DYNAMIC PROGRAMMING

Dynamic programming is a technique for solving a complex problem by breaking into a collection of simpler subproblems, solving each subproblem just once, and then storing their solutions to avoid repetitive computations. It is mainly an optimization over plain recursion. Wherever we see a recursive solution with repeated calls for the same inputs, we can optimize it using Dynamic Programming. This simple optimization reduces time complexities from exponential to polynomial.

For example:

Finding Fibonacci numbers-

- Recursion:

```
int fib(n){
    if(n<=1) return n;
    return fib(n-1)+fib(n-2);
}
```

Time Complexity: O(2^N)

- Dynamic Programming:

```
arr[0]=1;
arr[1]=1;
for(int i=2;i<=n;i++){
    arr[i]=arr[i-1]+arr[i-2];
}
return arr[n];
```

Time Complexity: O(N)

There are two approaches to dynamic programming:

1. Top-down approach
2. Bottom-up approach

Top-Down approach follows the memoization technique. Here memoization is equal to the sum of recursion and caching. Recursion means calling the function itself while caching means storing the intermediate results.

- Advantage- It solves the subproblems only when it is required.
- Disadvantage- It uses the recursion technique that occupies more memory in the call stack. Sometimes when the recursion is too deep, the stack overflow condition will occur.

```
int fib(n, vector<int>& dp){
        if(n<=1) return n;
        if(dp[n]!=-1) return dp[n];
        dp[n]= fib(n-1,dp)+fib(n-2,dp);
        Return dp[n];
    }
```

Bottom-up approach uses the tabulation technique to implement the dynamic programming approach. It solves the same kind of problems, but it removes the recursion. In this tabulation technique, we solve the problems and store the results in a matrix, so it is also known as the tabulation or table filling method.

- Advantage- Recursion is not involved; hence there is no stack overflow issue and no overhead of the recursive functions.
- Disadvantage- It solves all the subproblems.

```
arr[0]=1;
    arr[1]=1;
    for(int i=2;i<=n;i++){
        arr[i]=arr[i-1]+arr[i-2];
    }
    return arr[n];
```

We have been given a staircase of height 'N'. We can climb 1 or 2 steps at a time. Find the total number of ways to reach the top.

Input: 5

Output: 8

Approach:

The person can reach to ith stair from (i-1)th stair by climbing 1 step or (i-2)th stair by climbing two steps. i.e.

ways(i) = ways(i-1) + ways(i-2)

Also, if you notice, it matches with the states of Fibonacci numbers.

Brute Force - A simple method that is a direct recursive implementation.

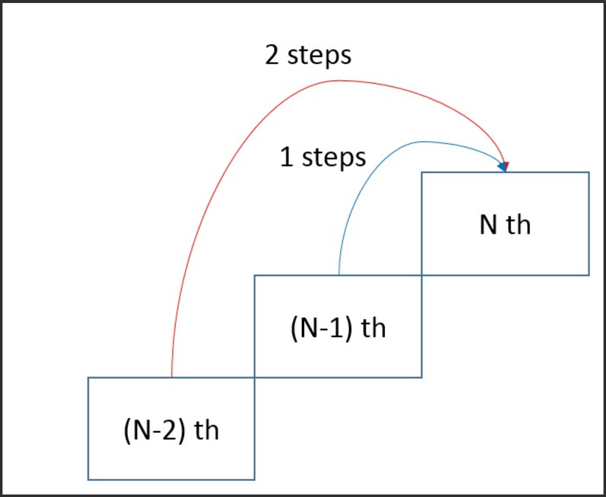
Time complexity: **O(2^N)**

Auxiliary Space: **O(1)**

Memoization - We can use the bottom-up approach of DP to solve this problem as well. We can create an array ways[] and initialize it with -1. Whenever we see that a subproblem is not solved, we can call the recursive method; else, we stop the recursion if the subproblem is solved already.

Time complexity: **O(N)**

Auxiliary Space: **O(N)**



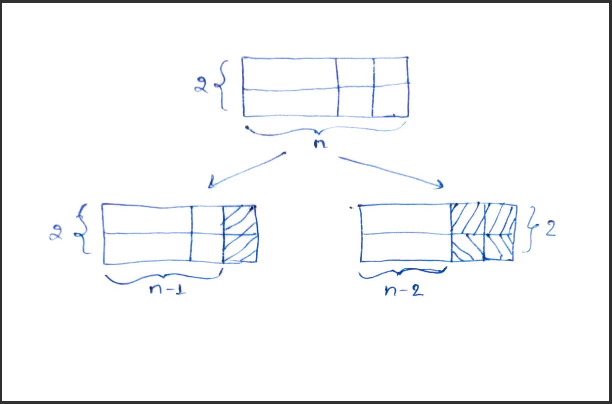
We have been given a lane of dimension ‘2x N’. We have to tile the lane with tiles of dimension 2x1. Find the total number of ways to tile the lane.

