question:

Mr. X is a skilled burglar who intends to loot houses along a street. Each home has a specific amount of money tucked away. All of the residences in this area are grouped in a circle. That is, the first home is next to the final one. Meanwhile, nearby houses are connected to a security system, which will immediately inform the police if two adjacent houses are broken into on the same night.

Given an integer array nums indicating the amount of money in each house. Return the greatest amount of money Mr. X can rob without notifying the cops.

Example:

Input:

[2, 3, 2]

Output:

3
*/

/*
   Ways to write recursive code

        1. try to represent the problem in a terms of index

        2.Do all possible stuffs on that index and accrodingly to the problem statement

        3. if question asking COUNT ALL WAYS == > Sum of all the stuffs if question asking MIN OR MAX ==
        > Then we will take max or
    min from the calculated stuffs
 */
#include<bits/stdc++.h>
using namespace std;

int recursiveRobber(vector<int>&input,int n,vector<int>&memo)
{
    if(n==0)
    {
        return input[0];
    }
    if(n<0)
    {
        return 0;
    }
    if(memo[n]!=-1)
    {
        return memo[n];
    }
    int taken=0;
    int notTaken=0;
    taken= input[n]+recursiveRobber(input,n-2,memo);
    notTaken= 0 + recursiveRobber(input,n-1,memo);
    return memo[n]=max(taken,notTaken);
}

int dpRobber(vector<int>&input)
{
    int dp[input.size()];
    dp[0]=input[0];
    for(int i=1;i<input.size();i++)
    {
        dp[i]= max(dp[i-1],(i-2<0?0:dp[i-2])+input[i]);
    }
    return dp[input.size()-1];
}

int main()
{
    int n;
    cin>>n;

    vector<int>v(n,0);
    vector<int>input1;
    vector<int>input2;
    // due to circular nature , 1st and last can not include in the answer thats why need to call the function by taking 1st element and not taking last elmenet and vice versa
    for(int i=0;i<n;i++)
    {
        cin>>v[i];
        if(i!=0)
        {
            input1.push_back(v[i]);
        }
        if(i!=n-1)
        {
            input2.push_back(v[i]);
        }
    }
    vector<int>memo(n+1,-1);
    vector<int>memo1(n+1,-1);
    int ans1=max(recursiveRobber(input1,input1.size()-1,memo),recursiveRobber(input2,input2.size()-1,memo1));
    int ans2=max(dpRobber(input1),dpRobber(input2));

    cout<<ans1<<" "<<ans2<<endl;
}
/*

In recusive + memoization

time complexity: O(n)
space complexity: O(n) for array  + O(n) because of recursion stack

In dp
time complexity: O(n);
space complexity: O(n);
*/
```

7. NINJA TRAINING

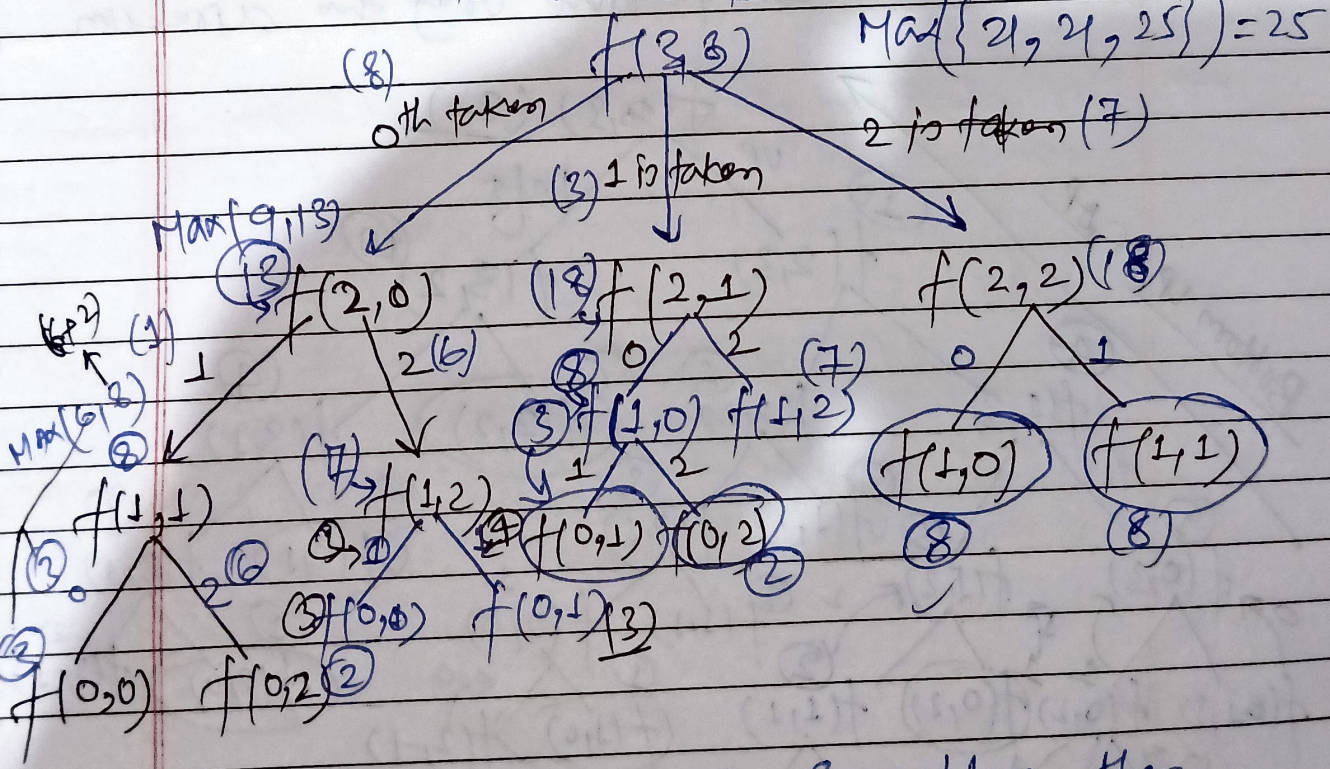
Input

n = 4

d_0	2	1	3
d_1	3	4	6
d_2	10	1	6
d_3	8	3	7

→ question description written in Coomeva section.

Lets make tree



→ There is overlapping of subproblem, then we need to do Memoization

→ What is f(2,0)

You have to Calculate upto ~~[illegible]~~ from 0 to 279 index provided for 3rd index we have skipped 0th item.

```cpp
/*
 question:
 Ninja is planing this ‘N’ days-long training schedule. Each day, he can perform any one of these three activities. (Running, Fighting Practice or Learning New Moves). Each activity has some merit points on each day. As Ninja has to improve all his skills, he can’t do the same activity in two consecutive days. Can you help Ninja find out the maximum merit points Ninja can earn?

 You are given a 2D array of size N*3 ‘POINTS’ with the points corresponding to each day and activity. Your last is to calculate the maximum number of merit points that Ninja can earn.
input:
3
10 40 70
20 50 80
30 60 90

output=210

 */

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


int recursiveTraining(vector<vector<int>>&input,int n,vector<vector<int>>&memo,int k)
{
    if(n==0)
    {
        int maxe=0;
        for(int i=0;i<3;i++)
        {
            if(k!=i)
            {
                maxe=max(maxe,input[n][i]);
            }
        }

        return maxe;
    }

    if(memo[n][k]!=-1)
    {
        return memo[n][k];
    }
    for(int i=0;i<3;i++)
    {
        // we can not take previosly taken activity
        if(k!=i)
        {
            memo[n][k]=max(memo[n][k],input[n][i]+recursiveTraining(input,n-1,memo,i));
        }
    }
    return memo[n][k];
}

int dpTraining(vector<vector<int>>&input)
{
```



```cpp
    int n=input.size();
  vector<vector<int>>dp(n,vector<int>(4,0));

    dp[0][0]=max(input[0][1],input[0][2]);
    dp[0][1]=max(input[0][0],input[0][2]);
    dp[0][2]=max(input[0][0],input[0][1]);
    dp[0][3]=max({input[0][0],input[0][1],input[0][2]});

    for(int i=1;i<n;i++)
    {
        for(int last=0;last<4;last++)
        {
            for(int task=0;task<3;task++)
            {
                if(last!=task)
                {
                    dp[i][last]=max(dp[i][last],input[i][task]+dp[i-1][task]);
                }
            }
        }
    }

    return dp[n-1][3];
}
int main()
{
   int n;
   cin>>n;
   vector<vector<int>> v(n,vector<int>(3));
    for(int i=0;i<n;i++)
    {
         for(int j=0;j<3;j++)
         {
             cin>>v[i][j];
         }
    }

    vector<vector<int>> memo(n,vector<int>(4,-1));

    int ans1= recursiveTraining(v,n-1,memo,3);
    int ans2= dpTraining(v);
    cout<<ans1<<" "<<ans2<<endl;

}
```

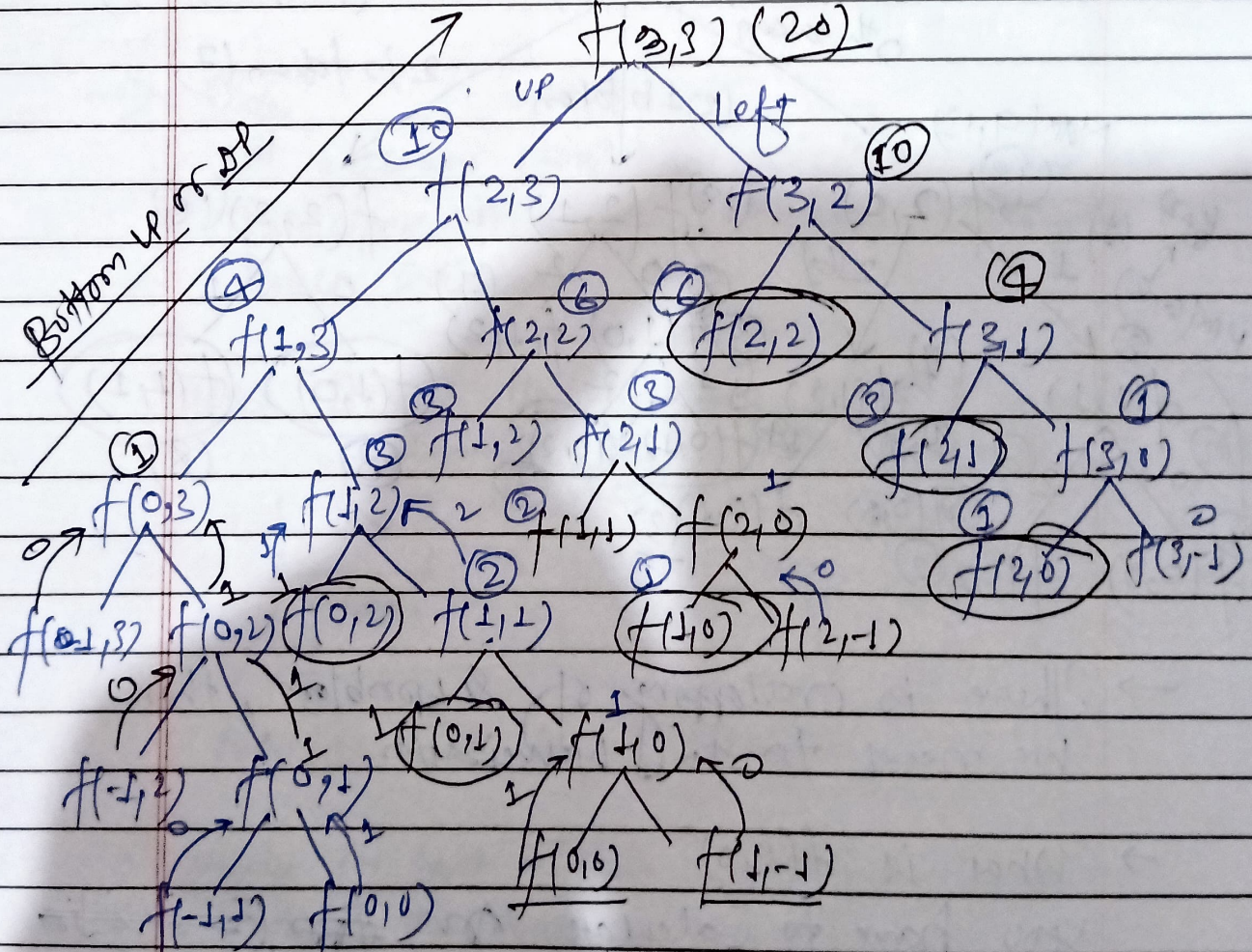
8. DP on Grid / 2D Matrix

→ Memoization to DP Conversion

i) Declare the Base Case
ii) Express all States in for Loops
iii) Copy the recurrance relation

Lets make recursive Tree for n=3, m=3

→ We are Constrain to move only two direction



Circle are overlapping of subproblem.

If n=0 and m==0 then return will be 1 because only position are there to move.

→ Here overlapping core in the score then memorization will taken place followed by bottom-up (dp).

9. Unique path 2.

This is slightly different from problem 2

In this case we need to keep one extra base case of dead cell.

→ In recursion you can put

```
if (Arr[i][j] == -1) { dead cell Condn.
    return 0;
}
```

→ In bottom-up DP approach we can put

```
if (Arr[i][j] == -1)
    dp[i][j] = 0;
```

And time and space complexity going to same.

```cpp
/*
Given a ‘N’ * ’M’ maze with obstacles, count and return the number of unique paths to reach the right-bottom cell from the top-left cell. A cell in the given maze has a value '-1' if it is a blockage or dead-end, else 0. From a given cell, we are allowed to move to cells (i+1, j) and (i, j+1) only. Since the answer can be large, print it modulo 10^9 + 7.
For Example :
Consider the maze below :
0 0 0
0 -1 0
0 0 0

There are two ways to reach the bottom left corner -

(1, 1) -> (1, 2) -> (1, 3) -> (2, 3) -> (3, 3)
(1, 1) -> (2, 1) -> (3, 1) -> (3, 2) -> (3, 3)

Hence the answer for the above test case is 2.
*/
#include <bits/stdc++.h>
using namespace std;


// top down approach
int recursiveCount(int m, int n,vector<vector<int>>&input, vector<vector<int>>&memo)
{
    if(n>=0 and m>=0 and input[n][m]==-1)
    {
        return 0;
    }
    if (n == 0 and m == 0)
    {
        return 1;
    }
    if (n < 0 or m < 0)
    {
        return 0;
    }



    if (memo[n][m] != -1)
    {
        return memo[n][m];
    }
    int upSideCall = recursiveCount(m, n - 1,input, memo);
    int leftSideCall = recursiveCount(m - 1, n,input, memo);

    memo[n][m] = upSideCall + leftSideCall;
}

/*
for bottom up approach
1. Declare the base case
2. express all the states(changing variable) in for loops
3. copy the recurrence relation
*/

```

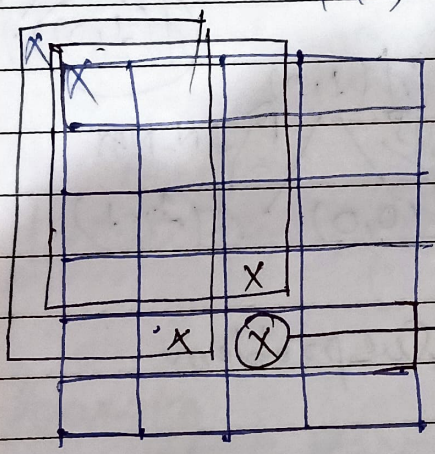


```cpp
int dpCount(int m, int n,vector<vector<int>>&input)
{
    vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));
    // dp[0][0]=1;

    for (int i = 0; i <=n; i++)
    {
        for (int j = 0; j <= m; j++)
        {
            if (input[i][j]==-1)
            {
                dp[i][j] = 0;
            }
            else if(i==0 and j==0)
            {
                dp[i][j]=1;
            }
            else
            {
                dp[i][j] = (j - 1 >= 0 ? dp[i][j - 1] : 0) + (i - 1 >= 0 ? dp[i - 1][j] : 0);
            }
        }
    }
    return dp[n][m];
}

int main()
{
    int m, n;
    cin >> m >> n;
    vector<vector<int>>input(n,vector<int>(m,0));
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<m;j++)
        {
            cin>>input[i][j];
        }
    }

    vector<vector<int>> memo(n + 1, vector<int>(m + 1, -1));

    int ans1 = recursiveCount(m - 1, n - 1,input, memo);
    int ans2 = dpCount(m - 1, n - 1,input);

    cout << ans1 << " " << ans2 << endl;
    // cout<<ans1<<endl;
}
/*
In recusive + memoization
time complexity: O(mn)
space complexity: O(mn) for memoization and O(n+m) recusion stack

In dp
time complexity: O(mn)
space complexity: O(mn)
*/
```

10. Minimum path Sum

→ For writing recurrence we need to take care of three things
i) Explore (i,j)
ii) Explore all path
iii) take the min path

f(n-1, m-1) → this will give cost from (0,0) to (n-1, m-1).

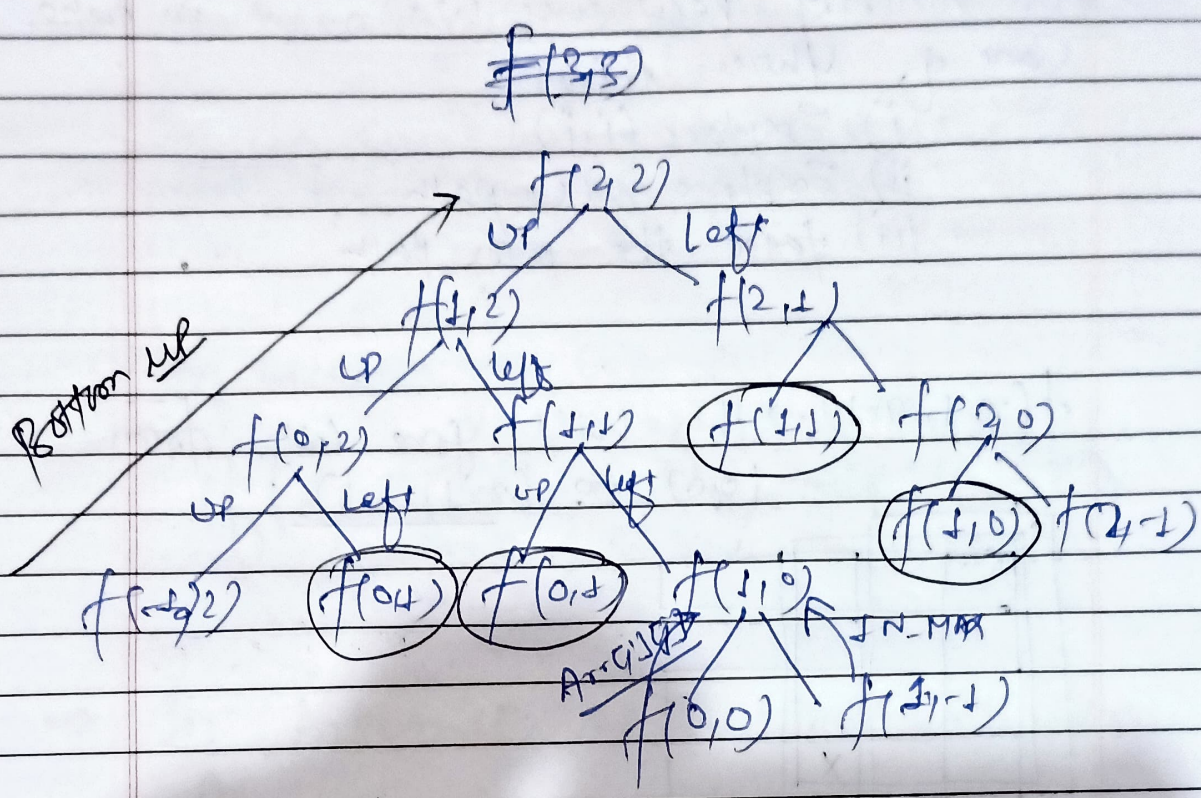


→ if you are calculating for this value at point (i,j) from (both sides)

→ This problem less bit similar to previous one..

→ Time and Space Complexity remain same as previous one.

Recursive Tree Visualization



overlapping of subproblem

→ from both branch we have option to choose prin (left, right) and add the right value.

```cpp
/*
Given a m x n grid filled with non-negative numbers, find a path from top left to bottom right, which minimizes the sum of all numbers along its path.

