1) **Fibonacci series**: 0 1 1 2 3 5 8 …

Observation: Every nth term is result of/sum of n-1th and n-2th term. To solve T(n), we need to solve
subproblems, can be solved by recursion method.
//Recurrence relation:
f(n) = f(n-1) + f(n-2), except n=0 and n=1(will be part of base case)

```
//Using recursion
int fib(int n)
{
        //Base case:
        if(n<2) //while n=0 and n=1
                return n;
        return f(n-1)+f(n-2);
}
```

Time complexity: o(2^n)

Memorization:

```
 int fibUtil(int n,  vector<int>&dp)
   {
      if(dp[n] !=-1)
         return dp[n];
       //Base case:
      if(n<2)
         return n;
      return dp[n] = fib(n-1)+fib(n-2);
   }
   int fib(int n)
   {
     vector<int>dp(n+1, -1); //n={0...n} =>total n+1
     return fibUtil(n, dp);
   }
```

Time Complexity: O(N)
Reason: The overlapping subproblems will return the answer in constant time O(1). Therefore the total number of new subproblems we solve is 'n'. Hence total time complexity is O(N).

Space Complexity: O(N)
Reason: We are using a recursion stack space(O(N)) and an array (again O(N)). Therefore total space complexity will be O(N) + O(N) ≈ O(N)

Tabulation:

```
 int fibUtil(int n,  vector<int>&dp)
   {
      if(dp[n] !=-1)
         return dp[n];
       //Base case:
      dp[0]=0;
      dp[1]=1;
      for(int i=2; i<=n; i++)
      {
         dp[i] = dp[i-1]+dp[i-2];
      }
      return dp[n];
   }
   int fib(int n)
   {
     vector<int>dp(n+1, -1); //n={0...n} =>total n+1
     return fibUtil(n, dp);
   }
```

Time Complexity: O(N)
Reason: We are running a simple iterative loop

Space Complexity: O(N)
Reason: We are using an external array of size 'n+1'.
----

2) **Climbing Stairs**: Given a number of stairs. Starting from the 0th stair we need to climb to the "Nth" stair. At a time, we can climb either one or two steps. We need to return the total number of distinct ways to reach from 0th to Nth stair.

//Observation: We have to find all distinct ways to reach to top stair.
Recurrence relation:
f(n)=f(n-1)+f(n-2)

Recursion:

```
int climbStairs(int n)
{
        //Base case:
        //if n=1, there is only one way(1 step move) and n=0 then there is only one way.

        if(n<=1)
                return 1;
        return climbStairs(n-1)+climbStairs(n-2);
}
```

//Memorization

```
int climbStairs(int n)
```

```
  {
    vector<int>dp(n+1, -1); //steps 0->n(n+1)
          //Base case:
          //if n=1, there is only one way(1 step move) and n=0 then there is only one way.
          if(n<=2)
                  return n;
    if(dp[n] != -1)
      return dp[n];
          return dp[n]=climbStairs(n-1)+climbStairs(n-2);
  }

//Tabulation:
int climbStairs(int n)
{
        vector<int>dp(n+1, -1);
        //Base case:
        //if n=1, there is only one way(1 step move) and n=0 then there is only one way.
        dp[0]=1;
        dp[1]=1;

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

3) Frog jump:

Problem statement

There is a frog on the '1st' step of an 'N' stairs long staircase. The frog wants to reach the 'Nth' stair. 'HEIGHT[i]' is the height of the '(i+1)th' stair.If Frog jumps from 'ith' to 'jth' stair, the energy lost in the jump is given by absolute value of (HEIGHT[i-1] - HEIGHT[j-1]). If the Frog is on 'ith' staircase, he can jump either to '(i+1)th' stair or to '(i+2)th' stair. Your task is to find the minimum total energy used by the frog to reach from '1st' stair to 'Nth' stair.

For Example

If the given 'HEIGHT' array is [10,20,30,10], the answer 20 as the frog can jump from 1st stair to 2nd stair (|20-10| = 10 energy lost) and then a jump from 2nd stair to last stair (|10-20| = 10 energy lost). So, the total energy lost is 20.

//Observation: Frog will jump from 1 to nth stair in two ways: jump step 1 or step 2. Basically, we need

to check all possible way and then minimum energy consumption from it, so recursion would be applied to solve the problem.

Recurrence relation:

```
f(index, height[])

{

  //Base case:

  if(index==0)

    return 0; //energy loss would be 0.

  jumpone = f(index-1, height) + abs(height[index-1] - height[index])

  if(index>1)

  jumptwo = f(index-2, height) + abs(height[index-2] - height[index])


    return min(jumpone, jumptwo)

}

int totalEnergyLost(int index, vector<int> &heights, vector<int>&dp)

{

  if(dp[index] != -1)

    return dp[index];

