Dynamic Programming

→ A bigger problem can be solved using the optimal soln of small subproblem.

→ Overlapping subproblems

⊛ Those who forget the past, are condemned to repeat it.

→ Top down (Recursion + Memoization)
→ Bottom up (Tabulation)

↓
Store the value of subproblem.

} Space Optimization

Fibonacci Sequence :-

```
// User function Template for C++
class Solution {
  public:
    int getFib(int n, vector<int> &dp) {
        if(n ≤ 1) return n;
        if(dp[n] ≠ -1) return dp[n];
        return dp[n] = getFib(n - 1, dp) + getFib(n - 2, dp);
    }
    int nthFibonacci(int n){
        vector<int> dp(n + 1, -1);
        return getFib(n, dp);
    }
};
```

→ Top down approach

Time Complexity : O(N)

Space Complexity : O(2*N)

```
class Solution {
  public:
    int nthFibonacci(int n){
        vector<int> dp(n + 1);
        dp[0] = 0;
        dp[1] = 1;

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

→ Tabulation

Time Complexity : O(N)

Space Complexity : O(N)

→ Space Optimization

```
class Solution {
  public:
    int mod = 1e9 + 7;
    int nthFibonacci(int n){
        long long prev2 = 0;
        long long prev1 = 1;

        for(int i = 2; i ≤ n; i++) {
            int curr = (prev1 + prev2) % mod;
            prev2 = prev1;
            prev1 = curr;
        }
        return prev1;
    }
};
```

Time Complexity : O(N)

Space Complexity : O(1)

Count ways to reach the nth stairs

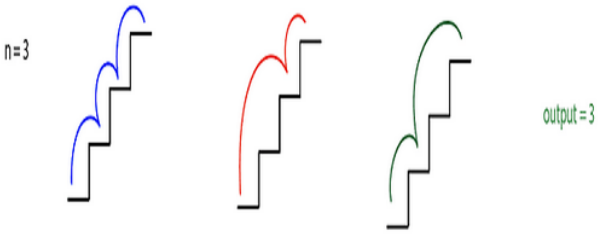
You have been given a number of stairs. Initially, you are at the 0th stair, and you need to reach the Nth stair.

Each time, you can climb either one step or two steps.

You are supposed to return the number of distinct ways you can climb from the 0th step to the Nth step.

Example :

N=3



We can climb one step at a time i.e. {(0, 1) ,(1, 2),(2,3)} or we can climb the first two-step and then one step i.e. {(0,2),(1, 3)} or we can climb first one step and then two step i.e. {(0,1), (1,3)}.

→ Recursion :-

```
long long countDistinctWays(long long nStairs) {
    if(nStairs < 0)
        return 0;
    else if(nStairs == 0)
        return 1;
    return countDistinctWays(nStairs - 2) + countDistinctWays(nStairs - 1);
}
```

→ Memoization :-

```
#define mod 1000000007;

int getWays(int nStairs, vector<long long> &dp) {
    if(nStairs <= 1) return 1;
    if(nStairs == 2) return 2;

    if(dp[nStairs] != -1) return dp[nStairs];

    return dp[nStairs] = (getWays(nStairs - 1, dp) + getWays(nStairs - 2, dp)) % mod;
}

int countDistinctWays(int nStairs) {
    vector<long long> dp(nStairs + 1, -1);
    return getWays(nStairs, dp);
}
```

→ Tabulation :-

```
#include <bits/stdc++.h>
#define mod 1000000007;

int getWays(int n) {
    vector<long long> dp(n + 1, -1);
    dp[0] = 1;
    dp[1] = 1;
    dp[2] = 2;

    for(int i = 3; i <= n; i++) {
        dp[i] = (dp[i - 1] + dp[i - 2]) % mod;
    }

    return dp[n];
}

int countDistinctWays(int nStairs) {
    return getWays(nStairs);
}
```

→ Space Optimization :-

```
#include <bits/stdc++.h>
#define mod 1000000007;

int getWays(int n) {
    long long prev0 = 1, prev1 = 1, prev2 = 2;

    for(int i = 3; i <= n; i++) {
        int curr = (prev2 + prev1) % mod;
        prev1 = prev2;
        prev2 = curr;
    }
    return prev2;
}

int countDistinctWays(int nStairs) {
    if(nStairs == 0 || nStairs == 1) return 1;
    return getWays(nStairs);
}
```

Minimum cost climbing stairs

You are given an integer array `cost` where `cost[i]` is the cost of `i^th` step on a staircase. Once you pay the cost, you can either climb one or two steps.

You can either start from the step with index `0`, or the step with index `1`.

Return *the minimum cost to reach the top of the floor.*

Example 1:

```
Input: cost = [10,15,20]
Output: 15
Explanation: You will start at index 1.
- Pay 15 and climb two steps to reach the top.
The total cost is 15.
```

Example 2:

```
Input: cost = [1,100,1,1,1,100,1,1,100,1]
Output: 6
Explanation: You will start at index 0.
- Pay 1 and climb two steps to reach index 2.
- Pay 1 and climb two steps to reach index 4.
- Pay 1 and climb two steps to reach index 6.
- Pay 1 and climb one step to reach index 7.
- Pay 1 and climb two steps to reach index 9.
- Pay 1 and climb one step to reach the top.
The total cost is 6.
```

Constraints:

- `2 <= cost.length <= 1000`
- `0 <= cost[i] <= 999`

→ Recursion :-

```
class Solution {
public:
    int solve(vector<int> cost, int n) {
        if(n == 0) return cost[0];
        if(n == 1) return cost[1];
        int ans = cost[n] + min(solve(cost, n - 1), solve(cost, n - 2));

        return ans;
    }


    int minCostClimbingStairs(vector<int>& cost) {
        int n = cost.size();
        int ans = min(solve(cost, n - 1), solve(cost, n - 2));
        return ans;
    }
};
```

→ Memoization :-

```
class Solution {
public:
    int solve(vector<int> cost, int n, vector<int> &dp) {
        if(n == 0) return cost[0];
        if(n == 1) return cost[1];

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

        dp[n] = cost[n] + min(solve(cost, n - 1, dp), solve(cost, n - 2, dp));

        return dp[n];
    }
    int minCostClimbingStairs(vector<int>& cost) {
        int n = cost.size();
        vector<int> dp(n + 1, -1);
        int ans = min(solve(cost, n - 1, dp), solve(cost, n - 2, dp));
        return ans;
    }
};
```

→ Tabulation :-

```
class Solution {
public:
    int solve(vector<int> &cost, int n) {
        vector<int> dp(n + 1);
        dp[0] = cost[0];
        dp[1] = cost[1];
        for(int i = 2; i < n; i++) {
            dp[i] = cost[i] + min(dp[i - 1], dp[i - 2]);
        }
        return min(dp[n - 1], dp[n - 2]);
    }
    int minCostClimbingStairs(vector<int>& cost) {
        int n = cost.size();
        return solve(cost, n);
    }
};
```

→ Space Optimization :-

```
class Solution {
public:
    int solve2(vector<int> &cost, int n) {
        int prev1 = cost[1];
        int prev2 = cost[0];
        for(int i = 2; i < n; i++) {
            int curr = cost[i] + min(prev1, prev2);
            prev2 = prev1;
            prev1 = curr;
        }
        return min(prev1, prev2);
    }
    int minCostClimbingStairs(vector<int>& cost) {
        int n = cost.size();
        return solve2(cost, n);
    }
};
```

Minimum Coins

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).

Detailed explanation (Input/output format, Notes, Images)

Constraints:

1 <= T <= 10
1 <= P <= 10^9

Time Limit: 1 sec

Sample Input 1:

3
60
10
24

Sample Output 1:

2
1
3

Explanation of Sample Input 1:

In the 1st test case, we need one coin of 50 cents and one coin of 10 cents.
In the 2nd test case, we need a coin of 10 cents.
In the 3rd test case, we need one coin of 20 cents and two coins of 2 cents.

→ Recursion :-

```
int solveRec(int denominations[], int x, int n) {
  if (x == 0)
    return 0;
  if (x < 0)
    return INT_MAX;

  int mini = INT_MAX;

  for (int i = 0; i < n; i++) {
    int ans = solveRec(denominations, x - denominations[i], n);
    if (ans != INT_MAX)
      mini = min(mini, 1 + ans);
  }
  return mini;
}

int minimumCoins(int p) {
  int denominations[9] = {1, 2, 5, 10, 20, 50, 100, 500, 1000};
  int ans = solveRec(denominations, p, 9);
  if (ans == INT_MAX)
    return -1;

  return ans;
}
```

→ Memoization :-

```
int solveRec(int denominations[], int x, int n, vector<int> &dp) {
  if (x == 0)
    return 0;
  if (x < 0)
    return INT_MAX;

  if (dp[x] != -1)
    return dp[x];

  int mini = INT_MAX;

  for (int i = 0; i < n; i++) {
    int ans = solveRec(denominations, x - denominations[i], n, dp);
    if (ans != INT_MAX)
      mini = min(mini, 1 + ans);
  }
  dp[x] = mini;
  return dp[x];
}

int minimumCoins(int p) {
  int denominations[9] = {1, 2, 5, 10, 20, 50, 100, 500, 1000};
  vector<int> dp(p + 1, -1);
  int ans = solveRec(denominations, p, 9, dp);
  if (ans == INT_MAX)
    return -1;

  return ans;
}
```

→ Tabulation :-

```
int solveRec(int denominations[], int x, int n) {
  vector<int> dp(x + 1, INT_MAX);

  dp[0] = 0;

  for (int i = 1; i <= x; i++) {
    for (int j = 0; j < 9; j++) {
        if( (i - denominations[j]) >= 0 && (dp[i - denominations[j]]) != INT_MAX )
        dp[i] = min(dp[i], 1 + dp[i - denominations[j]]);

