

## Fibonacci-dp

Easy

1. You are given a number  $n$ .
2. You are required to print the  $n$ th element of fibonacci sequence.

Note -> Notice precisely how we have defined the fibonacci sequence

0th element -> 0

1st element -> 1

2nd element -> 1

3rd element -> 2

4th element -> 3

5th element -> 5

6th element -> 8

### Constraints

$0 \leq n \leq 45$

### Format

#### Input

A number  $n$

#### Output

A number representing the  $n$ th element of fibonacci sequence

### Example

#### Sample Input

10

#### Sample Output

55

# Introduction to dp using fibonacci

■■■■■

LEVEL- 1



  
Sumeet Malik

# INTRODUCTION TO DYNAMIC PROGRAMMING

[www.nados.pepcoding.com](http://www.nados.pepcoding.com)

For better experience visit

```
#include <iostream>
#include<vector>
using namespace std;
//dp Tc=O(n) && SC = O(n)
int fib(int n, vector<int> & qb){
    // write your code here
    if( n == 0 || n == 1) {
        return n;
    }
    if(qb[n] != 0){
        return qb[n];
    }
    int a = fib( n - 1 ,qb) ;
    int b = fib( n - 2,qb) ;
    qb[n] = a+b;
    // cout<<"called for: "<<n<<endl;
    return a + b;
}
//recursion -> time limit exceed
// int fibonacci(int n) {
//     if( n == 0 || n == 1) {
//         return n;
//     }
//     return fibonacci( n - 1 ) + fibonacci( n - 2);
// }
```

```
int main(){
    int n;
    cin>>n;
    vector <int> qb(n+1,0);
    cout<<fib(n,qb)<<endl;

    //rec
    // cout<<fibonacci(n)<<endl;
    return 0;
}
```

# Climb Stairs

Easy

1. You are given a number  $n$ , representing the number of stairs in a staircase.
2. You are on the 0th step and are required to climb to the top.
3. In one move, you are allowed to climb 1, 2 or 3 stairs.
4. You are required to print the number of different paths via which you can climb to the top.

## Constraints

$0 \leq n \leq 20$

## Format

### Input

A number  $n$

### Output

A number representing the number of ways to climb the stairs from 0 to top.

## Example

### Sample Input

4

### Sample Output

7



```
#include<iostream>
#include<vector>
using namespace std;
//memoization
int nToZero(int n,vector<int> & cs) {
    if(n < 0) {
        return 0;
    }

    if(n == 0 || n == 1) {
        return 1;
    }
}
```

```

    if(cs[n] != 0) {
        return cs[n];
    }
    //at 1,2,3 steps allowed
    int a = nToZero(n-1,cs) + nToZero(n-2,cs) + nToZero(n-3,cs);
    cs[n] = a;
    return a;
}

//tabulation

int nToZeroTabulation(int n) {
    vector<int> dp(n+1, 0);

    dp[0] = 1;
    for(int i{1} ; i< dp.size();i++) {
        if(i == 1) { //base case
            dp[i] = dp[i-1];
        }else if(i == 2) { //base case
            dp[i] = dp[i-1] + dp[i-2];
        }else{
            dp[i] = dp[i-1] + dp[i-2] + dp[i-3];
        }
    }

    return dp[n];
}

int main() {
    int n{};
    cin>> n;
    vector<int> cs(n+1,0);

    // cout<<nToZero(n,cs)<<endl;
    cout<<nToZeroTabulation(n)<<endl;
    return 0;
}

```

# Climb Stairs With Variable Jumps

Easy

1. You are given a number  $n$ , representing the number of stairs in a staircase.
2. You are on the 0th step and are required to climb to the top.
3. You are given  $n$  numbers, where  $i$ th element's value represents - till how far from the step you could jump to in a single move.  
You can of course jump fewer number of steps in the move.
4. You are required to print the number of different paths via which you can climb to the top.

## Constraints

$0 \leq n \leq 20$

$0 \leq n_1, n_2, \dots \leq 20$

## Format

### Input

A number  $n$

..  $n$  more elements

### Output

A number representing the number of ways to climb the stairs from 0 to top.

## Example

### Sample Input

```
10
3
3
0
2
1
2
4
2
0
0
```

### Sample Output

```
5
```

```
#include <iostream>
#include <vector>
using namespace std;
```

```
int cs(int arr[], int n){
    // write your code here
    vector<int> dp(n+1,0);
    dp[n] = 1;
    for(int i{n-1}; i >= 0 ; i--) {
        int fromThis{};
        for(int j{1}; j <= arr[i]; j++) {
            // cout<<i+j<<"--"<<endl;
            if(i+j >= dp.size()){
                continue;
            }
            fromThis += dp[i+j];
        }
    }
}
```

```

        }
        // cout<<i<<" "<<fromThis<<endl;
        dp[i] = fromThis;
    }

    return dp[0];
}

int main(){
    int n;
    cin>>n;
    int arr[n];
    for(int i = 0 ; i < n ;i++){
        cin>>arr[i];
    }
    cout<<cs(arr,n)<<endl;
}

```

# Climb Stairs With Minimum Moves

Easy

1. You are given a number n, representing the number of stairs in a staircase.
2. You are on the 0th step and are required to climb to the top.
3. You are given n numbers, where ith element's value represents - till how far from the step you could jump to in a single move. You can of-course fewer number of steps in the move.
4. You are required to print the number of minimum moves in which you can reach the top of staircase.

Note -> If there is no path through the staircase print null.

## Constraints

$0 \leq n \leq 20$

$0 \leq n_1, n_2, \dots \leq 20$

## Format

### Input

A number n

.. n more elements

### Output

A number representing the number of ways to climb the stairs from 0 to top.

## Example

### Sample Input

```
10
3
3
0
2
1
2
4
2
0
0
```

### Sample Output

```
4
#include <iostream>
#include <vector>
#include <limits.h>
#include <algorithm>
using namespace std;

int cs(int arr[], int n, vector<int>& dp) {
    // write your code here
    dp[n] = 0;
    for(int i{n-1}; i >= 0 ; i--) {
        if(arr[i] == 0){
            dp[i] = 30;
            continue;
        }
    }
```



```

        int fromThis = 30; // 30 is like infinite here because
        can't be more than zero
        for(int j{1}; j <= arr[i]; j++) {
            if(i+j >= dp.size()){
                continue;
            }
            // cout<<i<<" "<<j<<" -> "<<dp[i + j] <<" "<<
            fromThis <<endl;

            fromThis = min(dp[i+j], fromThis);
        }
        // cout<<" ----"<<i<<" "<<fromThis <<endl;
        dp[i] = fromThis +1;
    }

    return dp[0];
}

```

//my solution

```

// int cs(int arr[], int n, vector<int>& dp) {
//     // write your code here
//     if( n== 0) {
//         return 0;
//     }
//     dp[n] = 1;
//     for(int i{n-1}; i >= 0 ; i--) {
//         int fromThis = INT_MIN;
//         int fesNoOFStep = 0;
//         for(int j{1}; j <= arr[i]; j++) {
//             if(i+j >= dp.size() || dp[i+j] == 0){
//                 continue;
//             }
//             // cout<<i<<" "<<j<<" -> "<<j + dp[i + j] <<" "<<
//             fromThis <<endl;
//             if(j + dp[i + j] > fromThis) {
//                 fromThis = j + dp[i+j];
//                 fesNoOFStep = j;
//             }
//         }
//         dp[i] = fesNoOFStep;
//     }
//     int a{};
//     int nos{};
//     // cout<<dp[0]<<" reached there"<<endl;
//     while(a < n && nos <= 10 ) { //second is just to make sure
//         that we have a path to destination
//         nos++;
//         a += dp[a];
//     }
// }

```

```

//      if(nos > n) {
//          cout<<"Can't go to final step from there"<<endl;
//      }
//      return nos;

// }

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

    vector<int> dp(n + 1, 0);
    //pepcoding c++ answer
    if(cs(arr, n, dp) >= 30) {
        cout<<"null"<<endl;
        return 0;
    }
    cout << cs(arr, n, dp) << endl;
}

```

# Min Cost In Maze Traversal

Easy

1. You are given a number  $n$ , representing the number of rows.
2. You are given a number  $m$ , representing the number of columns.
3. You are given  $n*m$  numbers, representing elements of 2d array  $a$ , which represents a maze.
4. You are standing in top-left cell and are required to move to bottom-right cell.
5. You are allowed to move 1 cell right (h move) or 1 cell down (v move) in 1 motion.
6. Each cell has a value that will have to be paid to enter that cell (even for the top-left and bottom-right cell).
7. You are required to traverse through the matrix and print the cost of path which is least costly.