Note: You can only move either down or right at any point in time.
Example 1:

Input: grid = [[1,3,1],[1,5,1],[4,2,1]]
Output: 7
Explanation: Because the path 1 → 3 → 1 → 1 → 1 minimizes the sum.
*/

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

int recursiveMinimuCost(vector<vector<int>> &input, int n, int m, vector<vector<int>> &memo)
{

    if (n == 0 and m == 0)
    {
        return input[n][m];
    }
    if (n < 0 or m < 0)
    {
        return 1e9;
    }
    if (memo[n][m] != -1)
    {
        return memo[n][m];
    }
    int left = recursiveMinimuCost(input, n, m - 1, memo);
    int up = recursiveMinimuCost(input, n - 1, m, memo);
    memo[n][m] = min(left, up) + input[n][m];
    return memo[n][m];
}

int dpMinimumCost(vector<vector<int>> &input)
{
    int n = input.size();
    int m = input[0].size();
    vector<vector<int>> dp(n, vector<int>(m, 0));
    dp[0][0] = input[0][0];
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            if(i==0 and j==0)
            {
                dp[i][j]=input[i][j];
            }
            else
            {
                dp[i][j] = min((i - 1 >= 0 ? dp[i - 1][j] : 1e9), (j - 1 >= 0 ? dp[i][j - 1] : 1e9)) + input[i][j];
            }

        }
    }
```



```cpp
    return dp[n - 1][m - 1];
}

int main()
{
    int n, m;
    cin >> n >> m;
    vector<vector<int>> input(n, vector<int>(m, 0));
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            cin >> input[i][j];
        }
    }
    vector<vector<int>> memo(n, vector<int>(m, -1));
    cout << recursiveMinimuCost(input, n - 1, m - 1, memo) << endl;
    cout << dpMinimumCost(input) << endl;
}


/*
In recursive + memoization

    time complexity: O(mn)
    space complexity: O(mn) for memoization and O(n+m) recusion stack

In dp

    time complexity: O(mn)
    space complexity: O(mn)

*/
```

11. Triangle

Starting point

1

2 3

? 6 7

8 9 6 10

Not fixed ending point here.

Move

→ find the minimum path from top to bottom.

→ Tabulation is opposite of Recursion

if Recursion start from (0,0) then tabulation will start from (n-1, n-1) · just opposite

Recursion

1 → starting point for Recursion

2 3

3 6 7

8 9 6 10

↓

Starting point for tabulation because it is totally opposite of Recursion.

You can move free for this problem easy 😊

```cpp
/*
Problem Statement
You are given a triangular array/list 'TRIANGLE'. Your task is to return the minimum path sum to reach from the top to the bottom row.
The triangle array will have N rows and the i-th row, where 0 <= i < N will have i + 1 elements.
You can move only to the adjacent number of row below each step. For example, if you are at index j in row i, then you can move to i or i + 1 index in row j + 1 in each step.
For Example :
If the array given is 'TRIANGLE' = [[1], [2,3], [3,6,7], [8,9,6,1]] the triangle array will look like:

1
2,3
3,6,7
8,9,6,10

For the given triangle array the minimum sum path would be 1->2->3->8. Hence the answer would be 14.
*/
#include <bits/stdc++.h>
using namespace std;

/*
In this question , in recursive we will start from top to bottom
*/

int recursiveTraingle(vector<vector<int>> &input, int row, int col, vector<vector<int>> &memo)
{
    if (col > row)
    {
        return 1e9;
    }
    if (row == input.size() - 1)
    {
        return input[row][col];
    }

    if (memo[row][col] != -1)
    {
        return memo[row][col];
    }

    int down = recursiveTraingle(input, row + 1, col, memo);
    int diag = recursiveTraingle(input, row + 1, col + 1, memo);
    memo[row][col] = min(down, diag) + input[row][col];
    return memo[row][col];
}

int dpTraingle(vector<vector<int>> &input)
{
    int n = input.size();

    /*
    As this is opposite of recursion then this will start from bottom to top

    Now we have to check States in Dp because of variable ending point .
```



```cpp

    for  i=3 then possible j= 0 1 2 3
    for i=2 then possible j= 0 1 2
    for i=1 then possible j= 0 1
    for i=0 then possible j= 0
     for different i or row we have different states of j .
    */
    //    now write base case from bottom
    vector<vector<int>> dp(n, vector<int>(n, 0));
    for (int i = 0; i < n; i++)
    {
        dp[n - 1][i] = input[n - 1][i];
    }
    for (int i = n - 2; i >= 0; i--)
    {
        for (int j = 0; j <= i; j++)
        {
            int down = dp[i + 1][j];
            int diag = dp[i + 1][j + 1];
            dp[i][j] = min(down, diag) + input[i][j];
        }
    }

    return dp[0][0];
}

int main()
{
    int n;
    cin >> n;
    vector<vector<int>> input(n);
    for (int i = 0; i < n; i++)
    {
        input[i].resize(i + 1);
        for (int j = 0; j <= i; j++)
        {
            cin >> input[i][j];
        }
    }
    vector<vector<int>> memo(n, vector<int>(n, -1));

    int ans1 = recursiveTraingle(input, 0, 0, memo);
    int ans2 = dpTraingle(input);
    cout << ans1 << " " << ans2 << endl;
}


```

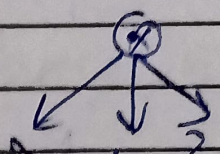
12. Max / Min Falling Sum.

→ Max path Sum

From any cell → 1st Row
to
any cell → Last Row

→ In this question starting and ending point are not fixed. We can start any cell from 1st row and similar if end at cell in last row.

In this we can move in three direction



direction of move.

(4,4)

→

1	2	10	4
100	3	2	1
1	1	20	2
1	2	2	1

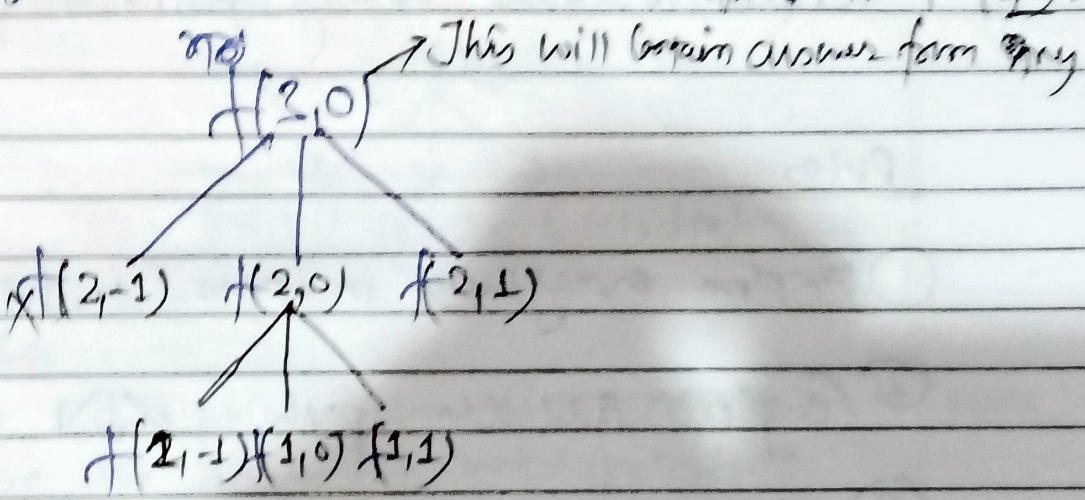
(n-1, m-1)

↓ ↓ ↓ ↓ } Any cell can be starting point.

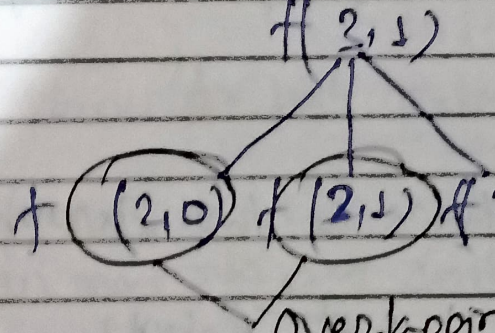
So we need to explore the path for last row column.

Let's see overlapping of subproblem

→ If we start from (2,0)



→ If we start from (3,1)



Overlapping from calling it (3,1).
That why memoization will store ignore.

f(3,1) → It will contain answer, starting from any column from 1st to ending of (2,1) in last row.

```cpp
/*
You have been given an N*M matrix filled with integer numbers, find the maximum sum
that can be obtained from a path starting from any cell in the first row to any cell
in the last row.

From a cell in a row, you can move to another cell directly below that row, or
diagonally below left or right. So from a particular cell (row, col), we can move in
three directions i.e.
*/

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

int recursiveFallsumUTIL(vector<vector<int>> &mat, int n, int m, vector<vector<int>>
&memo)
{
    if (m < 0 or m >= mat[0].size())
    {
        return -1e8;
    }
    if (n == 0)
    {
        return mat[0][m];
    }
    if (memo[n][m] != -1)
    {
        return memo[n][m];
    }
    // we have to move upper , upper right diagonal , upper left diagonal
    int up, up_right, up_left;
    up = up_right = up_left = INT_MIN;

    up = recursiveFallsumUTIL(mat, n - 1, m, memo) + mat[n][m];

    up_right = recursiveFallsumUTIL(mat, n - 1, m + 1, memo) + mat[n][m];

    up_left = recursiveFallsumUTIL(mat, n - 1, m - 1, memo) + mat[n][m];

    memo[n][m] = max(max(up, up_right), up_left);
    return memo[n][m];
}
/*
  In this recursive approach i need to traverse every col in last row for getting
answer.
  In this question starting or ending point not fixed.

*/
int recursiveFallingSum(vector<vector<int>> &input, int n, int m,
vector<vector<int>> &memo)
{
    int ans = INT_MIN;

    for (int i = 0; i <= m; i++)
    {
        ans = max(ans, recursiveFallsumUTIL(input, n, i, memo));
    }
    return ans;
}

```



```cpp
int dpFallingSum(vector<vector<int>>&input)
{
    int n = input.size();
    int m = input[0].size();
    vector<vector<int>>dp(n,vector<int>(m,0));
    /*
    Since tabulations completely opposite of recursion , here we start from starting

    lets think about states in the dp

    for i=0  what would be possible of j =0, 1,2.....m;
    for i=1  what would be possible of j =0, 1,2.....m;
    for i=2  what would be possible of j =0, 1,2.....m;
    for i=3  what would be possible of j =0, 1,2.....m;

    */
    //    base case ==> destination + outofbound;

    for(int j=0;j<m;j++)
    {
        dp[0][j]=input[0][j];
    }
    for(int i=1;i<n;i++)
    {
        for(int j=0;j<m;j++)
        {
            int up, up_right, up_left;
            up = up_right = up_left = INT_MIN;
            up = dp[i-1][j] + input[i][j];
            if(j+1<m)
            {
                up_right = dp[i-1][j+1] + input[i][j];
            }
            if(j-1>=0)
            {
                up_left = dp[i-1][j-1] + input[i][j];
            }
            dp[i][j] = max(max(up,up_right),up_left);
        }
    }
    int ans=INT_MIN;
    for(int i=0;i<m;i++)
    {
        ans=max(ans,dp[n-1][i]);
    }
    return ans;

}


int main()
{
    int n, m;
    cin >> n >> m;
    vector<vector<int>> mat(n, vector<int>(m));
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            cin >> mat[i][j];
```



```cpp
        }
    }
    vector<vector<int>> memo(n, vector<int>(m, -1));

    int ans1 = recursiveFallingSum(mat, n - 1, m - 1, memo);
    int ans2= dpFallingSum(mat);
    cout<<ans1<<" "<<ans2<<endl;
    // cout << ans1 << endl;
}

/*

In recursive + memoization

     time complexity = O(n*m)
    space complexity = O(n*m)for memoization array  + O(n) for recursion stack


In dp

    time complexity = O(n*m)
    space complexity = O(n*m)

*/
```

13. Cherry picker II
3D DP

→ Fixed starting point
→ ~~Variable~~ Variable ending point

Rules

① Explore everything in terms (i_1, j_1) & (i_2, j_2)

② Explore All the path ↙↓↘

③ MAX SUM

~~if~~ if we have only one starting point we should start recursion from fixed point.

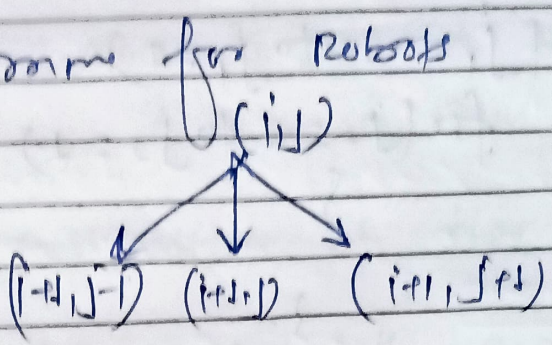
→ In this question there is two starting point

Robot 1 →

			← Robot 2
3	1	1	
2	5	1	
1	5	5	
2	1	1	

→ for Robot 1 Coordinate will be (i_1, j_1)
and for Robot 2 Coordinate will be (i_2, j_2)

→ possible move for Robots



Ok

At every point value of i or row no will be the same for both robots ?

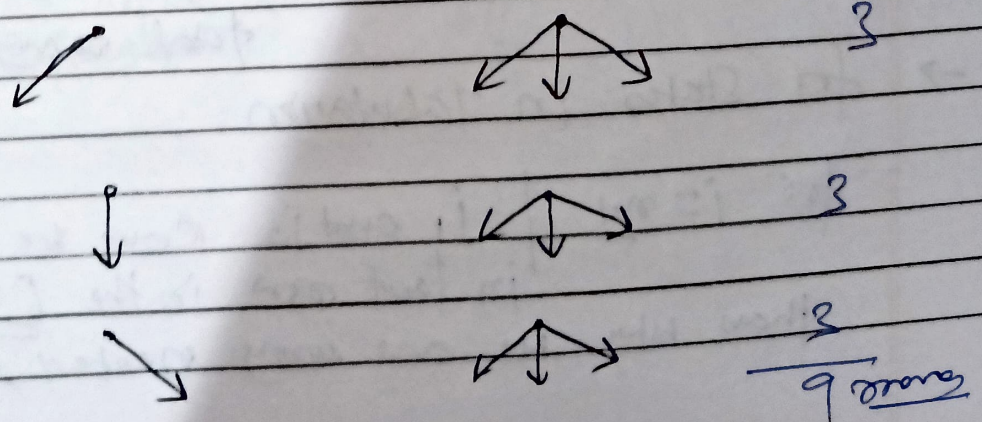
initially $f(i_1, j_1, i_2, j_2)$

↓ now

$f(i, j_1, j_2)$ because row no will same for both robot.

→ from one index one robot can move 3 blocks in next row and same for other

(robot 1 move) (robot 2 move) Moves



All Cordinate of there 9 moves, we will get. Like this

```
for(j1 = -1 to j1 = 1)
    for(j2 = -1 to j2 = 1)
        {
        }
}
```

Base case

① Out of Bound गया

② if j1 == j2 then pick one.

③ j1 != j2 then return grid[i][j1] + grid[i][j2]

On only recursion

$TC = (3^m \times 3^n)$

After Memoization

$TC = O(N \times M \times M) \times 9$

total moves.

→ for States in Tabulation

for i = m-1 | j1 and j2 can be anywhere in last row in the [0, m-1]

thats why we are using nested loop.

```cpp
/*
You are given a rows x cols matrix grid representing a field of cherries where grid[i][j] represents the number of cherries that you can collect from the (i, j) cell.

You have two robots that can collect cherries for you:

Robot #1 is located at the top-left corner (0, 0), and
Robot #2 is located at the top-right corner (0, cols - 1).
Return the maximum number of cherries collection using both robots by following the rules below:

From a cell (i, j), robots can move to cell (i + 1, j - 1), (i + 1, j), or (i + 1, j + 1).
When any robot passes through a cell, It picks up all cherries, and the cell becomes an empty cell.
When both robots stay in the same cell, only one takes the cherries.
Both robots cannot move outside of the grid at any moment.
Both robots should reach the bottom row in grid.
*/
#include <bits/stdc++.h>
using namespace std;

int recursiveCherryPickup(vector<vector<int>> &input, int i1, int j1, int j2, vector<vector<vector<int>>> &memo)
{
    if (i1 < 0 or i1 >= input.size() or j1 < 0 or j1 >= input[0].size() or j2 < 0 or j2 >= input[0].size())
    {
        return -1e8;
    }
    if (i1 == input.size() - 1)
    {
        if (j1 == j2)
        {
            return input[i1][j1];
        }
        else
        {
            return (input[i1][j1] + input[i1][j2]);
        }
    }

    if (memo[i1][j1][j2] != -1)
    {
        return memo[i1][j1][j2];
    }
    int ans = INT_MIN;
    for (int jj1 = -1; jj1 <= 1; jj1++)
    {
        for (int jj2 = -1; jj2 <= 1; jj2++)
        {
            int newj1 = j1 + jj1;
            int newj2 = j2 + jj2;
            int temp = 0;
            if (j1 == j2)
            {
                temp = recursiveCherryPickup(input, i1 + 1, newj1, newj2, memo) + input[i1][j1];
```



```cpp
            }
            else
            {
                temp = recursiveCherryPickup(input, i1 + 1, newj1, newj2, memo) + input[i1][j1] + input[i1][j2];
            }

            ans = max(ans, temp);
        }
    }
    memo[i1][j1][j2] = ans;
    return ans;
}
bool isvalid(int j1, int j2, int n, int m)
{
    if (j1 < 0 or j1 >= m or j2 < 0 or j2 >= m)
    {
        return false;
    }
    return true;
}

int dpCherryPickup(vector<vector<int>> &input)
{
    int n = input.size();
    int m = input[0].size();
    vector<vector<vector<int>>> dp(n, vector<vector<int>>(m, vector<int>(m, -1)));

    // first we need to write base , it will be motivated from recursion base case
    /*
    states like for i==n-1 then possible value of j1 and j2 may be they both point to same point or they may be on different point in same row or either they exchange their positions.
    */
    for (int j1 = 0; j1 < m; j1++)
    {
        for (int j2 = 0; j2 < m; j2++)
        {
            if (j1 == j2)
            {
                dp[n - 1][j1][j2] = input[n - 1][j1];
            }
            else
            {
                dp[n - 1][j1][j2] = input[n - 1][j1] + input[n - 1][j2];
            }
        }
    }
    //   now total three different states are there i,j1,j2 then three loops will be

    for (int i = n - 2; i >= 0; i--)
    {
        for (int j1 = 0; j1 < m; j1++)
        {
            for (int j2 = 0; j2 < m; j2++)
            {
                int ans = INT_MIN;
                for (int jj1 = -1; jj1 <= 1; jj1++)
                {
                    for (int jj2 = -1; jj2 <= 1; jj2++)
```



```cpp
                    {
                        int newj1 = j1 + jj1;
                        int newj2 = j2 + jj2;
                        if (!isvalid(newj1, newj2, n, m))
                        {
                            continue;
                        }
                        int temp = 0;
                        if (j1 == j2)
                        {

                            temp = dp[i + 1][newj1][newj2] + input[i][j1];
                        }
                        else
                        {

                            temp = dp[i + 1][newj1][newj2] + input[i][j1] + input[i][j2];
                        }

