When the question is like asking to find all the **possible ways / find the best or worst** etc , we can go for recursion.

# Climbing Stairs

**Problem Statement:** 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.

**Brute force** : either we can take steps of 1 stair or 2 stairs , hence the recurrence relation will be very similar to the fibonnacci series.

Time Complexity : O(2^n)

**Best Approach :** Use DP

TC : O(n)

# Frog Jump

**Problem Statement:**

Given a number of stairs and a frog, the frog wants to climb from the 0th stair to the (N-1)th stair. At a time the frog can climb either one or two steps. A height[N] array is also given. Whenever the frog jumps from a stair i to stair j, the energy consumed in the jump is abs(height[i]- height[j]), where abs() means the absolute difference. We need to return the minimum energy that can be used by the frog to jump from stair 0 to stair N-1.

**Approach :**

We have two options: we can jump from the previous stone (or latter one) to the i-th stone, or we can jump from the stone two positions back (or two positions after ith ) to the i-th stone. We choose the option with the minimum cost, where cost is the absolute difference in height between the two stones.

TC : O (2^n)

**Best Approach :** Use DP

TC : O(n)

# Frog Jump with k Distances

This is a follow-up question to “Frog Jump” In the previous question, the frog was allowed to jump either one or two steps at a time. In this question, the frog is allowed to jump up to ‘K’ steps at a time. If K=4, the frog can jump 1,2,3, or 4 steps at every index.

**Brute Force :** This is basically nothing but the extension of the same logic to k , by a for loop.

The time complexity of the plain recursive solution without memoization for this problem is O(K^N), where N is the number of stones and K is the maximum jump distance.

**Best Approach :** Use DP

If we use dynamic programming instead of recursion, we can reduce the time complexity to O(N\*K) as well, but without the overhead of function calls and recursion.

# Maximum sum of non-adjacent elements

**Problem Statement:** Given an array of ‘N’ positive integers, we need to return the maximum sum of the subsequence such that no two elements of the subsequence are adjacent elements in the array.

**Brute force :** For the general case, we have two options: either include the last element and skip the second-to-last element (since we cannot include adjacent elements), or skip the last element and consider the maximum sum up to the second-to-last element. We take the maximum of these two options.

TC : This recursive solution has a time complexity of O(2^n)

**Best Approach :** Use DP

TC : O(n)

# House Robber

**Problem Statement :** A thief needs to rob money in a street. The houses in the street are arranged in a circular manner. Therefore the first and the last house are adjacent to each other. The security system in the street is such that if adjacent houses are robbed, the police will get notified.

Given an array of integers “arr” which represents money at each house, we need to return the maximum amount of money that the thief can rob without alerting the police.

**Approach :** Same as above.

For a circular street, the first and last house are adjacent, therefore one thing we know for sure is that the answer will not consider the first and last element simultaneously (as they are adjacent).

Make two reduced arrays – arr1(arr-last element) and arr2(arr-first element).

# Ninja’s Training

A Ninja has an ‘N’ Day training schedule. He has to perform one of these three activities (Running, Fighting Practice, or Learning New Moves) each day. There are merit points associated with performing an activity each day. The same activity can’t be performed on two consecutive days. We need to find the maximum merit points the ninja can attain in N Days.

We are given a 2D Array POINTS of size ‘N\*3’ which tells us the merit point of specific activity on that particular day. Our task is to calculate the maximum number of merit points that the ninja can earn.

**BruteForce Approach :** We need an additional information like what activity has been done on the previous day so that we can avoid doing that activity in the current day.

**TC :** The time complexity of the recursive solution of this problem without memoization is O(3^N), where N is the number of days.

In the recursive solution, we are considering all three activities on each day and making a recursive call for each valid activity on the next day. Since we have three options at each step and we do this for N days, the total number of function calls can be as much as 3^N.

**Better Approach** : Using DP

Time Complexity: O(N\*4\*3)

Reason: There are N\*4 states and for every state, we are running a for loop iterating three times.