Input: 5

Output: 8

Approach:

Currently, we have a rectangle of dimension 2xN. If we put a tile vertically, we’ll be left with a rectangle of dimension 2x(N-1) and If we put two tiles horizontally, we’ll be left with a rectangle of dimension 2x(N-2).



ways(i) = ways(i-1) + ways(i-2)

Also, if you notice, it matches with the states of Fibonacci numbers.

Brute Force - A simple method that is a direct recursive implementation.

Time complexity: **O(2^N)**

Auxiliary Space: **O(1)**

Memoization - We can use the bottom-up approach of DP to solve this problem as well. For this, we can create an array ways[] and initialize it with -1. Whenever we see that a subproblem is not solved, we can call the recursive method; else, we stop the recursion if the subproblem is solved already.

Time complexity: **O(N)**

Auxiliary Space: **O(N)**

We have been given an array of integers. We have to find the maximum sum subsequence such that no two elements in the sequence are adjacent to each other in the array.

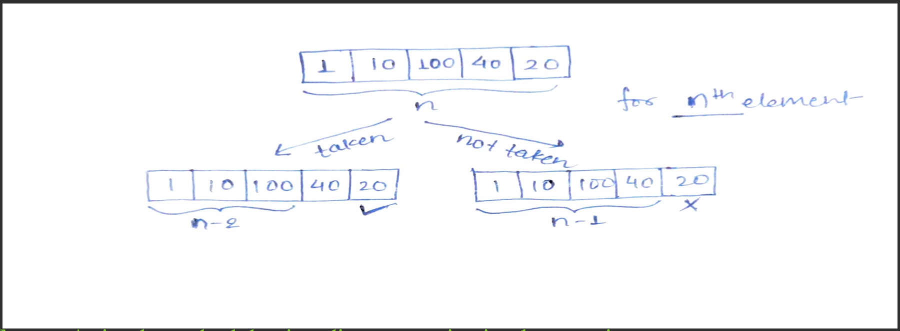
Input: [1, 10, 100, 40, 20]

Output: 121

Approach:

Every element in the array has two choices, either to be in the subsequence or to be left out. So the answer would be whatever of the two choices gives the maximum sum.

Maxsum[0,1,.........,n-1] = max (arr[0] + Maxsum[2,3,.....,n-1] , Maxsum[1,2,.....,n-1])



Brute Force - A simple method that is a direct recursive implementation.

Time complexity: **O(2^N)**

Auxiliary Space: **O(1)**

Memoization - We can use the Bottom-Up approach of DP to solve this problem as well.

We can create an array ans[] and initialize it with -1. If the subproblem is already solved, i.e., ans[i]!=-1, we'll stop the recursion by returning ans[i] else, we'll use the recursive approach i.e.

ans[i]= max(arr[i]+ maxsum(i+2,ans,arr), maxsum(i+1,ans,arr).

Time complexity: **O(N)**

Auxiliary Space: **O(N)**

Bottom-Top - We will create an array ans[] of size n. The state for ith index will be same here as well. i.e. **ans[i]=max(arr[i]+ans[i-2], ans[i-1])**. ans[i] is dependent on ans[i-1] ans ans[i-2] so we'll have to initialize ans[0] and ans[1]. Clearly, **ans[0]=max(0,arr[0])** and **ans[1]=max(ans[0], arr[1])**.

Time complexity: **O(N)**

Auxiliary Space: **O(N)**

Follow up: Can you come up with a solution that takes O(1) space?

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. We have to find the maximum revenue that can be generated by selling the rod after cutting(if required) into pieces.

Input: [2, 5, 9, 9, 10, 10, 11], N=7

Output: 20

Approach:

If the current length of the rod is ‘x’, then we have a choice of making a cut length of 1 to x. Suppose we make a cut of ‘y’ length, we can sell this y length for prices[y] and the remaining length is x-y. Again we have the choices to cut the remaining length. So we are generating all configurations of different pieces and finding the highest-priced configuration. The interesting thing to note is that If we cut the piece of length ‘y’, we can still cut another piece from the remaining length of size ‘y’ again.

rodcut(n) = max (prices[i] + rodCut(n – i)) where 1 <= i <= n

Brute Force - A simple method that is a direct recursive implementation.