## Constraints

$1 \leq n \leq 10^2$

$1 \leq m \leq 10^2$

$0 \leq e_1, e_2, \dots, n * m \text{ elements} \leq 1000$

## Format

### Input

A number  $n$

A number  $m$

$e_{11}$

$e_{12}..$

$e_{21}$

$e_{22}..$

.. $n * m$  number of elements

### Output

The cost of least costly path.

## Example

### Sample Input

```
6
6
0 1 4 2 8 2
4 3 6 5 0 4
1 2 4 1 4 6
2 0 7 3 2 2
3 1 5 9 2 4
2 7 0 8 5 1
```

### Sample Output

23

```
#include <iostream>
#include <limits.h>
#include <vector>
#include <algorithm>
```

```
using namespace std;
```

```

int minCost(int n, int m, vector<vector<int>> &arr,
vector<vector<int>> &dp ){

    // write your code here
    // cout<<n<<" "<<m<<endl;
    for(int i{n-1};i >=0 ;i--) {
        for(int j {m-1}; j >=0 ;j--) {
            int a = 0;
            if(i+1 < n && j+1 < m) {
                a = min(dp[i][j+1],dp[i+1][j]);
            }else if(i+1<n) {
                a = dp[i+1][j];
            }else if(j+1<m){
                a = dp[i][j+1];
            }
            // cout<<i<<j <<" " <<a + arr[i][j]<<endl;
            dp[i][j] = a + arr[i][j];
        }
    }

    return dp[0][0];

}

int main() {

    int n;
    int m;

    cin >> n >> m;

    vector<vector<int>> arr(n, vector<int>(m));

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            cin >> arr[i][j];
        }
    }

    vector<vector<int>> dp(n, vector<int>(m));

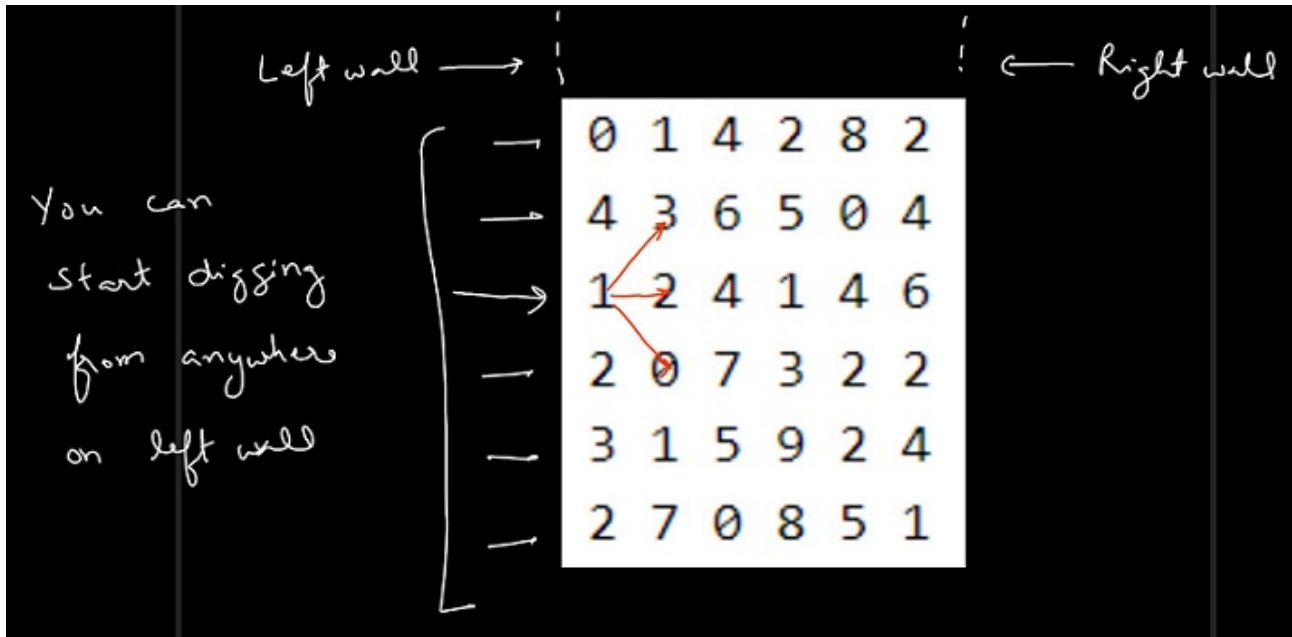
    cout << minCost(n, m, arr, dp);
}

```

# Goldmine

Easy

1. You are given a number  $n$ , representing the number of rows.
2. You are given a number  $m$ , representing the number of columns.
3. You are given  $n*m$  numbers, representing elements of 2d array  $a$ , which represents a gold mine.
4. You are standing in front of left wall and are supposed to dig to the right wall. You can start from any row in the left wall.
5. You are allowed to move 1 cell right-up (d1), 1 cell right (d2) or 1 cell right-down(d3).



6. Each cell has a value that is the amount of gold available in the cell.
7. You are required to identify the maximum amount of gold that can be dug out from the mine.

## Constraints

$$1 \leq n \leq 10^2$$

$$1 \leq m \leq 10^2$$

$$0 \leq e_1, e_2, \dots, n * m \text{ elements} \leq 1000$$

## Format

### Input

A number  $n$

A number  $m$

$e_{11}$

$e_{12}..$

$e_{21}$

$e_{22}..$

..  $n * m$  number of elements

### Output

An integer representing Maximum gold available.

## Example

### Sample Input

6

```

6
0 1 4 2 8 2
4 3 6 5 0 4
1 2 4 1 4 6
2 0 7 3 2 2
3 1 5 9 2 4
2 7 0 8 5 1

```

**Sample Output**

33

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
int collectGold(int n, int m, vector<vector<int>>& arr,
vector<vector<int>>& dp) {
```

```

    //write your code here
    for(int j{m-1} ; j >= 0 ; j--) {
        for(int i{n-1}; i>=0; i-- ) {
            // cout<<i<<" "<<j<<endl;
            if(j == m-1) {
                //last coloum
                dp[i][j] = arr[i][j];
            }else if(i == n-1) {
                if(n !=1){
                    dp[i][j] = arr[i][j] + max(dp[i][j+1],dp[i-1][j+1]);
                }else {
                    dp[i][j] = arr[i][j] + dp[i][j+1];
                }
            }else if(i == 0) {
                if(n !=1){
                    dp[i][j] = arr[i][j] + max(dp[i][j+1],dp[i+1][j+1]);
                }else {
                    dp[i][j] = arr[i][j] + dp[i][j+1];
                }
            }else {
                dp[i][j] = arr[i][j] + max(dp[i][j+1],max(dp[i+1]
[j+1],dp[i-1][j+1]));
            }
        }
    }

    int max = INT_MIN;
    for(int i{};i< n;i++) {
        // cout<<dp[i][0]<<"--"<<endl;
        if(dp[i][0] > max) {
            max = dp[i][0];
        }
    }
    return max;
}

```

```
int main() {  
    int n, m;  
    cin >> n >> m;  
  
    vector<vector<int>> arr(n, vector<int>(m, 0));  
    for (int i = 0; i < n; i++) {  
        for (int j = 0; j < m; j++) {  
            cin >> arr[i][j];  
        }  
    }  
  
    vector<vector<int>> dp(n, vector<int>(m, 0));  
  
    cout << collectGold(n, m, arr, dp);  
}
```

# Target Sum Subsets - Dp

Medium

1. You are given a number n, representing the count of elements.
2. You are given n numbers.
3. You are given a number "tar".
4. You are required to calculate and print true or false, if there is a subset the elements of which add up to "tar" or not.

## Constraints

1 <= n <= 30

0 <= n1, n2, .. n elements <= 20

0 <= tar <= 50

## Format

### Input

A number n

n1

n2

.. n number of elements

A number tar

### Output

true or false as required

## Example

### Sample Input