# Grid Unique Paths

Given two values M and N, which represent a matrix[M][N]. We need to find the total unique paths from the top-left cell (matrix[0][0]) to the rightmost cell (matrix[M-1][N-1]).

**BruteForce Approach :** Recursive solution exploring the two options we have.

TC : O(2^(m\*n)) , This is because at each step, the recursive function makes two recursive calls, one for the cell to the right and one for the cell below. As a result, the number of function calls grows exponentially with the dimensions of the grid.

**Better Approach** : Using DP

TC : O(m x n)

# Grid Unique Paths With Obstacles

We are given an “N\*M” Maze. The maze contains some obstacles. A cell is ‘blockage’ in the maze if its value is -1. 0 represents non-blockage. There is no path possible through a blocked cell.

We need to count the total number of unique paths from the top-left corner of the maze to the bottom-right corner. At every cell, we can move either down or towards the right.

**BruteForce Approach :** Recursive solution as above and we just need to avoid the obstacles.

TC : O(2^(m\*n))

**Better Approach** : Using DP

TC : O(m x n)

# Minimum Path Sum In a Grid

We are given an “N\*M” matrix of integers. We need to find a path from the top-left corner to the bottom-right corner of the matrix, such that there is a minimum cost path that we select.

At every cell, we can move in only two directions: right and bottom. The cost of a path is given as the sum of values of cells of the given matrix.

**BruteForce Approach :** Recursive solution exploring the two options we have (right , down ) and finding the minimum from them.

TC : O(2^(m\*n))

**Better Approach** : Using DP

TC : O(m x n)

# Minimum path sum in Triangular Grid

We are given a Triangular matrix. We need to find the minimum path sum from the first row to the last row.

At every cell we can move in only two directions: either to the bottom cell (↓) or to the bottom-right cell(↘)

**BruteForce Approach :** Recursive solution exploring the two options we have ( down , diagonal ) and finding the minimum from them.

TC : There will be a total of 1 + 2 + 3 + 4 + 5 + …….+ n = ~ n^2 cells in the triangular grid and each cell will have 2 options for recursion

O(2^(n^2))

**Better Approach** : Using DP

TC : O(n^2)

# Minimum/Maximum Falling Path Sum

We are given an ‘M\*N’ matrix. We need to find the maximum path sum from any cell of the first row to any cell of the last row.

At every cell we can move in three directions: to the bottom cell (↓), to the bottom-right cell(↘), or to the bottom-left cell(↙).

**Brute Force :**

We have a top row and a bottom row, we will be writing a recursion in the direction of the first row to the last row.

We need to make N calls for the first row and then find the maximum / minimum of them.

Time Complexity : O(3^(m\*n)) as we have 3 options for each cell in the matrix (diagonal left , diagonal right , down )

Better Approach : Use Dp

TC : O(m \* n)

# Cherry Pickup

We are given an ‘N\*M’ matrix. Every cell of the matrix has some chocolates on it, mat[i][j] gives us the number of chocolates. We have two friends ‘Alice’ and ‘Bob’. initially, Alice is standing on the cell(0,0) and Bob is standing on the cell(0, M-1). Both of them can move only to the cells below them in these three directions: to the bottom cell (↓), to the bottom-right cell(↘), or to the bottom-left cell(↙).

When Alica and Bob visit a cell, they take all the chocolates from that cell with them. It can happen that they visit the same cell, in that case, the chocolates need to be considered only once.

They cannot go out of the boundary of the given matrix, we need to return the maximum number of chocolates that Bob and Alice can together collect.

**Brute force Approach :**

We need to make sure that they both move the grid simultaneously ( otherwise we need to make sure that path of first person is traced and and second person also , subtract the intersecting cells ) to handle the case of collision better.

Initially Alice and Bob are at the first row, and they always move to the row below them every time, so they will always be in the same row. Therefore two different variables i1 and i2, to describe their positions are redundant. We can just use single parameter i, which tells us in which row of the grid both of them are.