Time complexity: O(N^N)

Auxiliary Space: O(1)

Memoization - We can use the Bottom-Up approach of DP to solve this problem as well.

We can create an array ans[] of size n+1 and initialize it with -1. If the subproblem is already solved, i.e., ans[i]!=-1, we’ll stop the recursion by returning ans[i] else, we’ll use the recursive approach i.e.

ans[i]=max(ans[i], prices[j-1]+maxval(i-j, ans, prices)) where 1<=j<=i

Time complexity: O(N^2)

Auxiliary Space: O(N)

Bottom-Top - We will create an array ans[] of size n+1. The state for ith index will be the same here as well. For every ith index, we will run a loop from 1 to i length and find out which cut gives us the maximum answer.

```
For i-> 1 to n
    For j-> 1 to i
        ans[i]=max of (ans[i], prices[j-1]+ans[i-j])
```

Time complexity: O(N^2)

Auxiliary Space: O(N)

We are given coins of several denominations and there is an infinite supply of each denomination. Find the minimum number of coins required to make an amount ‘N’.

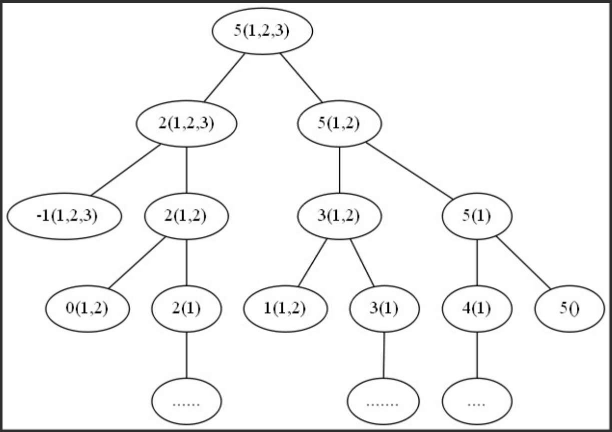
Input: [1, 5, 7] N=9

Output: 3

Approach:

If the current amount is ‘x’, then we have a choice of taking any denomination which is less than or equal to ‘x’. Suppose we take a coin of ‘y’ denomination, the count of taken coins increases by one and the remaining amount is x-y. Again we will have certain coins whose denomination will be less than the current amount and we’ll have the option of taking any. So we are generating all configurations of different coins and finding the configuration which takes the minimum number of coins.

ans(n) = min(1 + ans(n – coins[i])) where **0 <= i < size(coins)** and **coins[i]<=n**



Brute Force - A simple method that is a direct recursive implementation.

Time complexity: **O(N^N)**

Auxiliary Space: **O(1)**

Memoization- We can use the Bottom-Up approach of DP to solve this problem as well. We can create an array ans[] of size n+1 and initialize it with -1. If the subproblem is already solved, i.e., ans[i]!=-1, we’ll stop the recursion by returning ans[i] else, we’ll use the recursive approach i.e.

ans[i]=min(ans[i], 1+mincoin(i-coins[j], ans, coins)) where **0<=j< size(coins)**

Base condition:

```
1   if(curr_amount==0) return 0
2   if(curr_amount<0 ) return INT_MAX
```

Time complexity: **O(N^2)**

Auxiliary Space: **O(N)**

Bottom-Top - We will create an array ans[] of size n+1. The state for ith index will be the same here as well. For every ith index, we will run a loop from 1 to i length and find out which cut gives us the minimum answer.

```
      For i-> 1 to n

               For j-> 1 to size(coins]

                     ans[i]=min of (ans[i], 1+ans[i-coins[j])
```

Time complexity: **O(N^2)**

Auxiliary Space: **O(N)**

Coin Change - 2

We are given coins of several denominations, and there is an infinite supply of each denomination. Find the total number of ways to make an amount ‘N’.

Input: [1, 2, 3] N=4

Output: 4

Approach:

If the current amount is ‘x’, then we have a choice of taking any denomination which is less than or equal to ‘x’. We have the option of taking that coin in our configuration or not taking it. The summation of these possible two cases will give the total number of ways. Since the supply of each denomination is infinite, we can take it any number of times.