                        ans = max(ans, temp);
                    }
                }
                dp[i][j1][j2] = ans;
            }
        }
    }

    int result = 0;
    for (int j1 = 0; j1 < m; j1++)
    {
        for (int j2 = 0; j2 < m; j2++)
        {
            result = max(result, dp[0][j1][j2]);
        }
    }
    return result;
}

int main()
{
    int n, m;
    cin >> n >> m;

    vector<vector<int>> input(n, vector<int>(m));
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            cin >> input[i][j];
        }
    }
    vector<vector<vector<int>>> memo(n, vector<vector<int>>(m, vector<int>(m, -1)));

    int ans1 = recursiveCherryPickup(input, 0, 0, m - 1, memo);
    cout << ans1 << endl;
    int ans2 = dpCherryPickup(input);
    cout << ans2 << endl;
}
```

14. DP on Subset / Subsequence & target

Rules

i) Express every index in terms of (index, target)

ii) Explore possibilities of that index.

either Arr[i] will be the part of our Subset or Subsequence

iii) Return T or F According to condn

Question Name :- Subset equal to target

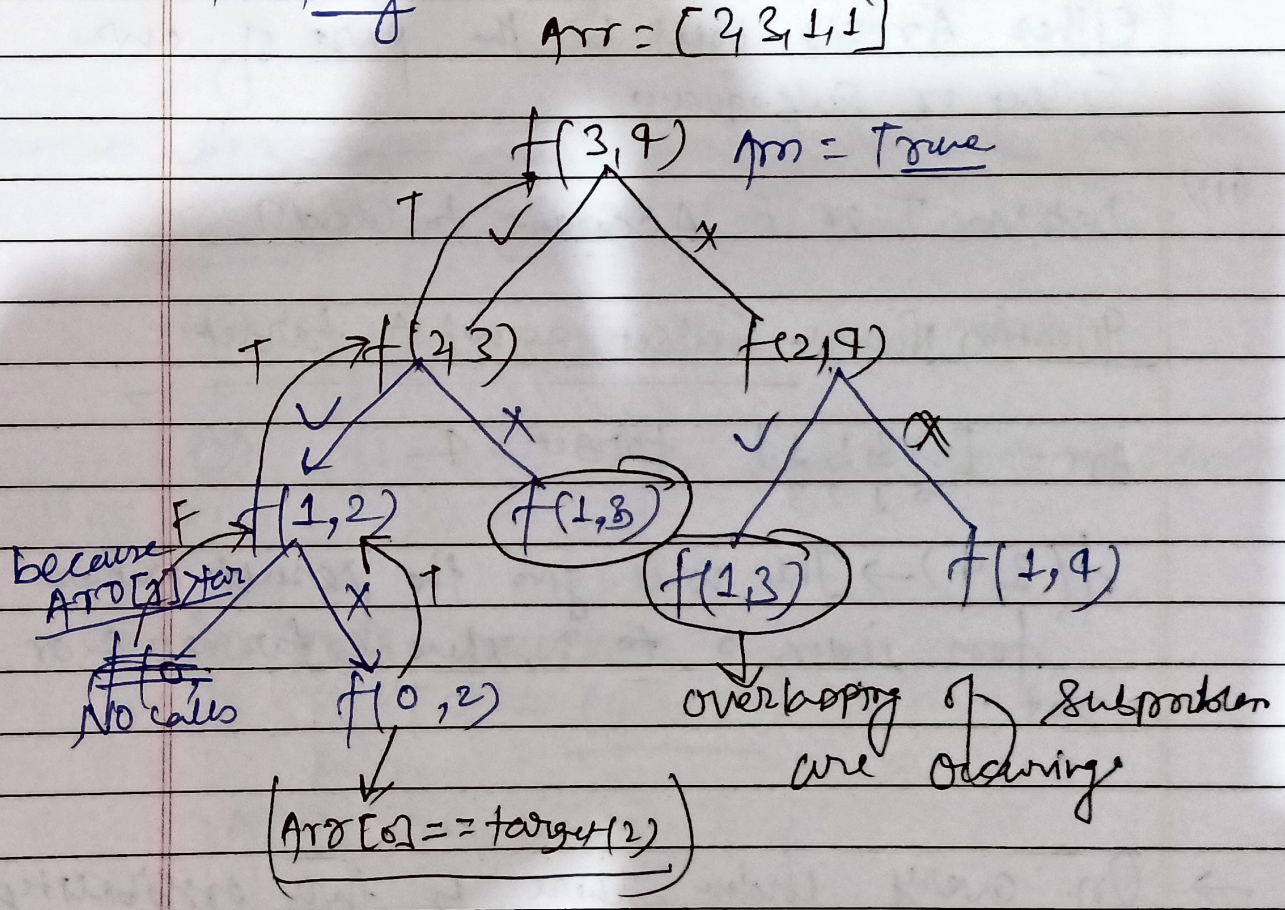
Arr = [4, 3, 1, 1] target = 4
0 1 2 3

f(3, 4) → This will give the result wether from index 0 to 3, there is target = 4 or not.

→ On every index, there is two possibility Wether we should include in our answer or not. And Accordingly need to handle the target

Let's see recursive tree

→ Base case will be
When target = 0 and i >= 0
if (i == 0)
return (arr[i] == target);
Because i = 0 means only one element in the Array.



f(1,2) = ? target
this function indicate that from index 0 to 1 there is target = 2 or not.
or
→ [2, 3], target = 2
0 1

```cpp
/*
You are given an array/list ‘ARR’ of ‘N’ positive integers and an integer ‘K’. Your task is to check if there exists a subset in ‘ARR’ with a sum equal to ‘K’.
Note: Return true if there exists a subset with sum equal to ‘K’. Otherwise, return false.
For Example :
If ‘ARR’ is {1,2,3,4} and ‘K’ = 4, then there exists 2 subsets with sum = 4. These are {1,3} and {4}. Hence, return true.
*/
#include<bits/stdc++.h>
using namespace std;

/*
In recursive we are starting from index n-1 to 0 that is top down appraoch  .
f(n-1,target)=this will give the answer wether any subset available equal to target or not .

In subset or subsequence problem use  inclusion or exclusion method .
*/
int recursiveTarget(vector<int>&input,int n ,int target)
{
    if(target==0)
    {
        return 1;
    }
    if(n==0)
    {
        return input[n]==target?1:0;
        // Agar array in ek hi element hai to oo simply target ke hi equal hi hona chahiye otherwise return false
    }
    int taken=0;
    int notaken=0;
    // for any particular , we have two option wether we take or not take the element
    notaken=recursiveTarget(input,n-1,target);

    if(input[n]<=target)
    {
        taken=recursiveTarget(input,n-1,target-input[n]);
    }

    return taken||notaken;
}

int tabulationCheck(vector<int> &input, int target)
{
    int n=input.size();
    vector<vector<int>>dp(n,vector<int>(target+1,0));
    /*
    dp[i][j]= the will give the answer by taking index [0....i]. any subset equal to target(j) or not

   for base case
   if target ==0 , its is possible everywhere because empty subset .

   if(i==0) then dp[0][arr[0]]=true; because of only one element are there so that element will be only true.
    */
```



```cpp
    for(int i=0;i<n;i++)
    {
        dp[i][0]=1;
    }
    dp[0][input[0]]=1;
    for(int i=1;i<n;i++)
    {
        for(int j=1;j<=target;j++)
        {
            bool nottaken = dp[i - 1][j];
            bool taken = false;
            if (input[i] <= j)
            {
                taken = dp[i - 1][j - input[i]];
            }
            dp[i][j] = taken || nottaken;
        }
    }
    return dp[n-1][target];
}


int main()
{
    int n;
    cin>>n;
    vector<int>input(n);
    for(int i=0;i<n;i++)
    {
        cin>>input[i];
    }
    int target;
    cin>>target;

    int ans1= recursiveTarget(input,n-1,target);
    cout<<ans1<<endl;
    int ans2= tabulationCheck(input,target);
    cout<<ans2<<endl;

}

/*
In recursion + memoization

time complexity: O(n*target)
Space complexity: O(n*target) + O(n) for recusion stack memory

In tabulation

time complexity: O(n*target)
Space complexity: O(n*target)


*/
```

15. Partition equal subset

Arr = [2, 3, 3, 8, 7, 5] target = 10

→ divide it into two subset such that sum of S_1 = sum of S_2.

$$S_1 = S_2 = \frac{S}{2}$$

→ if sum(Arr) is odd then Ans will be false

if sum is even then we should search for sum/2 as target in Arr Array.

→ same question as previous

① Array is given

② target = sum/2

```cpp
/*
Given a non-empty array nums containing only positive integers, find if the array can be partitioned into two subsets such that the sum of elements in both subsets is equal.

Example 1:

Input: nums = [1,5,11,5]
Output: true
Explanation: The array can be partitioned as [1, 5, 5] and [11].
*/
#include <bits/stdc++.h>
using namespace std;

int recursiveTarget(vector<int> &input, int n, int target)
{
    if (target == 0)
    {
        return 1;
    }
    if (n == 0)
    {
        return input[n] == target ? 1 : 0;
    }
    int taken = 0;
    int notaken = 0;

    notaken = recursiveTarget(input, n - 1, target);

    if (input[n] <= target)
    {
        taken = recursiveTarget(input, n - 1, target - input[n]);
    }

    return taken || notaken;
}

int tabulationCheck(vector<int> &input, int target)
{
    int n = input.size();
    vector<vector<int>> dp(n, vector<int>(target + 1, 0));

    for (int i = 0; i < n; i++)
    {
        dp[i][0] = 1;
    }
    dp[0][input[0]] = 1;
    for (int i = 1; i < n; i++)
    {
        for (int j = 1; j <= target; j++)
        {
            bool nottaken = dp[i - 1][j];
            bool taken = false;
            if (input[i] <= j)
            {
                taken = dp[i - 1][j - input[i]];
            }
            dp[i][j] = taken || nottaken;
        }
```



```cpp
    }
    return dp[n - 1][target];
}

int main()
{
    int n;
    cin >> n;
    vector<int> input(n);
    int sum=0;
    for (int i = 0; i < n; i++)
    {
        cin >> input[i];
        sum+=input[i];
    }
    int target;

    if(sum%2==1)
    {
        cout<<0<<endl;
        return 0;
    }
    else
    {
        target=sum/2;
    }
    int ans1 = recursiveTarget(input, n - 1, target);
    cout << ans1 << endl;
    int ans2 = tabulationCheck(input, target);
    cout << ans2 << endl;
}
/*
complexity will be same as previous question
*/

```

16.

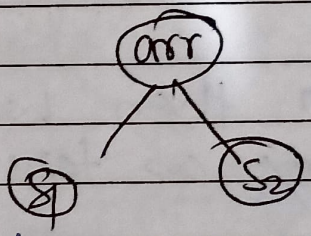
Partition A Set into two subsets with minimum Absolute difference.

Arr = [1, 2, 3, 7]

{1, 2} {3, 7} = |7-3| = 4

{1, 3} {2, 7} = |6-4| = 2

✓ {1, 7} {2, 3} = |5-5| = 0



Arr = [3, 2, 7]

Abs |S1-S2| is minimal

	0	1	2	3	4	5	6	7	8	9	10	11	12
fill 0th	0	T											
fill 1st ele	1	T											
fill 2nd ele	2	T	F	T	T	X	T	X	T	X	T	T	X	T

Here I have made dp[n][sum+1]

I have filled last row to get the answer.

In Array [3, 2, 7] Sum = 12

from dp table we set like
subset
possible sum using whole Array : last row of dp Array

→ 0 1 2 3 4 5 6 7 8 9 10 11 12
✓ ✗ ✓ ✓ ✗ ✓ ✗ T ✗ ✓ ✓ ✗ ✓

→ possible sum
S1 = 0 2 3 5 7 9 10 12

→ ~~if S1 = 0, S2 = 12 - 0 = 12~~

S1	(Sum - S1) S2	Abs diff
0	12	12
2	10	8
3	9	6
5	7	2 → min
7	5	2
9	3	6
10	2	8
12	0	12

Repeating

If sequence getting Repeat we traverse till half the sum only

→ It's just extension of problem 14.

* make dp[n][sum/2] And Col. min from last row in dp table

```
int ans = INT_MAX
for(int i=0 ; i<= sum/2 ; i++)
    ans = min(ans, abs(sum - dp[n-1][i]))
```

Code part

```cpp
/*
Given a set of integers, the task is to divide it into two sets S1 and S2 such that the absolute difference between their sums is minimum.

If there is a set S with n elements, then if we assume Subset1 has m elements, Subset2 must have n-m elements and the value of abs(sum(Subset1) – sum(Subset2)) should be minimum.
*/
#include <bits/stdc++.h>
using namespace std;

int tabulationCheck(vector<int> &input, int target)
{
    int n = input.size();
    vector<vector<int>> dp(n, vector<int>(target + 1, 0));

    for (int i = 0; i < n; i++)
    {
        dp[i][0] = 1;
    }
    dp[0][input[0]] = 1;
    for (int i = 1; i < n; i++)
    {
        for (int j = 1; j <= target; j++)
        {
           bool nottaken=dp[i-1][j];
           bool taken=false;
              if(input[i]<=j)
              {
                taken=dp[i-1][j-input[i]];
              }
            dp[i][j]=taken||nottaken;
        }
    }

    int ans = INT_MAX;
    target--;//we have already done target=sum+1; that is why we are decreasing it .


    for (int i = 0; i <= target / 2; i++)
    {
        if (dp[n - 1][i])
        {
            int s1=i;
            int s2=(target-i);
            ans= min(ans,abs(s1-s2));
        }
    }

    return ans;
}

int main()
{
    int n;
    cin >> n;
    vector<int> input(n);
    int sum = 0;
    for (int i = 0; i < n; i++)
    {
        cin >> input[i];
        sum += input[i];
    }
    /*
    As decribed above , we will use the last row in Dp table and out target will shift to sum+1; j++

    that means

    1. array will be there
    2. target= sum+1;

    */
   int target = sum + 1;

    int ans2 = tabulationCheck(input, target);
    cout << ans2 << endl;
}

```

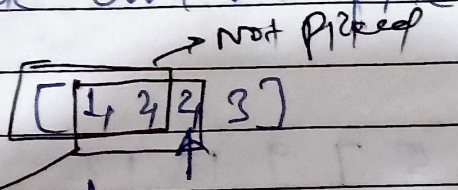
17. Number of Subsets

[1, 2, 2, 3] target = 3

→ Count the number of subset in array

{1, 2}, {1, 2}, {3} = 2 subsets

Pick and not pick

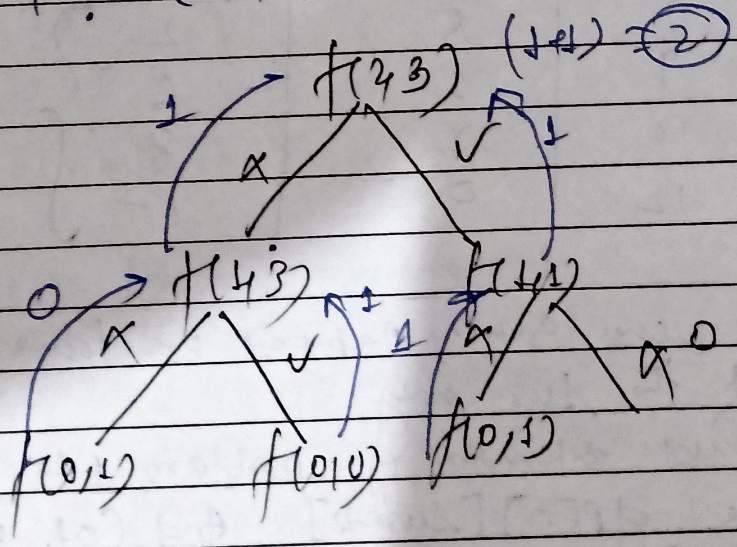


for index = 2 decision like pick and not pick.

→ for every element two choices

then Complexity $O(2^n)$

arr = [1, 2, 2] target = 3



```cpp
/*
Given an array ‘ARR’, partition it into two subsets (possibly empty) such that their union is the original array. Let the sum of the elements of these two subsets be ‘S1’ and ‘S2’.

Given a difference ‘D’, count the number of partitions in which ‘S1’ is greater than or equal to ‘S2’ and the difference between ‘S1’ and ‘S2’ is equal to ‘D’. Since the answer may be too large, return it modulo ‘10^9 + 7’.
*/
#include <bits/stdc++.h>
using namespace std;

int recursiveTarget(vector<int> &input, int n, vector<vector<int>> &memo, int target)
{
    if (target == 0)
    {
        return 1;
    }
    if (n == 0)
    {
        return input[n] == target;
    }

    if (memo[n][target] != -1)
    {
        return memo[n][target];
    }
    int nottaken = 0;
    int taken = 0;
    nottaken = recursiveTarget(input, n - 1, memo, target); // that means 0 to n-1th index tak kitna subset hai
    if (input[n] <= target)
    {
        taken = recursiveTarget(input, n - 1, memo, target - input[n]); // that means nth index kitna subset hai
    }
    memo[n][target] = taken + nottaken;

    return memo[n][target];
}

int tabulationCheck(vector<int> &input, int target)
{
    int n = input.size();
    vector<vector<int>> dp(n, vector<int>(target + 1, 0));
    for (int i = 0; i < n; i++)
    {
        dp[i][0] = 1;
    }
    dp[0][input[0]] = 1;
    for (int i = 1; i < n; i++)
    {
        for (int j = 1; j <= target; j++)
        {
            int nottaken = dp[i - 1][j];
            int taken = 0;
            if (input[i] <= j)
            {
                taken = dp[i - 1][j - input[i]];
            }
            dp[i][j] = taken + nottaken;
        }
    }
    return dp[n - 1][target];
}

int main()
{
    int n;
    cin >> n;
    vector<int> input(n);
    int difference;
    int s1pluss2=0;
    for (int i = 0; i < n; i++)
    {
        cin >> input[i];
        s1pluss2+=input[i];
    }
    cin >>difference;
    int s1minus2=difference;

    int s1= s1minus2+s1pluss2;
    if(s1%2)
    {
        cout<<0<<" "<<0<<endl;
        return 0;
    }
    int target=s1/2;
    vector<vector<int>> memo(n, vector<int>(target + 1, -1));

    int ans1 = recursiveTarget(input, n - 1, memo, target);

    int ans2 = tabulationCheck(input, target);
    cout << ans1 << " " << ans2 << endl;
}
```

18. Count partition With Given Difference

for a given Array we have to find

$S_1 - S_2 = d$

$S_1 + S_2 =$ Sum of all elements

$S_1 - S_2 = d$

$S_1 + S_2 = Sum$

$2S_1 = Sum + d$

$\Rightarrow S_1 = \left(\frac{Sum+d}{2}\right)$

So, now our question is like finding target $= \frac{(Sum+d)}{2}$ in Array.

Input:

Array
target $= \left(\frac{Sum+d}{2}\right)$

just apply our question Concept.

19. 0/1 Knapsack

Rules

→ Explore in terms of (ind, W).

→ Explore all possible pick and not picked.

→ Max (all possibility)

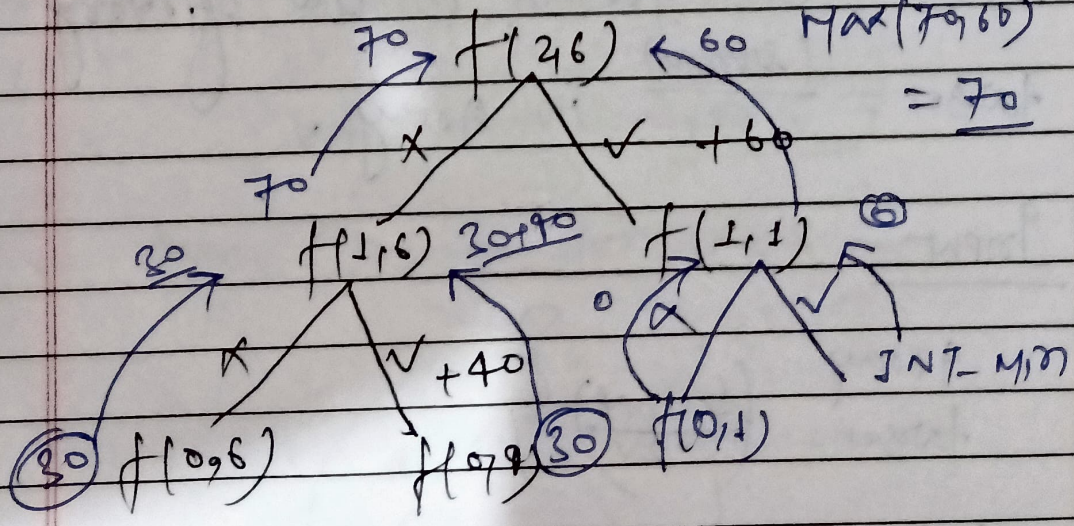
- for all possible combination we use recursion

Example

wt → 3 2 5 weight = 6

val → 30 40 60

0 1 2



Rules for Tabulation

1. Base Case
2. Use for loops according to the states
3. Copy recurrence relation

```cpp
/*
Problem Statement

A thief is robbing a store and can carry a maximal weight of W into his knapsack. There are N items and the ith item weighs wi and is of value vi. Considering the constraints of the maximum weight that a knapsack can carry, you have to find and return the maximum value that a thief can generate by stealing items.
*/
#include<bits/stdc++.h>
using namespace std;

int recursiveTarget( vector<int>&weight,vector<int>&value,int n ,vector<vector<int>>&memo,int capacity)
{
    if(n==0)
    {
        if(weight[n]>capacity)
            return 0;
        else
            return value[n];
    }
    if(capacity==0)
        return 0;

    if(memo[n][capacity]!=-1)
        return memo[n][capacity];

    int Exclude =  recursiveTarget(weight,value,n-1,memo,capacity);
    int Include = 0;
    if(weight[n]<=capacity)
        Include = value[n] + recursiveTarget(weight,value,n-1,memo,capacity-weight[n]);
    memo[n][capacity] = max(Exclude,Include);
    return memo[n][capacity];