    }
  }
  if(dp[x] == INT_MAX) return -1;
  return dp[x];
}

int minimumCoins(int p) {
  int denominations[9] = {1, 2, 5, 10, 20, 50, 100, 500, 1000};
  int ans = solveRec(denominations, p, 9);
  if (ans == INT_MAX)
    return -1;

  return ans;
}
```

Minimum Elements

You are given an array of 'N' distinct integers and an integer 'X' representing the target sum. You have to tell the minimum number of elements you have to take to reach the target sum 'X'.

Note:

You have an infinite number of elements of each type.

For example

If N=3 and X=7 and array elements are [1,2,3].
Way 1 - You can take 4 elements [2, 2, 2, 1] as 2 + 2 + 2 + 1 = 7.
Way 2 - You can take 3 elements [3, 3, 1] as 3 + 3 + 1 = 7.
Here, you can see in Way 2 we have used 3 coins to reach the target sum of 7.
Hence the output is 3.

Detailed explanation (Input/output format, Notes, Images)

Constraints:

1 <= T <= 10
1 <= N <= 15
1 <= nums[i] <= (2^31) - 1
1 <= X <= 10000

All the elements of the "nums" array will be unique.
Time limit: 1 sec

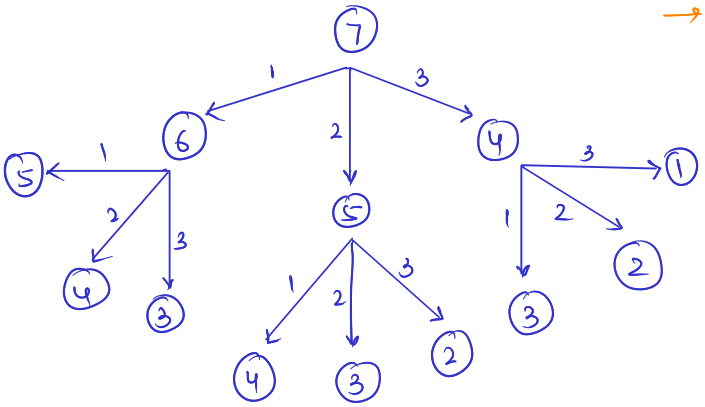
→ Recursion :-

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

int solveRec(vector<int> &num, int x) {
    if(x == 0) return 0;
    if(x < 0) return INT_MAX;

    int mini = INT_MAX;
    for(int i = 0; i < num.size(); i++) {
        int ans = solveRec(num, x - num[i]);
        if(ans != INT_MAX)
            mini = min(mini, 1 + ans);
    }
    return mini;
}

int minimumElements(vector<int> &num, int x)
{
    int ans = solveRec(num, x);
    if(ans == INT_MAX) {
        return -1;
    }
    return ans;
}
```



→ At every node, we will have N no. of choice where N is the size of array.

→ Memoization :-

```
int solveMem(vector<int> &num, int x, vector<int> &dp) {
    if(x == 0) return 0;
    if(x < 0) return INT_MAX;

    if(dp[x] != -1) return dp[x];

    int mini = INT_MAX;
    for(int i = 0; i < num.size(); i++) {
        int ans = solveMem(num, x - num[i], dp);
        if(ans != INT_MAX)
            mini = min(mini, 1 + ans);
    }
    dp[x] = mini;
    return dp[x];
}

int minimumElements(vector<int> &num, int x)
{
    vector<int> dp(x + 1, -1);
    int ans = solveMem(num, x, dp);

    if(ans == INT_MAX) return -1;
    return ans;
}
```

→ Tabulation :-

```
int solveTab(vector<int> &num, int x) {
  vector<int> dp(x + 1, INT_MAX);
  dp[0] = 0;

  // dp[i] represents minimum number of coins required to make target 'i'
  for (int i = 1; i <= x; i++) {
    // i am trying to solve for every amount figure from 1 to x
    for (int j = 0; j < num.size(); j++) {
      if ((i - num[j]) >= 0 && (dp[i - num[j]] != INT_MAX))
        dp[i] = min(dp[i], 1 + dp[i - num[j]]);
    }
  }
  if (dp[x] == INT_MAX)
    return -1;
  return dp[x];
}

int minimumElements(vector<int> &num, int x) {
  int ans = solveTab(num, x);
  return ans;
}
```

Maximum sum of non-adjacent elements

You are given an array/list of 'N' integers. You are supposed to return the maximum sum of the subsequence with the constraint that no two elements are adjacent in the given array/list.

Note:

A subsequence of an array/list is obtained by deleting some number of elements (can be zero) from the array/list, leaving the remaining elements in their original order.

Detailed explanation (Input/output format, Notes, Images)

Constraints:

1 <= T <= 500
1 <= N <= 1000
0 <= ARR[i] <= 10^5

Where 'ARR[i]' denotes the 'i-th' element in the array/list.

Time Limit: 1 sec.

Sample Input 1:

2
3
1 2 4
4
2 1 4 9

Sample Output 1:

5
11

Explanation to Sample Output 1:

In test case 1, the sum of 'ARR[0]' & 'ARR[2]' is 5 which is greater than 'ARR[1]' which is 2 so the answer is 5.

In test case 2, the sum of 'ARR[0]' and 'ARR[2]' is 6, the sum of 'ARR[1]' and 'ARR[3]' is 10, and the sum of 'ARR[0]' and 'ARR[3]' is 11. So if we take the sum of 'ARR[0]' and 'ARR[3]', it will give the maximum sum of sequence in which no elements are adjacent in the given array/list.

→ Recursion :-

```
int solve(vector<int> &nums, int n) {
    if(n < 0) {
        return 0;
    }
    if(n == 0)  {
        return nums[0];
    }

    int incl = solve(nums, n - 2) + nums[n];
    int excl = solve(nums, n - 1);
    return max(incl, excl);
}

int maximumNonAdjacentSum(vector<int> &nums){
    int n = nums.size();
    int ans = solve(nums, n - 1);
    return ans;
}
```

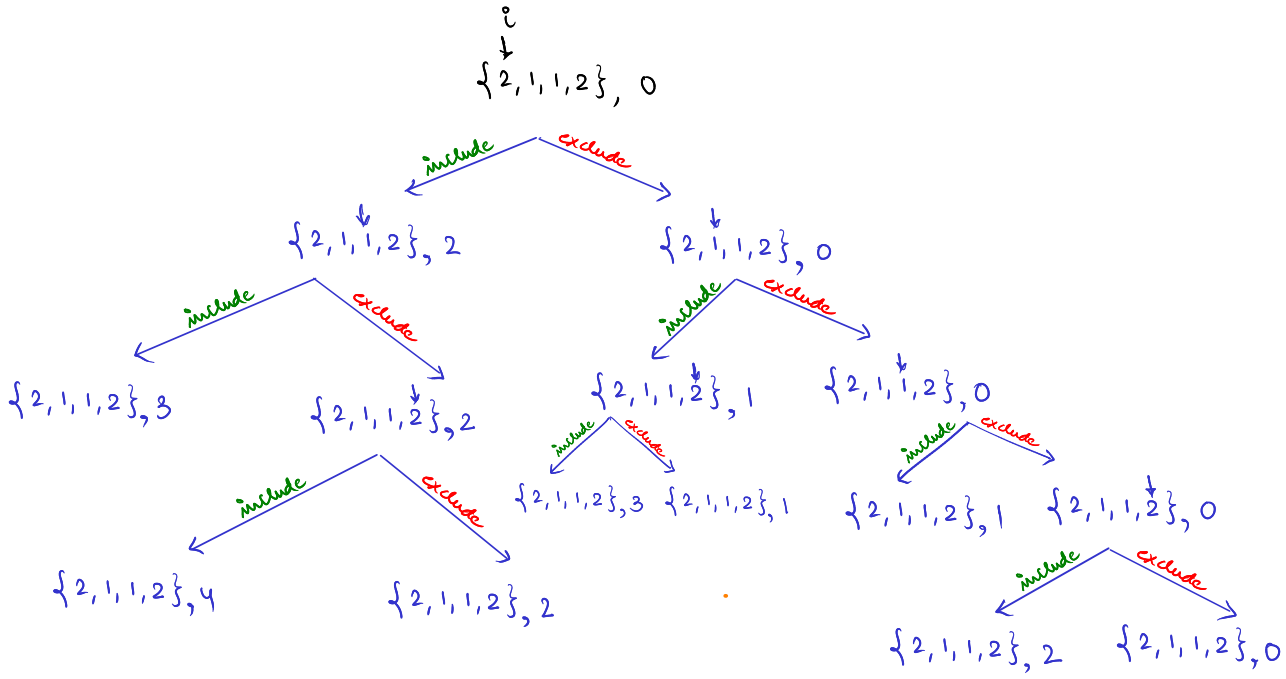
→ Memoization :-

```
int solve(vector<int> &nums, int n, vector<int> &dp) {
    if(n < 0) {
        return 0;
    }
    if(n == 0)  {
        return nums[0];
    }

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

    int incl = solve(nums, n - 2, dp) + nums[n];
    int excl = solve(nums, n - 1, dp);
    return dp[n] = max(incl, excl);
}

int maximumNonAdjacentSum(vector<int> &nums){
    int n = nums.size();
    vector<int> dp(n + 1, -1);
    int ans = solve(nums, n - 1, dp);
    return ans;
}
```



→ Tabulation :-

```
int solve(vector<int> &nums) {
    int n = nums.size();
    vector<int> dp(n, 0);

    dp[0] = nums[0];

    for(int i = 1; i < n; i++) {
        int incl = dp[i - 2] + nums[i];
        int excl = dp[i - 1];
        dp[i] = max(incl, excl);
    }
    return dp[n - 1];
}

int maximumNonAdjacentSum(vector<int> &nums){
    return solve(nums);
}
```

→ Space optimization :-

```
int solve(vector<int> &nums) {
    int n = nums.size();
    int prev2 = 0;
    int prev1 = nums[0];

    for(int i = 1; i < n; i++) {
        int incl = prev2 + nums[i];
        int excl = prev1;
        int ans = max(incl, excl);
        prev2 = prev1;
        prev1 = ans;
    }
    return prev1;
}

int maximumNonAdjacentSum(vector<int> &nums){
    return solve(nums);
}
```

House Robbery

Cut into segments

You are given an integer 'N' denoting the length of the rod. You need to determine the maximum number of segments you can make of this rod provided that each segment should be of the length 'X', 'Y', or 'Z'.