```
5
4
2
7
1
3
10
```

### Sample Output

```
true
```

```
#include <bits/stdc++.h>
using namespace std;
void input(vector<int> &arr)
{
    for (int i = 0; i < arr.size(); i++)
    {
        cin >> arr[i];
    }
}
void targetSumPair(vector<int> &arr, int target) {
    //write your code here
    vector<vector<bool>> > dp(arr.size()+1,vector<bool>(target +
1));

    for(int i {} ;i < dp.size() ;i++) {
        for(int j{};j < dp[0].size();j++) {
            if(i == 0 && j == 0) {
```



```

        dp[i][j] = true;
    } else if(i == 0) {
        dp[i][j] = false;
    } else if(j == 0) {
        dp[i][j] = true;
    } else {
        if(dp[i-1][j]){
            dp[i][j] = true;
        } else if(j - arr[i-1] >= 0 && dp[i-1][j -
arr[i-1]] == true) {
            dp[i][j] = true;
        } else{
            dp[i][j] = false;
        }
    }
}
}

cout<<std::boolalpha<<dp[arr.size()][target]<<endl;
return ;
}
int main()
{
    int n, target;
    cin >> n;
    vector<int> vec(n, 0);
    input(vec);
    cin >> target;
    targetSumPair(vec, target);
    return 0;
}

```

# Coin Change Combination

Easy

1. You are given a number  $n$ , representing the count of coins.
2. You are given  $n$  numbers, representing the denominations of  $n$  coins.
3. You are given a number "amt".
4. You are required to calculate and print the number of combinations of the  $n$  coins using which the amount "amt" can be paid.

Note1 -> You have an infinite supply of each coin denomination i.e. same coin denomination can be used for many installments in payment of "amt"

Note2 -> You are required to find the count of combinations and not permutations i.e.  
 $2 + 2 + 3 = 7$  and  $2 + 3 + 2 = 7$  and  $3 + 2 + 2 = 7$  are different permutations of same combination. You should treat them as 1 and not 3.

## Constraints

$1 \leq n \leq 30$   
 $0 \leq n_1, n_2, \dots, n_{\text{elements}} \leq 20$   
 $0 \leq \text{amt} \leq 50$

## Format

### Input

A number  $n$   
 $n_1$   
 $n_2$   
..  $n$  number of elements  
A number amt

### Output

A number representing the count of combinations of coins which can be used to pay the amount "amt"

## Example

### Sample Input

```
4
2
3
5
6
7
```

### Sample Output

```
2
```

```
#include <iostream>
#include <vector>
using namespace std;
```

```
int coinchangecombination(vector<int> coins, int amt, vector<int>
dp) {
```

```
// write your code here
```

```
dp[0] = 1;
for(int i{}; i < coins.size(); i++) {
```

```

        for(int j{coins[i]};j<dp.size();j++) {
            dp[j] += dp[j-coins[i]] ;
        }
    }
    cout<<dp[amt]<<endl;
    return dp[amt];
}

int main() {
    int n;
    cin >> n;
    vector<int> coins(n, 0);
    for (int i = 0; i < coins.size(); i++) {
        cin >> coins[i];
    }
    int amt;
    cin >> amt;
    vector<int> dp(amt + 1, 0);
    coinchangecombination(coins, amt, dp);
}

```

# Coin Change Permutations

Medium

1. You are given a number  $n$ , representing the count of coins.
2. You are given  $n$  numbers, representing the denominations of  $n$  coins.
3. You are given a number "amt".
4. You are required to calculate and print the number of permutations of the  $n$  coins using which the amount "amt" can be paid.

Note1 -> You have an infinite supply of each coin denomination i.e. same coin denomination can be used for many installments in payment of "amt"

Note2 -> You are required to find the count of permutations and not combinations i.e.  
 $2 + 2 + 3 = 7$  and  $2 + 3 + 2 = 7$  and  $3 + 2 + 2 = 7$  are different permutations of same combination. You should treat them as 3 and not 1.

## Constraints

$1 \leq n \leq 20$   
 $0 \leq n_1, n_2, \dots, n_{\text{elements}} \leq 20$   
 $0 \leq \text{amt} \leq 30$

## Format

### Input

A number  $n$   
 $n_1$   
 $n_2$   
..  $n$  number of elements  
A number amt

### Output

A number representing the count of combinations of coins which can be used to pay the amount "amt"

## Example

### Sample Input

4  
2  
3  
5  
6  
7

### Sample Output

5

```
#include <iostream>
#include <vector>
using namespace std;
```

```
int CCP(vector<int> coins, int amt, vector<int> dp) {
```

```
// write your code here
```

```
    dp[0] = 1;
    for(int i{} ; i < dp.size() ; i++) {
        for(int j{} ; j < coins.size() ; j++){
            if( i >= coins[j] ) {
                dp[i] += dp[i-coins[j]];
            }
        }
    }
}
```

```

    }
}

cout<<dp[amt];
return dp[amt];
}

int main() {
    int n;
    cin >> n;
    vector<int> coins(n, 0);
    for (int i = 0; i < coins.size(); i++) {
        cin >> coins[i];
    }
    int amt;
    cin >> amt;
    vector<int> dp(amt + 1, 0);
    CCP(coins, amt, dp);
}


```

## Analysis video

LEVEL- 1

COIN CHNGE PROBLEM


ANALYSIS



Sumeet Malik

For better experience visit

[www.nados.pepcoding.com](http://www.nados.pepcoding.com)



# Zero One Knapsack

Easy

1. You are given a number  $n$ , representing the count of items.
2. You are given  $n$  numbers, representing the values of  $n$  items.
3. You are given  $n$  numbers, representing the weights of  $n$  items.
3. You are given a number "cap", which is the capacity of a bag you've.
4. You are required to calculate and print the maximum value that can be created in the bag without overflowing it's capacity.

Note -> Each item can be taken 0 or 1 number of times. You are not allowed to put the same item again and again.

## Constraints

$1 \leq n \leq 20$   
 $0 \leq v_1, v_2, \dots, n \text{ elements} \leq 50$   
 $0 < w_1, w_2, \dots, n \text{ elements} \leq 10$   
 $0 < \text{cap} \leq 10$

## Format

### Input

A number  $n$   
 $v_1 v_2 \dots n \text{ number of elements}$   
 $w_1 w_2 \dots n \text{ number of elements}$   
A number cap

### Output

A number representing the maximum value that can be created in the bag without overflowing it's capacity

## Example

### Sample Input

```
5
15 14 10 45 30
2 5 1 3 4
7
```

### Sample Output

75

```
#include <iostream>
#include <vector>
```

```
using namespace std;
```

```
void zeroOneKnapsack(int n,vector<int> val, vector<int> weight,int cap){
```

```
// write your code here
vector< vector< int> > dp(val.size()+1,vector<int>(cap+1,0));
for(int i{1};i<dp.size(); i++) {
    for(int j {1}; j<dp[0].size(); j++){
        // cout<<i<<j<<endl;
        if(j - weight[i-1] < 0){
            dp[i][j] = dp[i-1][j]; //when that player doesn't bat
        }else {
```

```

        int remCap = j-weight[i-1];
        if(dp[i-1][remCap] + val[i-1] > dp[i-1][j]){
            dp[i][j] = dp[i-1][remCap] + val[i-1];
        }else{
            dp[i][j] = dp[i-1][j];
        }
    }
}
}
}
cout<<dp[dp.size()-1][cap]<<endl;

return ;
}

```

```

//my solution you can also get the number of ball at the end
//but bekaar solution"GHATIYA"
// void zeroOneKnapsack(int n,vector<int> val, vector<int>
weight,int cap){

```

```

// // write your code here
// //pair<val, weight>
// vector< vector< pair<int,int> *> > dp(val.size()
+1,vector<pair<int,int>*>(cap+1,NULL));
// // cout<<"check 2"<<endl;