}

int tabulationCheck(vector<int>&weight,vector<int>&value, int capacity)
{
    int n = weight.size();
    vector<vector<int>> dp(n,vector<int>(capacity+1,0));

    // Base case accroding to the recursive one
    for(int i=weight[0];i<=capacity;i++)
    {
        dp[0][i] = value[0];
    }
    if(capacity==0)
    {
        return 0;
    }

    for(int i=1;i<n;i++)
    {
        for(int j=0;j<=capacity;j++)
        {
            int taken=0;
            int notTaken=0;
            if(weight[i]<=j)
            {
                taken = value[i] + dp[i-1][j-weight[i]];
            }
            notTaken = dp[i-1][j];
            dp[i][j] = max(taken,notTaken);
        }
    }
    return dp[n-1][capacity];

}

int main()
{
    int n;
    cin >> n;
    vector<int>weight(n);
    vector<int>value(n);
    for(int i=0;i<n;i++)
    {
        cin>>weight[i];
    }
    for(int i=0;i<n;i++)
    {
        cin>>value[i];
    }
    int capacity;
    cin>>capacity;
    vector<vector<int>>memo(n,vector<int>(capacity+1,-1));
    int ans=recursiveTarget(weight,value,n-1,memo,capacity);
    cout<<ans<<endl;
    int ans2=tabulationCheck(weight,value,capacity);
    cout<<ans2<<endl;
    return 0;

}
```

20.

Minimum Coins

arr = [1, 4, 5] target = 7

(5, 1, 1) → len = 3 } Min = 3
(4, 2, 1) → len = 4 }

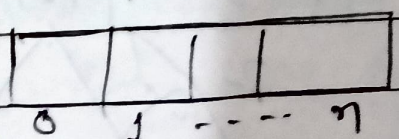
→ Try all possible combos, then return min of all possible combos

→ Recursion will come in

→ find out min coins required to represent the target

f(n, T) → What is minimum no of coins required to represent T

Counting the coins in given array from index 0 to n.



INFINITE

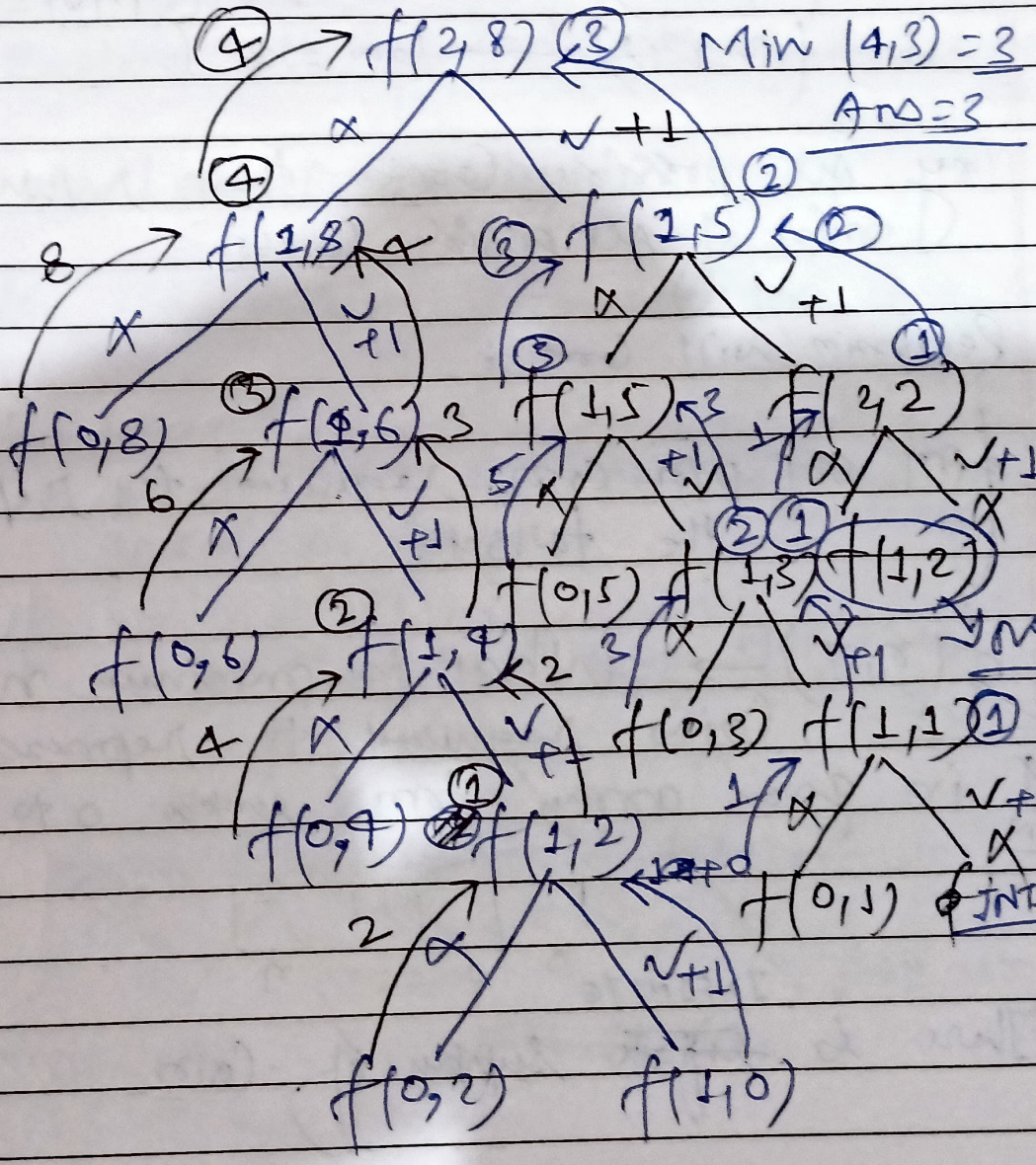
There is ~~infinite~~ supply of coins

→ For the base case always think about last element / single element.

Lets make tree diagram

arr = {1,2,3}

target = 8



→ This may look confusing but if you go through the tree you will get the tree properly

```cpp
/*
Bob went to his favourite bakery to buy some pastries. After picking up his favourite pastries his total bill was P cents. Bob lives in Berland where all the money is in the form of coins with denominations {1, 2, 5, 10, 20, 50, 100, 500, 1000}.

Bob is not very good at maths and thinks fewer coins mean less money and he will be happy if he gives minimum number of coins to the shopkeeper. Help Bob to find the minimum number of coins that sums to P cents (assume that Bob has an infinite number of coins of all denominations).
*/
#include<bits/stdc++.h>
using namespace std;

int recursiveCount(vector<int> &value, int target, int n,vector<vector<int>>&memo)
{
    // we are starting from last element then always think about first element for the base case ;
    if(n==0)
    {
      if(target%value[n]==0)
      {
        return target/value[n];
      }
      else
      {
        return 1e9;
      }
    }
     if(memo[n][target]!=-1)
    {
        return memo[n][target];
    }

    int taken=1e9;
    int notTaken=1e9;

    notTaken = recursiveCount(value, target, n-1,memo);
    if(value[n]<=target)
    {
        taken = recursiveCount(value, target - value[n], n,memo) + 1;
    }
    return memo[n][target]= min(taken,notTaken);
}
int tabulationCount(vector<int>&value,int target)
{
    int n = value.size();
    int dp[n][target+1];
    // think about the last element ;
    for(int i=0;i<=target;i++)
    {
        dp[0][i] = (i%value[0]==0)?(i/value[0]):1e9;
    }
    // we have already processed the 0th element then we need to start i==1
    for(int i=1;i<n;i++)
    {
        for(int j=0;j<=target;j++)
        {
            // After writing the different states of DP then just copy the recurrence relation
            int taken = 1e9;
            int notTaken = 1e9;

            notTaken = dp[i-1][j];
            if (value[i] <= j)
            {
                taken = dp[i][j-value[i]] + 1;
            }
            dp[i][j]= min(taken, notTaken);
        }
    }
    // due to bottom up result will store at (n-1) and target
    return dp[n-1][target];

}


int main()
{
    int target;
    cin>>target;

    vector<int>value{1, 2, 5, 10, 20, 50, 100, 500, 1000};
     int n= value.size();
    vector<vector<int>>memo(n,vector<int>(target+1,-1));
    int ans1= recursiveCount(value,target,n-1,memo);
    cout<<ans1<<endl;

    int ans2=tabulationCount(value,target);
    cout<<ans2<<endl;
}

/*
In recursion + memoization

time complexity : O(n*target);
space complexity : O(n*target) + recursion stack


In tabulation

time complexity : O(n*target);
space complexity : O(n*target)

*/
```

21. Target Sum

Arr[] = {1, 1, 2, 1}
target = 3

You have to assign it +1, -1 to every element to get target = 3.

+ + - - = -1
1 2 3 1
- + + - = 3

Q. How many ways we can assign the sign to achieve the given target.

→ We have already done this question before.
∴ Count the Subset with given difference.

Divide them into 2 subset named S1 and S2
Every element the part of S1 and S2

Then problem resolve like

[Sum(S1) - Sum(S2) = target]

↓

Such type of question we have already solved

22. Coin change 2

→ In a given Array you need to count the number of ways to represent target provided supply of the coin is infinite

Arr = [1, 2, 3, 9]
target = 4.

Here again we use principle of inclusion and exclusion

→ if we are adding any element to our answer then we can include same element in the answer again.

→ if we are not including to answer then we can move to next index.

You can see demonstration in the code section

```cpp
/*
You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money.

Return the number of combinations that make up that amount. If that amount of money cannot be made up by any combination of the coins, return 0.

You may assume that you have an infinite number of each kind of coin.
*/

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

int recursiveCount(vector<int>&value,int target,int n,vector<vector<int>>&memo)
{
    // base case just think about the case when n==0;that means only one element are there .
    if(n==0)
    {
        return target%value[0]==0;
    }

    if(memo[n][target]!=-1)
    {
        return memo[n][target];
    }
    int notinclude=0;
    int include=0;
    notinclude=recursiveCount(value,target,n-1,memo);
    if(value[n]<=target)
    {
        include=recursiveCount(value,target-value[n],n,memo);
    }
    return memo[n][target]=(notinclude+include);
}

int tabulationCount(vector<int>&input,int target)
{
    int n=input.size();
    vector<vector<int>>dp(n,vector<int>(target+1,0));
    for(int i=0;i<=target;i++)
    {
        dp[0][i]=(i%input[0]==0);
    }

    for(int i=1;i<n;i++)
    {
        for(int j=0;j<=target;j++)
        {
            int notinclude=0;
            int include=0;
            notinclude=dp[i-1][j];
            if(input[i]<=j)
            {
                include=dp[i][j-input[i]];
            }
            dp[i][j]=notinclude+include;
        }
    }
```



```cpp
    return dp[n-1][target];
}

int main()
{
   int n;
   cin>>n;
    vector<int>value(n);
    for(int i=0;i<n;i++)
    {
        cin>>value[i];
    }
    int target;
    cin>>target;
    vector<vector<int>>memo(n,vector<int>(target+1,-1));
    cout<<recursiveCount(value,target,n-1,memo)<<endl;
    cout<<tabulationCount(value,target)<<endl;
    return 0;
}
```

```cpp
/*
You are given ‘N’ items with certain ‘PROFIT’ and ‘WEIGHT’ and a knapsack with weight capacity ‘W’. You need to fill the knapsack with the items in such a way that you get the maximum profit. You are allowed to take one item multiple times.
For Example
Let us say we have 'N' = 3 items and a knapsack of capacity 'W' =  10
'PROFIT' = { 5, 11, 13 }
'WEIGHT' = { 2, 4, 6 }

We can fill the knapsack as:

1 item of weight 6 and 1 item of weight 4.
1 item of weight 6 and 2 items of weight 2.
2 items of weight 4 and 1 item of weight 2.
5 items of weight 2.

The maximum profit will be from case 3 i.e ‘27’. Therefore maximum profit = 27.
*/
#include<bits/stdc++.h>
using namespace std;

int recursiveCal(vector<int>&value,vector<int>&wt,vector<vector<int>>&memo,int n,int w)
{
    if(n==0)
    {
        int ans=(w/wt[0])*value[0];
        return ans;
    }
    if(memo[n][w]!=-1)
    {
        return memo[n][w];
    }
    int include=0;
    int exclude=0;
    exclude=recursiveCal(value,wt,memo,n-1,w);
    if(w>=wt[n])
    {
        include=recursiveCal(value,wt,memo,n,w-wt[n])+value[n];
    }
    memo[n][w]=max(include,exclude);
    return memo[n][w];
}

int tabulationCal(vector<int>&value,vector<int>&wt,int w)
{
    int n=value.size();
    vector<vector<int>>dp(n,vector<int>(w+1,0));
    for(int i=0;i<=w;i++)
    {
        dp[0][i]=(i/wt[0])*value[0];
    }
    for(int i=1;i<n;i++)
    {
        for(int j=0;j<=w;j++)
        {
            int include = 0;
            int exclude = 0;
            exclude = dp[i- 1][j];
```



```cpp
            if (j >= wt[i])
            {
                include = dp[i][j-wt[i]]+value[i];
            }
            dp[i][j]=max(include,exclude);
        }
    }
    return dp[n-1][w];
}

int main()
{
    int n,w;
    cin>>n>>w;
    vector<int> v(n);
    vector<int> wt(n);
    for(int i=0;i<n;i++)
    {
        cin>>v[i];
    }
    for(int i=0;i<n;i++)
    {
        cin>>wt[i];
    }
    vector<vector<int>>memo(n,vector<int>(w+1,-1));

    int ans1= recursiveCal(v,wt,memo,n-1,w);
    cout<<ans1<<endl;

    int ans2= tabulationCal(v,wt,w);
    cout<<ans2<<endl;

    return 0;

}
```

```cpp
/*
Problem Statement
Given a rod of length ‘N’ units. The rod can be cut into different sizes and each size has a cost associated with it. Determine the maximum cost obtained by cutting the rod and selling its pieces.

Note:
1. The sizes will range from 1 to ‘N’ and will be integers.

2. The sum of the pieces cut should be equal to ‘N’.

3. Consider 1-based indexing.
*/
#include <bits/stdc++.h>
using namespace std;

int recursiveCal(vector<int> &value, int index, int capacity, vector<vector<int>> &memo)
{
    // here length will index+1 because index is 0 based that is why we are doing +1;
    /*
    Here we are trying to make different combination of length to equal to capacity . That means reverse of the problem given;
    */
    //    base case if you have only one element then ? if you capacity=10 and n=0 then you have only one choice to cut the rod in length equal to 1. because index=0 is equal to length 1;
    if (index == 0)
    {
        return value[0] * capacity;
    }

    int include = 0;
    int exclude = 0;
    if (memo[index][capacity] != -1)
    {
        return memo[index][capacity];
    }
    exclude = 0 + recursiveCal(value, index - 1, capacity,memo);
    int length = index+ 1;
    if (capacity >= length)
    {
        include = recursiveCal(value, index, capacity - length,memo) + value[index];
    }
    return memo[index][capacity] = max(include, exclude);
}

int tabulationCheck(vector<int>&value)
{
    int n = value.size();
    int capacity = n + 1;
    vector<vector<int>>dp(n, vector<int>(capacity, 0));
    for(int i=0;i<=n;i++)
    {
        dp[0][i]=value[0]*i;
    }

    for(int i=1;i<n;i++)
    {
```



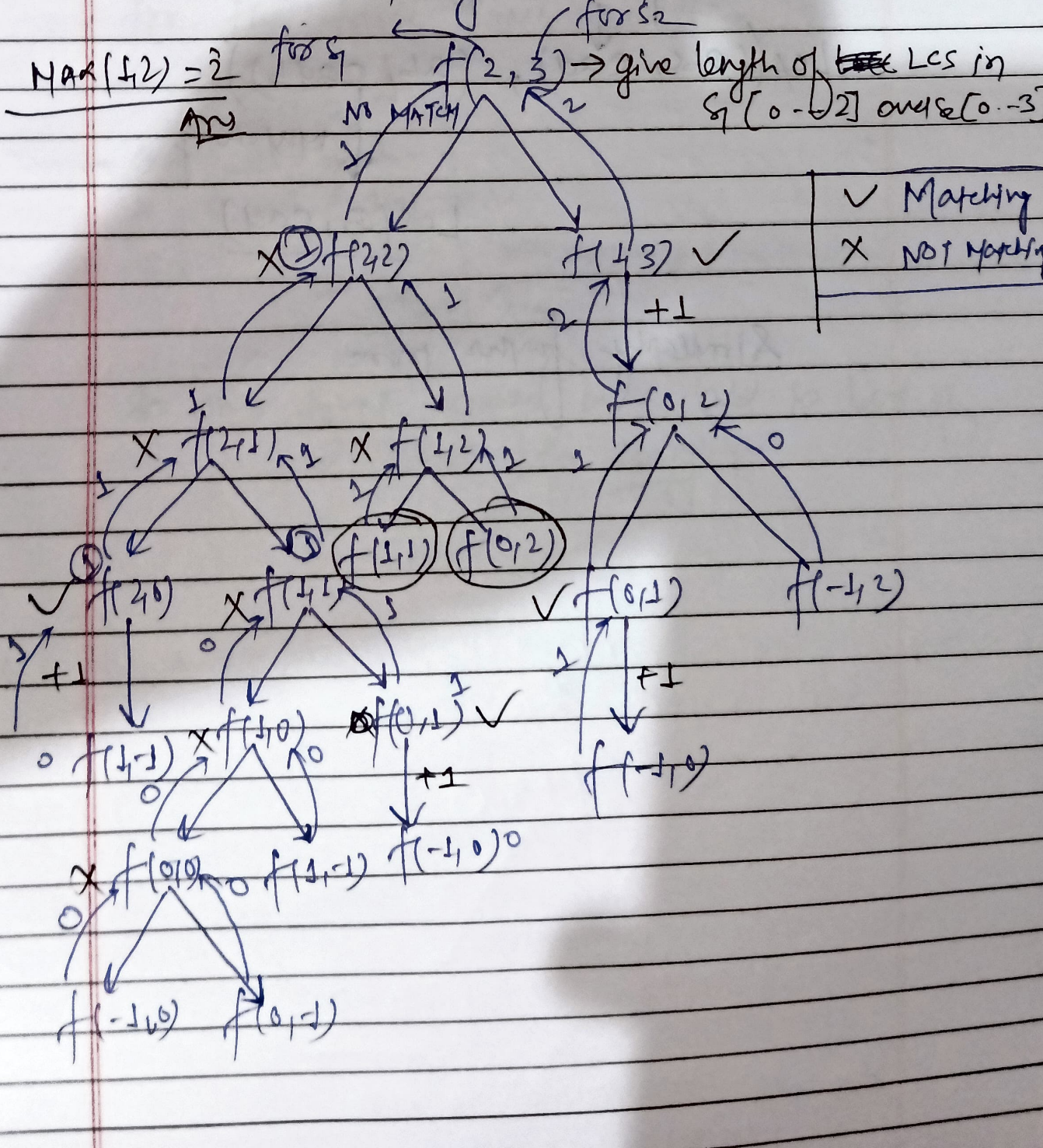
```cpp
        // states in dp , just think at i , what could be possible value of capacity i thin it would be 0 to n .
        for(int j=0;j<=n;j++)
        {
            int include = 0;
            int exclude = 0;
            exclude = 0 + dp[i-1][j];
            int length = i + 1;
            if (j >= length)
            {
                include = dp[i][j-length]+value[i];
            }
           dp[i][j] = max(include, exclude);
        }
    }
    return dp[n-1][n];
}

int main()
{
    int n;
    cin >> n;
    vector<int> v(n);
    for (int i = 0; i < n; i++)
    {
        cin >> v[i];
    }

    vector<vector<int>> memo(n, vector<int>(n + 1, -1));

    int ans1 = recursiveCal(v, n - 1, n, memo);
    cout << ans1 << endl;

    int ans2 = tabulationCheck(v);
    cout << ans2 << endl;
}
```

25. Longest Common Subsequence

→ $S_1 = abc$ $S_2 = cadb$

$LCS(S_1, S_2) =$ ab length = 2 Ans

Lets explore through the tree diagram.



if our strings matching at particular index then we decrement both of the index otherwise leave it

Lcs(abc, cadb)

Lcs((abc, cad), (ab, cadb))

matching

Lcs(a, cad)

Similarly further move

26. Print LCS

$S_1 = abc$ (n) $S_2 = cadb$ (m)

We are making dp of n+1, m+1 size because of ~~extra~~ negative index base case. thats why we are shifting it by 1 right side.

for this table will look like

		c	a	d	b
	0	0	0	0	0
a	0	0	1	1	1
b	0	0	1	1	2 → ✓
c	0	1	1	1	2 → Not Matching ✗

[3,4] index

So our final answer will be LCS of both the strings.