Detailed explanation (Input/output format, Notes, Images)

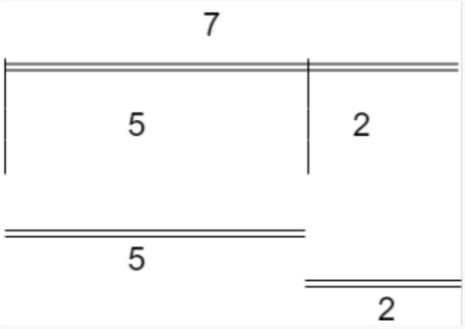
Sample Input 1:

```
2
7 5 2 2
8 3 3 3
```

Sample Output 1:

```
2
0
```

Explanation For Sample Input 1:



In the first test case, cut it into 2 parts of 5 and 2.

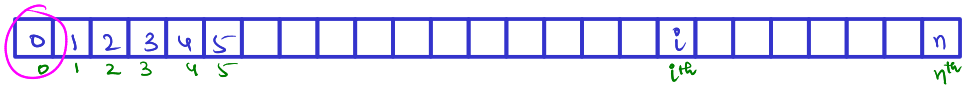
In the second case, there is no way to cut into segments of 3 length only as the length of the rod is less than the given length.

Count Dearrangements

A Derangement is a permutation of 'N' elements, such that no element appears in its original position. For example, an instance of derangement of {0, 1, 2, 3} is {2, 3, 1, 0}, because 2 present at index 0 is not at its initial position which is 2 and similarly for other elements of the sequence.
Given a number 'N', find the total number of derangements possible of a set of 'N' elements.
Note:

The answer could be very large, output answer %(10 ^ 9 + 7).

→ Approach:-

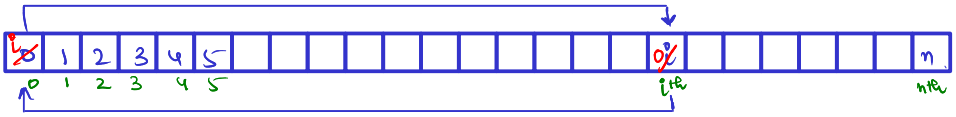


No. of ways to keep 0 is : (n-1)

Total dearrangements possible = (n-1) * (soln of subproblems)

→ Two cases of subproblems:-

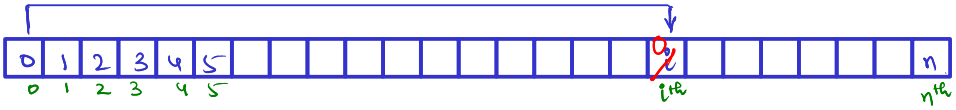
(I) i & 0 gets swapped:-



⇒ We have (n-2) indexes left and (n-2) numbers to be filled.

∴ The subproblem becomes f(n-2).

(II)



0 toh ith index me gaya, but main i ko 0th index pe nhi dalna chahta hu. Na hi 1 ko 1st index pe, 2 ko 2nd index pe and so on.

⇒ Indexes = n-1
& Numbers = n-1
} Because 0 is fixed at ith index.

∴ Subproblem is f(n-1)

$$f(n) = (n-1) * \{ f(n-1) + f(n-2) \}$$

→ Recursion :-

```
long long int countDerangements(int n) {
  if (n == 1)
    return 0;
  if (n == 2)
    return 1;

  return ((n - 1) * (((countDerangements(n - 1)) % MOD + (countDerangements(n - 2)) % MOD)) % MOD) % MOD;
}
```

→ Memoization :-

```
#define MOD 1000000007

long long int solveMem(int n, vector<long long int> &dp) {
  if (n == 1)
    return 0;
  if (n == 2)
    return 1;

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

    int ans = ((n - 1) * (( (solveMem(n - 1, dp))% MOD + (solveMem(n - 2, dp)) % MOD )) % MOD) % MOD;
    dp[n] = ans;
    return ans;
}

long long int countDerangements(int n) {
    vector<long long int> dp(n + 1, -1);
    return solveMem(n, dp);
}
```

→ Tabulation :-

```
long long int solveTab(int n) {
  vector<long long int> dp(n + 1, 0);

  dp[1] = 0;
  dp[2] = 1;

  for(int i = 3; i <= n; i++) {
    long long int first = dp[i - 1] % MOD;
    long long int second = dp[i - 2] % MOD;
    long long int sum = (first + second) % MOD;
    long long int ans = ( ( i - 1) * sum) % MOD;
    dp[i] = ans;
  }
  return dp[n];
}

long long int countDerangements(int n) {
    return solveTab(n);
}
```

→ Space Optimization :-

```
long long int solveTab(int n) {
  long long int prev2 = 0;
  long long int prev1 = 1;

  for(int i = 3; i <= n; i++) {
    long long int first = prev1 % MOD;
    long long int second = prev2 % MOD;
    long long int sum = (first + second) % MOD;
    long long int ans = ( ( i - 1) * sum) % MOD;
    prev2 = prev1;
    prev1 = ans;
  }
  return prev1;
}

long long int countDerangements(int n) {
    return solveTab(n);
}
```

Painting Fence Algorithm

Ninja has given a fence, and he gave a task to paint this fence. The fence has 'N' posts, and Ninja has 'K' colors. Ninja wants to paint the fence so that not more than two adjacent posts have the same color.
Ninja wonders how many ways are there to do the above task, so he asked for your help.
Your task is to find the number of ways Ninja can paint the fence. Print the answer modulo 10^9 + 7.

Example:

Input: 'N' = 3, 'K' = 2
Output: 6

Say we have the colors with the numbers 1 and 0. We can paint the fence with 3 posts with the following different combinations.

110
001
101
100
010
011

Detailed explanation (Input/output format, Notes, Images)

Constraints :

1 <= 'T' <= 10
1 <= 'N' <= 10^5
1 <= 'K' <= 10^5
Time Limit: 1 sec

→ Approach :-

Solve(n) → in how many different ways we can paint the post such that not more than 2 consecutive post have the same colour.

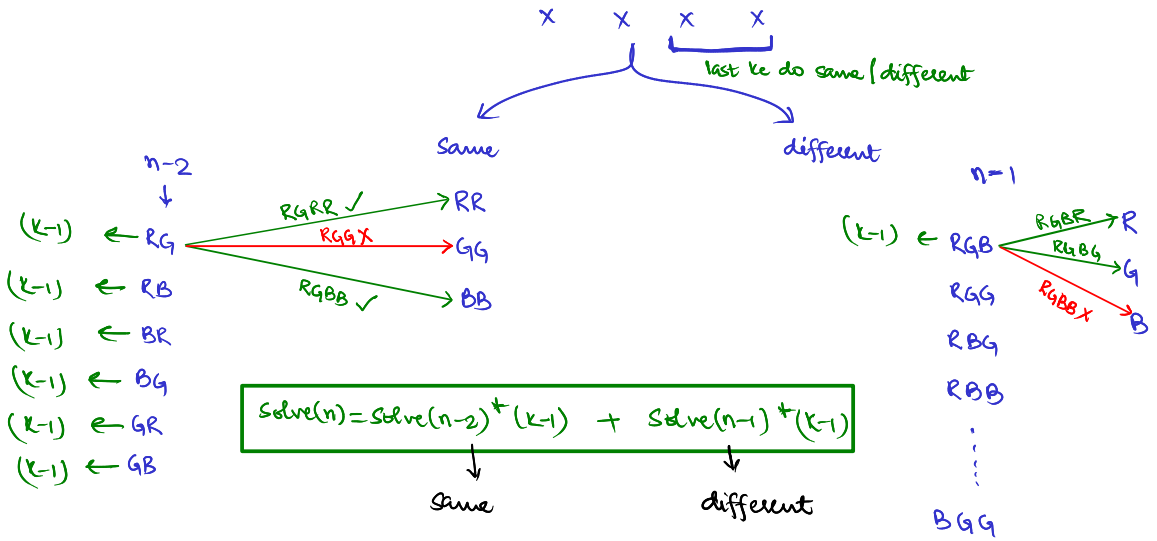
K = 3 (RGB)

	n=2 x x	n=3 xxx	n=4
Same colour on last two post	RR, GG, BB ③ ↓ K	RGG BRR, RBB BGG, GRR GBB ⑥	18
different colour on last two posts	RG BR, RB BG, GR GB ⑥ ↓ K*(K-1)	RR GB → K-1, GG BG → K-1, BB, RG, BR, RB, GR 9*(K-1) = 18	24 x K-1
∑	9	24	66

Solve(2) = K + K*(K-1) = K²

→ If n = 4

1st way → last 2 post colour will be different
2nd way → last 2 post colour will be same.