// for(int i{}; i< dp.size();i++) {
//     for(int j{}; j< dp[0].size();j++) {
//         cout<<i<<" "<<j<<endl;
//         if(i == 0 && j == 0){
//             // cout<<"con1 ";
//             pair<int,int>* np = new pair<int,int> (0,0);
//             dp[i][j] = np;
//         }else if(i == 0) {
//             // cout<<"con2 " <<i<<" "<<j<<endl;
//             continue;
//         }else if(j== 0) {
//             // cout<<"con2.6 ";
//             pair<int,int>* np = new pair<int,int> (0,0);
//             dp[i][j] = np;
//         }else if(j - weight[i-1] >= 0){
//             pair<int,int> *up = dp[i-1][j-weight[i-1]];//
upper previous
//             pair<int,int> *j_upper = dp[i-1][j];//just
upper
//             if(j_upper == NULL && up == NULL){
//                 // cout<<"con3.1 ";
//                 if(weight[i-1] <= cap){
//                     pair<int,int>* np = new pair<int,int>
(val[i-1],weight[i-1]);
//                     dp[i][j]= np;
//                 }
//             }
//         }
//     }
// }

```





```

//          dp[i][j] = j_upper;
//          // if(j_upper->second > weight[i-1]){
//          //      dp[i][j] = j_upper;
//          // }else if (j_upper->second ==
weight[i-1]) {
//          //      int smallerValue = val[i-1]<
j_upper->first ? val[i-1] : j_upper->first;
//          //      pair<int,int>* np = new
pair<int,int> (smallerValue,weight[i-1]);
//          //      dp[i][j]= np;
//          //      }
//          }
//      }

//      // cout<<i<<" "<<j<<"->";
//      // if(dp[i][j] != NULL) {
//      //      cout<<dp[i][j]->first<<" ~ "<<dp[i][j]->second;
//      //      }
//      // cout<<endl;
//      }
//  }
//  //      cout<<"check 3"<<endl;

//      // cout<<dp[dp.size()-1][dp[0].size()-1]->first<<"$
//      // cout<<dp[dp.size()-1][dp[0].size()-1]->second<<endl;
//      // cout<<dp[dp.size()-1][dp[0].size()-1]->first<<endl;
//      if(dp[dp.size()-1][dp[0].size()-1] != NULL) {
//      cout<<dp[dp.size()-1][dp[0].size()-1]->first<<endl;
//      }else{
//      cout<<0<<endl;
//      }
//      return ;
//  }

```

```

int main() {

    int n;
    cin >> n;

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

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

```

```
}
```

```
int cap;  
cin >> cap;
```

```
zeroOneKnapsack(n,val,weight,cap);
```

```
}
```

# Unbounded Knapsack

Easy

1. You are given a number  $n$ , representing the count of items.
2. You are given  $n$  numbers, representing the values of  $n$  items.
3. You are given  $n$  numbers, representing the weights of  $n$  items.
3. You are given a number "cap", which is the capacity of a bag you've.
4. You are required to calculate and print the maximum value that can be created in the bag without overflowing it's capacity.

Note -> Each item can be taken any number of times. You are allowed to put the same item again and again.

## Constraints

$1 \leq n \leq 20$   
 $0 \leq v_1, v_2, \dots, n \text{ elements} \leq 50$   
 $0 < w_1, w_2, \dots, n \text{ elements} \leq 10$   
 $0 < \text{cap} \leq 10$

## Format

### Input

A number  $n$

$v_1 v_2 \dots n$  number of elements

$w_1 w_2 \dots n$  number of elements

A number cap

### Output

A number representing the maximum value that can be created in the bag without overflowing it's capacity

## Example

### Sample Input

```
5
15 14 10 45 30
2 5 1 3 4
7
```

### Sample Output

```
100
```

```
#include <iostream>
```

```
#include <vector>
```

```
#include<algorithm>
```

```
using namespace std;
```

```
void unboundedKnapsack(int n,vector<int> val, vector<int>
weight,int cap){
```

```
// write your code here
```

```
// i did it like coin change combination
```

```
// sir did by coin change permutation both results same here
```

```
vector<int> dp (cap+1,0);
```

```
for(int i{}; i < n;i++) {
```

```

        for(int j{}; j< cap + 1;j++) {
            if(j >= weight[i] ){
                dp[j] = max(dp[j] , dp[j-weight[i]] + val[i]);
            }
        }
    }

    cout<<dp[cap]<<endl;
    return ;
}

```

```

int main() {

    int n;
    cin>>n;
    vector<int> val(n);
    for (int i = 0; i < n; i++) {

        cin>>val[i];
    }
    vector<int> weight(n);
    for (int i = 0; i < n; i++) {
        cin>>weight[i];
    }
    int cap;
    cin>>cap;

    unboundedKnapsack(n,val, weight, cap);

}

```

# Fractional Knapsack - Official

Easy

1. You are given a number  $n$ , representing the count of items.
2. You are given  $n$  numbers, representing the values of  $n$  items.
3. You are given  $n$  numbers, representing the weights of  $n$  items.
3. You are given a number "cap", which is the capacity of a bag you've.
4. You are required to calculate and print the maximum value that can be created in the bag without overflowing it's capacity.

Note1 -> Items can be added to the bag even partially. But you are not allowed to put same items again and again to the bag.

## Constraints

```
1 <= n <= 20
0 <= v1, v2, .. n elements <= 50
0 < w1, w2, .. n elements <= 10
0 < cap <= 10
```

## Format

### Input

A number  $n$

$v1\ v2\ \dots\ n$  number of elements

$w1\ w2\ \dots\ n$  number of elements

A number cap

### Output

A decimal number representing the maximum value that can be created in the bag without overflowing it's capacity

## Example

### Sample Input

```
10
33 14 50 9 8 11 6 40 2 15
7 2 5 9 3 2 1 10 3 3
5
```

### Sample Output

```
50.0
```

//all cases don't pass may be some issue

```
#include <iostream>
#include <vector>
#include<algorithm>
#include <iomanip>
#include <queue>
```

```
using namespace std;
```

```
class Pair{
```

```
public:
```

```
double val{};
```

```
int weight{};
```

```
double avg{};
```

```
Pair(double val, int weight, double avg) {
```

```
    this-> val = val;
```

```

        this-> weight = weight;
        this -> avg = avg;
    }
};
class my_comp{
public:
    bool operator()(Pair& a,Pair & b){
        return a.avg < b.avg;
    }
};
void fractionalKnapsack(int n,vector<double> val, vector<int>
weight,int cap){
// write your code here
    priority_queue < Pair,vector<Pair>, my_comp > pq;

    for(int i{};i < n;i++) {
        Pair np(val[i],weight[i],val[i]/weight[i]);
        pq.push(np);
    }
    // while(pq.empty() == false) {
    //     Pair a = pq.top();pq.pop();
    //     cout<<a.val<<" "<<a.weight<<" "<<a.avg<<endl;
    // }

    int ballsPlayed{};//weight till now
    double runMade{};//value gathered
    while(ballsPlayed < cap) {
        Pair a = pq.top();pq.pop();
        if(a.weight + ballsPlayed <= cap) {
            ballsPlayed += a.weight;
            runMade += a.val;
        }else{
            runMade += (cap - ballsPlayed) * a.avg;
            ballsPlayed += (cap - ballsPlayed);
        }
        // cout<<a.weight<<" "<<a.val<<" "<<a.avg<<"
"<<ballsPlayed<<" "<<runMade<<endl;
    }

    cout<<runMade<<endl;

// test case 0 and test case 1 problematic in just printing the
answer

    return ;
}

```

```
int main() {  
  
    int n;  
    cin >> n;  
  
    vector<double> val(n);  
    for (int i = 0; i < n; i++) {  
        cin >> val[i];  
    }  
  
    vector<int> weight(n);  
    for (int i = 0; i < n; i++) {  
        cin >> weight[i];  
    }  
  
    int cap;  
    cin >> cap;  
  
    fractionalKnapsack(n, val, weight, cap);  
  
}
```

# Count Binary Strings

Easy

1. You are given a number n.
2. You are required to print the number of binary strings of length n with no consecutive 0's.

## Constraints

$0 < n \leq 45$

## Format

### Input

A number n

### Output

A number representing the number of binary strings of length n with no consecutive 0's.

## Example

### Sample Input

6

### Sample Output

21