```cpp
/*
Given two strings text1 and text2, return the length of their longest common subsequence. If there is no common subsequence, return 0.

A subsequence of a string is a new string generated from the original string with some characters (can be none) deleted without changing the relative order of the remaining characters.

For example, "ace" is a subsequence of "abcde".
A common subsequence of two strings is a subsequence that is common to both strings.

Example 1:

Input: text1 = "abcde", text2 = "ace"
Output: 3
Explanation: The longest common subsequence is "ace" and its length is 3.
*/
#include <bits/stdc++.h>
using namespace std;

int recursiveLcs(string input1, string input2, int n, int m, vector<vector<int>> &dp)
{
    if (n < 0 || m < 0)
        return 0;
    if (dp[n][m] != -1)
        return dp[n][m];

    if (input1[n] == input2[m])
        return dp[n][m] = 1 + recursiveLcs(input1, input2, n - 1, m - 1,dp);
    else
        return dp[n][m] = max(recursiveLcs(input1, input2, n - 1, m,dp), recursiveLcs(input1, input2, n, m - 1,dp));
}

int tabulationLcs(string input1,string input2)
{
    int n = input1.length();
    int m = input2.length();
    vector<vector<int>> dp(n+1, vector<int>(m+1, 0));

    for(int i=0;i<=n;i++)
    {
        dp[i][0]=0;
    }
    for(int i=0;i<=m;i++)
    {
        dp[0][i]=0;
    }
    for(int i=1;i<=n;i++)
    {
        for(int j=1;j<=m;j++)
        {
            if(input1[i-1]==input2[j-1])
            {
                dp[i][j]=1+dp[i-1][j-1];
            }
            else
            {
                dp[i][j]=max(dp[i-1][j],dp[i][j-1]);
            }
        }
    }
    return dp[n][m];

}

int main()
{
    string input1, input2;
    cin >> input1 >> input2;
    vector<vector<int>> dp(input1.length() + 1, vector<int>(input2.length() + 1, -1));
    int ans1 = recursiveLcs(input1, input2, input1.size() - 1, input2.size() - 1, dp);
    cout << ans1 << endl;

    int ans2= tabulationLcs(input1,input2);
    cout << ans2 << endl;
}
```


6/28/22, 12:39 PM LongestCommonSubsequence.cpp

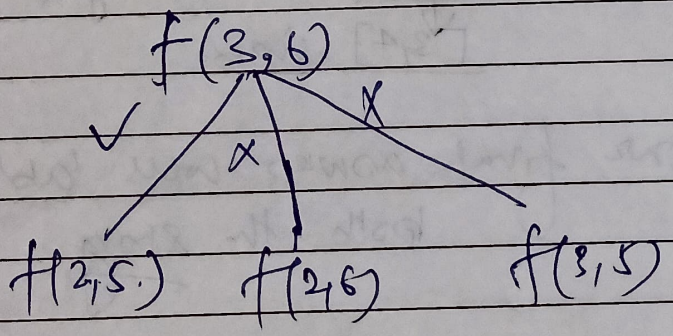
27. Longest Common Substring

→ $S_1 = \underset{0}{a}\underset{1}{b}\underset{2}{c}\underset{3}{d}\underset{4}{x}\underset{5}{y}\underset{6}{2}$

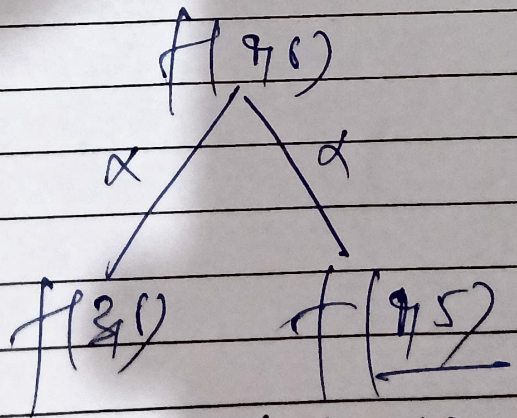
$S_2 = \underset{0}{x}\underset{1}{y}\underset{2}{2}\underset{3}{a}\underset{4}{b}\underset{5}{c}\underset{6}{d}$

then the longest Common substring will of length 4 (abcd)

In recursive, if matching, then we need to call in three branch.



→ If not matching then we need to call it in only two branches



In dp you need to do
dp[i][j] = 0 if not matching.

```cpp
/*
Given two strings ‘X’ and ‘Y’, find the length of the longest common substring.

Examples :

Input : X = “GeeksforGeeks”, y = “GeeksQuiz”
Output : 5
Explanation:
The longest common substring is “Geeks” and is of length 5.
*/
#include <bits/stdc++.h>
using namespace std;

int recursiveSubstring(string input1, string input2, int n, int m,
vector<vector<int>> &dp, int count)
{
    if (n < 0 || m < 0)
    {
        return count;
    }
    if (dp[n][m] != -1)
    {
        // return dp[n][m];
    }
    int temp = 0;
    if (input1[n] == input2[m])
    {
        temp = recursiveSubstring(input1, input2, n - 1, m - 1, dp, count + 1);
    }

    temp = max(temp, max(recursiveSubstring(input1, input2, n - 1, m, dp, 0),
recursiveSubstring(input1, input2, n, m - 1, dp, 0)));

    return dp[n][m] = temp;
}

int tabulationSubstring(string input1, string input2)
{
    int n = input1.length();
    int m = input2.length();
    vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));
    for (int i = 0; i <= n; i++)
    {
        dp[i][0] = 0;
    }
    for (int i = 0; i <= m; i++)
    {
        dp[0][i] = 0;
    }
    int ans = 0;
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= m; j++)
        {
            if (input1[i - 1] == input2[j - 1])
            {
                dp[i][j] = 1 + dp[i - 1][j - 1];
                ans = max(ans, dp[i][j]);
            }
```



```cpp
            else
            {
                dp[i][j] = 0;
            }
        }
    }
    return ans;
}

int main()
{
    string input1, input2;
    cin >> input1 >> input2;
    vector<vector<int>> dp(input1.length() + 1, vector<int>(input2.length() + 1,
-1));
    int ans1 = recursiveSubstring(input1, input2, input1.length() - 1,
input2.length() - 1, dp, 0);
    cout << ans1 << endl;

    int ans2 = tabulationSubstring(input1, input2);
    cout << ans2 << endl;
}
```

```cpp
/*
Given a string s. In one step you can insert any character at any index of the string.

Return the minimum number of steps to make s palindrome.

A Palindrome String is one that reads the same backward as well as forward.
*/
#include <bits/stdc++.h>
using namespace std;

int recursiveLcs(string input1, string input2, int n, int m, vector<vector<int>> &dp)
{
    if (n < 0 || m < 0)
        return 0;
    if (dp[n][m] != -1)
        return dp[n][m];

    if (input1[n] == input2[m])
        return dp[n][m] = 1 + recursiveLcs(input1, input2, n - 1, m - 1, dp);
    else
        return dp[n][m] = max(recursiveLcs(input1, input2, n - 1, m, dp), recursiveLcs(input1, input2, n, m - 1, dp));
}

int tabulationLcs(string input1, string input2)
{
    int n = input1.length();
    int m = input2.length();
    vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));

    for (int i = 0; i <= n; i++)
    {
        dp[i][0] = 0;
    }
    for (int i = 0; i <= m; i++)
    {
        dp[0][i] = 0;
    }
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= m; j++)
        {
            if (input1[i - 1] == input2[j - 1])
            {
                dp[i][j] = 1 + dp[i - 1][j - 1];
            }
            else
            {
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }
    return dp[n][m];
}

int main()
{
    string input1, input2;
```



```cpp
    cin >> input1;
    input2 = input1;
    reverse(input2.begin(), input2.end());
    vector<vector<int>> dp(input1.length() + 1, vector<int>(input2.length() + 1, -1));
    int ans1 = recursiveLcs(input1, input2, input1.size() - 1, input2.size() - 1, dp);
    cout <<input1.size()-ans1 << endl;

    int ans2 = tabulationLcs(input1, input2);
    cout << input1.size()-ans2 << endl;
}
```

28. Longest ~~Pali~~ Palindrome Subsequence (LPS)

→ Let suppose you have given String

~~input = "Raushan";~~

input = "raushan";

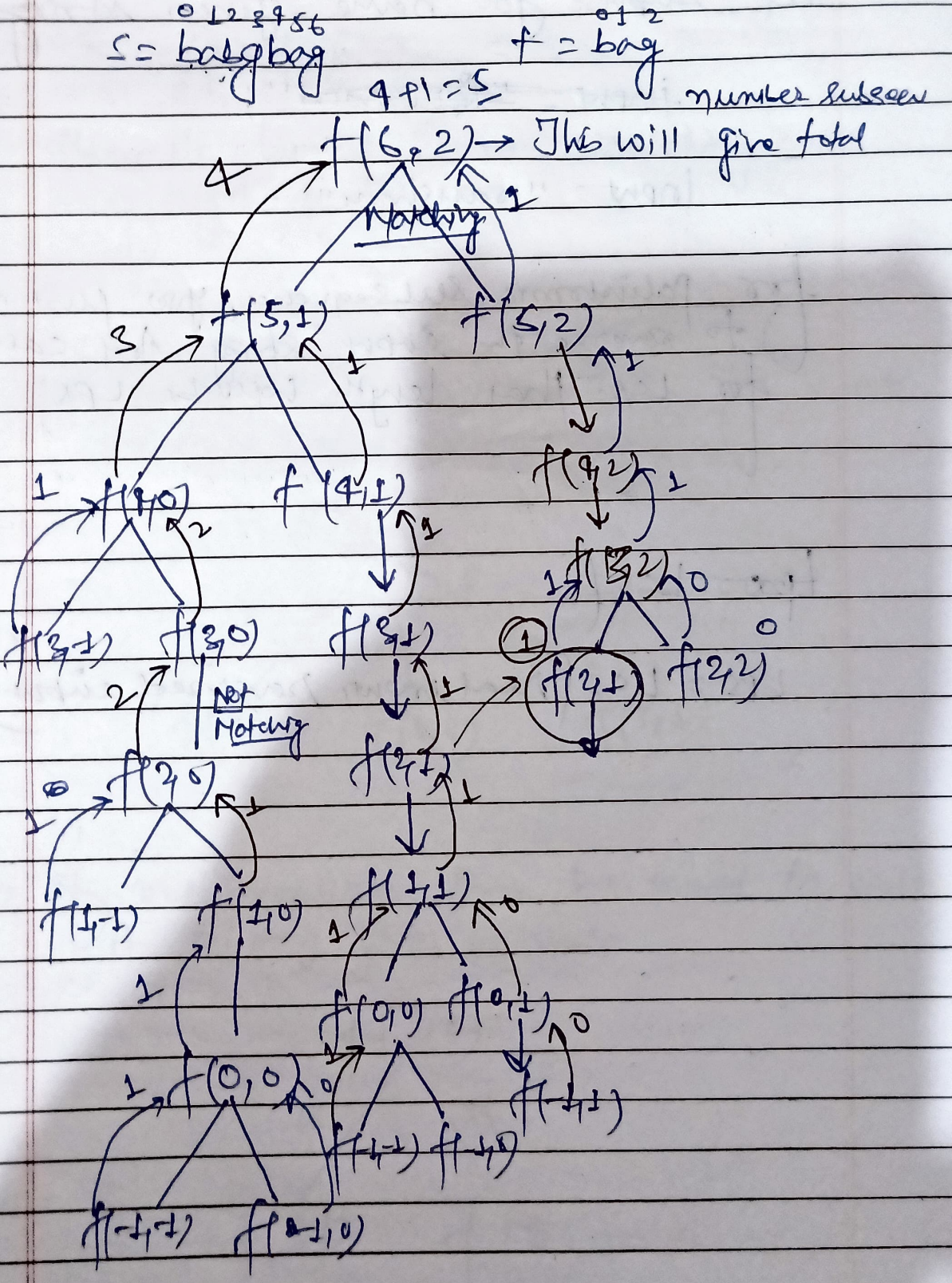
~~for~~ palindrome subsequence, you just need to reverse the input string and call for LCS. that length will be LPS.

~~LPS LCS(~~

LPS = LCS(real input, reversed input);

If Agar ek g Mil v gaya toh v chalega
Second g ko tone S Main Search kro

33. Distinct Subsequence



```cpp
/*
Input: s = "babgbag", t = "bag"
Output: 5
Explanation:
As shown below, there are 5 ways you can generate "bag" from S.
babgbag
babgbag
babgbag
babgbag
babgbag
*/

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

int recursiveDistinctSequence(string input1, string input2, int n, int m,
vector<vector<int>> &dp)
{
    if (m < 0)
    {
        return 1;
    }
    if (n < 0)
    {
        return 0;
    }

    if (dp[n][m] != -1)
    {
        return dp[n][m];
    }
    int ans = 0;
    if (input1[n] == input2[m])
    {
        ans = (recursiveDistinctSequence(input1, input2, n - 1, m - 1, dp) +
recursiveDistinctSequence(input1, input2, n - 1, m, dp));
    }
    else
    {
        ans = recursiveDistinctSequence(input1, input2, n - 1, m, dp);
    }
    return dp[n][m] = ans;
}

int tabulation(string input1, string input2)
{
    int n = input1.length();
    int m = input2.length();
    vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));

    for (int i = 0; i <= m; i++)
    {
        dp[0][i] = 0;
    }
    for (int i = 0; i <= n; i++)
    {
        // for any i whose j==0 then we should always put dp[i][0]=1;
                dp[i][0] = 1;
    }
```



```cpp
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= m; j++)
        {
            int ans = 0;
            if (input1[i - 1] == input2[j - 1])
            {
                ans = dp[i- 1][j- 1] + dp[i - 1][j];
            }
            else
            {
                ans = dp[i - 1][j];
            }
            dp[i][j] = ans;
        }
    }
    return dp[n][m];
}

int main()
{
    string input1, input2;
    cin >> input1 >> input2;

    int n = input1.length();
    int m = input2.length();
    vector<vector<int>> dp(n + 1, vector<int>(m + 1, -1));

    int ans1 = recursiveDistinctSequence(input1, input2, n - 1, m - 1, dp);
    cout << ans1 << endl;

    int ans2 = tabulation(input1, input2);
    cout << ans2 << endl;
}
```

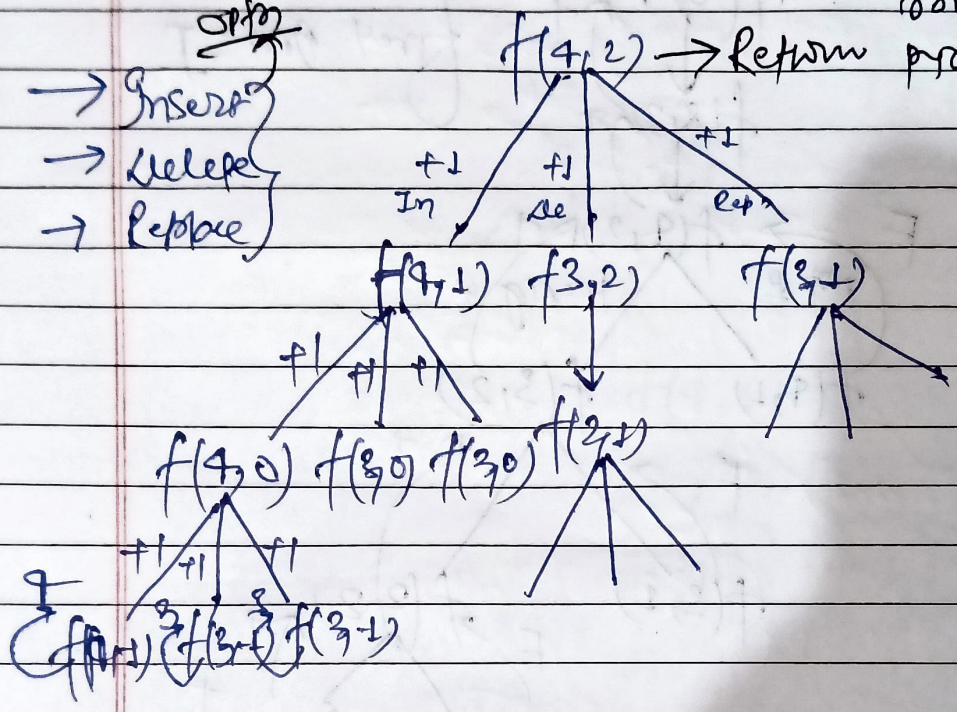
72. Edit distance

→ We have to change it from word1 → word2

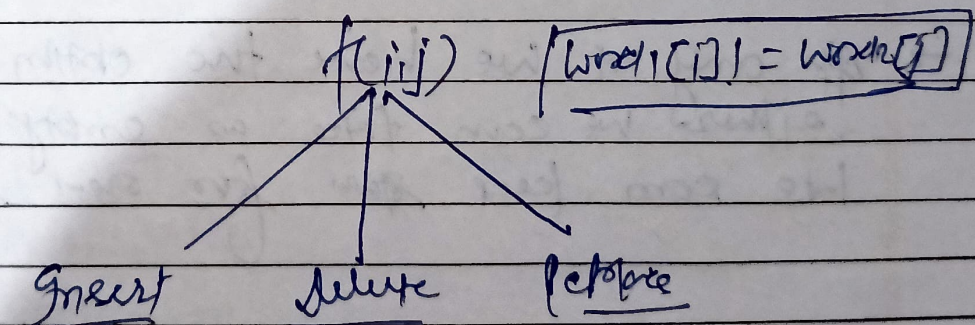
word1 = horse (0 1 2 3 4) word2 = ros (0 1 2)

→ Insert
→ Delete
→ Replace

opn



→ Similarly if character is not match for any given parameter index then to find option upon.



If match then go straight down by (i-1, j-1) index.

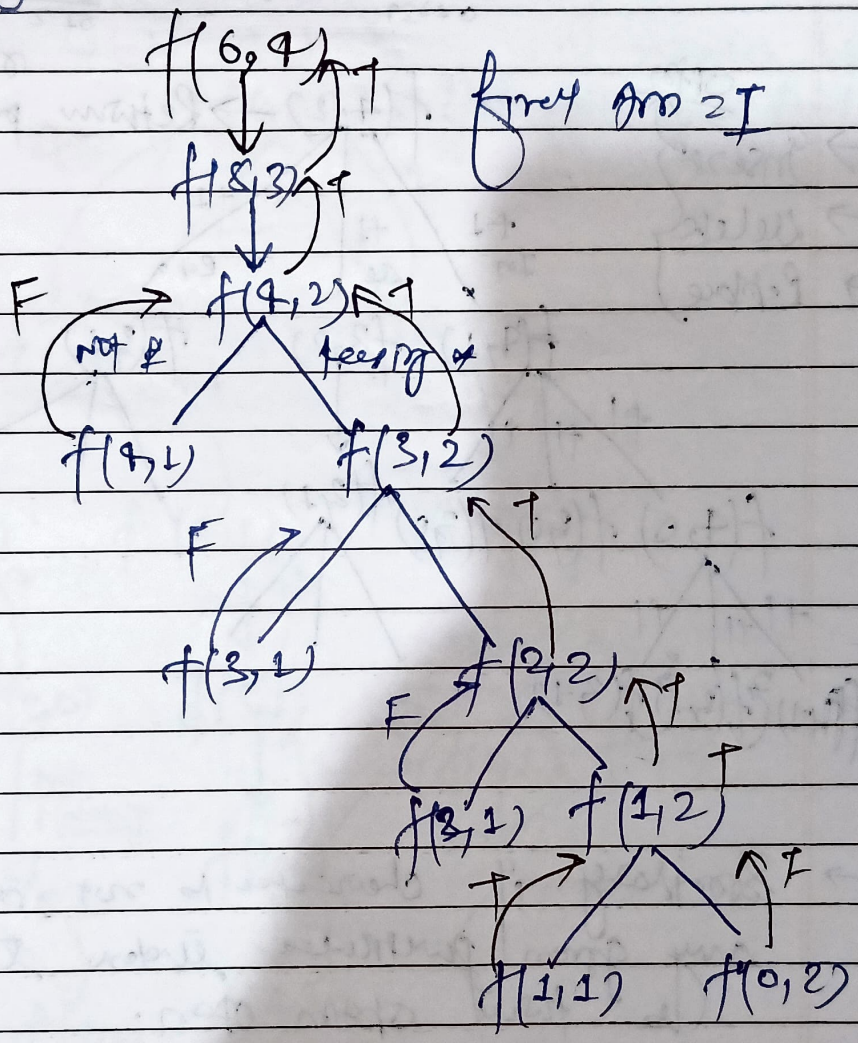
```cpp
/*
Given two strings word1 and word2, return the minimum number of operations required to convert word1 to word2.