→ Recursion :-

```
#include <bits/stdc++.h>
#define MOD 1000000007

int add(int a, int b) {
    return (a % MOD + b % MOD) % MOD;
}

int mul(int a, int b) {
    return ((a % MOD) * (b % MOD)) % MOD;
}

int solve(int n, int k) {
    if(n == 1) return k;
    if(n == 2) return add(k, mul(k, k - 1));

    int ans = add(mul(solve(n - 2, k), k - 1), mul(solve(n - 1, k), k - 1));
    return ans;
}

int numberOfWays(int n, int k) {
    return solve(n, k);
}
```

→ Memoization :-

```
#include <bits/stdc++.h>
#define MOD 1000000007

int add(int a, int b) { return (a % MOD + b % MOD) % MOD; }

int mul(int a, int b) { return ((a % MOD) * 1LL * (b % MOD)) % MOD; }

int solve(int n, int k, vector<int> &dp) {
  if (n == 1)
    return k;
  if (n == 2)
    return add(k, mul(k, k - 1));

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

  dp[n] = add(mul(solve(n - 2, k, dp), k - 1), mul(solve(n - 1, k, dp), k - 1));
  return dp[n];
}

int numberOfWays(int n, int k) {
  vector<int> dp(n + 1, -1);
  return solve(n, k, dp);
}
```

→ Tabulation :-

```
#include <bits/stdc++.h>
#define MOD 1000000007

int add(int a, int b) { return (a % MOD + b % MOD) % MOD; }

int mul(int a, int b) { return ((a % MOD) * 1LL * (b % MOD)) % MOD; }

int solve(int n, int k) {

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

  dp[1] = k;
  dp[2] = add(k, mul(k, k - 1));

  for (int i = 3; i <= n; i++) {
    dp[i] = add(mul(dp[i - 2], k - 1), mul(dp[i - 1], k - 1));
  }
  return dp[n];
}

int numberOfWays(int n, int k) {
    return solve(n, k);
}
```

→ Space optimization :-

```
#include <bits/stdc++.h>
#define MOD 1000000007

int add(int a, int b) { return (a % MOD + b % MOD) % MOD; }

int mul(int a, int b) { return ((a % MOD) * 1LL * (b % MOD)) % MOD; }

int solve(int n, int k) {
    int prev2 = k;
    int prev1 = add(k, mul(k, k - 1));

  for (int i = 3; i <= n; i++) {
    int ans = add(mul(prev2, k - 1), mul(prev1, k - 1));
    prev2 = prev1;
    prev1 = ans;
  }
  return prev1;
}

int numberOfWays(int n, int k) {
    return solve(n, k);
}
```

0/1 Knapsack

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.

Detailed explanation (Input/output format, Notes, Images)

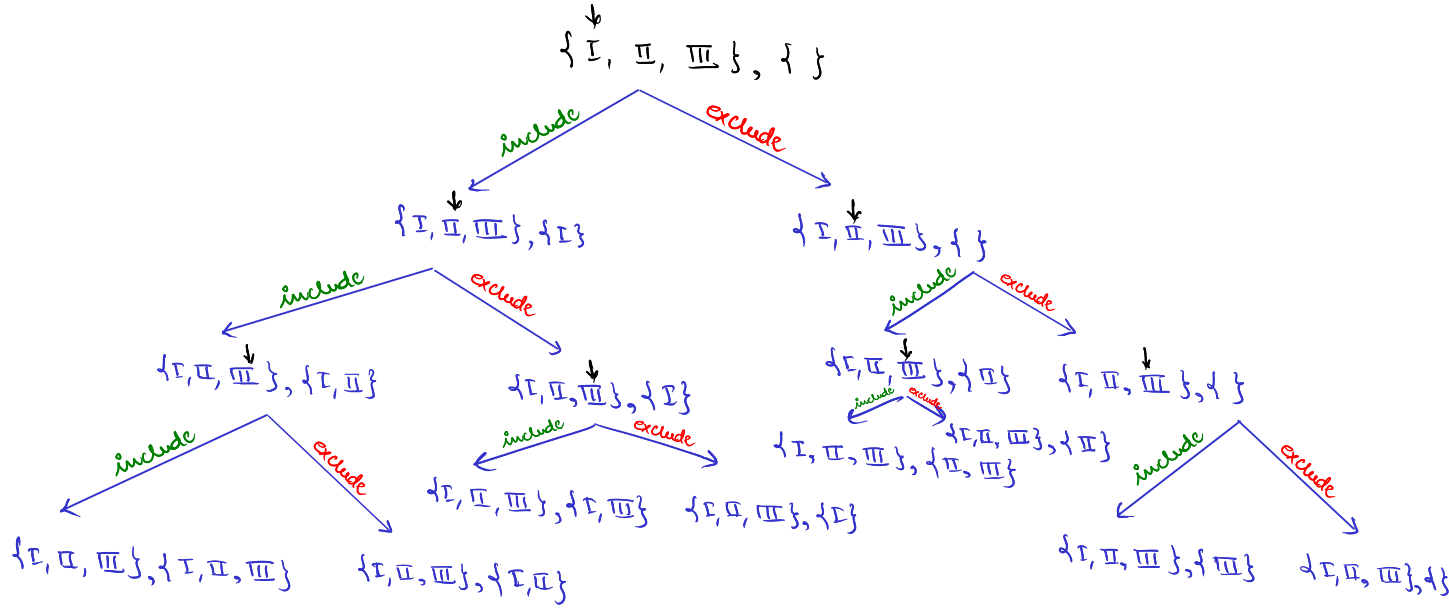
Constraints:

1 <= T <= 10
1 <= N <= 10^2
1<= wi <= 50
1 <= vi <= 10^2
1 <= W <= 10^3

Time Limit: 1 second

→ Brute force:-

- Combination of n items.
- Return the combination with max value.



→ Recursion:-

```
int solve(vector<int> &weight, vector<int> &value, int index, int capacity) {
    if(index == 0) {
        if(weight[0] <= capacity) return value[0];
        else return 0;
    }
    int include = 0;
    if(weight[index] <= capacity) {
        include = value[index] + solve(weight, value, index - 1, capacity - weight[index]);
    }
    int exclude = solve(weight, value, index - 1, capacity);

    return max(include, exclude);
}

int knapsack(vector<int> weight, vector<int> value, int n, int maxWeight)
{
        return solve(weight, value, n - 1, maxWeight);
}
```

→ Memoization:-

- DP array (-1)
- Recursive call: dp ans store & return
- After base case: if dp array contains ans then return

```cpp
int solve(vector<int> &weight, vector<int> &value, int index, int capacity, vector<vector<int>> &dp) {
    if(index == 0) {
        if(weight[0] <= capacity) return value[0];
        else return 0;
    }

    if(dp[index][capacity] != -1) return dp[index][capacity];

    int include = 0;
    if(weight[index] <= capacity) {
        include = value[index] + solve(weight, value, index - 1, capacity - weight[index], dp);
    }
    int exclude = solve(weight, value, index - 1, capacity, dp);

    return dp[index][capacity] = max(include, exclude);
}

int knapsack(vector<int> weight, vector<int> value, int n, int maxWeight)
{
    vector<vector<int>> dp(n, vector<int>(maxWeight + 1, -1));
        return solve(weight, value, n - 1, maxWeight, dp);
}
```

two variable are changing ⇒ 2D - dp

→ Tabulation :-

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

int solve(vector<int> &weight, vector<int> &value, int n, int capacity) {
  vector<vector<int>> dp(n, vector<int>(capacity + 1, 0));

  for (int w = weight[0]; w <= capacity; w++) {
    if (weight[0] <= capacity) {
      dp[0][w] = value[0];
    } else
      dp[0][w] = 0;
  }

  for (int index = 1; index < n; index++) {
    for (int w = 0; w <= capacity; w++) {
      int include = 0;
      if (weight[index] <= w) {
        include = value[index] + dp[index - 1][w - weight[index]];
      }
      int exclude = dp[index - 1][w];
      dp[index][w] = max(exclude, include);
    }
  }
  return dp[n - 1][capacity];
}

int knapsack(vector<int> weight, vector<int> value, int n, int maxWeight) {
  return solve(weight, value, n, maxWeight);
}
```

→ Space Optimization :-

```cpp
int solve(vector<int> &weight, vector<int> &value, int n, int capacity) {
  vector<int> prev(capacity + 1, 0);
  vector<int> curr(capacity + 1, 0);

  for (int w = weight[0]; w <= capacity; w++) {
    if (weight[0] <= capacity) {
      prev[w] = value[0];
    } else
      prev[w] = 0;
  }

  for (int index = 1; index < n; index++) {
    for (int w = 0; w <= capacity; w++) {
      int include = 0;
      if (weight[index] <= w) {
        include = value[index] + prev[w - weight[index]];
      }
      int exclude = prev[w];
      curr[w] = max(exclude, include);
    }
    prev = curr;
  }
  return prev[capacity];
}

int knapsack(vector<int> weight, vector<int> value, int n, int maxWeight) {
  return solve(weight, value, n, maxWeight);
}
```

→ More optimization :-

```
int solve(vector<int> &weight, vector<int> &value, int n, int capacity) {
  vector<int> curr(capacity + 1, 0);

  for (int w = weight[0]; w ⩽ capacity; w++) {
    if (weight[0] ⩽ capacity) {
      curr[w] = value[0];
    } else
      curr[w] = 0;
  }

  for (int index = 1; index < n; index++) {
    for (int w = capacity; w ⩾ 0; w--) {
      int include = 0;
      if (weight[index] ⩽ w) {
        include = value[index] + curr[w - weight[index]];
      }
      int exclude =.curr[w];
      curr[w] = max(exclude, include);
    }
  }
  return curr[capacity];
}

int knapsack(vector<int> weight, vector<int> value, int n, int maxWeight) {
  return solve(weight, value, n, maxWeight);
}
```

Note: Same pattern is utilized by many questions like:
- Equal subset sum partition
- Subset sum
- Minimum subset sum difference
- Count of subset sum
- Target Sum

Minimum Cost

Ninja is willing to take some time off from his training and planning a year-long tour. You are given a DAYS array consisting of 'N' days when Ninjas will be traveling during the year. Each Day is an integer between 1 to 365 (both inclusive). Train tickets are sold in three different ways:

A 1-day pass is sold for 'COST'[0] coins,
A 7-day pass is sold for 'COST'[1] coins, and
A 30-day pass is sold for 'COST'[2] coins.
The passes allow for many days of consecutive travel.

Your task is to help the Ninja to find the minimum number of coins required to complete his tour. For example,

If Ninja gets a 7-day pass on day 2, then he can travel for 7 days: 2, 3, 4, 5, 6, 7, and 8.

Detailed explanation (Input/output format, Notes, Images)

Constraints:

1 <= T <= 10
1 <= N <= 365
1 <= DAYS[i] <= 365

Time Limit: 1 sec

Sample Input 1:

```
2
2
2 5
1 4 25
7
1 3 4 5 7 8 10
2 7 20
```

Sample Output 1:

```
2
11
```

→ Recursion :-

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

int solve(int n, vector<int> &days, vector<int>& cost, int index) {
    if(index >= n) {
        return 0;
    }
    int option1 = cost[0] + solve(n, days, cost, index + 1);

    int i;

    for(i = index; i < n && days[i] < days[index] + 7; i++);

    int option2 = cost[1] + solve(n, days, cost, i);

    for(i = index; i < n && days[i] < days[index] + 30; i++);

    int option3 = cost[2] + solve(n, days, cost, i);

    return min(option1, min(option2, option3));
}

int minimumCoins(int n, vector<int> days, vector<int> cost)
{
    return solve(n, days, cost, 0);
}
```

→ Memoization :-

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

int solve(int n, vector<int> &days, vector<int>& cost, int index, vector<int> &dp) {
    if(index >= n) {
        return 0;
    }
    if(dp[index] != -1) return dp[index];
    int option1 = cost[0] + solve(n, days, cost, index + 1, dp);

    int i;

    for(i = index; i < n && days[i] < days[index] + 7; i++);
    int option2 = cost[1] + solve(n, days, cost, i, dp);

    for(i = index; i < n && days[i] < days[index] + 30; i++);
    int option3 = cost[2] + solve(n, days, cost, i, dp);

    return dp[index] = min(option1, min(option2, option3));
}

int minimumCoins(int n, vector<int> days, vector<int> cost)
{
    vector<int> dp(n + 1, -1);
    return solve(n, days, cost, 0, dp);
}
```

→ Tabulation :-

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

int solve(int n, vector<int> &days, vector<int> &cost) {
  vector<int> dp(n + 1, INT_MAX);
  dp[n] = 0;

  for (int k = n - 1; k >= 0; k--) {

    int option1 = cost[0] + dp[k + 1];

    int i;

    for (i = k; i < n && days[i] < days[k] + 7; i++);
    int option2 = cost[1] + dp[i];

    for (i = k; i < n && days[i] < days[k] + 30; i++);
    int option3 = cost[2] + dp[i];

    dp[k] = min(option1, min(option2, option3));
  }
  return dp[0];
}

int minimumCoins(int n, vector<int> days, vector<int> cost) {
  return solve(n, days, cost);
}
```

→ Space Optimization :-

{1, 3, 4, 5, 7, 8, 10} {2, 7, 20}

int ans = INT_MAX;

day cost till that day

queue <pair < int, int >> monthly; → at max 30 days

— — — — — — — weekly; → at max 7 days } O(1)

for(day : days){

// remove expired days from queue.

monthly ke andar 30 se zyada hai to remove, similar for weekly.

while(size >0 && monthly.front.first +30 ≤ days) monthly.pop()

//same for weekly.

weekly.push({ day, ans + cost[1]});

//same for monthly.

ans = min(ans + cost[0], min(monthly.front.second ___));

Time Complexity :- O(n)

Space Complexity : O(1)

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

int minimumCoins(int n, vector<int> days, vector<int> cost) {
  int ans = 0;
  queue<pair<int, int>> month;
  queue<pair<int, int>> week;
  for (int day : days) {
    while (!month.empty() && month.front().first + 30 ≤ day)
      month.pop();
    while (!week.empty() && week.front().first + 7 ≤ day)
      week.pop();

    week.push({day, ans + cost[1]});
    month.push({day, ans + cost[2]});

    ans = min(ans + cost[0], min(week.front().second, month.front().second));
  }
  return ans;
}
```

Largest Square Area in Matrix

You have been given a non-empty grid 'MAT' consisting of only 0s and 1s. Your task is to find the area of maximum size square sub-matrix with all 1s.
If there is no such sub-matrix, print 0.
For example, for the following grid:

```
[[1, 0, 1, 0, 0],
 [1, 0, 1, 1, 1],
 [1, 1, 1, 1, 1],
 [1, 0, 0, 1, 0]]
```

The area of the largest square submatrix with all 1s is 4.

Detailed explanation (Input/output format, Notes, Images)

Constraints:

1 <= T <= 100
1 <= N <= 50
1 <= M <= 50
0 <= MAT[i][j] <= 1

Time limit: 1 sec

Sample Input 1:

1
2 2
1 0
0 0

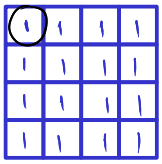
Sample Output 1:

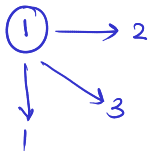
1

Explanation of the Sample Input 1:

For the given grid, the largest square with all 1s has a side of length 1. So, the area will be 1 * 1 = 1.

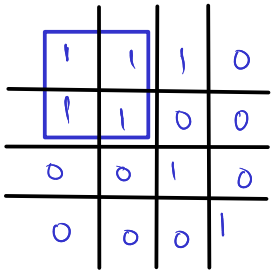
→ Approach :-





ie

Only 2 size can be formed.



→ Memoization :-

```
class Solution{
public:
    int solve(vector<vector<int>> &mat, int i, int j, int &maxi, vector<vector<int>> &dp) {
        if(i >= mat.size() || j >= mat[0].size()) {
            return 0;
        }

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

        int right = solve(mat, i, j + 1, maxi, dp);
        int diagonal = solve(mat, i + 1, j + 1, maxi, dp);
        int bottom = solve(mat, i + 1, j, maxi, dp);

        if(mat[i][j] == 1) {
            dp[i][j] = 1 + min(right, min(diagonal, bottom));
            maxi = max(maxi, dp[i][j]);
            return dp[i][j];
        } else {
            return 0;
        }
    }
    int maxSquare(int n, int m, vector<vector<int>> mat){
        int maxi = 0;
        vector<vector<int>> dp(n, vector<int> (m, -1));
        solve(mat, 0, 0, maxi, dp);
        return maxi;
    }
};
```

```
class Solution{
public:
    int solve(vector<vector<int>> &mat, int i, int j, int &maxi) {
        if(i >= mat.size() || j >= mat[0].size()) {
            return 0;
        }

        int right = solve(mat, i, j + 1, maxi);
        int diagonal = solve(mat, i + 1, j + 1, maxi);
        int bottom = solve(mat, i + 1, j, maxi);

        if(mat[i][j] == 1) {
            int ans = 1 + min(right, min(diagonal, bottom));
            maxi = max(maxi, ans);
            return ans;
        } else {
            return 0;
        }
    }
    int maxSquare(int n, int m, vector<vector<int>> mat){
        int maxi = 0;
        solve(mat, 0, 0, maxi);
        return maxi;
    }
};
```

→ Tabulation :-

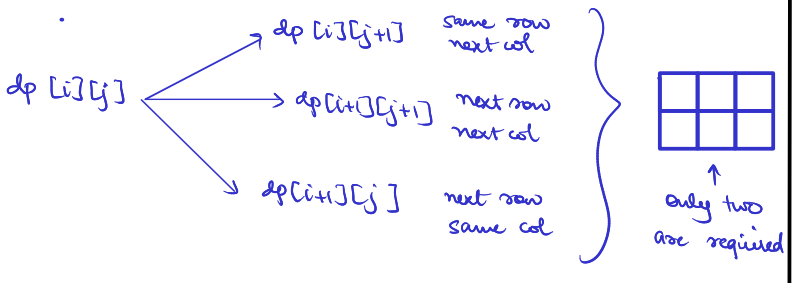
```
class Solution{
public:
    int solve(vector<vector<int>> &mat, int &maxi) {
        int row = mat.size(), col = mat[0].size();

        vector<vector<int>> dp(row + 1, vector<int> (col + 1, 0));

        for(int i = row - 1; i >= 0; i--) {
            for(int j = col - 1; j >= 0; j--) {
                int right = dp[i][j + 1];
                int diagonal = dp[i + 1][j + 1];
                int bottom = dp[i + 1][j];

                if(mat[i][j] == 1) {
                    dp[i][j] = 1 + min(right, min(diagonal, bottom));
                    maxi = max(maxi, dp[i][j]);
                } else {
                    dp[i][j] = 0;
                }
            }
        }
        return dp[0][0];
    }
    int maxSquare(int n, int m, vector<vector<int>> mat){
        int maxi = 0;
        solve(mat, maxi);
        return maxi;
    }
};
```

→ Space Optimization :-