```
#include <iostream>
#include <vector>
```

```
using namespace std;
```

```
long int noOfBinaryStringWithoutConsecutiveZerosOfLenght(int n) {
```

```
    //pepcoding solution
```

```
    if(n == 0){
```

```
        return 1;
```

```
    }
```

```
    long int old0{1}; //
```

```
    long int old1{1};
```

```
    long int new0{1};
```

```
    long int new1{1};
```

```
    // old0 = 1;
```

```
    // old1 = 1;
```

```
    for(int i=2;i<= n;i++) {
```

```
        new0 = old1;
```

```
        new1 = old0 + old1;
```

```
        old0 = new0;
```

```
        old1 = new1;
```

```
    }
```

```
    return new0 + new1;
```

```
    //mysolution
```

```
    // if(n == 0){
```



```

//     return 1;
// }
//     vector<long int> dp(n+1,0);
//     dp[0] = 1;
//     dp[1] = 2;
//     for(int i{2};i< n+1;i++) {
//         dp[i] = dp[i-1]+dp[i-2];
//         // cout<<i<<"-> "<<dp[i]<<"      ";
//     }
//     return dp[n];
}

int main () {
    int n{};
    cin>>n;
    //using long just for 45 ,till 44 int works fine
    long int ans =
noOfBinaryStringWithoutConsecutiveZerosOfLenght(n);
    cout<< ans<<endl;
}
//GHATIYA
// #include <iostream>
// #include <vector>

// using namespace std;

// vector<int> factorial(int a){
//     vector<int> fac(a+1); // creating vector of lenght 8 cause for
// facotrial 7 =>0 1 2 3 4 5 6 7
//     fac[0]=1;
//     for(int i{1};i<a+1;i++ ) {
//         fac[i] = fac[i-1]*i;
//     }
//     return fac;
// }

// int noOfBinaryStringWithoutConsecutiveZerosOfLenght(int n) {
//     int result{};
//     int t = n+1/2;
//     int noOfZeros{};
//     int places{n+1}; //for zeros

//     vector<int> fac = factorial(n+1); //tell n = 6 then we
// requires factorial till 7
//     for(int a:fac){
//         cout<<a<<" ";
//     }
//     cout<<endl;
//     while(noOfZeros <= t) {

```

```

//      result += fac[places] / (fac[places-noOfZeros] *
fac[noOfZeros]);
//      places--;
//      noOfZeros++;
//  }
//  return result;
//  }

// int main() {
//     int n{};
//     cin>>n;
//     int ans =
noOfBinaryStringWithoutConsecutiveZerosOfLenght(n);

//     cout<<ans<<endl;
//  }

```

# Arrange Buildings

Easy

1. You are given a number  $n$ , which represents the length of a road. The road has  $n$  plots on it's each side.
2. The road is to be so planned that there should not be consecutive buildings on either side of the road.
3. You are required to find and print the number of ways in which the buildings can be built on both side of roads.

## Constraints

$$0 < n \leq 45$$

## Format

### Input

A number  $n$

### Output

A number representing the number of ways in which the buildings can be built on both side of roads.

## Example

### Sample Input

6

### Sample Output

441

```
#include <iostream>
#include <vector>
```

```
using namespace std;
```

```
long int noOfWayOfHavingNotConsecutiveBuildingOnTheSideOfRoad(int
n) {
```

```
    //pepcoding solution
    if(n == 0){
        return 1;
    }
```

```
    long int oldB{1}; //
    long int oldS{1};
    long int newB{1};
    long int newS{1};
```

```
    for(int i=2;i<= n;i++) {
        newB = oldS;
        newS = oldB + oldS;

        oldB = newB;
        oldS = newS;
    }
```

```
    return (newB + newS) *(newB + newS);
```

```
}
```

```
int main () {  
    int n{};  
    cin>>n;  
    //using long just for 45 ,till 44 int works fine  
    long int ans =  
noOfWayOfHavingNotConsecutiveBuildingOnTheSideOfRoad(n);  
    cout<< ans<<endl;  
}
```

# Count Encodings

Easy

1. You are given a string str of digits. (will never start with a 0)
2. You are required to encode the str as per following rules
  - 1 -> a
  - 2 -> b
  - 3 -> c
  - ..
  - 25 -> y
  - 26 -> z
3. You are required to calculate and print the count of encodings for the string str.

For 123 -> there are 3 encodings. abc, aw, lc

For 993 -> there is 1 encoding. iic

For 013 -> This is an invalid input. A string starting with 0 will not be passed.

For 103 -> there is 1 encoding. jc

For 303 -> there are 0 encodings. But such a string maybe passed. In this case print 0.

## Constraints

$0 < \text{str.length} \leq 10$

## Format

### Input

A string str

### Output

count of encodings

## Example

### Sample Input

123

### Sample Output

3

//Don't have solution (I think similar  
//question of leet code "decode ways")

# Count A+b+c+ Subsequences

Easy

1. You are given a string str.
2. You are required to calculate and print the count of subsequences of the nature a+b+c+.

For abbc -> there are 3 subsequences. abc, abc, abbc

For abcabc -> there are 7 subsequences. abc, abc, abbc, aabc, abcc, abc, abc.

## Constraints

$0 < \text{str.length} \leq 10$

## Format

### Input

A string str

### Output

count of subsequences of the nature a+b+c+

## Example

### Sample Input

abcabc

### Sample Output

7

```
#include<iostream>
#include<string>

using namespace std;

int countAPlusBPlusCSubsequence(string s) {
    int a{};
    int ab{};
    int abc{};

    for(int i{}; i< s.length();i++) {
        if(s[i] == 'a'){
            a = 2 * a +1;
        }else if( s[i] == 'b') {
            ab = 2 * ab + a;
        }else if (s[i] == 'c') {
            abc = 2 * abc + ab;
        }
    }

    return abc;
}

int main() {
    string s{};
    cin>> s;
    int ans = countAPlusBPlusCSubsequence(s);
    cout<<ans <<endl;
}
```

# Maximum Sum Non Adjacent Elements

Easy

1. You are given a number n, representing the count of elements.
2. You are given n numbers, representing n elements.
3. You are required to find the maximum sum of a subsequence with no adjacent elements.

## Constraints

$1 \leq n \leq 1000$

$-1000 \leq n_1, n_2, \dots, n_{\text{elements}} \leq 1000$

## Format

### Input

A number n

n1

n2

.. n number of elements

### Output

A number representing the maximum sum of a subsequence with no adjacent elements.

## Example

### Sample Input

```
6
5
10
10
100
5
6
```

### Sample Output

```
116
```

```
#include<iostream>
#include<vector>
#include<algorithm>
```

```
using namespace std;
```

```
int main() {
    int n{};
    cin>>n;
    vector<int> vec(n);
    for(int i{};i<n;i++) {
        cin>>vec[i];
    }
    int pinc{vec[0]}; //include 0th
    int pexc{}; //exclude 0th
    int ninc{vec[0]};
    int nexc{};

    for(int i{1} ; i < n ; i++) {
        ninc = pexc + vec[i];
        nexc = max(pinc,pexc);
    }
}
```

```
    pinc = ninc;  
    pexc = nexc;  
    // cout<<ninc<<" "<<nexc<<endl;  
}  
  
cout<<max(ninc,nexc)<<endl;  
  
}
```



# Paint House

Easy

1. You are given a number  $n$ , representing the number of houses.
2. In the next  $n$  rows, you are given 3 space separated numbers representing the cost of painting  $n$ th house with red or blue or green color.
3. You are required to calculate and print the minimum cost of painting all houses without painting any consecutive house with same color.

## Constraints

$1 \leq n \leq 1000$

$0 \leq n1red, n1blue, \dots \leq 1000$

## Format

### Input

A number  $n$

$n1red\ n1blue\ n1green$

$n2red\ n2blue\ n2green$

..  $n$  number of elements

### Output

A number representing the minimum cost of painting all houses without painting any consecutive house with same color.

## Example

### Sample Input