Now, we need to understand that we want to move Alice and Bob together. Both of them can individually move three moves but say Alice moves to bottom-left, then Bob can have three different moves for Alice’s move, and so on.

Hence we have a total of 9 different options at every cell. if (j1===j2), we will only consider chocolates collected by one of them otherwise we will consider chocolates collected by both of them.

**TC :** Alice has 3 options for every column and Bob also does have 3 .

O(3^ n x 3^n )

**Better Approach** : Using DP

TC : O(M x N x N ) x 9 since for each cell we have to handle 9 cases.

# Subset sum equal to target

Given an array of non-negative integers, and a value sum, determine if there is a subset of the given set with sum equal to given sum.

**Brute force Approach :** Same pick or not approach solved using recursion.

TC : O( 2^n ) , because every index has 2 choices of recursive calls.

**Better Approach :** Using DP

TC : O(n \* sum )

# Partition Equal Subset Sum

We are given an array ‘ARR’ with N positive integers. We need to find if we can partition the array into two subsets such that the sum of elements of each subset is equal to the other.

If we can partition, return true else return false

**Approach :** same as above we just need to check if there is sum/2 target in the array, another preliminary check should be to check if sum is odd or even , if odd it is not possible to split array into 2 equal halves.

# Partition Set Into 2 Subsets With Min Absolute Sum Diff

We are given an array ‘ARR’ with N positive integers. We need to partition the array into two subsets such that the absolute difference of the sum of elements of the subsets is minimum.

We need to return only the minimum absolute difference of the sum of elements of the two partitions.

**Approach1 :** Inorder to minimize the difference, we need to maximize the sum of subset close to half of the sum of the total array.

So , basically in the dp solution the n th row will have the possible range of values of sum from 0 to totalSum , we just need to start figuring out that which sum is possible from the totalSum/2 back to 0.

TC : O(n x sum) + O(sum /2 )

**Approach2 :** we have two choices for each element in A: we can either add it to the first subset (sum1), or we can add it to the second subset (sum2). We recursively compute the minimum difference between the two subsets for each of these choices, and return the minimum of the two.

But here we need to maintain 3 states , ie for sum1 , sum2 and ind.

# Count Subsets with Sum K

We are given an array ‘ARR’ with N positive integers and an integer K. We need to find the number of subsets whose sum is equal to K.

**Approach :** Same approach as subset sum equal to target , but here we should not be using the base case ***if(sum == 0 ) return true ;*** as this will not cover the cases like [1 , 0 ] , sum = 1 , we will be missing {1 ,0 } pair here

# Count Partitions with Given Difference

We are given an array ‘ARR’ with N positive integers and an integer D. We need to count the number of ways we can partition the given array into two subsets, S1 and S2 such that S1 – S2 = D and S1 is always greater than or equal to S2.

**Approach :** Same as above , S1 + S2 = totalSum , S1 – S2 = d , based on this we can find S1 = (totalSum + d )/2.

S1 , S2 should not be fractions therefore if (totalSum + d ) is odd we will return 0.

# 0/1 Knapsack

A thief wants to rob a store. He is carrying a bag of capacity W. The store has ‘n’ items. Its weight is given by the ‘wt’ array and its value by the ‘val’ array. He can either include an item in its knapsack or exclude it but can’t partially have it as a fraction. We need to find the maximum value of items that the thief can steal.

**Approach :** Same include and exclude approach

# Minimum Coins

We are given a target sum of ‘X’ and ‘N’ distinct numbers denoting the coin denominations. We need to tell the minimum number of coins required to reach the target sum. We can pick a coin denomination for any number of times we want.

Return the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return -1.

**Brute force Approach :** Unbounded knapsack , since we are returning the min , we should consider large value for odd case in base case.