```cpp
class Solution{
public:
    int solve(vector<vector<int>> &mat, int &maxi) {
        int row = mat.size(), col = mat[0].size();

        vector<int> curr(col + 1, 0);
        vector<int> next(col + 1, 0);

        for(int i = row - 1; i >= 0; i--) {
            for(int j = col - 1; j >= 0; j--) {

                int right = curr[j + 1];
                int diagonal = next[j + 1];
                int bottom = next[j];

                if(mat[i][j] == 1) {
                    curr[j] = 1 + min(right, min(diagonal, bottom));
                    maxi = max(maxi, curr[j]);
                } else {
                    curr[j] = 0;
                }
            }
            next = curr;
        }
        return curr[0];
    }
    int maxSquare(int n, int m, vector<vector<int>> mat){
        int maxi = 0;
        solve(mat, maxi);
        return maxi;
    }
};
```

Optimize space to O(1)

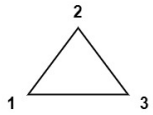
Minimum Score Triangulation

You have a convex `n`-sided polygon where each vertex has an integer value. You are given an integer array `values` where `values[i]` is the value of the `i^th` vertex (i.e., **clockwise order**).

You will **triangulate** the polygon into `n - 2` triangles. For each triangle, the value of that triangle is the product of the values of its vertices, and the total score of the triangulation is the sum of these values over all `n - 2` triangles in the triangulation.

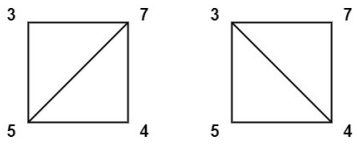
Return *the smallest possible total score that you can achieve with some triangulation of the polygon.*

Example 1:

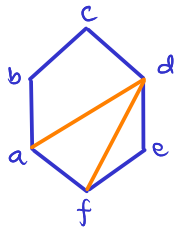


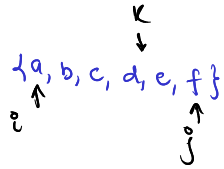

```
Input: values = [1,2,3]
Output: 6
Explanation: The polygon is already triangulated, and the score of the only triangle is 6.
```

Example 2:




```
Input: values = [3,7,4,5]
Output: 144
Explanation: There are two triangulations, with possible scores: 3*7*5 + 4*5*7 = 245, or 3*4*5 + 3*4*7 = 144.
The minimum score is 144.
```





→ Both the extremes of the array will be adjacent i & j.

→ Let's make a triangle by taking 3rd point as d.

→ Now, subproblem

fde & abcd

Soln: Solve(i,j) = a*d*f + Solve(i,K) + Solve(K,j)

if(i+1 == j)

return 0;

→ Recursion :-

```
class Solution {
public:
    int solve(vector<int> &v, int i, int j) {
        if(i + 1 == j) return 0;
        int ans = INT_MAX;
        for(int k = i + 1; k < j; k++) {
            ans = min(ans, v[i] * v[j] * v[k] + solve(v, i, k) + solve(v, k, j));
        }
        return ans;
    }
    int minScoreTriangulation(vector<int>& values) {
        int n = values.size();
        return solve(values, 0, n - 1);
    }
};
```

→ Memoization :-

```
class Solution {
public:
    int solve(vector<int> &v, int i, int j, vector<vector<int>> &dp) {
        if(i + 1 == j) return 0;
        if(dp[i][j] != -1) dp[i][j];
        int ans = INT_MAX;

        for(int k = i + 1; k < j; k++) {
            ans = min(ans, v[i] * v[j] * v[k] + solve(v, i, k, dp) + solve(v, k, j, dp));
        }
        dp[i][j] = ans;
        return dp[i][j];
    }
    int minScoreTriangulation(vector<int>& values) {
        int n = values.size();
        vector<vector<int>> dp(n, vector<int> (n, -1));
        return solve(values, 0, n - 1, dp);
    }

};
```

→ Matrix chain multiplication

→ Catlan series

→ Tabulation :-

```
class Solution {
public:
    int solve(vector<int> &v) {
        int n = v.size();
        vector<vector<int>> dp(n, vector<int> (n, 0));

        for(int i = n - 1; i >= 0; i--) {
            for(int j = i + 2; j < n; j++) {
                int ans = INT_MAX;
                for(int k = i + 1; k < j; k++) {
                    ans = min(ans, v[i] * v[j] * v[k] + dp[i][k] + dp[k][j]);
                }
                dp[i][j] = ans;
            }
        }
        return dp[0][n - 1];
    }
    int minScoreTriangulation(vector<int>& values) {
        int n = values.size();
        return solve(values);
    }
};
```

j = i

can't form a triangle

j = i+1

still can't Δ

∴ j = i+2

Minimum Sideways Jump

There is a **3 lane road** of length `n` that consists of `n + 1` **points** labeled from `0` to `n`. A frog **starts** at point `0` in the **second** lane and wants to jump to point `n`. However, there could be obstacles along the way.

You are given an array `obstacles` of length `n + 1` where each `obstacles[i]` (**ranging from 0 to 3**) describes an obstacle on the lane `obstacles[i]` at point `i`. If `obstacles[i] == 0`, there are no obstacles at point `i`. There will be **at most one** obstacle in the 3 lanes at each point.

- For example, if `obstacles[2] == 1`, then there is an obstacle on lane 1 at point 2.

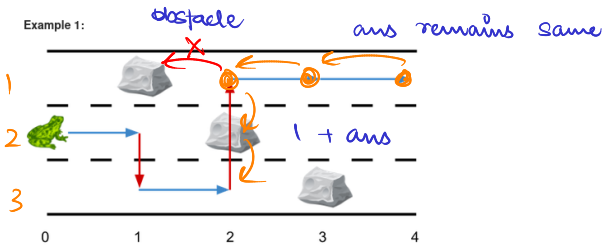
The frog can only travel from point `i` to point `i + 1` on the same lane if there is not an obstacle on the lane at point `i + 1`. To avoid obstacles, the frog can also perform a **side jump** to jump to **another** lane (even if they are not adjacent) at the **same** point if there is no obstacle on the new lane.

- For example, the frog can jump from lane 3 at point 3 to lane 1 at point 3.

Return *the **minimum number of side jumps** the frog needs to reach **any lane** at point n starting from lane `2` at point 0.*

Note: There will be no obstacles on points `0` and `n`.

Example 1:




```
Input: obstacles = [0,1,2,3,0]
Output: 2
Explanation: The optimal solution is shown by the arrows above. There are 2 side jumps (red arrows).
Note that the frog can jump over obstacles only when making side jumps (as shown at point 2).
```

→ Recursion :-

```
class Solution {
public:
int solve(vector<int> &obstacles, int i, int lane) {
    if(i == 1 && lane != 2) return 1;
    if(i == 1 && lane == 2) return 0;

    if(obstacles[i - 1] != lane) {
        return solve(obstacles, i - 1, lane);
    } else {
        int ans = INT_MAX;
        for(int l = 1; l < 4; l++) {
            if(l != lane && obstacles[i] != l) {
                int jumps = solve(obstacles, i, l);
                if(jumps != -1)
                    ans = min(ans, 1 + jumps);
            }
        }
        if(ans == INT_MAX) return -1;
        return ans;
    }
}

    int minSideJumps(vector<int>& obstacles) {
        int n = obstacles.size();
        int ans = INT_MAX;
        for(int i = 1; i < 4; i++) {
            ans = min(ans, solve(obstacles, n - 1, i));
        }
        return ans;
    }
};
```

Base case

It's given that it starts from lane 2

Therefore, if its not in the lane 2 we need to add 1 while returning from n-1 to 0.

→ Memoization :-

```
class Solution {
public:
    int solve(vector<int> &obstacles, int i, int lane, vector<vector<int>> &dp) {
        if(i == 1 && lane != 2) return 1;
        if(i == 1 && lane == 2) return 0;

        if(dp[lane][i] != -1) return dp[lane][i];

        if(obstacles[i - 1] != lane) {
            return solve(obstacles, i - 1, lane, dp);
        } else {
            int ans = INT_MAX;
            for(int l = 1; l < 4; l++) {
                if(l != lane && obstacles[i] != l) {
                    int jumps = solve(obstacles, i, l, dp);
                    if(jumps != -1)
                        ans = min(ans, 1 + jumps);
                }
            }
            if(ans == INT_MAX) return -1;
            dp[lane][i] = ans;
            return dp[lane][i];
        }
    }

    int minSideJumps(vector<int>& obstacles) {
        int n = obstacles.size();
        vector<vector<int>> dp(4, vector<int>(n + 1, -1));
        int ans = INT_MAX;
        for(int i = 1; i < 4; i++) {
            ans = min(ans, solve(obstacles, n - 1, i, dp));
        }
        return ans;
    }
};
```

#Reducing Dishes

A chef has collected data on the `satisfaction` level of his `n` dishes. Chef can cook any dish in 1 unit of time.

Like-time coefficient of a dish is defined as the time taken to cook that dish including previous dishes multiplied by its satisfaction level i.e. `time[i] * satisfaction[i]`.

Return the maximum sum of **like-time coefficient** that the chef can obtain after preparing some amount of dishes.

Dishes can be prepared in **any** order and the chef can discard some dishes to get this maximum value.

Example 1:

```
Input: satisfaction = [-1,-8,0,5,-9]
Output: 14
Explanation: After Removing the second and last dish, the maximum total like-time coefficient will be equal to (-1*1 + 0*2 + 5*3 = 14).
Each dish is prepared in one unit of time.
```

Example 2:

```
Input: satisfaction = [4,3,2]
Output: 20
Explanation: Dishes can be prepared in any order, (2*1 + 3*2 + 4*3 = 20)
```

Example 3:

```
Input: satisfaction = [-1,-4,-5]
Output: 0
Explanation: People do not like the dishes. No dish is prepared.
```

Constraints:

- `n == satisfaction.length`
- `1 <= n <= 500`
- `-1000 <= satisfaction[i] <= 1000`

points towards Sorting.

→ Approach :-

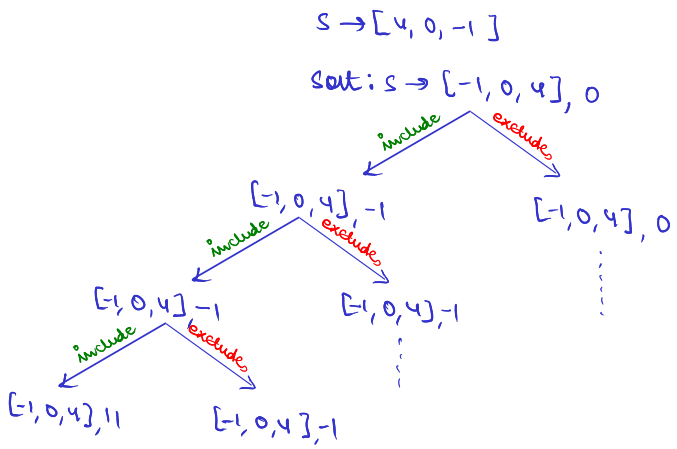
We need to maximize the value of dishes.

- Keep +ve values at last.
- Try to increase the like time[i] for +ve values & ↓ for −ve values.