```
4
1 5 7
5 8 4
3 2 9
1 2 4
```

### Sample Output

8

```
#include<iostream>
#include<vector>
#include<algorithm>
```

```
using namespace std;
```

```
int main() {
    int n{};
    cin>>n;
    vector<vector<int>> colour(n,vector<int> (3));
    for(int i{};i< n;i++) {
        for(int j{};j<3;j++) {
            cin>>colour[i][j];
        }
    }

    long int pr{colour[0][0]};
    long int pg{colour[0][1]};
    long int pb{colour[0][2]};
    long int nr{colour[0][0]};
```

```
long int ng{colour[0][1]};
long int nb{colour[0][2]};

for(int i{1} ; i < n ; i++) {
    nr = min(pg,pb) + colour[i][0];
    ng = min(pr,pb) + colour[i][1];
    nb = min(pr,pg) + colour[i][2];

    pr = nr;
    pg = ng;
    pb = nb;
}

cout<<min(min(nr,ng),nb)<<endl;

}
```

# Paint House - Many Colors

Easy

1. You are given a number  $n$  and a number  $k$  separated by a space, representing the number of houses and number of colors.
2. In the next  $n$  rows, you are given  $k$  space separated numbers representing the cost of painting  $n$ th house with one of the  $k$  colors.
3. You are required to calculate and print the minimum cost of painting all houses without painting any consecutive house with same color.

## Constraints

$1 \leq n \leq 1000$   
 $1 \leq k \leq 10$   
 $0 \leq n1-0th, n1-1st, \dots \leq 1000$

## Format

### Input

A number  $n$

$n1-0th$   $n1-1st$   $n1-2nd$  ..  $n1-kth$

$n2-0th$   $n2-1st$   $n2-2nd$  ..  $n2-kth$

..  $n$  number of elements

### Output

A number representing the minimum cost of painting all houses without painting any consecutive house with same color.

## Example

### Sample Input

```
4 3
1 5 7
5 8 4
3 2 9
1 2 4
```

### Sample Output

8

```
#include <iostream>
#include <bits/stdc++.h>
```

```
using namespace std ;
```

```
// int minExcept(vector<int> vec, int e){
//     int min = INT_MAX;
//     for(int i{};i<vec.size();i++) {
//         if(i == e){
//             continue;
//         }
//         if(vec[i]< min){
//             min = vec[i];
//         }
//     }
//     return min;
// }
```

```

int main()
{
    int n ;
    cin >> n ;
    int k ;
    cin >> k ;
    //write your code from here

    vector<vector<int>> colour(n,vector<int> (k));
    for(int i{};i< n;i++) {
        for(int j{};j<k;j++) {
            cin>>colour[i][j];
        }
    }

    int least = INT_MAX;
    int sleast = INT_MAX;
    int leastJ {} ;
    vector<vector<int>> dp(n,vector<int> (k));
    for(int i{};i<k;i++) {
        dp[0][i] = colour[0][i];
        if(dp[0][i] < least ){
            leastJ = i;
            sleast = least;
            least = dp[0][i];
        }else if(dp[0][i] < sleast ){
            sleast = dp[0][i];
        }
    }
    // cout<<"least "<<least<<" "<<sleast<<leastJ<<endl;
    for(int i {1}; i<n;i++) {
        int l = least;
        int sl = sleast;
        int li = leastJ;
        least = INT_MAX;
        sleast = INT_MAX;
        leastJ = 0;
        for(int j {};j< k;j++) {
            if(j != li){
                dp[i][j] = colour[i][j] + l;
            }else{
                dp[i][j] = colour[i][j] + sl;
            }
            if(dp[i][j] <= least ){
                leastJ = j;
                sleast = least;
                least = dp[i][j];
            }else if(dp[i][j] <= sleast ){
                sleast = dp[i][j];
            }
        }
    }
}

```

```
    // cout<<"least "<<least<<" "<<least<<leastJ<<endl;
}
// for(auto a:dp){
//     for(auto i:a){
//         cout<<i<<" ";
//     }
//     cout<<endl;
// }

cout<<least<<endl;
return 0;
}
```

# Paint Fence

Easy

1. You are given a number n and a number k in separate lines, representing the number of fences and number of colors.
2. You are required to calculate and print the number of ways in which the fences could be painted so that not more than two consecutive fences have same colors.

## Constraints

1 <= n <= 10

1 <= k <= 10

## Format

### Input

A number n

A number k

### Output

A number representing the number of ways in which the fences could be painted so that not more than two fences have same colors.

## Example

### Sample Input

8

3

### Sample Output

3672

```
#include <iostream>
#include <vector>

using namespace std;

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

    //write your code here
    if(n== 0){
        cout<<0<<endl;
        return 0;
    }else if(n == 1) {
        cout<<k<<endl;
        return 0;
    }else if( n== 2){
        cout<<k*k<<endl;
        return 0;
    }
    int secondLast {1};
```

```

int last{k};
int removedFromLast  {};
int current  {};

for(int i = 3 ;i<= n;i++) {
    current = last*k -(secondLast - removedFromLast);

    removedFromLast = secondLast - removedFromLast;
    secondLast = last;
    last = current;
}

cout<<current*k<<endl;

// vector<int> dp(n);
// dp[0] = 1;
// dp[1] = k;
// int removedFromLast  {};
// // cout<<"check1"<<endl;
// for(int i = 2; i< dp.size();i++) {
//     dp[i] = (dp[i-1] * k) - (dp[i-2] - removedFromLast);
//     removedFromLast = dp[i-2] - removedFromLast;
//     // cout<<"till " <<i<<" "<<dp[i]<<endl;
// }
// cout<<dp[n-1]*k<<endl;

return 0;
}

```

# Tiling With 2 \* 1 Tiles

Easy

1. You are given a number n representing the length of a floor space which is 2m wide. It's a 2 \* n board.
2. You've an infinite supply of 2 \* 1 tiles.
3. You are required to calculate and print the number of ways floor can be tiled using tiles.

## Constraints

1 <= n <= 100

## Format

### Input

A number n

### Output

A number representing the number of ways in which the number of ways floor can be tiled using tiles.

## Example

### Sample Input

8

### Sample Output

34

```
#include <iostream>
```

```
using namespace std;
```

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

    //pepcoding solution good read this
    if(n == 0) {
        cout<<0<<endl;
        return 0;
    }else if(n == 1){
        cout<<1<<endl;
        return 0;
    }else if(n == 2){
        cout<<2<<endl;
        return 0;
    }
    int secondLast = 1;
    int last = 2;
    int curr {};
    for(int i = 3 ; i <= n;i++) {
        curr = last + secondLast;
        secondLast = last;
        last = curr;
    }
    cout<<curr<<endl;

    //mysolution
```



```
//  if(n == 0) {  
//      cout<<0<<endl;  
//      return 0;  
//  }  
  
//  int lastTileHorizontal {0};  
//  int lastTileVertical{1};  
//  int newLastTileHorizontal {0};  
//  int newLastTileVertical{1};  
//  for(int i = 2 ; i <= n;i++) {  
//      newLastTileHorizontal = lastTileVertical;  
//      newLastTileVertical = lastTileHorizontal +  
lastTileVertical;  
  
//      lastTileHorizontal = newLastTileHorizontal;  
//      lastTileVertical = newLastTileVertical;  
//  }  
  
//  cout<<newLastTileHorizontal + newLastTileVertical<<endl;  
return 0;  
}
```

# Tiling With $M * 1$ Tiles

Easy

1. You are given a number  $n$  and a number  $m$  separated by line-break representing the length and breadth of a  $m * n$  floor.
2. You've an infinite supply of  $m * 1$  tiles.
3. You are required to calculate and print the number of ways floor can be tiled using tiles.

## Constraints

$1 \leq n \leq 100$

$1 \leq m \leq 50$

## Format

### Input

A number  $n$

A number  $m$

### Output

A number representing the number of ways in which the number of ways floor can be tiled using tiles.