  //Base case:

  if(index==0)

    return 0; //energy lost would be 0

  int oneStepJump= totalEnergyLost(index-1, heights, dp) + abs(heights[index-1] - heights[index]);

  int twoStepJump=INT_MAX;

  if(index>1)

    twoStepJump= totalEnergyLost(index-2, heights, dp) + abs(heights[index-2] - heights[index]);

  return dp[index] =min(oneStepJump, twoStepJump);
}

int frogJump(int n, vector<int> &heights)
{
   vector<int>dp(n, -1); //DP array size would be n(1->n)
   int cost =totalEnergyLost(n-1, heights, dp); //we are considering array from 0 to n-1, so
max index would be n-1
   return cost;
}
```

----

4) **Knapsack problem**:

A thief is robbing a store and can carry a maximum weight of ‘W’ into his knapsack.

There are 'N' items available in the store and the weight and value of each item is known to the thief. Considering the constraints of the maximum weight that a knapsack can carry, you have to find the maximum profit that a thief can generate by stealing items.

Note: The thief is not allowed to break the items.

For example, N = 4, W = 10 and the weights and values of items are weights = [6, 1, 5, 3] and values = [3, 6, 1, 4]. Then the best way to fill the knapsack is to choose items with weight 6, 1 and 3. The total value of knapsack = 3 + 6 + 4 = 13.

Observation:

we have indexes-> 0,1,2,3, will start from n-1th would be 3

Recurrence_relation:

```
f(index, W)
{
   //Base case:
   if(index==0) //this is first item, so we will see if weight[index]<=W, then take it else return 0
   {
        if(wt[0]<W)
        {
          return val[0];
        }
        else
        {
          return 0;
        }
      int notpick= 0 + f(index-1, W) (if not picking current weight of n-1th index, adding 0 value
and weight  will be w, as not picked
      //We can pick only if weight of current index <=W
      int take = INT_MIN
      if(wt[index]<W)
        pick= val[index] + f(index-1, W-wt[index])
      return max(pick, notpick);
    }
    int knapsackUtil(int W, int wt[], int val[], int index, vector<vector<int>>&dp)
      {
       //Base case:
       if (index < 0)
       {
          return 0;
       }
      // If the result for this state is already calculated, return it
      if (dp[index][W] != -1) {
        return dp[index][W];
      }


      // Calculate the maximum value by either excluding the current item or including it
```

```
  int notTaken = knapsackUtil(W, wt, val, index - 1, dp);
  int taken = 0;

  // Check if the current item can be included without exceeding the knapsack's capacity
  if (wt[index] <= W) {
    taken = val[index] + knapsackUtil(W - wt[index], wt, val, index - 1, dp);
  }
  // Store the result in the DP table and return
  return dp[index][W] = max(notTaken, taken);
}
  //Function to return max value that can be put in knapsack of capacity W.
  int knapSack(int W, int wt[], int val[], int n)
  {
   vector<vector<int>>dp(n, vector<int>(W+1, -1));
   return knapsackUtil(W, wt, val, n-1, dp);
  }
```

---

5) **Longest increasing Sequence**:

Input: nums = [10,9,2,5]
Output: 2

```
                  f(index=0, prev_index=-1)
                                 /\
                 Pick /  \ Don't pick
                    /    \
                              f(1,0)   f(1,-1)
                                 /\
        pick /  \ Don't pick
           /    \
      f(2,1) f(2, -1)
```

Incase of pick, will increase length by 1 and incase of don't pick, will consider 0.

Recurrence_relation:
f(ind, prev_ind) //f(0, -1) //first call to this method would be

```
{
  //Base case: Reached at last index and no other index is left to pick
  if(ind==n)
  {
   return 0;
  }
  //Not take:
  notTake = 0+  f(ind+1, prev_ind)
  //Take:
   take=0
   take= 1+ f(ind+1, ind)
   return max(notTake, take)
 }

// Function to find the length of the longest increasing subsequence
int getAns(int ind, int prev_index, vector<int>nums, int n, vector<vector<int>>& dp) {
   // Base condition
   if (ind == n)
      return 0;

   if (dp[ind][prev_index + 1] != -1)
      return dp[ind][prev_index + 1];

   int notTake = 0 + getAns(ind + 1, prev_index, nums, n, dp);

   int take = 0;

   if (prev_index == -1 || nums[ind] > nums[prev_index]) {
      take = 1 + getAns(ind + 1, ind, nums, n, dp);
   }

   return dp[ind][prev_index + 1] = max(notTake, take);
}

   int lengthOfLIS(vector<int>& nums)
   {
      int n=nums.size();
       // Create a 2D DP array initialized to -1
      vector<vector<int>> dp(n, vector<int>(n + 1, -1));
      return getAns(0, -1, nums, n, dp); //getAns(ind, prev_ind, num_array, n, dp)
   }
```
----

6) **max_sum_non_adjacent_element:**
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.
Ex:
Input: arr[] = {5, 5, 10, 100, 10, 5}
Output: 110
Explanation: Pick the subsequence {5, 100, 5}.
The sum is 110 and no two elements are adjacent. This is the highest possible sum.

Input: arr[] = {3, 2, 7, 10}
Output: 13
Explanation: The subsequence is {3, 10}. This gives sum = 13.

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.

Recurrence relation:
```
f(ind,arr[])
{
  //Base case:
  if(ind==0) return 0
  pick=arr[ind]+f(ind-1,arr)
  notpick=0+f(ind-1, arr)
  return max(pick,notpick)
}
```