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')
*/
#include<bits/stdc++.h>
using namespace std;

int recursiveEditDistance(string input1,string input2,int n,int m,vector<vector<int>>&dp)
{
   if(n<0 and m<0)
   {
    return 0;
   }
   if(n<0 and m>=0)
   {
    return m+1;
   }
    if(n>=0 and m<0)
    {
     return n+1;
    }
    if(dp[n][m]!=-1)
    {
     return dp[n][m];
    }
    int ans=0;
    if(input1[n]==input2[m])
    {
     ans=recursiveEditDistance(input1,input2,n-1,m-1,dp);
    }
    else
    {
     ans=min(recursiveEditDistance(input1,input2,n-1,m,dp),min(recursiveEditDistance(input1,input2,n,m-1,dp),recursiveEditDistance(input1,input2,n-1,m-1,dp)))+1;
    }
    return dp[n][m]=ans;
}

int tabulationEditDistance(string input1, string input2)
{
    vector<vector<int>>dp(input1.length()+1,vector<int>(input2.length()+1,0));
```



```cpp
    int n= input1.size();
    int m= input2.size();
    for(int i=0;i<=n;i++)
    {
        dp[i][0]=i+1;
    }
    for(int i=0;i<=m;i++)
    {
        dp[0][i]=i+1;
    }
    dp[0][0]=0;

    for(int i=1;i<=n;i++)
    {
        for(int j=1;j<=m;j++)
        {
            int ans=0;
            if(input1[i-1]==input2[j-1])
            {
                ans=dp[i-1][j-1];
            }
            else
            {
                ans=min(dp[i-1][j],min(dp[i][j-1],dp[i-1][j-1]))+1;
            }
            dp[i][j]=ans;
        }
    }
    return dp[n][m];
}

int main()
{
    string input1,input2;
    cin>>input1>>input2;
    vector<vector<int>> dp(input1.length()+1,vector<int>(input2.length()+1,-1));

    int ans=recursiveEditDistance(input1,input2,input1.length()-1,input2.length()-1,dp);
    cout<<ans<<endl;
    ans=tabulationEditDistance(input1,input2);
    cout<<ans<<endl;
}
```

36 Wild Card matching

String a b d e f c d pattern a b * c d
0 1 2 3 4 5 6 0 1 2 3 4



→ If any of we have two options either we can take as empty or we can keep * for next sequence

```cpp
#include <bits/stdc++.h>
using namespace std;
/*
  Given an input string (s) and a pattern (p), implement wildcard pattern matching with support for '?' and '*' where:
'?' Matches any single character.
'*' Matches any sequence of characters (including the empty sequence).
The matching should cover the entire input string (not partial).
Example 1:
Input: s = "aa", p = "a"
Output: false
Explanation: "a" does not match the entire string "aa".
*/
int recursiveWildCard(string input, string required, int n, int m, vector<vector<int>> &dp)
{
    if (n < 0 and m < 0)
        return 1;
    if (m >= 0 and n < 0)
    {
        return 0;
    }
    if (n >= 0 and m < 0)
    {
        bool flag = true;
        for (int i = 0; i <= m; i++)
        {
            if (input[i] != '*')
            {
                flag = false;
                break;
            }
        }
        if (flag)
            return 1;
        else
            return 0;
    }
    if (dp[n][m] != -1)
        return dp[n][m];

    if (input[n] == required[m] or input[n] == '?')
    {
        dp[n][m] = recursiveWildCard(input, required, n - 1, m - 1, dp);
    }

    else if (input[n] == '*')
    {
        dp[n][m] = recursiveWildCard(input, required, n - 1, m, dp) || recursiveWildCard(input, required, n, m - 1, dp);
    }
    else
    {
        dp[n][m] = 0;
    }
    return dp[n][m];
}
int tabulationWildCard(string input,string required)
{
```



```cpp
    int n = input.length();
    int m = required.length();
    vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));
     for(int i=0;i<=m;i++)
     {
        dp[0][i]=0;
     }
     for(int i=0;i<=n;i++)
     {
         int flag = 1;
         for (int j = 0; j <= i; j++)
         {
             if (input[j] != '*')
             {
                 flag = 0;
                 break;
             }
         }
            if (flag)
                dp[i][0] =1;
            else
                dp[i][0] = 0;
     }
     dp[0][0]=1;
    for(int i=1;i<=n;i++)
    {
        for(int j=1;j<=m;j++)
        {
            if (input[i] == required[j] or input[i] == '?')
            {
                dp[i][j] = dp[i - 1][j- 1];
            }

            else if (input[i] == '*')
            {
                dp[i][j] = dp[i - 1][j] || dp[i][j - 1];
            }
            else
            {
                dp[i][j] = 0;
            }
        }
    }
    return dp[n][m];
}
int main(){
    string input;
    string required;
    cin >> input;
    cin >> required;
    int n = input.length();
    int m = required.length();
    vector<vector<int>> dp(n + 1, vector<int>(m + 1, -1));
    int ans = recursiveWildCard(input, required, n - 1, m - 1, dp);
    int ans1= tabulationWildCard(input,required);
    if (ans1 and ans)
        cout << "YES";
    else
        cout << "NO";
}
```

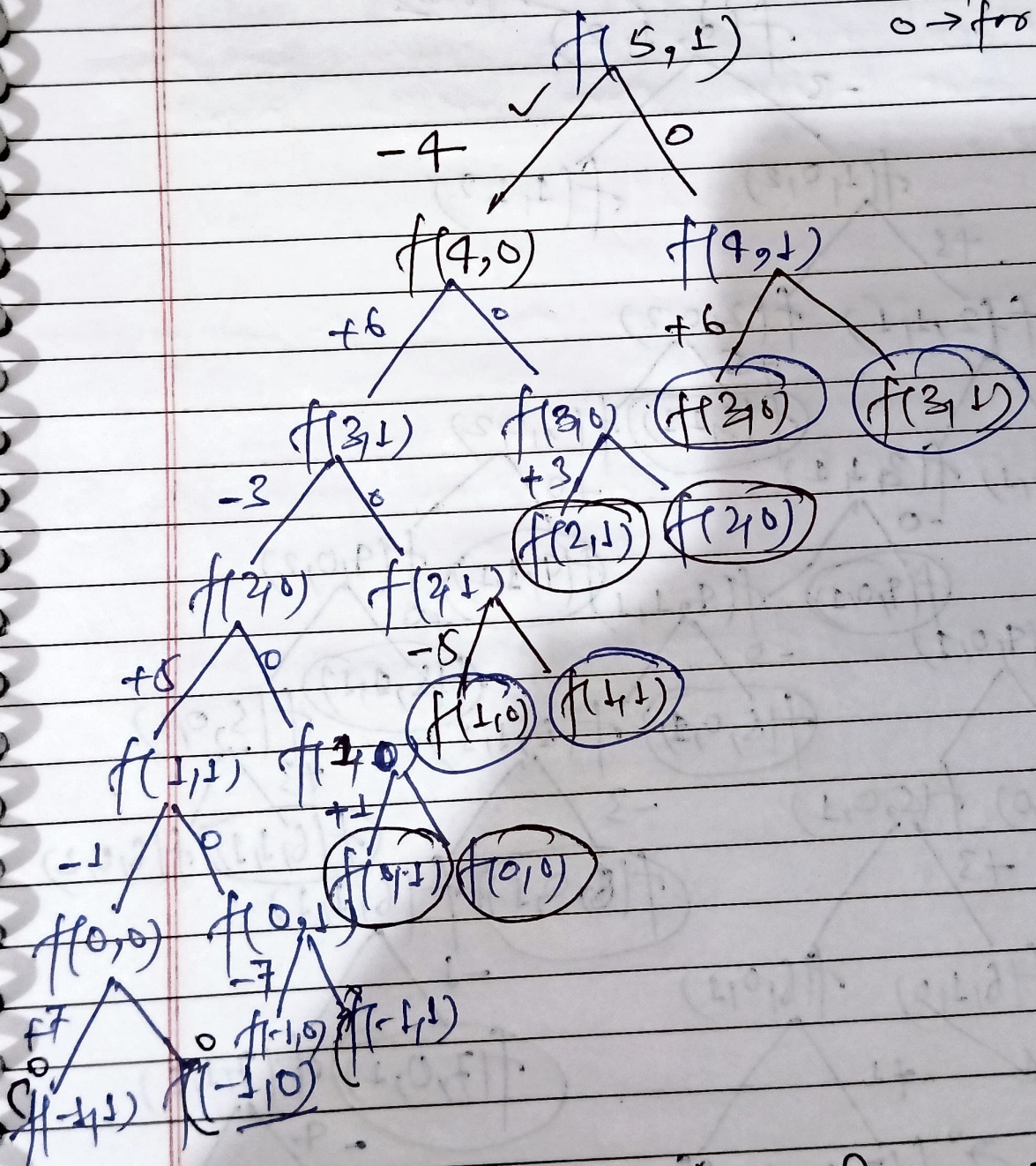
```cpp
/*
You are given an array prices where prices[i] is the price of a given stock on the ith day.

You want to maximize your profit by choosing a single day to buy one stock and choosing a different day in the future to sell that stock.

Return the maximum profit you can achieve from this transaction. If you cannot achieve any profit, return 0.
*/
#include<bits/stdc++.h>
using namespace std;

int BestTimeToBuyAndSellStock(vector<int>&prices)
{
    int n = prices.size();

    int maxe = prices[0];
    int mine = prices[0];

    int maxecost = 0;

    for (int i = 0; i < n; i++)
    {
        maxe = prices[i];

        maxecost = max(maxecost, maxe - mine);

        mine = min(mine, prices[i]);
    }

    return maxecost;
}

int main()
{
    int n;
    cin>>n;
    vector<int>input(n);
    for(int i=0;i<n;i++)
    {
        cin>>input[i];
    }
    int ans=BestTimetoBuyandSellStock(input);
    cout<<ans;
}

/*
time complexity: O(n)
space complexity: O(1)
*/

```

36. Buy and Sell Stock II

→ Every stock have two choices either sell or buy

Arr = [7, 1, 5, 3, 6, 4]

1 → for buying
0 → for selling



→ Because we have two choice 9 on two branch from every stocks.

```cpp
/*
You are given an integer array prices where prices[i] is the price of a given stock on the ith day.

On each day, you may decide to buy and/or sell the stock. You can only hold at most one share of the stock at any time. However, you can buy it then immediately sell it on the same day.

Find and return the maximum profit you can achieve.
*/
#include<bits/stdc++.h>
using namespace std;

int calculateans(vector<int> &price, int buyorsell, int index, vector<vector<int>> &dp)
{
    if (index == price.size())
    {
        return 0;
    }
    if (dp[index][buyorsell] != -1)
    {
        return dp[index][buyorsell];
    }
    if (buyorsell)
    {
        int ans = max(-price[index] + calculateans(price, !buyorsell, index + 1, dp), calculateans(price, buyorsell, index + 1, dp));
        return dp[index][buyorsell] = ans;
    }
    else
    {
        int ans1 = max(price[index] + calculateans(price, !buyorsell, index + 1, dp), calculateans(price, buyorsell, index + 1, dp));

        return dp[index][buyorsell] = ans1;
    }
}

int tabulation(vector<int> &price)
{
    int n = price.size();
    // vector<vector<int>>dp(n+1,vector<int>(2,0));
    vector<int> ahead(2, 0);
    vector<int> curr(2, 0);
    for (int index = n - 1; index >= 0; index--)
    {
        for (int buyorsell = 0; buyorsell <= 1; buyorsell++)
        {
            if (buyorsell)
            {
                int ans = max(-price[index] + ahead[!buyorsell], ahead[buyorsell]);
                curr[buyorsell] = ans;
            }
            else
            {
                int ans1 = max(price[index] + ahead[!buyorsell], ahead[buyorsell]);

                curr[buyorsell] = ans1;
            }
        }
        ahead = curr;
    }

    return ahead[1];
}

int main()
{
    int n;
    cin >> n;
    vector<int> price(n);
    for (int i = 0; i < n; i++)
    {
        cin >> price[i];
    }
    vector<vector<int>> dp(n + 1, vector<int>(2, -1));
    int ans = calculateans(price, 1, 0, dp);
    int ans1 = tabulation(price);
    cout << ans << " " << ans1;
    return 0;
}

/*
In recursive + memoization
    time complexity= O(n*m)
    space complexity= O(n*m) +recurisve Stack

In dp
  time complexity= O(n*2)
  space complexity= O(2)
*/
```

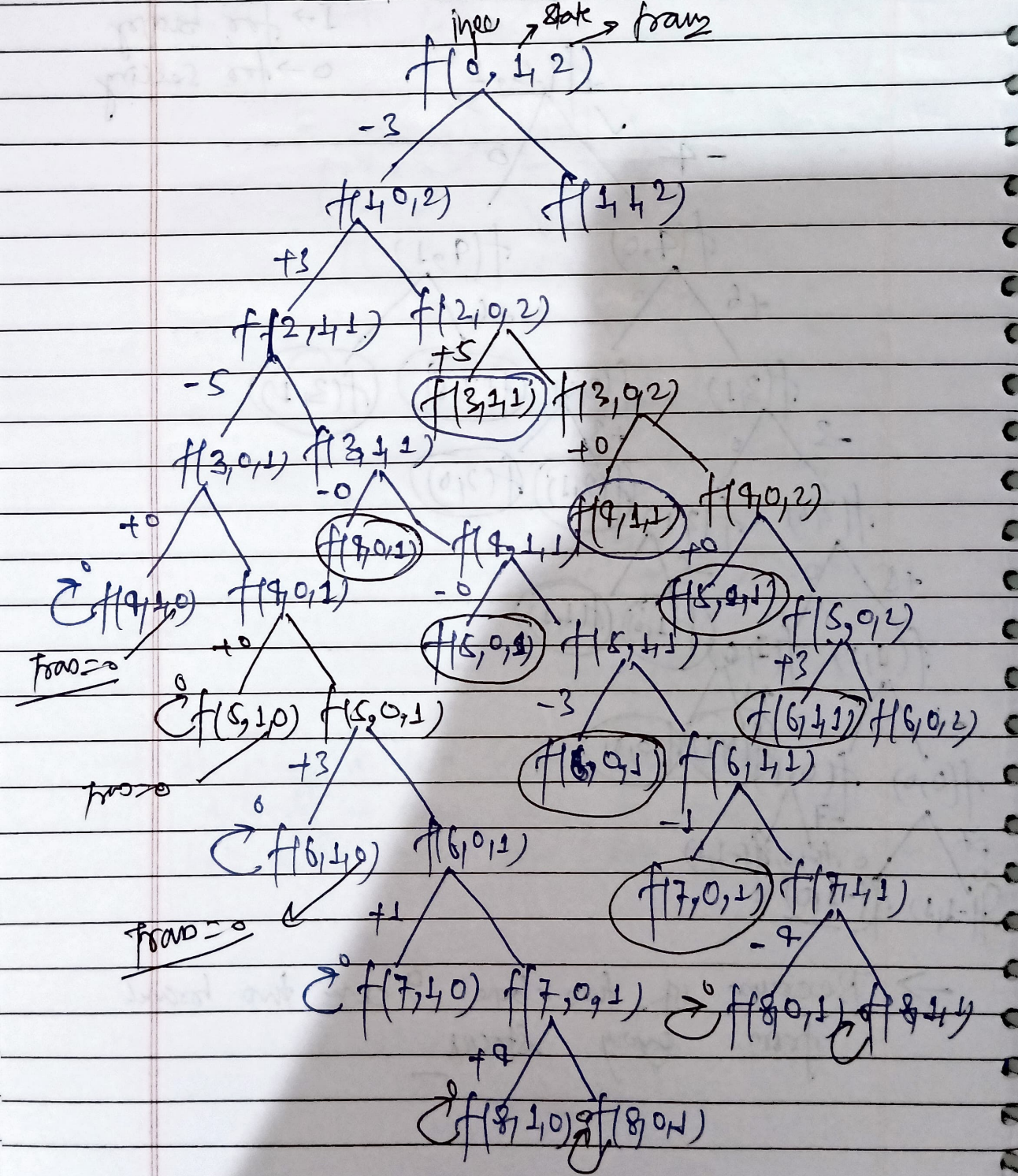
37.

Best time to buy and sell III

→ Here we need to take at most 2 transaction

0 1 2 3 4 5 6 7

Arr = [3, 3, 5, 0, 0, 3, 1, 4]



```cpp
/*
You are given an array prices where prices[i] is the price of a given stock on the ith day.

Find the maximum profit you can achieve. You may complete as many transactions as you like (i.e., buy one and sell one share of the stock multiple times) with the following restrictions:

After you sell your stock, you cannot buy stock on the next day (i.e., cooldown one day).
Note: You may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).
*/
#include<bits/stdc++.h>
using namespace std;
int calculateans(vector<int> &price, int buyorsell, int index, vector<vector<int>> &dp)
{
    if (index >= price.size())
    {
        return 0;
    }
    if (dp[index][buyorsell] != -1)
    {
        return dp[index][buyorsell];
    }
    if (buyorsell)
    {
        int ans = max(-price[index] + calculateans(price, !buyorsell, index + 1, dp), calculateans(price, buyorsell, index + 1, dp));
        return dp[index][buyorsell] = ans;
    }
    else
    {
        int ans1 = max(price[index] + calculateans(price, !buyorsell, index + 2, dp), calculateans(price, buyorsell, index + 1, dp));

        return dp[index][buyorsell] = ans1;
    }
}

int tabulation(vector<int> &price)
{
    int n = price.size();
    vector<vector<int>> dp(n + 2, vector<int>(2, 0));
    // space optimization
    //  vector<int>ahead(2,0);
    //  vector<int>curr(2,0);
    for (int index = n - 1; index >= 0; index--)
    {
        for (int buyorsell = 0; buyorsell <= 1; buyorsell++)
        {
            if (buyorsell)
            {
                int ans = max(-price[index] + dp[index + 1][!buyorsell], dp[index + 1][buyorsell]);
                dp[index][buyorsell] = ans;
            }
            else
```



```cpp
            {
                int ans1 = max(price[index] + dp[index + 2][!buyorsell], dp[index + 1][buyorsell]);
                dp[index][buyorsell] = ans1;
            }
        }
        // ahead=curr;
    }

    return dp[0][1];
}

int main()
{
    int n;
    cin>>n;
    vector<int>input(n);
    for(int i=0;i<n;i++)
    {
        cin>>input[i];
    }
    vector<vector<int>> dp(prices.size(), vector<int>(2, -1));
    int ans=calculateans(input,0,0,dp);
    cout<<ans<<endl;
    int ans1=tabulation(input);
    cout<<ans1<<endl;
}

/*
In recursion +memoization
  time complexity: O(n*2);
  space complexity: O(n*2)+ recursion stack;

In tabulation
    time complexity: O(n*2);
    space complexity: O(n*2);
*/
```

38. Buy and Stock sell IV

→ In this problem you need to do k transactions at most

→ It is almost similar to prev problem just change 2 to k. And you will get the answer

```cpp
/*
You are given an array prices where prices[i] is the price of a given stock on the ith day.

Find the maximum profit you can achieve. You may complete at most two transactions.