```
class Solution {
public:
    int maxSatisfaction(vector<int>& satisfaction) {
        sort(satisfaction.begin(), satisfaction.end());
        int n = satisfaction.size();
        int ans = INT_MIN;
        for(int i = 0; i < n; i++) {
            int turn = 1;
            int sum = 0;
            for(int j = i; j < n; j++) {
                sum += turn * satisfaction[j];
                turn++;
            }
            ans = max(sum, ans);
        }
        if(ans < 0) return 0;
        return ans;
    }
};
```

→ DP approach :-

- Sort the array.



→ Recursion :-

```
class Solution {
public:
    int solve(vector<int> &satisfaction, int index, int time) {
        if(index == satisfaction.size()) return 0;

        int include = (time + 1) * satisfaction[index] + solve(satisfaction, index + 1, time + 1);
        int exclude = solve(satisfaction, index + 1, time);

        return max(include, exclude);
    }
    int maxSatisfaction(vector<int>& satisfaction) {
        sort(satisfaction.begin(), satisfaction.end());
        return solve(satisfaction, 0, 0);
    }
};
```

→ Memoization :-

```
class Solution {
public:
    int solve(vector<int> &satisfaction, int index, int time, vector<vector<int>> &dp) {
        if(index == satisfaction.size()) return 0;

        if(dp[index][time] != -1) return dp[index][time];

        int include = (time + 1) * satisfaction[index] + solve(satisfaction, index + 1, time + 1, dp);
        int exclude = solve(satisfaction, index + 1, time, dp);

        return dp[index][time] = max(include, exclude);
    }
    int maxSatisfaction(vector<int>& satisfaction) {
        sort(satisfaction.begin(), satisfaction.end());
        vector<vector<int>> dp(satisfaction.size(), vector<int>(satisfaction.size(), -1));
        return solve(satisfaction, 0, 0, dp);
    }
};
```

```
class Solution {
public:
    int solve(vector<int> &satisfaction, int index, int time, vector<vector<int>> &dp) {
        if(index == satisfaction.size()) return 0;

        if(dp[index][time] != -1) return dp[index][time];

        int include = (time) * satisfaction[index] + solve(satisfaction, index + 1, time + 1, dp);
        int exclude = solve(satisfaction, index + 1, time, dp);

        return dp[index][time] = max(include, exclude);
    }
    int maxSatisfaction(vector<int>& satisfaction) {
        sort(satisfaction.begin(), satisfaction.end());
        vector<vector<int>> dp(satisfaction.size(), vector<int>(satisfaction.size() + 1, -1));
        return solve(satisfaction, 0, 1, dp);
    }
};
```

→ Both the solutions are identical, we are just shifting the time coordinates.
→ In the first sol", time starts with 0 (the matrix value of 0 corresponds to 1).
→ In second sol", time starts with 1.

→ Tabulation :-

```
class Solution {
public:
    int solve(vector<int> &satisfaction) {

        vector<vector<int>> dp(satisfaction.size() + 1, vector<int>(satisfaction.size() + 1, 0));

        for(int i = satisfaction.size() - 1; i >= 0; i--) {
            for(int j = satisfaction.size() - 1; j >= 0; j--) {
                int include = (j + 1) * satisfaction[i] + dp[i + 1][j + 1];
                int exclude = dp[i + 1][j];

                dp[i][j] = max(include, exclude);
            }
        }
        return dp[0][0];
    }
    int maxSatisfaction(vector<int>& satisfaction) {
        sort(satisfaction.begin(), satisfaction.end());
        return solve(satisfaction);
    }
};
```

dp[i][j] → dp[i+1][j+1] (Next row, next col)
dp[i][j] → dp[i+1][j] (Next row same col)

→ Space Optimization :-

```
class Solution {
public:
    int solve(vector<int> &satisfaction) {

        vector<vector<int>> dp(satisfaction.size() + 1, vector<int>(satisfaction.size() + 1, 0));
        vector<int> next(satisfaction.size() + 1, 0);
        vector<int> curr(satisfaction.size() + 1, 0);

        for(int i = satisfaction.size() - 1; i >= 0; i--) {
            for(int j = satisfaction.size() - 1; j >= 0; j--) {
                int include = (j + 1) * satisfaction[i] + next[j + 1];
                int exclude = next[j];

                curr[j] = max(include, exclude);
            }
            next = curr;
        }
        return curr[0];
    }
    int maxSatisfaction(vector<int>& satisfaction) {
        sort(satisfaction.begin(), satisfaction.end());
        return solve(satisfaction);
    }
};
```

Longest Increasing Subsequence

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

Example 1:

```
Input: nums = [10,9,2,5,3,7,101,18]
Output: 4
Explanation: The longest increasing subsequence is [2,3,7,101], therefore the length is 4.
```

Example 2:

```
Input: nums = [0,1,0,3,2,3]
Output: 4
```

Example 3:

```
Input: nums = [7,7,7,7,7,7,7]
Output: 1
```

Constraints:

- `1 <= nums.length <= 2500`
- `-10^4 <= nums[i] <= 10^4`

Follow up: Can you come up with an algorithm that runs in `O(n log(n))` time complexity?

→ Approach :-

```
class Solution {
public:
    void solve(vector<int> &nums, int index, int lastIncludedValue, int currLen, int &maxLen) {
        int n = nums.size();
        if(index == n) {
            maxLen = max(maxLen, currLen);
            return;
        }
        if(nums[index] > lastIncludedValue) {
            solve(nums, index + 1, nums[index], currLen + 1, maxLen);
        }

        solve(nums, index + 1, lastIncludedValue, currLen, maxLen);

        return;
    }
    int lengthOfLIS(vector<int>& nums) {
        int currLen = 0, maxLen = 0;

        solve(nums, 0, INT_MIN, currLen, maxLen);
        return maxLen;
    }
};
```

→ Recursion :-

```
class Solution {
public:
    int solve(vector<int> &nums, int curr, int prev) {
        int n = nums.size();
        if(curr == n) return 0;

        int take = 0;
        if(prev == -1 || nums[curr] > nums[prev]) {
            take = 1 + solve(nums, curr + 1, curr);
        }
        int notTake = solve(nums, curr + 1, prev);

        return max(take, notTake);
    }
    int lengthOfLIS(vector<int>& nums) {
        return solve(nums, 0, -1);
    }
};
```

→ Tabulation :-

```
class Solution {
public:
    int solve(vector<int> &nums) {
        int n = nums.size();
        vector<vector<int>> dp(n + 1, vector<int>(n + 1, 0));

        for(int curr = n - 1; curr >= 0; curr--) {
            for(int prev = curr - 1; prev >= -1; prev--) {
                int take = 0;
                if(prev == -1 || nums[curr] > nums[prev]) {
                    take = 1 + dp[curr + 1][curr + 1];
                }
                int notTake = dp[curr + 1][prev + 1];
                dp[curr][prev + 1] = max(take, notTake);
            }
        }
        return dp[0][0];
    }
    int lengthOfLIS(vector<int>& nums) {
        return solve(nums);
    }
};
```

} just to change the coordinate.

if we will make only n size then

dp[curr+1][curr+1];

go out of bound

∵ curr = n-1 initially.

→ Memoization :-

```
class Solution {
public:
    int solve(vector<int> &nums, int curr, int prev, vector<vector<int>> &dp) {
        int n = nums.size();
        if(curr == n) return 0;

        if(dp[curr][prev + 1] != -1) return dp[curr][prev + 1];

        int take = 0;
        if(prev == -1 || nums[curr] > nums[prev]) {
            take = 1 + solve(nums, curr + 1, curr, dp);
        }
        int notTake = solve(nums, curr + 1, prev, dp);

        return dp[curr][prev + 1] = max(take, notTake);
    }
    int lengthOfLIS(vector<int>& nums) {
        int n = nums.size();
        vector<vector<int>> dp(n, vector<int>(n + 1, -1));
        return solve(nums, 0, -1, dp);
    }
};
```

→ Space Optimization :-

```
class Solution {
public:
    int solve(vector<int> &nums) {
        int n = nums.size();

        vector<int> currRow(n + 1, 0);
        vector<int> next(n + 1, 0);

        for(int curr = n - 1; curr >= 0; curr--) {
            for(int prev = curr - 1; prev >= -1; prev--) {
                int take = 0;
                if(prev == -1 || nums[curr] > nums[prev]) {
                    take = 1 + next[curr + 1];
                }
                int notTake = next[prev + 1];
                currRow[prev + 1] = max(take, notTake);
            }
            next = currRow;
        }
        return next[0];
    }
    int lengthOfLIS(vector<int>& nums) {
        return solve(nums);
    }
};
```

DP with Binary Search :-

Ip : 5, 8, 3, 7, 9, 1

1
3 7
5, 8, 9 ⇒ size = 3.

1, 7, 9 is not LIS, but it's size is the size of LIS