```
// Function to solve the problem using dynamic programming
int solveUtil(int ind, vector<int>& arr, vector<int>& dp) {
  // If the result for this index is already computed, return it
  if (dp[ind] != -1)
    return dp[ind];

  // Base cases: if you reaching index 0, means you have not picked 1st index, you must
pick 0th index array value.
  if (ind == 0)
    return arr[ind];
  if (ind < 0)
    return 0;

  // Choose the current element or skip it, and take the maximum
  int pick = arr[ind] + solveUtil(ind - 2, arr, dp); // Choosing the current element
  int nonPick = 0 + solveUtil(ind - 1, arr, dp);     // Skipping the current element
```

```
   // Store the result in the DP table and return it
   return dp[ind] = max(pick, nonPick);
}

// Function to initiate the solving process
int solve(int n, vector<int>& arr) {
   vector<int> dp(n, -1); // Initialize the DP table with -1
   return solveUtil(n - 1, arr, dp); // Start solving from the last element
}
```
----

7) minimum_path_sum_grid:

Given a m x n grid filled with non-negative numbers, find a path from top left to bottom right,
which minimizes the sum of all numbers along its path.
Note: You can only move either down or right at any point in time.

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

Input: grid = [[1,2,3],[4,5,6]]
Output: 12

Recurrence relation:

```
f(i, j)
{
    //Base case:
    if(i==0 && j==0)
        return arr[0][0];
    if(i<0 || j<0)
        return INT_MAX;
    up = arr[i][j] + f(i, j-1)
    left = arr[i][j] + f(i-1, j)
    return min(up, left)

}
```

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

```
int solve(vector<vector<int>> & dp, vector<vector<int>> &grid, int row, int col) {
      if(row == 0 && col == 0) return grid[0][0];
      if(row < 0 || col < 0) return INT_MAX;
      if(dp[row][col] != -1) return dp[row][col];

      return dp[row][col] = grid[row][col] + min(solve(dp, grid, row - 1, col), solve(dp, grid,
row, col - 1));

   }
   int minPathSum(vector<vector<int>>& grid) {
      int ans = 0;
      int m = grid.size();
      int n = grid[0].size();

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

      return solve(dp, grid, m-1, n-1);

   }

int main() {
   vector<vector<int>> matrix{
      {5, 9, 6},
      {11, 5, 2}
   };

   int n = matrix.size();
   int m = matrix[0].size();

   cout << "Minimum sum path: " << minSumPath(n, m, matrix) << endl;
   return 0;
}
```
----

8. Coin Change:

We are given an array Arr with N distinct coins and a target.
We have an infinite supply of each coin denomination.
We need to find the number of ways we sum up the coin values to give us the target.
Each coin can be used any number of times.

Example 1:

Input: coins = [1,2,5], amount = 11
Output: 3
Explanation: 11 = 5 + 5 + 1

Example 2:

Input: coins = [2], amount = 3
Output: -1
Example 3:

Input: coins = [1], amount = 0
Output: 0

Recurrence_relation:

```
 f(ind, T)  //f(n-1, T)
 {
   //Base case: reached to last index and should return T/coins[0](divisible) or
INT_MAX(not divisible)
       if(index==0)
      {
         if(T%coins[index]==0)
         {
            return T/coins[index];
         }
         else
         {
            return INT_MAX;
         }
        //Not take:
       notTake = f(ind-1, T)
      //Take:
      take=0
      if(arr[ind]<=T) //we can take it as current index value is less than or equal to Target
sum
      take= f(ind, T-arr[ind]) //Don't jump to ind-1 index as coins have many
denominations of it.
      return min(take, notTake);

 }
```

```
int coinChangeUtil(int index, int T, vector<vector<int>>& dp, vector<int>& coins)
  {
      if(index==0)
      {
         if(T%coins[index]==0)
         {
            return T/coins[index];
         }
```

```
            else
            {
               return INT_MAX;
            }
         }
         if(dp[index][T]!=-1)
         {
            return dp[index][T];
         }
         int notTake=coinChangeUtil(index-1, T, dp, coins);
         int take=INT_MAX;
         if(T>=coins[index])
         {
            take=1+coinChangeUtil(index, T-coins[index], dp, coins);
         }
         return dp[index][T]=min(take, notTake);
      }
      int coinChange(vector<int>& coins, int amount)
      {
         int n=coins.size();
         vector<vector<int>>dp(n, vector<int>(amount+1, -1));
         int ans=coinChangeUtil(n-1, amount, dp, coins);

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

9. **Subset Sum:**

Given a set of non-negative integers and a value sum, the task is to check if there is a subset of the given set whose sum is equal to the given sum.

Examples:
Input: set[] = {3, 34, 4, 12, 5, 2}, sum = 9
Output: True
Explanation: There is a subset (4, 5) with sum 9.

Input: set[] = {3, 34, 4, 12, 5, 2}, sum = 30
Output: False
Explanation: There is no subset that add up to 30.