## Example

### Sample Input

39

16

### Sample Output

61

```
#include <iostream>
```

```
#include <vector>
```

```
using namespace std ;
```

```
int main() {
    int n ;
    cin >> n ;
    int m ;
    cin >> m ;
    // write your code here
    if(n < m){
        cout<<1<<endl;
        return 0;
    }else if(n == m) {
        cout<<2<<endl;
        return 0;
    }
    vector<int> dp(n+1);
    for(int i{0};i<m;i++){ //0 = 0tile space
        dp[i] = 1;
    }

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

```
cout<<dp[n]<<endl;  
return 0;
```

```
}
```

# Friends Pairing

Easy

1. You are given a number n, representing the number of friends.
2. Each friend can stay single or pair up with any of it's friends.
3. You are required to print the number of ways in which these friends can stay single or pair up.

E.g.

1 person can stay single or pair up in 1 way.

2 people can stay singles or pair up in 2 ways. 12 => 1-2, 12.

3 people (123) can stay singles or pair up in 4 ways. 123 => 1-2-3, 12-3, 13-2, 23-1.

## Constraints

$0 \leq n \leq 20$

## Format

### Input

A number n

### Output

A number representing the number of ways in which n friends can stay single or pair up.

## Example

### Sample Input

4

### Sample Output

10

```
#include <iostream>
#include <vector>
```

```
using namespace std ;
```

```
int main() {
    int n ;
    cin >> n ;
    // write your code here
    if(n == 0){
        return 0;
    }else if( n==1) {
        cout<<1<<endl;
        return 0;
    }else if(n == 2) {
        cout<<2<<endl;
        return 0;
    }

    int secondLast = 1;
    int last = 2;
    int curr{};

    for(int i = 3 ; i <= n;i++) {
        curr = last + (i-1) * secondLast;

        secondLast = last;
```

```
    last = curr;  
}  
  
cout<<curr<<endl;  
return 0;  
}
```

# Partition Into Subsets

Easy

1. You are given a number  $n$ , representing the number of elements.
2. You are given a number  $k$ , representing the number of subsets.
3. You are required to print the number of ways in which these elements can be partitioned in  $k$  non-empty subsets.

E.g.

For  $n = 4$  and  $k = 3$  total ways is 6

12-3-4

1-23-4

13-2-4

14-2-3

1-24-3

1-2-34

## Constraints

$0 \leq n \leq 20$

$0 \leq k \leq n$

## Format

### Input

A number  $n$

A number  $k$

### Output

A number representing the number of ways in which these elements can be partitioned in  $k$  non-empty subsets.

## Example

### Sample Input

4

3

### Sample Output

6

```
#include <iostream>
```

```
#include <vector>
```

```
using namespace std ;
```

```
int main() {
```

```
    int n ;
```

```
    cin >> n ;
```

```
    int k{};
```

```
    cin>> k;
```

```
    if(k == 0 || n == 0 || k > n){
```

```
        cout<<0<<endl;
```

```
        return 0;
```

```
    }else if(k == 1){
```

```
        cout<<1<<endl;
```

```
        return 0;
```

```
    }
```

```
    vector<vector<long int>> dp (k+1,vector<long int>(n+1));
```

```

for(int t{1};t <= k ; t++) { //teams
    for(int p{1}; p<= n ;p++) { //people
        if(p < t){
            dp[t][p] = 0;
        }else if( t == p){
            dp[t][p] = 1;
        }else {
            dp[t][p] = t * dp[t][p-1] + dp[t-1][p-1];
        }
    }
}

cout<<dp[k][n]<<endl;
return 0;
}

```

# Buy And Sell Stocks - One Transaction Allowed

Easy

1. You are given a number n, representing the number of days.
2. You are given n numbers, where ith number represents price of stock on ith day.
3. You are required to print the maximum profit you can make if you are allowed a single transaction.

## Constraints

$0 \leq n \leq 20$

$0 \leq n_1, n_2, \dots \leq 10$

## Format

### Input

A number n

.. n more elements

### Output

A number representing the maximum profit you can make if you are allowed a single transaction.

## Example

### Sample Input

```
9
11
6
7
19
4
1
6
18
4
```

### Sample Output

```
17
```

```
#include <iostream>
```

```
#include <vector>
```

```
using namespace std;
```

```
void Transaction(vector<int> arr) {
    //write your code here
    int min = arr[0];
    int profit {};
    for(int i{1}; i<arr.size(); i++) {
        if(arr[i] < min) {
            min = arr[i];
            continue;
        }
        int diff = arr[i] - min ;
        if(diff > profit){
            profit = diff;
        }
    }
}
```



```
    }  
  }  
  cout<<profit<<endl;  
}
```

```
int main() {  
    int n;  
    cin >> n;  
    vector<int>arr(n, 0);  
    for (int i = 0; i < arr.size(); i++) {  
        cin >> arr[i];  
    }  
    Transaction(arr);  
    return 0;  
}
```

# Buy And Sell Stocks - Infinite Transactions Allowed

Easy

1. You are given a number n, representing the number of days.
  2. You are given n numbers, where ith number represents price of stock on ith day.
  3. You are required to print the maximum profit you can make if you are allowed infinite transactions.
- Note - There can be no overlapping transaction. One transaction needs to be closed (a buy followed by a sell) before opening another transaction (another buy)

## Constraints

$0 \leq n \leq 20$

$0 \leq n_1, n_2, \dots \leq 10$

## Format

### Input

A number n

.. n more elements

### Output

A number representing the maximum profit you can make if you are allowed infinite transactions.

## Example

### Sample Input

```
9
11
6
7
19
4
1
6
18
4
```

### Sample Output

```
30
```

```
#include <iostream>
```

```
#include <vector>
```

```
using namespace std;
```

```
void Transaction(vector<int> arr) {
    //write your code here
```

```
    int bd{};
    int sd{};
    int profit{};
    for(int i{1}; i< arr.size();i++) {
        if(arr[i] > arr[i-1]) {
            sd++;
        }else{
            profit += arr[sd]- arr[bd];
            sd = bd = i;
        }
    }
    return profit;
}
```

```

    }
}
profit += arr[sd]- arr[bd];
cout<<profit<<endl;

// int totalProfit = 0;
//     // int profit {};
//     int previous{arr[0]};
//     for(int i{1};i<arr.size(); i++) {
//         if(arr[i] > previous) {
//             totalProfit += arr[i] - previous;
//         }
//         previous = arr[i];
//     }
// cout<<totalProfit<<endl;

```

```

//little difficult
// int min = arr[0];
// int totalProfit = 0;
// int profit {};
// int previous{arr[0]};
// for(int i{1};i<arr.size(); i++) {
//     if(arr[i] < previous) {
//         min = arr[i];
//         profit = 0;
//         previous = arr[i];
//         continue;
//     }
//     int diff = arr[i] - min ;
//     if(diff > profit){
//         totalProfit += diff - profit;
//         profit = diff;
//     }
//     previous = arr[i];
// }
// cout<<totalProfit<<endl;
}

```

```

int main() {
    int n;
    cin >> n;
    vector<int>arr(n, 0);
    for (int i = 0; i < arr.size(); i++) {
        cin >> arr[i];
    }
    Transaction(arr);
    return 0;
}

```

# Buy And Sell Stocks With Transaction Fee - Infinite Transactions Allowed

Medium

1. You are given a number  $n$ , representing the number of days.
2. You are given  $n$  numbers, where  $i$ th number represents price of stock on  $i$ th day.
3. You are given a number  $fee$ , representing the transaction fee for every transaction.
4. You are required to print the maximum profit you can make if you are allowed infinite transactions, but has to pay " $fee$ " for every closed transaction.

Note - There can be no overlapping transaction. One transaction needs to be closed (a buy followed by a sell) before opening another transaction (another buy).

## Constraints

$0 \leq n \leq 20$

$0 \leq n_1, n_2, \dots \leq 10$

$0 \leq fee \leq 5$

## Format

### Input

A number  $n$

..  $n$  more elements

A number  $fee$

### Output

A number representing the maximum profit you can make if you are allowed infinite transactions with transaction  $fee$ .

## Example

### Sample Input