TC : The time complexity of this solution is exponential since we are exploring all possible combinations of coins. Specifically, at each level of the recursion tree, we have two recursive calls: one where we use the current coin (which reduces the amount by the value of the coin), and one where we skip the current coin (which moves to the next coin in the vector). This leads to a branching factor of 2, and the maximum depth of the recursion tree is equal to the amount of money we are trying to make change for.

Therefore, the total number of nodes in the recursion tree is 2^depth, which is equal to **2^amount** in the worst case , when the coin is 1.

**Best Approach :** Using DP

TC : O(n x amount )

# Target Sum

We are given an array ‘ARR’ of size ‘N’ and a number ‘Target’. Our task is to build an expression from the given array where we can place a ‘+’ or ‘-’ sign in front of an integer. We want to place a sign in front of every integer of the array and get our required target. We need to count the number of ways in which we can achieve our required target.

**Approach :** We are given the difference of the two subsets of the array in otherwords , i.e S1 – S2 , we can also find the totalSum of the array which will be S1 + S2 , S1 will be (totalSum + target) /2 , since we are dealing with the integers (totalSum + target) should not be odd.

This is same as **Count Partitions with Given Difference** problem.

# Coin Change 2

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.

# Unbounded Knapsack

A thief wants to rob a store. He is carrying a bag of capacity W. The store has ‘n’ items of infinite supply. Its weight is given by the ‘wt’ array and its value by the ‘val’ array. He can either include an item in its knapsack or exclude it but can’t partially have it as a fraction. We need to find the maximum value of items that the thief can steal. He can take a single item any number of times he wants and put it in his knapsack.

Approach : Unbounded knapsack

# Rod Cutting Problem

We are given a rod of size ‘N’. It can be cut into pieces. Each length of a piece has a particular price given by the price array. Our task is to find the maximum revenue that can be generated by selling the rod after cutting( if required) into pieces.

**Approach :** Here we need to consider n itself as the length of the rod , and to follow 1 based indexing , we must subtract the rob with sizes of (ind + 1) , and it is just the same version of Unbounded knapsack.

# Longest Common Subsequence

A subsequence of a string is a list of characters of the string where some characters are deleted ( or not deleted at all) and they should be in the same order in the subsequence as in the original string.

Note: For a string of length n, the number of subsequences will be 2^n.

The longest Common Subsequence is defined for two strings. It is the common subsequence that has the greatest length.

**Bruteforce approach :** We need to check if the characters at current indices of both strings matching then definitely we will have a subsequence of length 1 more , we will start comparing strings from their next characters , if they do not match at a current index , we need to move explore the two choices by excluding each character once from both strings.

**Time Complexity :** At a given point of time we are either exploring 1 matching case or the two non matching cases of the characters , so we need to consider the dominant one i.e 2 non matching cases, here for this case the no of function calls depends upon maximum length of the string of the both , L O(2 ^ L ).

**Better Approach :** Use DP

**TC :** O(len1 x len2)

# Print Longest Common Subsequence

**Approach :** In the above problem ,after forming the dp array , dp[0][0] ---> will represent the length of lcs , so we shall start from there and build the same logic , and at any cell if the corresponding row number and column number characters matching , we shall add that char to our lcs , otherwise we shall take max of two options as in the recurrence relation and go to that cell.

# Longest Common Substring

A substring of a string is a subsequence in which all the characters are consecutive. Given two strings, we need to find the longest common substring.

**Brute force Approach :** we have to add an extra parameter count to the LCS function, which is the length of the current common substring. In the base case, if either m or n is zero, we return the current count. In the recursive case, if the last characters of s1 and s2 match, we increase the count by 1 and make a recursive call with the remaining substrings. Otherwise, we make two recursive calls, one with s1 and s2 without the last character of s1, and the other with s1 and s2 without the last character of s2. We take the maximum of the three values returned by the recursive calls and return it as the final result.

The point to be noted here is that , we need to pass the count varaible which we are returning as an argument to the recursive function call , that way we will be able to keep track of the consecutive length of matching strings so far.

This will have 3 states for dp , ind1 , ind2 , and count , hence not recommended to follow this approach .