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

```
using namespace std;

// Function to check if there is a subset of 'arr' with a sum equal to 'target'
bool subsetSumUtil(int ind, int target, vector<int>& arr, vector<vector<int>>& dp) {
  // If the target sum is 0, we have found a subset
  if (target == 0)
    return true;

  // If we have reached the first element in 'arr'
  if (ind == 0)
    return arr[0] == target;

  // If the result for this subproblem has already been computed, return it
  if (dp[ind][target] != -1)
    return dp[ind][target];

  // Try not taking the current element into the subset
  bool notTaken = subsetSumUtil(ind - 1, target, arr, dp);

  // Try taking the current element into the subset if it doesn't exceed the target
  bool taken = false;
  if (arr[ind] <= target)
    taken = subsetSumUtil(ind - 1, target - arr[ind], arr, dp);

  // Store the result in the dp array to avoid recomputation
  return dp[ind][target] = notTaken || taken;
}

// Function to check if there is a subset of 'arr' with a sum equal to 'k'
bool subsetSumToK(int n, int k, vector<int>& arr) {
  // Initialize a 2D DP array for memoization
  vector<vector<int>> dp(n, vector<int>(k + 1, -1));

  // Call the recursive subsetSumUtil function
  return subsetSumUtil(n - 1, k, arr, dp);
}

int main() {
  vector<int> arr = {1, 2, 3, 4};
  int k = 4;
  int n = arr.size();

  // Call the subsetSumToK function and print the result
  if (subsetSumToK(n, k, arr))
```

```
    cout << "Subset with the given target found";
  else
    cout << "Subset with the given target not found";

  return 0;
}
```
----

10. Longest common subsequence:
Example 1:
Input: text1 = "abcde", text2 = "ace"
Output: 3
Explanation: The longest common subsequence is "ace" and its length is 3.
Example 2:
Input: text1 = "abc", text2 = "abc"
Output: 3
Explanation: The longest common subsequence is "abc" and its length is 3.
Example 3:
Input: text1 = "abc", text2 = "def"
Output: 0
Explanation: There is no such common subsequence, so the result is 0.

```
// Function to find the length of the Longest Common Subsequence (LCS)
  int lcsUtil(string& s1, string& s2, int ind1, int ind2, vector<vector<int>>& dp)
  {
    // Base case: If either string reaches the end, return 0
    if (ind1 < 0 || ind2 < 0)
      return 0;

    // If the result for this pair of indices is already calculated, return it
    if (dp[ind1][ind2] != -1)
      return dp[ind1][ind2];

    // If the characters at the current indices match, increment the LCS length
    if (s1[ind1] == s2[ind2])
      return dp[ind1][ind2] = 1 + lcsUtil(s1, s2, ind1 - 1, ind2 - 1, dp);
     else
    // If the characters don't match, consider two options: moving either left or up in the
strings
    return dp[ind1][ind2] = max(lcsUtil(s1, s2, ind1, ind2 - 1, dp), lcsUtil(s1, s2, ind1 - 1,
ind2, dp));
}
  int longestCommonSubsequence(string text1, string text2)
        {
  int n = text1.size();
  int m = text2.size();

  vector<vector<int>> dp(n, vector<int>(m, -1)); // Create a DP table
  return lcsUtil(text1, text2, n - 1, m - 1, dp);
  }
```
----

11. Edit Distance:

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

You have the following three operations permitted on a word:

- Insert a character
- Delete a character
- Replace a character

Example 1:

Input: word1 = "horse", word2 = "ros"

Output: 3

Explanation:

horse -> rorse (replace 'h' with 'r')

rorse -> rose (remove 'r')

rose -> ros (remove 'e')

Example 2:

Input: word1 = "intention", word2 = "execution"

Output: 5

Explanation:

intention -> inention (remove 't')

inention -> enention (replace 'i' with 'e')

enention -> exention (replace 'n' with 'x')

exention -> exection (replace 'n' with 'c')

exection -> execution (insert 'u')

```
// Function to calculate the edit distance between two strings
int editDistanceUtil(string& S1, string& S2, int i, int j, vector<vector<int>>& dp) {
  // Base cases
  if (i < 0)
    return j + 1;
  if (j < 0)
    return i + 1;

  // If the result for this state has already been calculated, return it
  if (dp[i][j] != -1)
    return dp[i][j];

  // If the characters at the current positions match, no operation is needed
```

```
    if (S1[i] == S2[j])
        return dp[i][j] = 0 + editDistanceUtil(S1, S2, i - 1, j - 1, dp);

    // Minimum of three choices:
    // 1. Replace the character at S1[i] with the character at S2[j]
    // 2. Delete the character at S1[i]
    // 3. Insert the character at S2[j] into S1
    else
        return dp[i][j] = 1 + min(editDistanceUtil(S1, S2, i - 1, j - 1, dp),
                                min(editDistanceUtil(S1, S2, i - 1, j, dp),
                                    editDistanceUtil(S1, S2, i, j - 1, dp)));
}

// Function to calculate the minimum number of operations required to transform S1 into S2
int editDistance(string& S1, string& S2) {
    int n = S1.size();
    int m = S2.size();

    // Create a DP table to memoize results
    vector<vector<int>> dp(n, vector<int>(m, -1));

    // Call the utility function with the last indices of both strings
    return editDistanceUtil(S1, S2, n - 1, m - 1, dp);
}

int main() {
    string s1 = "horse";
    string s2 = "ros";

    // Call the editDistance function and print the result
    cout << "The minimum number of operations required is: " << editDistance(s1, s2);
    return 0;
}
```
----