Note: You may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).
*/

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

int recursive(vector<int> &prices, int index, int trans, int buyorsell, vector<vector<vector<int>>> &dp)
{
    if (index == prices.size() or trans == 0)
    {
        return 0;
    }

    if (dp[index][trans][buyorsell] != -1)
    {
        return dp[index][trans][buyorsell];
    }

    if (buyorsell)
    {
        int ans = 0;
        ans = max(-prices[index] + recursive(prices, index + 1, trans, !buyorsell, dp), recursive(prices, index + 1, trans, buyorsell, dp));
        return dp[index][trans][buyorsell] = ans;
    }
    else
    {
        int ans = 0;
        ans = max(prices[index] + recursive(prices, index + 1, trans - 1, !buyorsell, dp), recursive(prices, index + 1, trans, buyorsell, dp));
        return dp[index][trans][buyorsell] = ans;
    }
}

int tabulation(vector<int> &prices)
{
    int n = prices.size();
    // vector<vector<vector<int>>>dp(n+1,vector<vector<int>>(3,vector<int>(2,0)));
    /*      here my matrix size is of n*3*2;
           we can matrix of size 3*2

    */
    vector<vector<int>> ahead(3, vector<int>(2, 0));
    vector<vector<int>> curr(3, vector<int>(2, 0));

    for (int index = n - 1; index >= 0; index--)
    {
        for (int trans = 1; trans < 3; trans++)
        {
            for (int buyorsell = 0; buyorsell < 2; buyorsell++)
            {
```



```cpp
                if (buyorsell)
                {
                    int ans = 0;
                    ans = max(-prices[index] + ahead[trans][!buyorsell], ahead[trans][buyorsell]);
                    curr[trans][buyorsell] = ans;
                }
                else
                {
                    int ans = 0;
                    ans = max(prices[index] + ahead[trans - 1][!buyorsell], ahead[trans][buyorsell]);
                    curr[trans][buyorsell] = ans;
                }
            }
        }
        ahead = curr;
    }

    return curr[2][1];
}

int main()
{
    int n;
    cin>>n;
    vector<int>input(n);
    for(int i=0;i<n;i++)
    {
        cin>>input[i];
    }
    vector<vector<vector<int>>>dp(n+1,vector<vector<int>>(3,vector<int>(2,-1)));
    int ans=recursive(input,0,3,1,dp);
    cout<<ans<<endl;
    int ans1=tabulation(input);
    cout<<ans1<<endl;
}

/*
In recursion + memoization
  time complexity= O(n*3*2);
  space complexity= O(n*3*2)+ recusive stack

In tabulation
    time complexity= O(n*3*2);
    space complexity= O(3*2);
*/
```

39. Buy and sell stock with Cooldown

→ This problem just provides to Buy and Sell Stock II.

Only difference

:→ After you sell your stock, you cannot buy stock on the next day.

Modification to its second version II

→ After selling the stock just do index+2.

index+2 ? Because of cooldown after selling the stocks.

```cpp
/*
You are given an integer array prices where prices[i] is the price of a given stock on the ith day, and an integer k.

Find the maximum profit you can achieve. You may complete at most k transactions.

Note: You may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).

Example 1:

Input: k = 2, prices = [2,4,1]
Output: 2
Explanation: Buy on day 1 (price = 2) and sell on day 2 (price = 4), profit = 4-2 = 2.
*/

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

int recursive(vector<int> &prices, int index, int trans, int buyorsell, vector<vector<vector<int>>> &dp)
{
    if (index == prices.size() or trans == 0)
    {
        return 0;
    }

    if (dp[index][trans][buyorsell] != -1)
    {
        return dp[index][trans][buyorsell];
    }

    if (buyorsell)
    {
        int ans = 0;
        ans = max(-prices[index] + recursive(prices, index + 1, trans, !buyorsell, dp), recursive(prices, index + 1, trans, buyorsell, dp));
        return dp[index][trans][buyorsell] = ans;
    }
    else
    {
        int ans = 0;
        ans = max(prices[index] + recursive(prices, index + 1, trans - 1, !buyorsell, dp), recursive(prices, index + 1, trans, buyorsell, dp));
        return dp[index][trans][buyorsell] = ans;
    }
}
int tabulation(vector<int> &prices, int k)
{
    int n = prices.size();

    vector<vector<int>> ahead(k + 1, vector<int>(2, 0));
    vector<vector<int>> curr(k + 1, vector<int>(2, 0));

    for (int index = n - 1; index >= 0; index--)
    {
        for (int trans = 1; trans < k + 1; trans++)
        {
```



```cpp
            for (int buyorsell = 0; buyorsell < 2; buyorsell++)
            {
                if (buyorsell)
                {
                    int ans = 0;
                    ans = max(-prices[index] + ahead[trans][!buyorsell], ahead[trans][buyorsell]);
                    curr[trans][buyorsell] = ans;
                }
                else
                {
                    int ans = 0;
                    ans = max(prices[index] + ahead[trans - 1][!buyorsell], ahead[trans][buyorsell]);
                    curr[trans][buyorsell] = ans;
                }
            }
        }
        ahead = curr;
    }

    return curr[k][1];
}

int main()
{
    int n;
    cin>>n;
    vector<int> prices(n,0);

    for(int i=0; i<n; i++)
    {
        cin>>prices[i];
    }
    int k;
    cin>>k;

     vector<vector<vector<int>>>dp(n,vector<vector<int>>(k+1,vector<int>(2,-1)));
     int ans1= recursive(prices,0,k,1,dp);
    int ans = tabulation(prices, k);
    cout<<ans1<<endl;
    cout<<ans<<endl;
}

/*
In recursion +memoization
  time complexity: O(n*k*2);
  space complexity: O(n*k*2)+ recursion stack;

In tabulation
    time complexity: O(n*k*2);
    space complexity: O(k*2);
*/
```

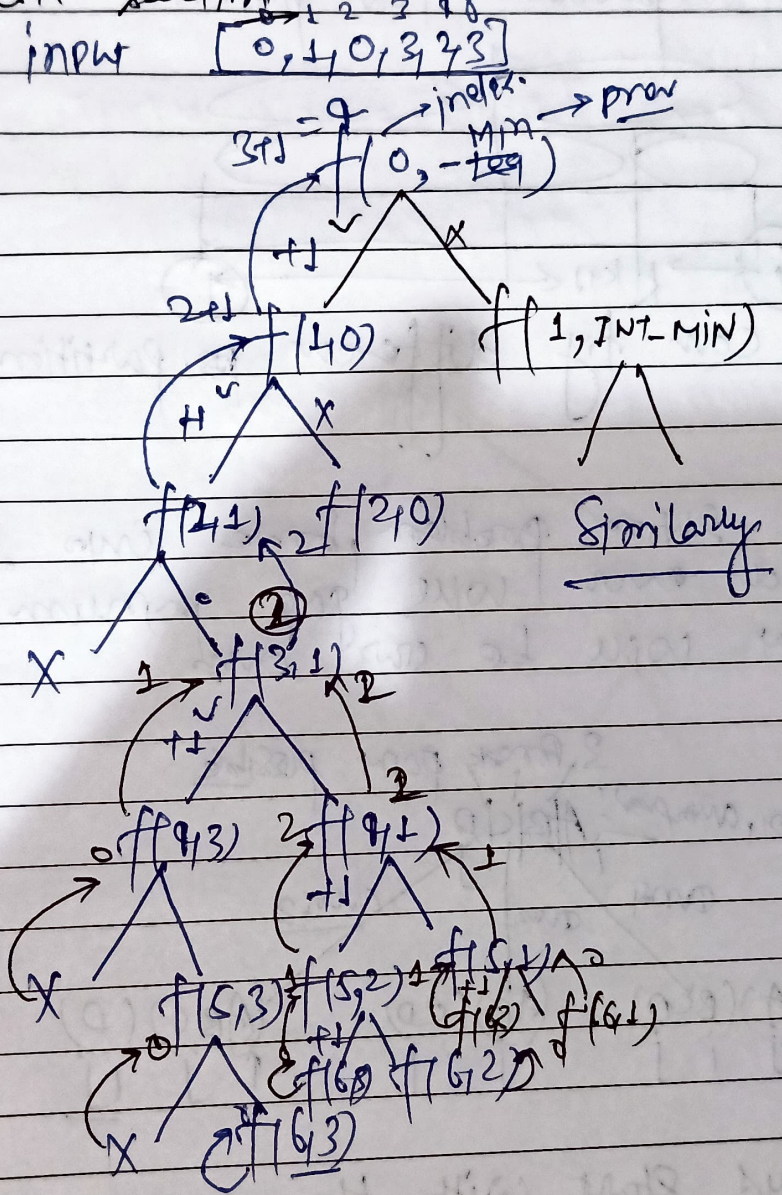
```

/**
You are given an array prices where prices[i] is the price of a given stock on the ith day, and an integer fee representing a transaction fee.

Find the maximum profit you can achieve. You may complete as many transactions as you like, but you need to pay the transaction fee for each transaction.

Note: You may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).

Example 1:

Input: prices = [1,3,2,8,4,9], fee = 2
Output: 8
Explanation: The maximum profit can be achieved by:
- Buying at prices[0] = 1
- Selling at prices[3] = 8
- Buying at prices[4] = 4
- Selling at prices[5] = 9
The total profit is ((8 - 1) - 2) + ((9 - 4) - 2) = 8.
 */
using namespace std;

int calculateans(vector<int> &price, int buyorsell, int index, vector<vector<int>> &dp, int fee)
{
    if (index == price.size())
    {
        return 0;
    }
    if (dp[index][buyorsell] != -1)
    {
        return dp[index][buyorsell];
    }
    if (buyorsell)
    {
        int ans = max(-price[index] + calculateans(price, !buyorsell, index + 1, dp, fee), calculateans(price, buyorsell, index + 1, dp, fee));
        return dp[index][buyorsell] = ans;
    }
    else
    {
        int ans1 = max(price[index] - fee + calculateans(price, !buyorsell, index + 1, dp, fee), calculateans(price, buyorsell, index + 1, dp, fee));

        return dp[index][buyorsell] = ans1;
    }
}

int tabulation(vector<int> &price, int fee)
{
    int n = price.size();
    // vector<vector<int>>dp(n+1,vector<int>(2,0));
    // space optimization
    vector<int> ahead(2, 0);
    vector<int> curr(2, 0);
    for (int index = n - 1; index >= 0; index--)
    {
```



```
        for (int buyorsell = 0; buyorsell <= 1; buyorsell++)
        {
            if (buyorsell)
            {
                int ans = max(-price[index] + ahead[!buyorsell], ahead[buyorsell]);
                curr[buyorsell] = ans;
            }
            else
            {
                int ans1 = max(price[index] - fee + ahead[!buyorsell], ahead[buyorsell]);

                curr[buyorsell] = ans1;
            }
        }
        ahead = curr;
    }

    return curr[1];
}

int main()
{
    int n;
    cin >> n;
    vector<int> input(n);
    for (int i = 0; i < n; i++)
    {
        cin >> input[i];
    }
    int fee;
    cin >> fee;
    int ans=calculateans(input,1,0,vector<vector<int>>(n+1,vector<int>(2,-1)),fee);
    cout<<ans<<endl;
    int ans1=tabulation(input,fee);
    cout<<ans1<<endl;
}
/*
In recursion +memoization
  time complexity: O(n*2);
  space complexity: O(n*2)+ recursion stack;

In tabulation
    time complexity: O(n*2);
    space complexity: O(2);
*/
```

40. Longest Increasing Subsequence

→ There are three methods to solve it. All three method are mentioned in the code section.



Remember method no code section. Please read carefully

```cpp
/*
Given an integer array nums, return the length of the longest strictly increasing subsequence.

A subsequence is a sequence that can be derived from an array by deleting some or no elements without changing the order of the remaining elements. For example, [3,6,2,7] is a subsequence of the array [0,3,1,6,2,2,7].
*/
#include<bits/stdc++.h>
using namespace std;
int calculateans(vector<int> &nums, int index, int prev, vector<vector<int>> &dp)
{

    if (index == nums.size())
    {
        return 0;
    }

    int include = 0;
    int exclude = 0;
    if (dp[index][prev + 1] != -1)
    {
        return dp[index][prev + 1];
    }
    exclude = calculateans(nums, index + 1, prev, dp);
    if (prev == -1 or nums[index] > nums[prev])
    {
        include = 1 + calculateans(nums, index + 1, index, dp);
    }

    return dp[index][prev + 1] = max(include, exclude);
}

int tabulation(vector<int> &nums)
{
    int n = nums.size();
    vector<vector<int>> dp(n + 1, vector<int>(n + 1, 0));
    //         here we are shifting the second parameter by 1 right because of prev=-1 , thats why we are doing +1 in every second parameter in the dp array
    for (int index = n - 1; index >= 0; index--)
    {
        for (int prev = index - 1; prev >= -1; prev--)
        {
            int include = 0;
            int exclude = 0;

            exclude = dp[index + 1][prev + 1];
            if (prev == -1 or nums[index] > nums[prev])
            {
                include = 1 + dp[index + 1][index + 1];
            }

            dp[index][prev + 1] = max(include, exclude);
        }
    }

    return dp[0][0];
}

```

```cpp
int arraySolution(vector<int> &nums)
{
    int n = nums.size();
    vector<int> dp(n, 1);
    vector<int> parent(n, 0);
    for (int i = 0; i < n; i++)
    {
        parent[i] = i;
    }
    dp[0] = 1;
    int ans = 1;
    for (int i = 1; i < n; i++)
    {
        for (int j = 0; j < i; j++)
        {
            if (nums[j] < nums[i])
            {
                dp[i] = max(dp[i], dp[j] + 1);
            }
        }
        ans = max(ans, dp[i]);
    }

    return ans;
}

void printLcs(vector<int> &nums)
{
    int n = nums.size();
    vector<int> dp(n, 1);
    vector<int> parent(n, 0);
    for (int i = 0; i < n; i++)
    {
        parent[i] = i;
    }
    dp[0] = 1;
    int ans = 1;
    int maxe = 0;
    int index = -1;
    for (int i = 1; i < n; i++)
    {
        for (int j = 0; j < i; j++)
        {
            if (nums[j] < nums[i])
            {
                // dp[i]=max(dp[i],dp[j]+1);
                if (dp[j] + 1 > dp[i])
                {
                    dp[i] = dp[j] + 1;
                    parent[i] = j;
                }
            }
        }
        if (dp[i] > maxe)
        {
            maxe = dp[i];
            index = i;
        }
        ans = max(ans, dp[i]);
    }
```

```
116
117     cout << nums[index] << " ";
118
119     while (index != parent[index])
120     {
121         cout << nums[parent[index]] << " ";
122         index = parent[index];
123     }
124     cout << endl;
125 }
126
127 int LcsNlogN(vector<int> &nums)
128 {
129     vector<int> dp;
130     int n = nums.size();
131     dp.push_back(nums[0]);
132     for (int i = 1; i < n; i++)
133     {
134         if (nums[i] > dp.back())
135         {
136             dp.push_back(nums[i]);
137         }
138         else
139         {
140             auto it = lower_bound(dp.begin(), dp.end(), nums[i]) - dp.begin();
141             dp[it] = nums[i];
142         }
143     }
144     return dp.size();
145 }
146
147 int lengthOfLIS(vector<int> &nums)
148 {
149
150     int n = nums.size();
151
152     // vector<vector<int>>dp(n,vector<int>(n+1,-1));
153     // int ans= calculateans(nums,0,-1,dp);
154     // int ans= tabulation(nums);
155     // int ans= arraySolution(nums);
156     // printLcs(nums);
157     int ans = LcsNlogN(nums);
158     return ans;
159 }
160
161 int main()
162 {
163     int n;
164     cin>>n;
165     vector<int>input(n);
166     for(int i=0;i<n;i++)
167     {
168         cin>>input[i];
169     }
170
171     cout<<lengthOfLIS(input)<<endl;
172 }
```

```cpp
/*
You are given an array of words where each word consists of lowercase English letters.

wordA is a predecessor of wordB if and only if we can insert exactly one letter anywhere in wordA without changing the order of the other characters to make it equal to wordB.

For example, "abc" is a predecessor of "abac", while "cba" is not a predecessor of "bcad".
A word chain is a sequence of words [word1, word2, ..., wordk] with k >= 1, where word1 is a predecessor of word2, word2 is a predecessor of word3, and so on. A single word is trivially a word chain with k == 1.

Return the length of the longest possible word chain with words chosen from the given list of words.

Example 1:

Input: words = ["a","b","ba","bca","bda","bdca"]
Output: 4
Explanation: One of the longest word chains is ["a","ba","bda","bdca"].
*/
#include<bits/stdc++.h>
using namespace std;
const bool static comp(string &input1, string &input2)
{
    return input1.size() < input2.size();
}
bool isvalid(string &larger, string &smaller)
{
    int nl = larger.size();
    int ns = smaller.size();
    if (abs(nl - ns) != 1)
    {
        return false;
    }

    int i = 0;
    int j = 0;
    while (i != nl)
    {
        if (larger[i] == smaller[j])
        {
            i++;
            j++;
        }
        else
        {
            i++;
        }
    }
    return j == ns;
}
int longestStrChain(vector<string> &words)
{

    sort(words.begin(), words.end(), comp);
    int n = words.size();
```

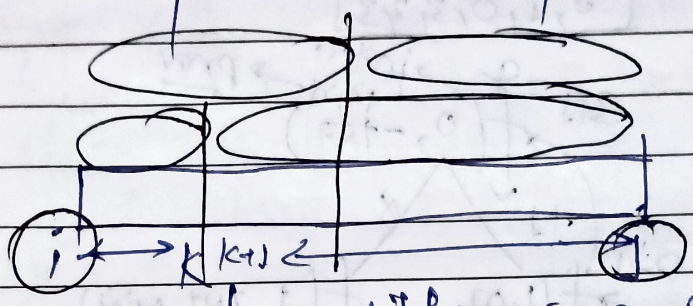


```cpp

    vector<int> dp(n, 1);
    int maxe = 0;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < i; j++)
        {
            if (isvalid(words[i], words[j]))
            {
                if (dp[i] < dp[j] + 1)
                {
                    dp[i] = dp[j] + 1;
                }
            }
        }
        maxe = max(maxe, dp[i]);
    }
    return maxe;
}
int main()
{
    int n;
    cin >> n;
    vector<string> words(n);
    for (int i = 0; i < n; i++)
    {
        cin >> words[i];
    }
    cout << longestStrChain(words) << endl;
    return 0;
}
```

41. Matrix chain multiplication (MCM)

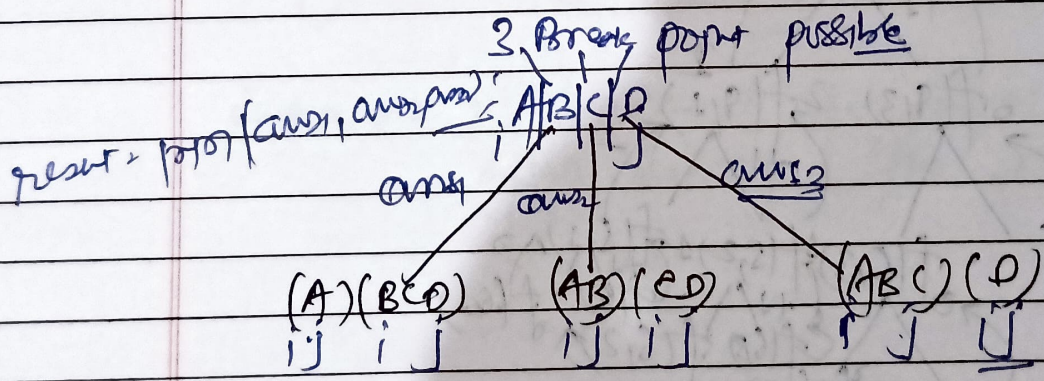
→ Partition DP

→ Solve problem in pattern



We can try different possible partitions.

→ Take whole problem, break into smaller which ever will give minimum that will be our ans



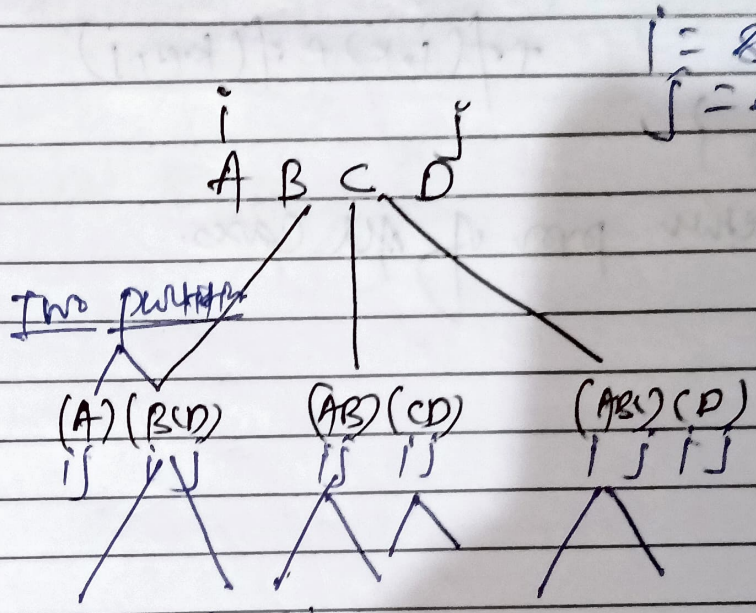
→ Always start with the whole problem i will be the start point and j will be the end point and k will be the partition.

Rules

i) Start with entire block [array f(i,j)

2) Try all partitions (Run a loop to try all partition...)

3. Return the best possible for 2 partition



→

→ Let suppose you have given

Array [10, 20, 30, 40, 50]
0 1 2 3 4

i j (0+1)

A B C D

f(1,4) → return the min multiplications to multiply matrix from 1 to 4

Then our loop will run for

f(k → i → j-1)

{

steps = (A[i-1] × Arr[k] × Arr[j]) → self Calculation

+ f(i, k) + f(k+1, j)

}

Return min of All cases.

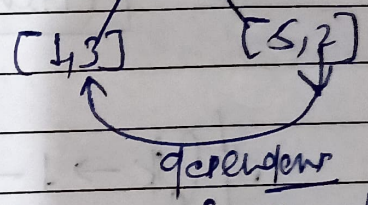
```cpp
#include<bits/stdc++.h>
using namespace std;

