Dynamic Programming

→ A bigger problem can be solved using the optimal sol^n of small subproblem.

→ Overlapping subproblems

(✶) Those who forget the past, are condemned to repeat it.

- → Top down (Recursion + Memoization)
  - ↓ Store the value of subproblem.
- → Bottom up (Tabulation)

} Space Optimization

Fibonacci Sequence :-

```cpp
// 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)

```cpp
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

```cpp
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

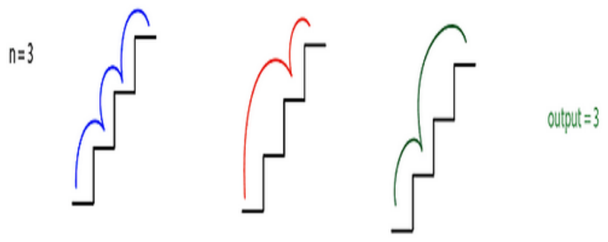
You have been given a number of stairs. Initially, you are at the 0th stair, and you need to reach the Nth stair.

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;
    }
};
```

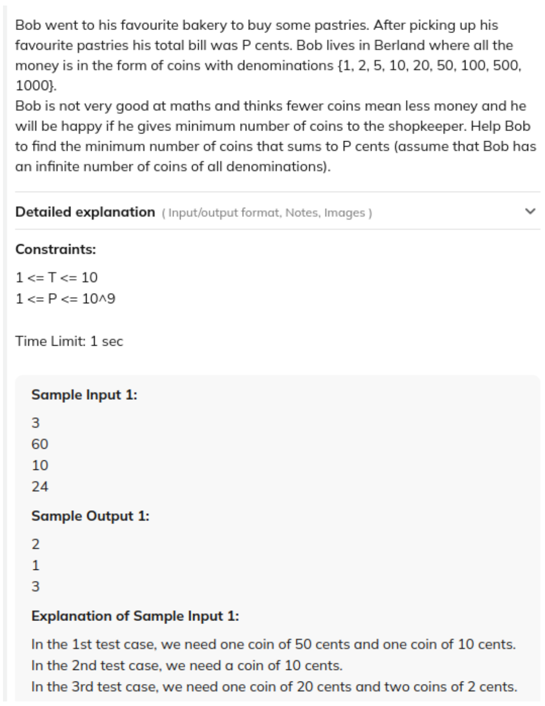
→ Tabulation :-

```
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.

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

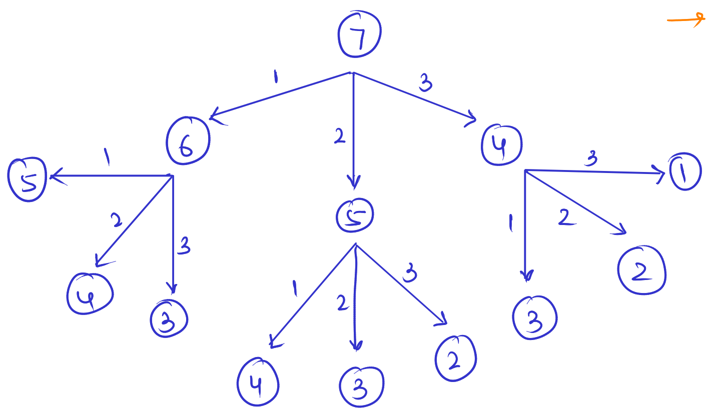
If N=3 and X=7 and array elements are [1,2,3].
Way 1 - You can take 4 elements [2, 2, 2, 1] as 2 + 2 + 2 + 1 = 7.
Way 2 - You can take 3 elements [3, 3, 1] as 3 + 3 + 1 = 7.
Here, you can see in Way 2 we have used 3 coins to reach the target sum of 7.
Hence the output is 3.

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



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

→ 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;
}
```

→ 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;
}
```

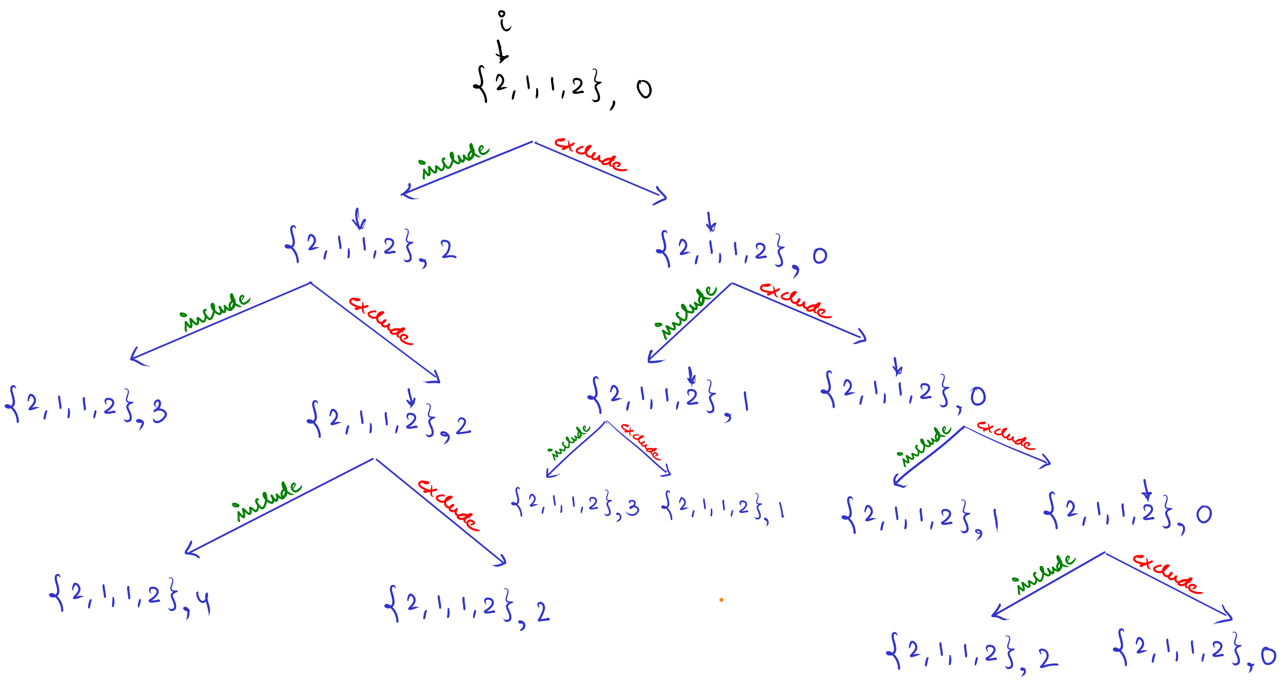
→ Memoization :-

```
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)

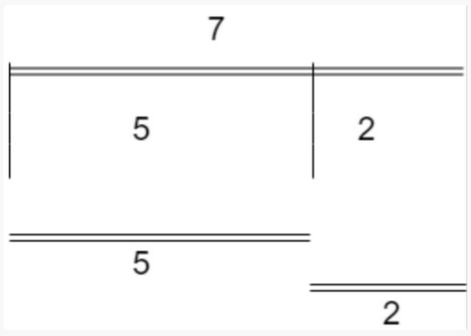
Sample Input 1:

```
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.