12. Rod cutting problem:
Given a rod of length ‘N’ units. The rod can be cut into different sizes and each size has a cost associated with it. Determine the maximum cost obtained by cutting the rod and selling its pieces.

```
int cutRod(vector<int> &price, int n)
{
            // since length array is not given therefore let us make one
    int length_cap[n];
    for(int i=1;i<=n;i++)
```

```
      length_cap[i-1]=i;

   // Here W->n , val[]->price[] , wt[]=>length[]

            // Now unbounded Knapsack as usual no changes
   int dp[n+1][n+1];
    for(int i=0;i<=n;i++)
      dp[i][0] = 0;

    for(int j=0;j<=n;j++)
      dp[0][j] = 0;

     for(int i=1;i<=n;i++){
      for(int j=1;j<=n;j++){
         if(length_cap[i-1] <= j){
           dp[i][j] = max(price[i-1] + dp[i][j-length_cap[i-1]],
                       dp[i-1][j]);
         }
         else
            dp[i][j] = dp[i-1][j];
      }
   }
   return dp[n][n];
}
```
----

13. **Super Egg Drop**:
You are given k identical eggs and you have access to a building with n floors labeled from 1 to n.
You know that there exists a floor f where 0 <= f <= n such that any egg dropped at a floor **higher** than f will **break**, and any egg dropped **at or below** floor f will **not break**.
Each move, you may take an unbroken egg and drop it from any floor x (where 1 <= x <= n). If the egg breaks, you can no longer use it. However, if the egg does not break, you may **reuse** it in future moves.

```
 int dp[101][10001];
   int dfs(int e,int n){
      if( n== 0 || n == 1){
         return n;
      }
      if(e == 1) return n;
      if(dp[e][n] != -1) return dp[e][n];
      int ans = 1e9;
      int l = 1, r = n;
      while(l <= r){
         int mid = (l+r)/2;
         int left = dfs(e-1,mid-1);
         int right = dfs(e,n-mid);
         ans = min(ans,1 + max(left,right));
         if(left < right) l = mid+1;
         else r = mid-1;
      }
      return dp[e][n] = ans;
   }
   int superEggDrop(int k, int n) {
      memset(dp,-1,sizeof(dp));
      return dfs(k,n);

   }
```

----

14. Word Break: Given a string s and a dictionary of strings wordDict, return true if s can be segmented into a space-separated sequence of one or more dictionary words.
Note that the same word in the dictionary may be reused multiple times in the segmentation.

Example 1:
Input: s = "leetcode", wordDict = ["leet","code"]
Output: true
Explanation: Return true because "leetcode" can be segmented as "leet code".
Example 2:
Input: s = "applepenapple", wordDict = ["apple","pen"]
Output: true
Explanation: Return true because "applepenapple" can be segmented as "apple pen apple".
Note that you are allowed to reuse a dictionary word.
Example 3:
Input: s = "catsandog", wordDict = ["cats","dog","sand","and","cat"]
Output: false

```
bool wordBreak(string s, unordered_set<string> &set){
        if(s.size() == 0){
                return true;
    }
    for(int i=0; i<s.size(); i++){
      if(set.count(s.substr(0, i+1)) && wordBreak(s.substr(i+1), set)){
        return true;
      }
    }
```

```
        return false;
    }
    bool wordBreak(string s, vector<string>& wordDict) {
        unordered_set<string> set(wordDict.begin(), wordDict.end());
        return wordBreak(s, set);
    }
```

Time Complexity : O(2^N), Given a string of length N, there are N+1 ways to split it into two parts. At each
step, we have a choice: to split or not to split. In the worse case, when all choices are to be checked, that
results in O(2^N).

Space Complexity : O(N), The depth of the recursion tree can go upto N.

Using String + Backtracking + Hash Table.

```
bool wordBreak(string s, unordered_set<string> &set, int start){
        if(start == s.size()){
            return true;
        }
        for(int i=start; i<s.size(); i++){
            if(set.count(s.substr(start, i+1-start)) && wordBreak(s, set, i+1)){
                return true;
            }
        }
        return false;
    }
    bool wordBreak(string s, vector<string>& wordDict) {
        unordered_set<string> set(wordDict.begin(), wordDict.end());
        return wordBreak(s, set, 0);
    }
```

Time Complexity : O(N^3), Size of recursion tree can go up to N^2.
Space Complexity : O(N), The depth of the recursion tree can go upto N.

using String + DP(Memoisation) + Hash Table.

```
bool wordBreak(string s, unordered_set<string> &set, vector<int> &memo, int start){
        if(start == s.size()){
            return true;
        }
        if(memo[start] != -1){
            return memo[start];
        }
        for(int i=start; i<s.size(); i++){
            if(set.count(s.substr(start, i+1-start)) && wordBreak(s, set, memo, i+1)){
                memo[start] = true;
                return true;
            }
        }
        return memo[start] = false;
    }
    bool wordBreak(string s, vector<string>& wordDict) {
        vector<int> memo(s.size(), -1);
        unordered_set<string> set(wordDict.begin(), wordDict.end());
        return wordBreak(s, set, memo, 0);
    }
```