**Best Approach :** Using 2D grid, note this is not a dp solutuion to the recursive code , rather we are just putting the strings across the row and column as for the consective ness we just need to check

If ind1 , ind2 matching ind1+1 , ind2+1 also should be matching.

And also the maximum length does not necessarily found in the dp[0][0] , This is because there is no restriction that the longest common substring is present at the end of both the strings.

So we maintain a max variable each time and return the max at the end.

# Longest Palindromic Subsequence

Given a string S, find the common palindromic sequence ( A sequence that does not need to be contiguous and is a palindrome), which is common in itself.

Input: S = “BEBEEED”

Output: 4

Explanation:

The longest common palindromic subsequence is “EEEE”, which has a length of 4

Approach 1 :

The idea is to check and compare the first and last characters of the string. There are only two possibilities for the same:

If the first and the last characters are the same, it can be ensured that both the characters can be considered in the final palindrome and hence, add 2 to the result, since we have found a sequence of length 2 and recurse the remaining substring S[i + 1, j – 1].

In case, the first and the last characters aren’t the same, the following operation must be performed:

Recurse S[i + 1, j]

Recurse S[ i, j – 1]

Find the maximum length obtained amongst them.

Approach 2 :

The longest palindromic subsequence of a string is the longest common subsequence of the given string and its reverse.

# Minimum insertions to make string palindrome

We are given a string, we need to find the minimum insertions that we can make in that string to make it a palindrome.

**Approach :**

Let us take **abcaa**

To minimize the insertions, we will first try to refrain from adding those characters again which are already making the given string palindrome. For the given example, “aaa”, “aba”,”aca”, any of these are themselves palindromic components of the string. We can take any of them( as all are of equal length) and keep them intact. (let’s say “aaa”).

Now, there are two characters(‘b’ and ‘c’) remaining which prevent the string from being a palindrome.If we can add them at proper places we should be able to get the palindrome

Like abca**cb**a

In order to minimize the insertions, we need to find the length of the longest palindromic component or in other words, the longest palindromic subsequence.

**Minimum Insertion required = n(length of the string) – length of longest palindromic subsequence.**

# Minimum Insertions/Deletions to Convert String

We are given two strings, str1 and str2. We are allowed the following operations:

Delete any number of characters from string str1.

Insert any number of characters in string str1.

We need to tell the minimum operations required to convert str1 to str2.

**Approach:**

We will first try to refrain from deleting those characters from str1 which are already present in str2. More extensively, we refrain from deleting those characters which are common and come in the same order.

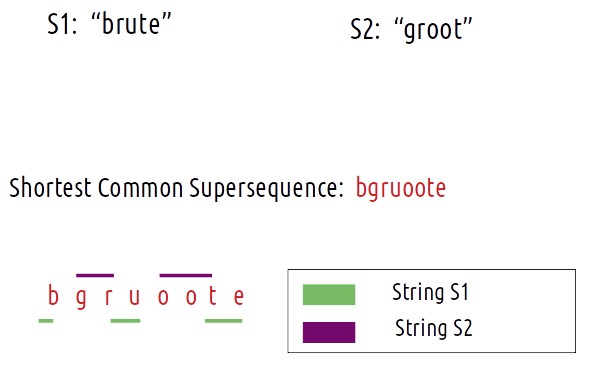
To minimize the operations, we would like to keep the maximum common characters coming in the same order intact. These maximum characters are the characters of the longest common subsequence.

We will first keep the longest common subsequence of str1 and str2 intact in str1 and delete all other characters from str1. This will take (len1-lcs) operations.