Brute Force - A simple method that is a direct recursive implementation.

 -> ways = func(rem_amount-coin[i], i) + func(rem_amount], i-1)

- Base condition:

 Case 1: If the rem_amount is 0, we return 1 as it is one of the possible configurations.
 Case 2: if(i<0) return 0; no coin denomination left, so not possible.
 Case 3: if(rem_amount<0) return 0; no way we can make a negative amount.

Time complexity: O(N^N)

Auxiliary Space: O(1)

Memoization- We can use the Bottom-Up approach of DP to solve this problem as well.

We can create an array matrix[size(coins)][N+1] and initialize it with -1. If the subproblem is already solved, i.e., ans[i][rem_amount]!=-1, we’ll stop the recursion by returning ans[i][rem_amount] else, we’ll use the recursive approach i.e.

Time complexity: O(N^2)

Auxiliary Space: O(N^2)

0/1 Bounded Knapsack - 1

We are given two arrays prices[n] and wt[n] which represent prices and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum profit such that the total weight of items is smaller than or equal to W. You cannot break an item, either pick the complete item or don't pick it (0-1 property).

Input: wt[]= {4,3,5}, prices[]={30,20,40}, W=7

Output: 50

Why a Greedy Solution doesn't work?

The first approach that comes to our mind is greedy. A greedy solution will fail in this problem because there is no 'uniformity' in data. While selecting a local better choice we may choose an item that will in long term give less value.

Let us understand this with help of an example:

N=3, W=7

wt[]= {4,3,5}

prices[]={30,20,40}

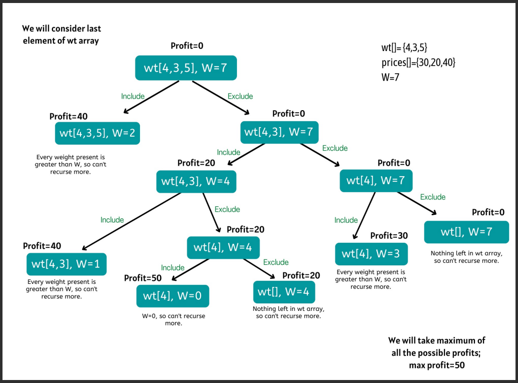
A Greedy solution will be to take the most valuable item first, so we will take an item on index 2, with a value of 40, and put it in the knapsack. Now the remaining capacity of the knapsack will be 2. Therefore we cannot add any other item. So a greedy solution gives us the answer 40.

Now we can clearly see that a non-greedy solution of taking the first two items will give us the value of 50 (30+20) in the given capacity of the knapsack.

Approach:

Every item has two choices, To be taken in the knapsack, And Not to be taken in the knapsack. Therefore, the maximum value that can be obtained from 'n' items is the max of the following two values.

Maximum value obtained by n-1 items and W weight (excluding nth item).
Value of nth item plus maximum value obtained by n-1 items and W minus the weight of the nth item (including nth item).
Also, note that If the weight of 'nth' item is greater than 'W', then the nth item cannot be included and Case 1 is the only possibility.


We will consider last element of wt array
Profit=0
wt[4,3,5], W=7
wt[]={4,3,5}
prices[]={30,20,40}
W=7
Include
Exclude
Profit=40
wt[4,3,5], W=2
Every weight present is greater than W, so can't recurse more.
Profit=0
wt[4,3], W=7
Include
Exclude
Profit=20
wt[4,3], W=4
Profit=0
wt[4], W=7
Include
Exclude
Profit=40
wt[4,3], W=1
Every weight present is greater than W, so can't recurse more.
Profit=20
wt[4], W=4
Include
Exclude
Profit=50
wt[4], W=0
W=0, so can't recurse more.
Profit=20
wt[], W=4
Nothing left in wt array, so can't recurse more.
Include
Exclude
Profit=30
wt[4], W=3
Every weight present is greater than W, so can't recurse more.
Profit=0
wt[], W=7
Nothing left in wt array, so can't recurse more.
We will take maximum of all the possible profits; max profit=50

Brute Force - A simple method that is a direct recursive implementation. We will have two parameters here. We are given 'n' items. So clearly one parameter will be index up to which the array items are being considered. The second parameter is 'W'. We need the capacity of the knapsack to decide whether we can pick an item or not in the knapsack.

maxprofit[n]=max(maxprofit(n-1,w) , prices[n] + maxprofit(n-1, w-wt[n]))

- Base condition:
  Case 1 : If w<0, that means there is any invalid pick so return -inf to avoid considering this case in answer.
  Case 2 : if(i<0 || w==0) return 0; this is a valid case.