Time Complexity : O(N^3), There are two nested loops, and substring computation at each iteration. Overall
that results in O(N^3) time complexity.
Space Complexity : O(N), Length of dp array is N+1.

using String + DP(Tabulation) + Hash Table.

```
bool wordBreak(string s, vector<string>& wordDict) {
        vector<bool> dp(s.size()+1, 0);
        dp[0] = true;
        unordered_set<string> set(wordDict.begin(), wordDict.end());
        for(int i=1; i<=s.size(); i++){
            for(int j=0; j<i; j++){
                if(dp[j] && set.count(s.substr(j, i-j))){
                    dp[i] = true;
                    break;
                }
            }
        }
        return dp[s.size()];
    }
```

----

15. Ones and Zeroes: You are given an array of binary strings strs and two integers m and n.
Return *the size of the largest subset of strs such that there are **at most*** m *0's and* n *1's in the subset*.
A set x is a **subset** of a set y if all elements of x are also elements of y.

Example 1:

Input: strs = ["10","0001","111001","1","0"], m = 5, n = 3

Output: 4

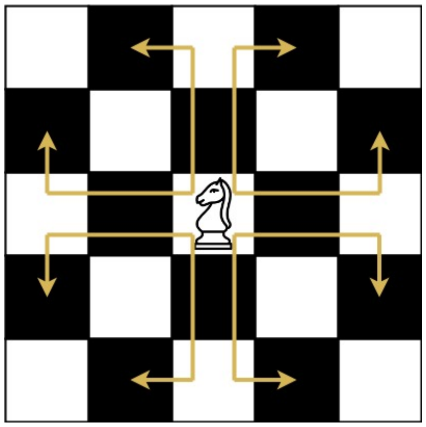
Explanation: The largest subset with at most 5 0's and 3 1's is {"10", "0001", "1", "0"}, so the answer is 4.
Other valid but smaller subsets include {"0001", "1"} and {"10", "1", "0"}.
{"111001"} is an invalid subset because it contains 4 1's, greater than the maximum of 3.
Example 2:
Input: strs = ["10","0","1"], m = 1, n = 1
Output: 2
Explanation: The largest subset is {"0", "1"}, so the answer is 2.

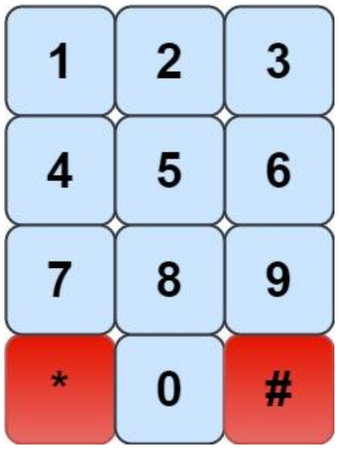
```
int findMaxForm(vector<string>& strs, int m, int n) {
    vector<vector<vector<int>>> memo(strs.size(), vector<vector<int>>(m + 1,
vector<int>(n + 1, -1)));
    return dp(strs, m, n, 0, memo);
  }

private:
  int dp(vector<string>& strs, int m, int n, int i, vector<vector<vector<int>>>& memo) {
    if (m == 0 && n == 0) return 0;
    if (i == strs.size()) return 0;
    if (memo[i][m][n] != -1) return memo[i][m][n];

    int ones = count(strs[i].begin(), strs[i].end(), '1');
    int zeros = strs[i].size() - ones;
    int take = 0;
    if (m >= zeros && n >= ones)
      take = 1 + dp(strs, m - zeros, n - ones, i + 1, memo);
    int dontTake = dp(strs, m, n, i + 1, memo);

    return memo[i][m][n] = max(take, dontTake);
  }
```
----
16. **Counting Bits**: Given an integer n, return *an array* ans *of length* n + 1 *such that for each* i (0 <= i <= n)*,* ans[i] *is the* ***number of*** 1***'s*** *in the binary representation of* i.

Example 1:
Input: n = 2
Output: [0,1,1]
Explanation:
0 --> 0
1 --> 1
2 --> 10
Example 2:
Input: n = 5
Output: [0,1,1,2,1,2]

Explanation:
0 --> 0
1 --> 1
2 --> 10
3 --> 11
4 --> 100
5 --> 101

```
vector <int> countBits(int num) {
      vector<int> res(num);
      res.push_back(0);  // for num = 0
      if (num == 0) return res;

      for (int i = 1; i <= num; i++) {
         if (i % 2 == 0)
            res[i] = res[i / 2];
         else
            res[i] = res[i - 1] + 1;
      }
      return res;
   }
```

Knight Dialer:
The chess knight has a **unique movement**, it may move two squares vertically and one square horizontally, or two squares horizontally and one square vertically (with both forming the shape of an **L**). The possible movements of chess knight are shown in this diagram:
A chess knight can move as indicated in the chess diagram below:

We have a chess knight and a phone pad as shown below, the knight **can only stand on a numeric cell** (i.e. blue cell).



Given an integer n, return how many distinct phone numbers of length n we can dial.
You are allowed to place the knight **on any numeric cell** initially and then you should perform n - 1 jumps to dial a number of length n. All jumps should be **valid** knight jumps.
As the answer may be very large, **return the answer modulo** $10^9 + 7$.