int calculate(int i, int j, int *input, vector<vector<int>> &dp)
{
    if (i == j)
    {
        return 0;
    }

    if (dp[i][j] != -1)
    {
        return dp[i][j];
    }

    int ans = INT_MAX;
    for (int k = i; k <= j - 1; k++)
    {
        int leftans = calculate(i, k, input, dp);
        int rightans = calculate(k + 1, j, input, dp);
        int additionalans = input[i - 1] * input[k] * input[j];

        ans = min(ans, (leftans + rightans + additionalans));
    }
    return dp[i][j] = ans;
}
int matrixMultiplication(int n, int arr[])
{
    // code here

    vector<vector<int>> dp(n, vector<int>(n, -1));

    int result = calculate(1, n - 1, arr, dp);

    return result;
}


int main()
{
    int n;
    cin >> n;
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        cin >> arr[i];
    }
    int result = matrixMultiplication(n, arr);
    cout << result;
    return 0;
}
```

50. Minimum Cost to Cut the Stick

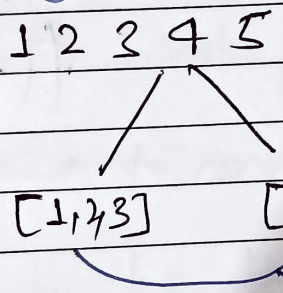
arr[] = [1,3,4,5] n=4

→ For this problem we just need to process sub problem independent of each other.

if our array like [1,3,4,5,2]



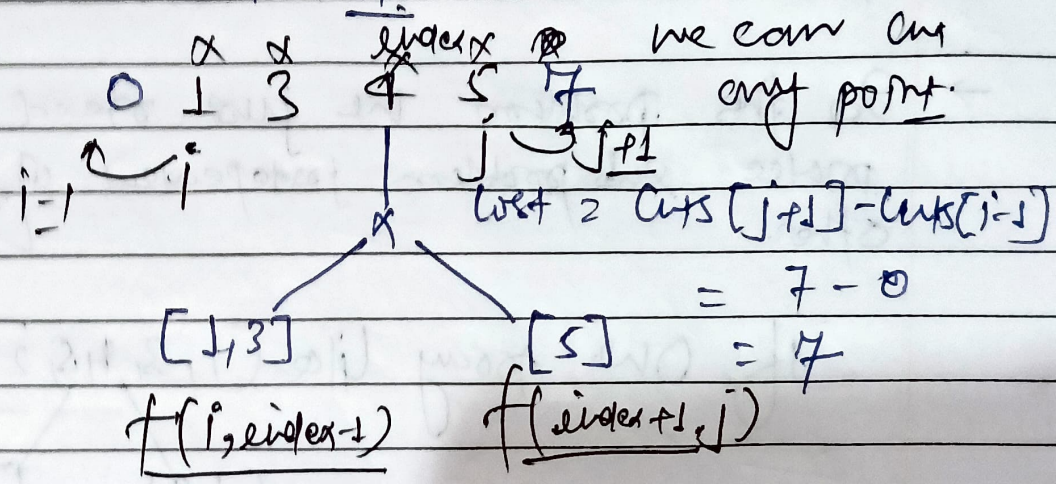
for processing independent we need to sort them first



→ Now how to find the length of stick

Cost = length of that stick in which array that index exist

for calculating length we will pass i & j only in f.



f(ik → 1 → index 2)

{

cost = cuts[j+1] - cost[i-1] + f(i, index-1) + f(index+1, j)

}

if (i>j)
return 0

TC = (M² × M) = M³

for i, j → for a loop

```cpp
/*
Given a wooden stick of length n units. The stick is labelled from 0 to n. For example, a stick of length 6 is labelled as follows:


Given an integer array cuts where cuts[i] denotes a position you should perform a cut at.

You should perform the cuts in order, you can change the order of the cuts as you wish.

The cost of one cut is the length of the stick to be cut, the total cost is the sum of costs of all cuts. When you cut a stick, it will be split into two smaller sticks (i.e. the sum of their lengths is the length of the stick before the cut). Please refer to the first example for a better explanation.

Return the minimum total cost of the cuts.
Example 1:
Input: n = 7, cuts = [1,3,4,5]
Output: 16
Explanation: Using cuts order = [1, 3, 4, 5] as in the input leads to the following scenario:

The first cut is done to a rod of length 7 so the cost is 7. The second cut is done to a rod of length 6 (i.e. the second part of the first cut), the third is done to a rod of length 4 and the last cut is to a rod of length 3. The total cost is 7 + 6 + 4 + 3 = 20.
Rearranging the cuts to be [3, 5, 1, 4] for example will lead to a scenario with total cost = 16 (as shown in the example photo 7 + 4 + 3 + 2 = 16).
*/
#include<bits/stdc++.h>
using namespace std;

int calculate(vector<int> &cuts, int start, int end, vector<vector<int>> &dp)
{
    if (start > end)
    {
        return 0;
    }
    if (dp[start][end] != -1)
    {
        return dp[start][end];
    }

    int ans = INT_MAX;
    for (int k = start; k <= end; k++)
    {
        int leftans = calculate(cuts, start, k - 1, dp);
        int rightans = calculate(cuts, k + 1, end, dp);
        int aditionalans = cuts[end + 1] - cuts[start - 1];
        ans = min(ans, leftans + rightans + aditionalans);
    }

    return dp[start][end] = ans;
}

int minCost(int n, vector<int> &cuts)
{

```



```cpp
    int m = cuts.size();
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, -1));
    sort(cuts.begin(), cuts.end());
    cuts.push_back(n);
    cuts.insert(cuts.begin(), 0);
    int ans = calculate(cuts, 1, m, dp);

    return ans;
}

int main()
{
    int n;
    cin >> n;
    vector<int> cuts(n);
    for (int i = 0; i < n; i++)
    {
        cin >> cuts[i];
    }
    int result = minCost(n, cuts);
    cout << result;
    return 0;
}
```

51. Burst Balloons

→ Place ~~Sub~~ Sub problem ~~do~~ ~~step~~ independent is important.

Arr [3, 1, 5, 8] n = 4

1 × 3 × 1 = 3

[1, 5, 8] 1 × 1 × 5 = 5

[5, 8] 1 × 5 × 8 = 40

[8] 1 × 8 × 1 = 8

56

We can burst any ballon at first.

We need to maximize it

[b1 b2 b3 b4 b5 b6]

[b1 b2 b3] + [b2 × b4 × b5] + [b5 × b6]

Subproblem, Left, Subproblem

This will not give Correct result.

→ After burst [b1 b2 b3 b5 b6]

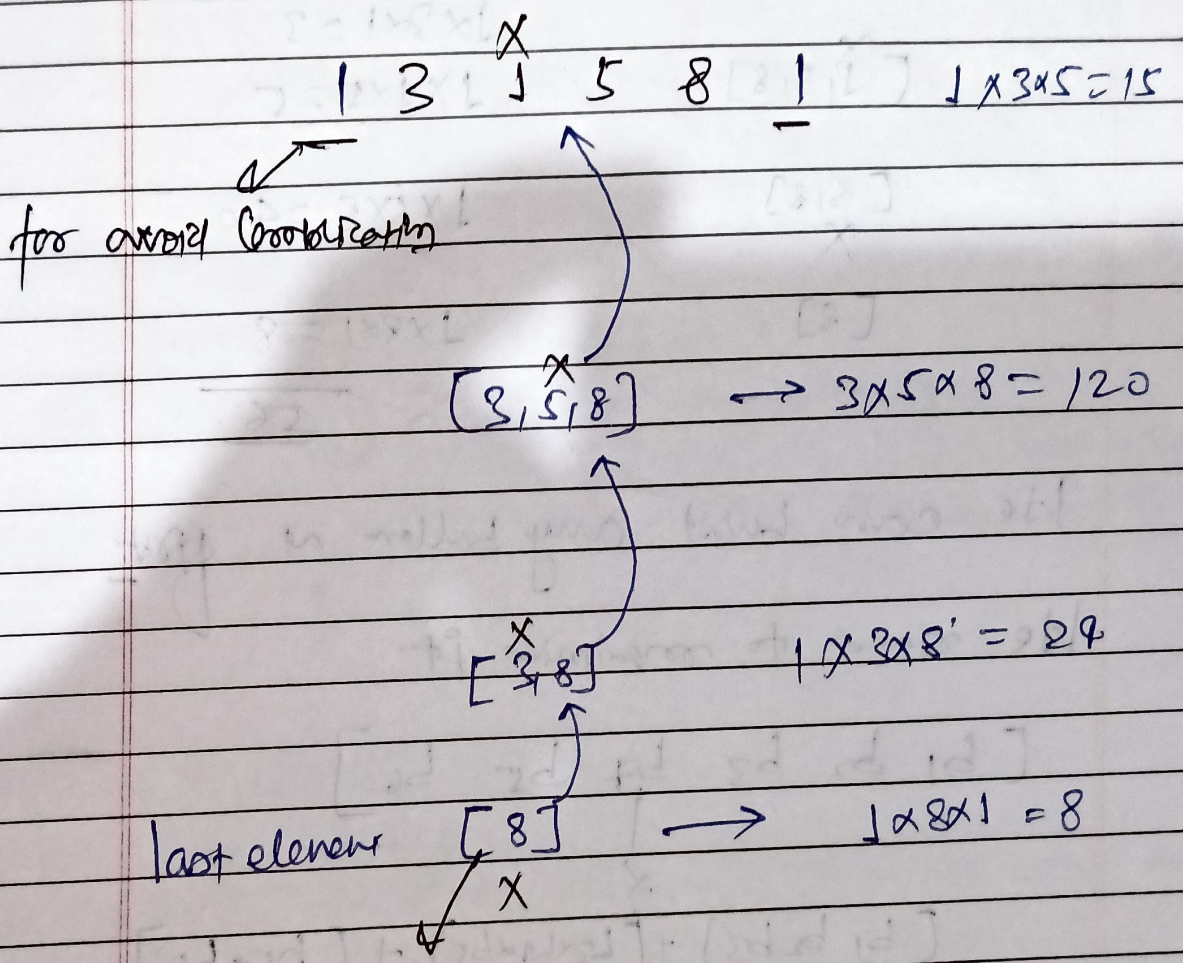
(b2 × b3 × b5)

Depending on b5

Dependent on each other.

Then we need to choose it independent

→ for that we go from last element



Assuming this is last element to solve.

b1 b2 b3 b4 b5 b6

a[i-1] × a[end] × a[j+1] + f(i, end-1) + f(end+1, j)

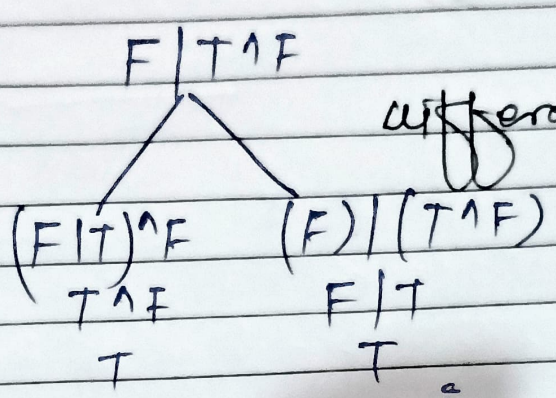
```cpp
/*
You are given n balloons, indexed from 0 to n - 1. Each balloon is painted with a number on it represented by an array nums. You are asked to burst all the balloons.

If you burst the ith balloon, you will get nums[i - 1] * nums[i] * nums[i + 1] coins. If i - 1 or i + 1 goes out of bounds of the array, then treat it as if there is a balloon with a 1 painted on it.

Return the maximum coins you can collect by bursting the balloons wisely.
*/
#include<bits/stdc++.h>
using namespace std;

int calculate(vector<int> &nums, int start, int end, vector<vector<int>> &dp)
{
    if (start > end)
    {
        return 0;
    }

    int ans = INT_MIN;
    //  we are starting from bottom that means there will be only one element in the array and their left and right side will be
    if (dp[start][end] != -1)
    {
        return dp[start][end];
    }
    for (int k = start; k <= end; k++)
    {
        int leftans = calculate(nums, start, k - 1, dp);
        int rightans = calculate(nums, k + 1, end, dp);

        int additional = nums[start - 1] * nums[k] * nums[end + 1];

        ans = max(ans, leftans + rightans + additional);
    }
    return dp[start][end] = ans;
}

int maxCoins(vector<int> &nums)
{

    int n = nums.size();
    nums.push_back(1);
    nums.insert(nums.begin(), 1);
    vector<vector<int>> dp(n + 1, vector<int>(n + 1, -1));
    int ans = calculate(nums, 1, n, dp);

    return ans;
}

int main()
{
    int n;
    cin >> n;
    vector<int> nums(n);
    for (int i = 0; i < n; i++)
    {
        cin >> nums[i];
```



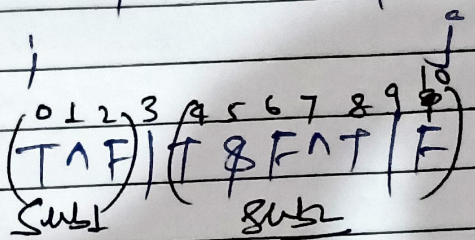
```cpp
    }
    int result = maxCoins(nums);
    cout << result;
    return 0;
}


// This question based on Matrix chain multiplication

```

52. Evaluate to true

Input: F|T^F



Input: (T^F)|(T&F^T|F)



f(0,0) and f(2,10) + Some Calculation
Sub1 Sub2

f(0,4) and f(6,10) + Some Calculation

Now need to try every possible partition

f(i, index-1) + C + f(index+1, j)
x y

if (str[index] == F)

Total way = xy

```cpp
/*
Given an expression, A, with operands and operators (OR , AND , XOR), in how many ways can you evaluate the expression to true, by grouping in different ways?

Operands are only true and false.

Return the number of ways to evaluate the expression modulo 103 + 3.

Input Format:

The first and the only argument of input will contain a string, A.

The string A, may contain these characters:
    '|' will represent or operator
    '&' will represent and operator
    '^' will represent xor operator
    'T' will represent true operand
    'F' will false
*/
#include <bits/stdc++.h>
using namespace std;
long long calculate(string input, int start, int end, int istrue, vector<vector<vector<long long>>> &dp)
{
    if (start > end)
    {
        return 0;
    }
    if (start == end)
    {
        if (istrue)
        {
            return input[start] == 'T';
        }
        else
        {
            return input[start] == 'F';
        }
    }
    if (dp[start][end][istrue] != -1)
    {
        return dp[start][end][istrue];
    }
    int ways = 0;
    long long mode = 1e3 + 3;
    for (int k = start + 1; k <= end - 1; k += 2)
    {
        long long lefttrue = calculate(input, start, k - 1, 1, dp);
        long long leftfalse = calculate(input, start, k - 1, 0, dp);
        long long righttrue = calculate(input, k + 1, end, 1, dp);
        long long rightfalse = calculate(input, k + 1, end, 0, dp);

        // appan calculation
        if (input[k] == '&')
        {
            if (istrue)
                ways = (ways + (lefttrue * righttrue) % (mode)) % (mode);
            else
            {
                ways = (ways + (lefttrue * rightfalse) % (mode) + (leftfalse * righttrue) % (mode) + (leftfalse * rightfalse) % (mode)) % (mode);
            }
        }
        else if (input[k] == '|')
        {
            if (istrue)
            {
                ways = (ways + (lefttrue * righttrue) % (mode) + (lefttrue * rightfalse) % (mode) + (leftfalse * righttrue) % (mode)) % (mode);
            }
            else
            {
                ways = (ways + (leftfalse * rightfalse) % (mode)) % (mode);
            }
        }
        else
        {
            if (istrue)
            {
                ways = (ways + (lefttrue * rightfalse) % (mode) + (leftfalse * righttrue) % (mode)) % (mode);
            }
            else
            {
                ways = (ways + (lefttrue * righttrue) % (mode) + (leftfalse * rightfalse) % (mode)) % (mode);
            }
        }
    }
    return dp[start][end][istrue] = ways;
}

int cnttrue(string input)
{

    long long n = input.size();

    vector<vector<vector<long long>>> dp(n + 1, vector<vector<long long>>(n + 1, vector<long long>(2, -1)));

    long long ans = calculate(input, 0, n - 1, 1, dp);

    return ans;
}

int main()
{
    int n;
    cin >> n;
    string input;
    cin >> input;
    int result = cnttrue(input);
    cout << result;
    return 0;
}

```

```cpp
/*
Given a string s, partition s such that every substring of the partition is a palindrome.

Return the minimum cuts needed for a palindrome partitioning of s.

Example 1:

Input: s = "aab"
Output: 1
Explanation: The palindrome partitioning ["aa","b"] could be produced using 1 cut.
*/
#include<bits/stdc++.h>
using namespace std;


bool ispalindrome(string &input, int start, int end)
{
    while (start < end)
    {
        if (input[start] != input[end])
        {
            return false;
        }
        start++;
        end--;
    }
    return true;
}

int calculate(string &input, int start, vector<int> &dp)
{
    if (start == input.size())
    {
        return 0;
    }

    int ans = INT_MAX;
    if (dp[start] != -1)
    {
        return dp[start];
    }

    for (int k = start; k < input.size(); k++)
    {
        if (ispalindrome(input, start, k))
        {
            int nextcall = calculate(input, k + 1, dp);
            int ans1 = 1 + nextcall;
            ans = min(ans, ans1);
        }
    }
    return dp[start] = ans;
}
int minCut(string s)
{

    int n = s.size();
    vector<int> dp(n + 1, -1);
```



```cpp
    int result = calculate(s, 0, dp);

    return result - 1;
}

int main()
{
    string s;
    cin >> s;
    int result = minCut(s);
    cout << result;
    return 0;
}
```

```cpp
/*
Given an integer array arr, partition the array into (contiguous) subarrays of length at most k. After partitioning, each subarray has their values changed to become the maximum value of that subarray.

Return the largest sum of the given array after partitioning. Test cases are generated so that the answer fits in a 32-bit integer.

Example 1:

Input: arr = [1,15,7,9,2,5,10], k = 3
Output: 84
Explanation: arr becomes [15,15,15,9,10,10,10]
*/
#include<bits/stdc++.h>
using namespace std;
int calculate(vector<int> &arr, int start, int k, vector<int> &dp)
{
    if (start == arr.size())
    {
        return 0;
    }
    int ans = INT_MIN;

    int maxe = INT_MIN;
    int len = 0;
    if (dp[start] != -1)
    {
        return dp[start];
    }
    for (int k1 = start; k1 < min(start + k, (int)arr.size()); k1++)
    {
        maxe = max(maxe, arr[k1]);
        len++;
        int nextcall = calculate(arr, k1 + 1, k, dp);

        ans = max(ans, nextcall + len * maxe);
    }

    return dp[start] = ans;
}

int maxSumAfterPartitioning(vector<int> &arr, int k)
{

    int n = arr.size();
    vector<int> dp(n, -1);
    int ans = calculate(arr, 0, k, dp);

    return ans;
}

int main()
{
    int n;
    cin >> n;
    vector<int> arr(n);
    for (int i = 0; i < n; i++)
    {
```



```cpp
        cin >> arr[i];
    }
    int k;
    cin>>k;
    int result = maxSumAfterPartitioning(arr, k);
    cout << result;
}
```