```
12
10
15
17
20
16
18
22
20
22
20
23
25
3
```

### Sample Output

```
13
```

```
#include <iostream>
#include <vector>
```

```
using namespace std;
```

```
void transactions(vector<int> arr, int fee){
    // write your code here
```

```

    int pbsp{-arr[0]}; //previous buy state profit(balance,in buy
state we have to pay)
    int pssp{}; //previous sold state profit
    int bsp{}; //buy state profit
    int ssp{}; //sold state profit
    for(int i = 1 ; i < arr.size(); i++) {
        bsp = max( pbsp , pssp - arr[i] ); //max because -10 is
bigger than -15(-10 is a better state to be in )
        ssp = max (pssp,arr[i]+pbsp-fee );

        pbsp = bsp;
        pssp = ssp;
    }
    cout<<ssp<<endl;
    return ;
}

int main() {
    int n;
    cin>> n;
    vector<int> arr(n,0);
    for (int i = 0; i < arr.size(); i++) {
        cin>> arr[i];
    }
    int fee;
    cin>> fee;

    transactions(arr,fee);

    return 0;
}

```

# Buy And Sell Stocks With Cooldown - Infinite Transaction Allowed

Medium

1. You are given a number  $n$ , representing the number of days.
  2. You are given  $n$  numbers, where  $i$ th number represents price of stock on  $i$ th day.
  3. You are required to print the maximum profit you can make if you are allowed infinite transactions, but have to cooldown for 1 day after 1 transaction  
i.e. you cannot buy on the next day after you sell, you have to cooldown for a day at-least before buying again.
- Note - There can be no overlapping transaction. One transaction needs to be closed (a buy followed by a sell) before opening another transaction (another buy).

## Constraints

$0 \leq n \leq 20$

$0 \leq n_1, n_2, \dots \leq 10$

## Format

### Input

A number  $n$

..  $n$  more elements

### Output

A number representing the maximum profit you can make if you are allowed infinite transactions with cooldown of 1 day.

## Example

### Sample Input

```
12
10
15
17
20
16
18
22
20
22
20
23
25
```

### Sample Output

```
19
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

void transaction(vector<int> arr){
    // write your code here
    if(arr.size() == 1){
```

```

        cout<<0<<endl;
        return ;
    }else if ( arr.size() == 2 ){
        if(arr[1] - arr[0] >= 0){
            cout<<arr[1] - arr[0]<<endl;
            return ;
        }else {
            cout<<0<<endl;
            return ;
        }
    }
}

int lastToLastSsp{};
int pbsp = max (-arr[0],-arr[1]); //previous buy state
profit(balance,in buy state we have to pay)
int pssp = max (0, arr[1] - arr[0]); //previous sold state
profit
int bsp{}; //buy state profit
int ssp{}; //sold state profit
for(int i = 2 ; i < arr.size(); i++) {
    bsp = max( pbsp , lastToLastSsp - arr[i] ); //max because -10
    is bigger than -15(-10 is a better state to be in )
    ssp = max(pssp, arr[i]+pbsp);

    lastToLastSsp = pssp;
    pbsp = bsp;
    pssp = ssp;
}
cout<<ssp<<endl;
return ;
}

int main(){
    int n;
    cin>>n;
    vector<int> arr(n,0);
    for (int i = 0; i < arr.size(); i++) {
        cin>> arr[i] ;
    }

    transaction(arr);
    return 0;
}

```

# Buy And Sell Stocks - K Transactions Allowed

Easy

1. You are given a number n, representing the number of days.
  2. You are given n numbers, where ith number represents price of stock on ith day.
  3. You are given a number k, representing the number of transactions allowed.
  3. You are required to print the maximum profit you can make if you are allowed k transactions at-most.
- Note - There can be no overlapping transaction. One transaction needs to be closed (a buy followed by a sell) before opening another transaction (another buy).

## Constraints

$0 \leq n \leq 20$

$0 \leq n_1, n_2, \dots \leq 10$

$0 \leq k \leq n / 2$

## Format

### Input

A number n

.. n more elements

A number k

### Output

A number representing the maximum profit you can make if you are allowed a single transaction.

## Example

### Sample Input

```
6
9
6
7
6
3
8
1
```

### Sample Output

```
5
```

```
#include <iostream>
#include <vector>
#include <limits>
```

```
using namespace std;
```

```
void transactions(vector<int> arr,int k){
    //write your code here
    vector<vector<int>> dp (k+1,vector<int>(arr.size()));
    //    cout<<k<<"    size1"<<dp.size()<<" "<<dp[0].size()<<endl;
```

//not good more time complexity but can see to understand  
below code

```
// for(int t{1};t<=k;t++ ) {
//     for(int d{1}; d < arr.size();d++) {
//         //    cout<<t<<d<<endl;
```



```

        //          dp[t][d] = dp[t][d-1]; //putting last day
profit with same transaction(will be compared in loop)
        //          for(int pd {};; pd < d;pd++ ){
        //              int pdp = dp[t-1][pd]; //profit of past day
with one less transaction
        //              int lt = arr[d] - arr[pd]; //that last
transaction
        //              if(pdp + lt > dp[t][d]){
        //                  dp[t][d] = pdp + lt;
        //              }
        //          }
        //      }
        //  }

```

```

for(int t{1};t<=k;t++ ) {
    int mx = INT_MIN; //mx = max;
    for(int d{1}; d < arr.size();d++) {
        mx = max(mx,dp[t-1][d-1] - arr[d-1]);

        dp[t][d] = max(mx+arr[d] ,dp[t][d-1]);
    }
}

```

```

cout<<dp[k][arr.size()-1]<<endl;
return ;

```

```

}

int main(){

    int n;
    cin>>n;
    vector<int> arr(n,0);

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

    transactions(arr,k);

    return 0;
}

```

# Buy And Sell Stocks - Two Transactions Allowed

Easy

1. You are given a number n, representing the number of days.
  2. You are given n numbers, where ith number represents price of stock on ith day.
  3. You are required to print the maximum profit you can make if you are allowed two transactions at-most.
- Note - There can be no overlapping transaction. One transaction needs to be closed (a buy followed by a sell) before opening another transaction (another buy).

## Constraints

$0 \leq n \leq 20$

$0 \leq n_1, n_2, \dots \leq 10$

## Format

### Input

A number n

.. n more elements

### Output

A number representing the maximum profit you can make if you are allowed a single transaction.

## Example

### Sample Input

```
9
11
6
7
19
4
1
6
18
4
```

### Sample Output

```
30
```

```
#include <iostream>
#include <vector>
#include <climits>
```

```
using namespace std;
```

```
void transactions(vector<int> arr){
    // write your code here
    int n = arr.size();

    //first iteration going left to right
    int min{arr[0]};
    int maxProfit{};
    vector<int> leftToRight(n); //store the max profit going left
    to right if sold today or before
    leftToRight[0] = 0;
```

```

for(int i{1}; i< n;i++) {
    if(arr[i] < min) {
        min = arr[i];
    }
    int diff = arr[i]- min;
    if(diff > maxProfit){
        maxProfit = diff;
        leftToRight[i] = diff;
    }else {
        //for if [[min is changed in this iteration ]] at that
position is will store 0 cause arr[i] will be eqaul to min after
it got changed
        leftToRight[i] = maxProfit;
    }
}

// for(int a : leftToRight){
//     cout<<a<<' ';
// }
// cout<<endl;

//second iteration right to left
int max{arr[n-1]};
maxProfit = 0;
vector<int> rightToLeft(arr.size()); //store the max profit
going left to right if sold today or before
rightToLeft[arr.size() - 1] = 0;
for(int i{n - 2}; i >= 0;i--) {
    if(arr[i] > max) {
        max = arr[i];
    }
    int diff = max - arr[i];
    if(diff > maxProfit){
        maxProfit = diff;
        rightToLeft[i] = diff;
    }else {
        //for if [[min is changed in this iteration ]] at that
position is will store 0 cause arr[i] will be eqaul to min after
it got changed
        rightToLeft[i] = maxProfit;
    }
}

// for(int a : rightToLeft){
//     cout<<a<<' ';
// }
// cout<<endl;

int ans = INT_MIN;
for(int i{}; i< n;i++) {

```

```

        if(leftToRight[i] + rightToLeft[i] > ans){
            ans = leftToRight[i] + rightToLeft[i];
        }
    }
    cout<<ans<<endl;
    return ;
    /*
    19
    30 40 43 50 45 20 26 40 80 50 30 15 10 20 40 45 71 50 55
    */
}

int main() {
    int n ;
    cin>> n;
    vector<int> arr(n,0);
    for (int i = 0; i < arr.size(); i++) {
        cin>>arr[i] ;
    }

    transactions(arr);

    return 0;
}

```