Example 1:
Input: n = 1
Output: 10
Explanation: We need to dial a number of length 1, so placing the knight over any numeric cell of the 10 cells is sufficient.
Example 2:
Input: n = 2
Output: 20
Explanation: All the valid number we can dial are [04, 06, 16, 18, 27, 29, 34, 38, 40, 43, 49, 60, 61, 67, 72, 76, 81, 83, 92, 94]
Example 3:
Input: n = 3131
Output: 136006598
Explanation: Please take care of the mod.

```
const int MOD = 1e9 + 7;
std::vector<std::vector<int>> dp(5001, std::vector<int>(10, -1));

class Solution {
public:
  int knightDialer(int n) {
    int res = 0;
    for (int i = 0; i < 10; i++) {
      res = (res + knight_iter(i, n, dp)) % MOD;
    }
    return res;
  }
  int knight_iter(int begin, int n, std::vector<std::vector<int>>& dp) {
    if (n == 1) return 1;
    if (dp[n][begin] != -1) return dp[n][begin];

    switch (begin) {
      case 1:
        return dp[n][begin] = (knight_iter(6, n - 1, dp) + knight_iter(8, n - 1, dp)) % MOD;
      case 2:
        return dp[n][begin] = (knight_iter(7, n - 1, dp) + knight_iter(9, n - 1, dp)) % MOD;
      case 3:
        return dp[n][begin] = (knight_iter(4, n - 1, dp) + knight_iter(8, n - 1, dp)) % MOD;
      case 4:
        return dp[n][begin] = ((knight_iter(0, n - 1, dp) + knight_iter(3, n - 1, dp)) % MOD +
knight_iter(9, n - 1, dp)) % MOD;
      case 6:
        return dp[n][begin] = ((knight_iter(0, n - 1, dp) + knight_iter(1, n - 1, dp)) % MOD +
knight_iter(7, n - 1, dp)) % MOD;
      case 7:
        return dp[n][begin] = (knight_iter(2, n - 1, dp) + knight_iter(6, n - 1, dp)) % MOD;
      case 8:
        return dp[n][begin] = (knight_iter(1, n - 1, dp) + knight_iter(3, n - 1, dp)) % MOD;
      case 9:
        return dp[n][begin] = (knight_iter(2, n - 1, dp) + knight_iter(4, n - 1, dp)) % MOD;
      case 0:
        return dp[n][begin] = (knight_iter(4, n - 1, dp) + knight_iter(6, n - 1, dp)) % MOD;
      default:
        return 0; // case 5
    }
  }
};
```
----

17. Maximize The Cut Segments:
Given an integer n denoting the Length of a line segment. You need to cut the line segment in such a way that the cut length of a line segment each time is either x , y or z. Here x, y, and z are integers.
After performing all the cut operations, your total number of cut segments must be maximum. Return the maximum number of cut segments possible.
Note: if no segment can be cut then return 0.
Examples:
Input: n = 4, x = 2, y = 1, z = 1
Output: 4
Explanation: Total length is 4, and the cut
lengths are 2, 1 and 1. We can make
maximum 4 segments each of length 1.

Input: n = 5, x = 5, y = 3, z = 2
Output: 2
Explanation: Here total length is 5, and
the cut lengths are 5, 3 and 2. We can
make two segments of lengths 3 and 2.
Expected Time Complexity : O(n)
Expected Auxiliary Space: O(n)

```
int solve(int n, int x, int y, int z, vector<int> &dp){
    if(n == 0) return 0;
    // if(n < 0) return INT_MIN;
    if(dp[n] != -1) return dp[n];
    int ans = INT_MIN;
    if(n-x >= 0) ans = max(ans, 1+solve(n-x,x,y,z,dp));
    if(n-y >= 0) ans = max(ans, 1+solve(n-y,x,y,z,dp));
    if(n-z >= 0) ans = max(ans, 1+solve(n-z,x,y,z,dp));

    return dp[n] = ans;
  }

  int maximizeTheCuts(int n, int x, int y, int z)
  {
    vector<int> dp(n+1,-1);
    int ans = solve(n, x, y, z, dp);

    if(ans < 0) return 0;
    return ans;
  }
```
---

18. **Decode Ways**: You have intercepted a secret message encoded as a string of numbers. The message is **decoded** via the following mapping:
"1" -> 'A'
"2" -> 'B'
...
"25" -> 'Y'
"26" -> 'Z'
However, while decoding the message, you realize that there are many different ways you can decode the message because some codes are contained in other codes ("2" and "5" vs "25").
For example, "11106" can be decoded into:

- "AAJF" with the grouping (1, 1, 10, 6)
- "KJF" with the grouping (11, 10, 6)
- The grouping (1, 11, 06) is invalid because "06" is not a valid code (only "6" is valid).

Note: there may be strings that are impossible to decode.

Given a string s containing only digits, return the **number of ways** to **decode** it. If the entire string cannot be decoded in any valid way, return 0.
The test cases are generated so that the answer fits in a **32-bit** integer.