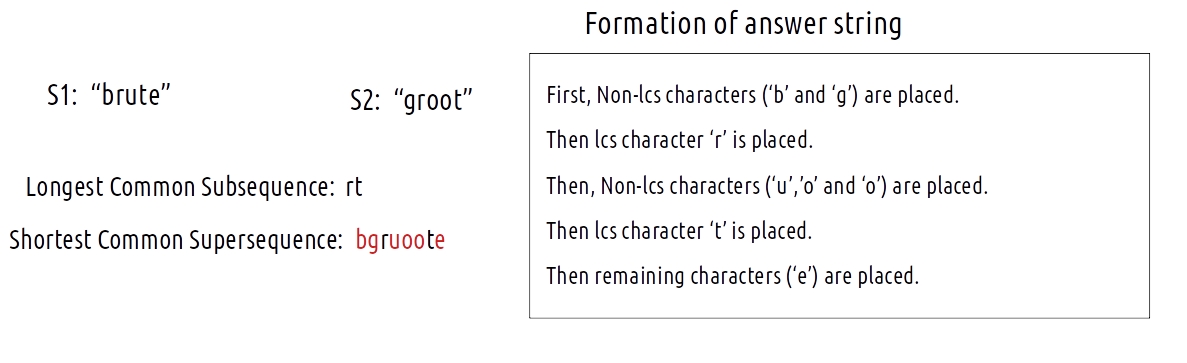
We will insert all the remaining characters of str2 to str1. This will take (len2-lcs) operations.

# Shortest Common Supersequence

We are given two strings ‘S1’ and ‘S2’. We need to return their shortest common supersequence. A supersequence is defined as the string which contains both the strings S1 and S2 as subsequences.



The characters of the longest common subsequence are written only once and other characters are placed around them. For every character that belongs to the longest common subsequence, the non-lcs characters coming before them in the strings S1 and S2 are placed before the lcs-character in the answer string. The below figure explains this:



**length of the shortest Common supersequence = len1 + len2 – lcs** , where (len1 and len2 are lengths of S1 and S2, and lcs is the length of the lcs string).

When we form the DP table to calculate the longest common subsequence (as done in Print Longest Common Subsequence) we have all the information of characters that are coming in the lcs string and characters that don’t. We use this same DP table to form the shortest common supersequence.

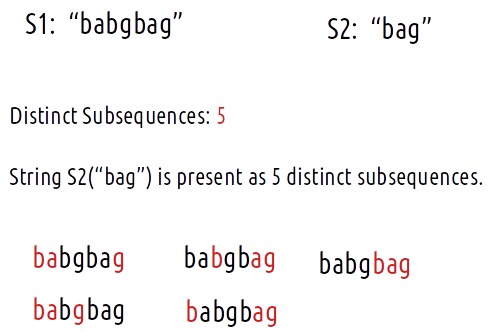
If(S1[ind1] == S2[ind2]), this means the character is an lcs character and needs to be included only once from both the strings, so we add it to the ans string .

If(S1[ind1] == S2[ind2]), this means that the character is a non-lcs character .This way non-lcs characters will be included separately in the right order.

After breaking, either ind1 == len1 or ind2 == len2 (only one condition will fail to break from the while loop), if(ind1 < len1 ) we push all the characters from S1 to ans string, else if(ind2 < len2 ), we push all the remaining characters from S2.

# Distinct Subsequences

We are given two strings S1 and S2, we want to know how many distinct subsequences of S2 are present in S1.



Approach :

**Case 1: When the characters match**

If S1[ind1] == S2[ind2], now as the characters at ind1 and ind2 match, we would want to check the possibility of the remaining characters of S2 in S1 therefore we increase the length of both the strings by 1 and call the function recursively.

If we only make the above single recursive call, we are rejecting the opportunities to find more than one subsequences because it can happen that the ind2 th character may match with more characters in S1

To explore all such possibilities, we make another recursive call in which we increase the length of the S1 string by 1 but keep the S2 string the same.

**Case 2: When the characters don’t match**

It means that we don’t have any other choice than to try the next character of S1 and match it with the current character S2.

**Base Cases**

We are increasing ind1 and ind2 in our recursive relation, there can be two possibilities, either ind1 becomes len1 or ind2 becomes len2.

if ind2== len2 , it means we have matched all characters of S2 with characters of S1, so we return 1.

if ind1 == len1, it means we have checked all characters of S1 but we are not able to match all characters of S2, therefore we return 0.