```
class Solution {
public:
    int solveOptimal(vector<int> &nums) {
        int n = nums.size();
        if(n == 0) return 0;

        vector<int> ans;
        ans.push_back(nums[0]);

        for(int i = 1; i < n; i++) {
            if(nums[i] > ans.back()) {
                ans.push_back(nums[i]);
            } else {
                int index = lower_bound(ans.begin(), ans.end(), nums[i]) - ans.begin();
                ans[index] = nums[i];
            }
        }
        return ans.size();
    }
    int lengthOfLIS(vector<int>& nums) {
        return solveOptimal(nums);
    }
};
```

Time complexity : O(NlogN)

Space complexity : O(N)

Russian Dolls Envelope

You are given a 2D array of integers `envelopes` where `envelopes[i] = [wᵢ, hᵢ]` represents the width and the height of an envelope.

One envelope can fit into another if and only if both the width and height of one envelope are greater than the other envelope's width and height.

Return *the maximum number of envelopes you can Russian doll (i.e., put one inside the other).*

Note: You cannot rotate an envelope.

Example 1:

```
Input: envelopes = [[5,4],[6,4],[6,7],[2,3]]
Output: 3
Explanation: The maximum number of envelopes you can Russian doll is 3 ([2,3] => [5,4] => [6,7]).
```

Example 2:

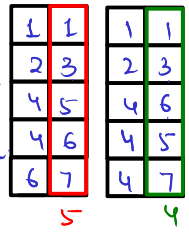
```
Input: envelopes = [[1,1],[1,1],[1,1]]
Output: 1
```

Constraints:

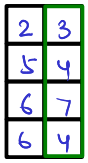
- `1 <= envelopes.length <= 10⁵`
- `envelopes[i].length == 2`
- `1 <= wᵢ, hᵢ <= 10⁵`

→ Approach :-

How about we sort the given array based on increasing width and decreasing height if width is same.



In the LIS (4,5) & (4,6) are also calculated but they can't be included. ∴ Sort height in descending order.



→ Now apply LIS (DP + Binary) on this array.

```cpp
class Solution {
public:
    static bool comp(vector<int> &a, vector<int> &b){
        if(a[0] == b[0]){
            return a[1] > b[1];
        }
        else{
            return a[0] < b[0];
        }
    }

    int solveOptimal(vector<int> &nums) {
        int n = nums.size();
        if(n == 0) return 0;

        vector<int> ans;
        ans.push_back(nums[0]);

        for(int i = 1; i < n; i++) {
            if(nums[i] > ans.back()) {
                ans.push_back(nums[i]);
            } else {
                int index = lower_bound(ans.begin(), ans.end(), nums[i]) - ans.begin();
                ans[index] = nums[i];
            }
        }
        return ans.size();
    }

    int maxEnvelopes(vector<vector<int>>& env) {
        sort(env.begin(),env.end(),comp);

        vector<int> temp(env.size());

        for(int i = 0; i < env.size(); i++) {
            temp[i] = env[i][1];
        }
        return solveOptimal(temp);
    }
};
```

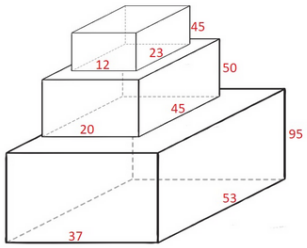
Maximum height by stacking cuboids

Given `n` `cuboids` where the dimensions of the `i`th cuboid is `cuboids[i] = [width_i, length_i, height_i]` (**0-indexed**). Choose a **subset** of `cuboids` and place them on each other.

You can place cuboid `i` on cuboid `j` if `width_i <= width_j` and `length_i <= length_j` and `height_i <= height_j`. You can rearrange any cuboid's dimensions by rotating it to put it on another cuboid.

Return *the **maximum height** of the stacked* `cuboids`.

Example 1:




```
Input: cuboids = [[50,45,20],[95,37,53],[45,23,12]]
Output: 190
Explanation:
Cuboid 1 is placed on the bottom with the 53x37 side facing down with height 95.
Cuboid 0 is placed next with the 45x20 side facing down with height 50.
Cuboid 2 is placed next with the 23x12 side facing down with height 45.
The total height is 95 + 50 + 45 = 190.
```

Example 2:

```
Input: cuboids = [[38,25,45],[76,35,3]]
Output: 76
Explanation:
You can't place any of the cuboids on the other.
We choose cuboid 1 and rotate it so that the 35x3 side is facing down and its height is
76.
```

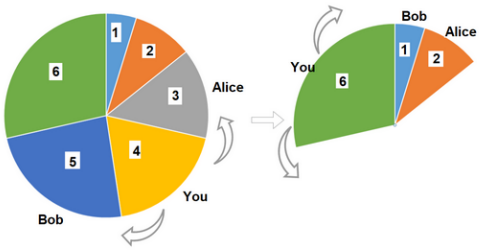
Pizza with 3n slices

There is a pizza with `3n` slices of varying size, you and your friends will take slices of pizza as follows:

- You will pick **any** pizza slice.
- Your friend Alice will pick the next slice in the anti-clockwise direction of your pick.
- Your friend Bob will pick the next slice in the clockwise direction of your pick.
- Repeat until there are no more slices of pizzas.

Given an integer array `slices` that represent the sizes of the pizza slices in a clockwise direction, return *the maximum possible sum of slice sizes that you can pick.*

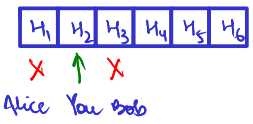
Example 1:

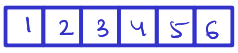



```
Input: slices = [1,2,3,4,5,6]
Output: 10
Explanation: Pick pizza slice of size 4, Alice and Bob will pick slices
with size 3 and 5 respectively. Then Pick slices with size 6, finally Alice
and Bob will pick slice of size 2 and 1 respectively. Total = 4 + 6.
```

→ Approach :-

- If observed carefully, it's similar to house robbery.



- 

Case 1 : If we take index 0 then we can't take last index. ∵ The array is in circular format.

Case 2 : If we take last index we can't take 0th.

```
class Solution {
public:
    int solve(int index, int endIndex, vector<int>& slices, int n) {
        if(n == 0 || index > endIndex) return 0;

        int take = slices[index] + solve(index + 2, endIndex, slices, n - 1);
        int notTake = solve(index + 1, endIndex, slices, n);

        return max(take, notTake);
    }
    int maxSizeSlices(vector<int>& slices) {
        int k = slices.size();
        int case1 = solve(0, k - 2, slices, k / 3);
        int case2 = solve(1, k - 1, slices, k / 3);
        return max(case1, case2);
    }
};
```

→ Memoization :-

```
class Solution {
public:
    int solve(int index, int endIndex, vector<int>& slices, int n, vector<vector<int>> &dp) {
        if(n == 0 || index > endIndex) return 0;

        if(dp[index][n] != -1) return dp[index][n];

        int take = slices[index] + solve(index + 2, endIndex, slices, n - 1, dp);
        int notTake = solve(index + 1, endIndex, slices, n, dp);

        return dp[index][n] = max(take, notTake);
    }
    int maxSizeSlices(vector<int>& slices) {
        int k = slices.size();
        vector<vector<int>> dp(k, vector<int>(k + 1, -1));
        int case1 = solve(0, k - 2, slices, k / 3, dp);
        vector<vector<int>> dp2(k, vector<int>(k + 1, -1));
        int case2 = solve(1, k - 1, slices, k / 3, dp2);
        return max(case1, case2);
    }
};
```

} Can be K, it will just reduce the index by 1.

→ Tabulation :-

```
class Solution {
public:
    int solve(vector<int>& slices) {
        int k = slices.size();
        vector<vector<int>> dp1(k + 2, vector<int>(k, 0));
        vector<vector<int>> dp2(k + 2, vector<int>(k, 0));

        for(int i = k - 2; i >= 0; i--) {
            for(int n = 1; n <= k/3; n++) {
                int take = slices[i] + dp1[i + 2][n - 1];
                int notTake = dp1[i + 1][n];

                dp1[i][n] = max(take, notTake);
            }
        }

        int case1 = dp1[0][k/3];

        for(int i = k - 1; i >= 1; i--) {
            for(int n = 1; n <= k/3; n++) {
                int take = slices[i] + dp2[i + 2][n - 1];
                int notTake = dp2[i + 1][n];

                dp2[i][n] = max(take, notTake);
            }
        }

        int case2 = dp2[1][k/3];
        return max(case1, case2);

    }
    int maxSizeSlices(vector<int>& slices) {
        return solve(slices);
    }
};
```

just to make sure dp1[i+2][n-1] does not go out of bound.

→ Space optimization :-

```
class Solution {
public:
    int solve(vector<int>& slices) {
        int k = slices.size();
        vector<int> prev1(k + 2, 0);
        vector<int> curr1(k + 2, 0);
        vector<int> next1(k + 2, 0);
        vector<int> prev2(k + 2, 0);
        vector<int> curr2(k + 2, 0);
        vector<int> next2(k + 2, 0);

        for(int i = k - 2; i >= 0; i--) {
            for(int n = 1; n <= k/3; n++) {
                int take = slices[i] + next1[n - 1];
                int notTake = curr1[n];
                prev1[n] = max(take, notTake);
            }
            next1 = curr1;
            curr1 = prev1;
        }

        int case1 = curr1[k/3];

        for(int i = k - 1; i >= 1; i--) {
            for(int n = 1; n <= k/3; n++) {
                int take = slices[i] + next2[n - 1];
                int notTake = curr2[n];
                prev2[n] = max(take, notTake);
            }
            next2 = curr2;
            curr2 = prev2;
        }

        int case2 = curr2[k/3];
        return max(case1, case2);

    }
    int maxSizeSlices(vector<int>& slices) {
        return solve(slices);
    }
};
```