Example 1:
Input: s = "12"
Output: 2
Explanation:
"12" could be decoded as "AB" (1 2) or "L" (12).
Example 2:
Input: s = "226"
Output: 3
Explanation:
"226" could be decoded as "BZ" (2 26), "VF" (22 6), or "BBF" (2 2 6).
Example 3:
Input: s = "06"
Output: 0
Explanation:
"06" cannot be mapped to "F" because of the leading zero ("6" is different from "06"). In this case, the string is not a valid encoding, so return 0.

```
class Solution {
public:
  int numDecodingsRecursive(string s, int index) {
    if (index == s.length()) {
      return 1; // If we reach the end of the string, one way to decode it.
    }

    if (s[index] == '0') {
      return 0; // If the current digit is '0', it cannot be decoded individually.
    }

    int ways = numDecodingsRecursive(s, index + 1); // Decode current digit individually

    if (index + 1 < s.length() && (s[index] == '1' || (s[index] == '2' && s[index + 1] <= '6'))) {
      ways += numDecodingsRecursive(s, index + 2); // Decode current and next digit together
    }

    return ways;
  }
  int numDecodings(string s) {
    return numDecodingsRecursive(s, 0);
  }
};
```
----

19) **Palindrome Partitioning**:

Given a string s, partition s such that every
substring
of the partition is a
palindrome
. Return *all possible palindrome partitioning of* s.

Example 1:
Input: s = "aab"
Output: [["a","a","b"],["aa","b"]]
Example 2:
Input: s = "a"
Output: [["a"]]

```
class Solution {
 public:
  vector < vector < string >> partition(string s) {
   vector < vector < string > > res;
   vector < string > path;
   partitionHelper(0, s, path, res);
   return res;
  }

 void partitionHelper(int index, string s, vector < string > & path,
  vector < vector < string > > & res) {
  if (index == s.size()) {
   res.push_back(path);
   return;
  }
  for (int i = index; i < s.size(); ++i) {
   if (isPalindrome(s, index, i)) {
    path.push_back(s.substr(index, i - index + 1));
    partitionHelper(i + 1, s, path, res);
    path.pop_back();
   }
  }
 }

 bool isPalindrome(string s, int start, int end) {
    while (start <= end) {
      if (s[start++] != s[end--])
        return false;
    }
    return true;
  }
};
int main() {
  string s = "aabb";
  Solution obj;
  vector < vector < string >> ans = obj.partition(s);
  int n = ans.size();
  cout << "The Palindromic partitions are :-" << endl;
  cout << " [ ";
  for (auto i: ans) {
    cout << "[ ";
    for (auto j: i) {
      cout << j << " ";
    }
    cout << "] ";
  }
  cout << "]";

  return 0;
}
```
Time Complexity: O((2^n) *k*(n/2))
Reason: O(2^n) to generate every substring and **O(n/2)** to check if the substring generated is a palindrome. O(k) is for inserting the palindromes in another data structure, where k is the average length of the palindrome list.
Space Complexity: O(k * x)
Reason: The space complexity can vary depending upon the length of the answer. k is the average length of the list of palindromes and if we have x such list of palindromes in our final answer. The depth of the recursion tree is n, so the auxiliary space required is equal to the O(n).

Maximum sub array: Kadane's algo:
/*
Given an integer array nums, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum.
A subarray is a contiguous part of an array.

Input: nums = [-2,1,-3,4,-1,2,1,-5,4]
Output: 6

Explanation: [4,-1,2,1] has the largest sum = 6.

Brute Force approach:
We will check the sum of every possible subarray and consider the maximum among them.
To get every possible subarray sum, we will be using three nested loops. The first loops(say i and j) will iterate over
every possible starting index and ending index of a subarray. Basically, in each iteration, the subarray range will be
from index i to index j. Using another loop we will get the sum of the elements of the subarray [i.....j]. Among all values
of the sum calculated, we will consider the maximum one.

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

int maxSubarraySum(int arr[], int n) {
  int maxi = INT_MIN; // maximum sum

  for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {
      // subarray = arr[i.....j]
      int sum = 0;

      //add all the elements of subarray:
      for (int k = i; k <= j; k++) {
        sum += arr[k];
      }

      maxi = max(maxi, sum);
    }
  }

  return maxi;
}

int main()
{
  int arr[] = { -2, 1, -3, 4, -1, 2, 1, -5, 4};
  int n = sizeof(arr) / sizeof(arr[0]);
  int maxSum = maxSubarraySum(arr, n);
  cout << "The maximum subarray sum is: " << maxSum << endl;
  return 0;
}
```

Time Complexity: O(N3), where N = size of the array.
Reason: We are using three nested loops, each running approximately N times.
Space Complexity: O(1) as we are not using any extra space.

DP Approach:

```
int maxSubArray(vector<int>& nums)
{
        int totalMaxSum = nums[0], currMaxSum = nums[0];
        for (int i = 1; i < nums.size(); i++)
        {
                currMaxSum = max(currMaxSum + nums[i], nums[i]);
                totalMaxSum = max(totalMaxSum, currMaxSum);
        }
        return totalMaxSum;
}
```

Time Complexity: O(N), where N = size of the array.
Space Complexity: O(1) as we are not using any extra space.