# Edit Distance

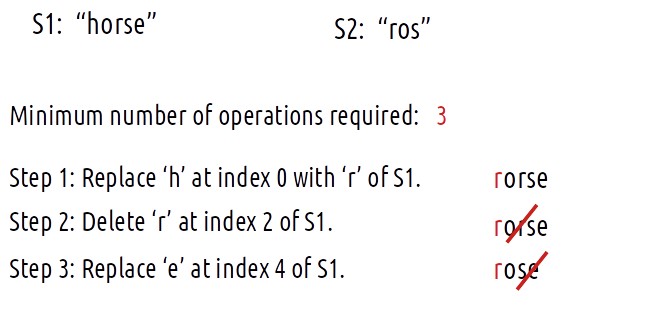
We are given two strings ‘S1’ and ‘S2’. We need to convert S1 to S2. The following three operations are allowed:

Deletion of a character.

Replacement of a character with another one.

Insertion of a character.

We have to return the minimum number of operations required to convert S1 to S2 as our answer.



**Approach :**

**(i) When the characters match**

if(S1[ind1]==S2[ind2]),

If this is true, now as the characters at ind1 and ind2 match, we would not want to do any operations to make them match, so we will just increment both ind1 and ind2 by 1 and recursively find the answer for the remaining string portion.

We return 0+f(ind1 +1 , ind2 + 1).

**(ii) When the characters don’t match**

if(S1[ind1] != S2[ind2] ) is true, then we have to do any of three operations to match the characters. We have three options, we will analyze each of them one by one.

Case 1: Inserting a character

If we insert whatever the character in S2[ind2] in S1 before ind1 , hypothetically they both match and we move ind2 to the next character

**h**orse

**r**ose

r**h**orse

r**o**se

Case 2 : Deleting a character

If we delete whatever the character S1[ind1] is , we would need to move to the next character after ind1

**h**orse

**r**ose

**o**rse

**r**ose

Case 3: Replacing a character

If we replace the character of S1[ind1] with S2[ind2] , We again hit the case of character matching , so again we would need to move indices both by 1 ahead

**h**orse

**r**ose

r**o**rse

r**o**se

finally , we would need to consider the minimum of all these 3.

**Base Cases:**

If ind1 == len1 , then we would have exhausted the string 1 and whatever the remaining characters in the string 2 needs to be inserted at the end of string1.

If ind2 == len2 , then we would have exhausted the string 2 , and whatever the remaining characters in the string 1 needs to be deleted from string1.

If ind1 == len1 and ind2 == len2 met then , it will be automatically taken care by returning 0.

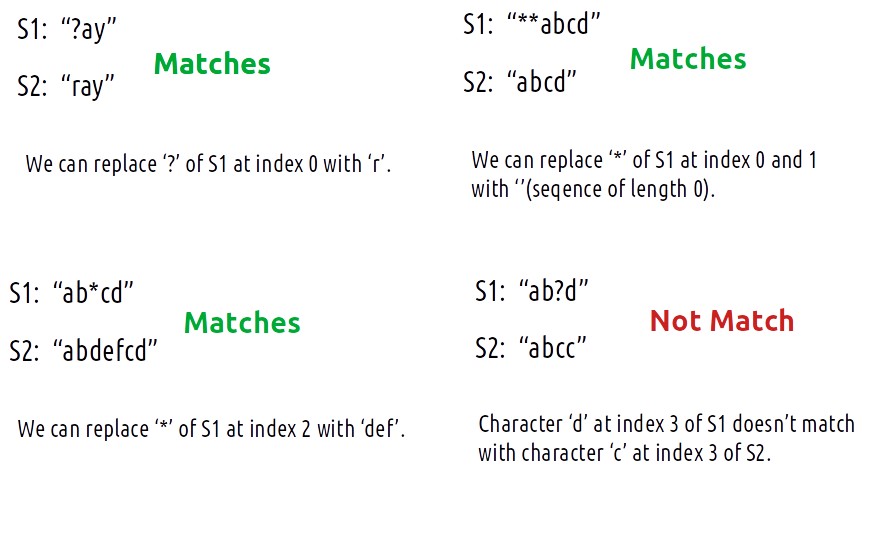
# Wildcard Matching

We are given two strings ‘S1’ and ‘S2’. String S1 can have the following two special characters:

‘?’ can be matched to a single character of S2.