Time complexity: **O(2^N)**

Auxiliary Space: **O(1)**

Memoization- We can use the Bottom-Up approach of DP to solve this problem as well.

We can create an array ans[N][W+1] and initialize it with -1. If the subproblem is already solved, i.e., ans[i][w]!=-1, we'll stop the recursion by returning ans[i][rem_amount] else, we'll use the recursive approach i.e.

Time complexity: **O(N*W)**

Auxiliary Space: **O(N*W)**

Bottom-Top - We will create an array ans[N+1][W+1]. States will remain the same here as well.

ans[i][j]= max(taken, not_taken)

Where, taken is if the current item is picked, so **taken=ans[i-1][j-wt[i-1]] + prices[i-1]**

And, not_taken is if the current item is not taken, so **not_taken= ans[i-1][j]**

Initializations:

1. For 0th column, if the weight (i.e. W) is 0, then the maximum profit will be 0. Hence, ans[0][i]=0, where 0<=i<=N
2. For 0th row, if there is no item, then the maximum profit will be 0. Hence, ans[i][0]=0, where 0<=i<=W

Time complexity: **O(N*W)**

Auxiliary Space: **O(N*W)**

Space optimization for Bottom-top:

If we closely look at the relation,

ans[i][j] = max(ans[i-1][j] ,ans[i-1][j-wt[i-1]]

We see that to calculate a value of a cell of the ansarray, we need only the previous row values. So, we don't need to store an entire array. Hence we can space optimize it.

We'll take two arrays here, ans[W+1] and temp[W+1]. Where ans array will store values for ith row and temp will store values of (i-1)th row. After calculating values for ith row, we'll assign its values to temp array, so that for calculation of (i+1)th row it can be considered as ith row.

Time complexity: **O(N*W)**

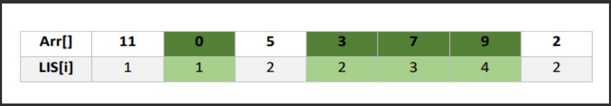
Auxiliary Space: **O(W)**

Longest Increasing Subsequence

We are given an array of integers. We have to find the length of the longest increasing subsequence.

Input: 11, 0, 5, 3, 7, 9, 2

Output: 4



Arr[]	11	0	5	3	7	9	2
LIS[i]	1	1	2	2	3	4	2

Brute Force - A simple method that is a direct recursive implementation. We will generate all the subsequences and whichever subsequence is longest and increasing will be the answer.

Time complexity: **O(2^N)**

Auxiliary Space: **O(1)**

Bottom-top - Suppose we have to find LIS ending up to index i, and we know LIS at every index which is smaller than i. We can iterate from 0 to i-1 and let's assume index varies in j. So if arr[j] is smaller than arr[i], we can say LIS up to i is equal to 1 + LIS up to j. We will take the max of all possible values of j and that will be LIS at index i.

We will take an array lis of size n, and initialize it with 1. Because at every index, the minimum length of an increasing subsequence is 1, i.e. that element itself.

```
For i-> 0 to n
        For j-> 0 to i
            if(arr[j]<arr[i]) lis[i]=max(lis[i], 1+ lis[j])
```

Time complexity: **O(N^2)**

Auxiliary Space: **O(N)**

We are given 2 strings of length n and m. Find the length of their longest common subsequence (LCS).

Input: s1= “aabcb”, s2= “abab”

Output: 3

Brute Force - A simple method that is a direct recursive implementation. Two strings S1 and S2 (suppose of same length n) are given, the simplest approach will be to generate all the subsequences and store them, then manually find out the longest common subsequence.

Time complexity: **O(2^N)**

Auxiliary Space: **O(1)**

Approach for DP:

A single variable can’t express both the strings at the same time, so we will use two variables ind1 and ind2. They mean that we are considering string S1 from index 0 to ind1 and string S2 from index 0 to S2.

f(ind1,ind2) represents the LCS of s1[0….ind1] and s2[0….ind2].