‘\*’ can be matched to any sequence of characters of S2. (sequence can be of length zero or more).

We need to check whether strings S1 and S2 match or not.



**Approach :**

Let ind1 and ind2 represent two characters from strings S1 and S2 respectively. There are only two options that make sense: either the characters represented by ind1 and ind2 match or they don’t.

**(i) When the characters match**

if(S1[ind1]==S2[ind2]),

If this is true, the characters at i and j match, we can simply move to the next characters of both the strings.

So we will just increment both ind1 and ind2 by 1 and recursively find the answer for the remaining string portions.

**(ii) When the characters don’t match**

If the characters don’t match, there are three possible scenarios:

S1[ind1] == ‘?’

S1[ind1] == ‘\*’

S1[ind1] is some other character

**(i) If S1[ind1] == ‘?’**

we can explicitly match ‘?’ at index i of S1 with the corresponding character at index ind2 of S2. And then recursively call next portion of the string (ind1+1 , ind2+1)

**(ii) If S1[ind1] == ‘\*’**

This is an interesting case as now ‘\*’ can be replaced with any sequence of characters( of length 0 or more) from S2.

1. replace ‘\*’ with nothing and act as if it was not present. , i.e we can have ( ind1+1 , ind2 )

2. replace ‘\*’ with a single character at index ind2 and make the ind1 pointer to still point at index ind1.

(ind1 , ind2+1 ) In this, we matched it with a single character (one of the many options that need to be tried).

**(iii) If S1[ind1] is neither ‘?’ nor ‘\*’,** then we can say as the characters at ind1 and ind2 don’t match then the strings don’t match, so we return false.

**Base Cases:**

**(i) When S1 is exhausted:**

When S1 is exhausted (ind1 == len1 ), we know that in order for the strings to match, String S2 should also exhaust simultaneously. If it does, we return true, else we return false.

If(ind1 == len1 ) {

If(ind2 == len2 ) return true ;

else return false ;

}

**(ii) When S2 is exhausted:**

When S2 is exhausted (ind2 == len2) and S1 has not, there is only one pattern that can account for true(matching of strings). i.e: S1 contains only stars further from ind1. Then we can replace every star with a sequence of length 0 and say that the string match.

i.e If S1 is all-stars, we return true, else return false.

If( ind2 == len2 ){

If(isAllStars(ind1 , S1 ) ) return true ;

Else return false ;

}

# Stock Buy and Sell

We are given an array Arr[] of length n. It represents the price of a stock on ‘n’ days. The following guidelines need to be followed:

We can buy and sell a stock only once.

We can buy and sell the stock on any day but to sell the stock, we need to first buy it on the same or any previous day.

We need to tell the maximum profit one can get by buying and selling this stock.

**Brute Force:**

This problem can not be modelled to solve using the recursive approach.

We can simply use 2 loops and track of tranaction difference (If arr[j] > arr[i]) and maintain a variable maxPro to contain the max value among all transactions.

Time complexity: O(n^2)

Space Complexity: O (1)

**Optimal/Best solution:**

One way to approach this problem is by iterating over the array of stock prices and keeping track of the minimum price seen so far and the maximum profit that can be obtained by selling at the current price. To find the maximum profit, we can take the difference between the current price and the minimum price seen so far.

By iterating over the array and updating the minimum price and maximum profit accordingly, we can find the maximum profit that can be obtained from a single transaction.