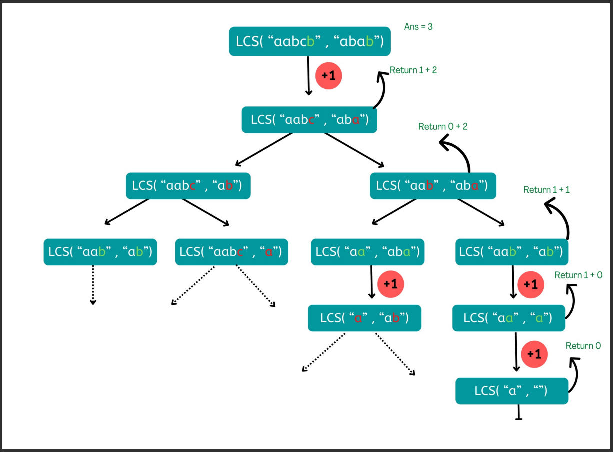
In the function f(ind1,ind2), ind1 and ind2 are representing two characters from strings S1 and S2 respectively. Now, there can be two possibilities:

1 **if(S1[ind1] == S2[ind2])**: In this case, this common element will represent a unit length common subsequence, so we can say that we have found one character and we can shrink both the strings by 1 to find the longest common subsequence in the remaining pair of strings.

2 **if(S1[ind1] != S2[ind2])**: In this case, we know that the current characters represented by ind1 and ind2 will be different. So, we need to compare the ind1 character with shrunk S2 and ind2 with shrunk S1. But how do we make this comparison? If we make a single recursive call as we did above to f(ind1-1,ind2-1), we may lose some characters of the subsequence. Therefore we make two recursive calls: one to f(ind1,ind2-1) (shrinking only S1) and one to f(ind1-1,ind2) (shrinking only S2). We will take the maximum of both choices.

Base case:

- If (ind1<0 || ind2<0) return 0.
  We will make a call to f(0-1,1), i.e f(-1,1) but a negative index simply means that there are no more indexes to be explored, so we simply return 0.



```
f(ind1, s1, ind2, s2){
        if(ind1<0 || ind2<0) return 0;
        if(s1[ind1]==s2[ind2]) return 1+f(ind1-1, s1, ind2-1, s2);
        return max(f(ind1-1, s1, ind2, s2), f(ind1, s1, ind2-1, s2));
    }
```

Memoization: We will create a ans matrix of size [N][M] where N and M are lengths of S1 and S2 respectively and initialize the matrix to -1. For findinf the answer of particular parameters (say f(ind1,ind2)), we first check whether the answer is already calculated using the ans matrix(i.e ans[ind][ind2]!= -1). If yes, simply return the value of ans[ind][ind2]. If not, then we are finding the answer for the given value for the first time, we will use the recursive relation as usual.

Time complexity: **O(N*M)**

Auxiliary Space: **O(N*M)**

We are given two strings w1 and w2, find the minimum number of operations required to convert w1 to w2. We are permitted to do the following three operations:

1 Insert a character
2 Delete a character
3 Replace a character

Input: w1= “folotbol”, w2= “football”

Output: 3

Approach:

We will process all characters one by one starting from either from left or right sides of both strings. Suppose the length of w1 and w2 is m and n respectively.

Let us traverse from the right corner, there are two possibilities for every pair of characters being traversed.

1 If the last characters of two strings are same, nothing much to do. Ignore the last characters and get the count for the remaining strings. So we recur for lengths m-1 and n-1.
2 If the last characters are not same, we consider all operations on ‘w1’, consider all three operations on the last character of first string, recursively compute the minimum cost for all three operations, and take minimum of three values. Each of the given operations would cause 1 unit.

Now, Let's talk about the operations that we have to perform on w1. Consider w1 = "abc" , w2 = "bcd" , m = 3 , n = 3.

1 So, for the Insert operation, we will insert a character from right side in w1 and after inserting character string will be "abcd" and the m which was pointing to c in s1 will be at the same position but the n which was pointing to d in w2 now will point to c in w2 i.e. m, n-1 for the rest of the function calls.
2 Now, for the Delete operation, after deleting a character from w1, the m will be m-1, but the n will be same i.e. m-1, n for the rest of the function calls.
3 Now, for the Replace operation, Ultimately, the character at corresponding positions in strings will be the same after replacing the character in w1 with the character in w2, Now, call for the rest of the string, so here m will be m-1 and n will be n-1, i.e. m-1, n-1 for the rest of the function calls.

Base conditions:

- if m = 0, we need to insert n characters from s2 in s1 to make s2.
- if n = 0, we need to delete m characters in s1 to make s2.

Brute Force - A simple method that is a direct recursive implementation.

```
if(m == 0) return n;

if(n == 0) return m;



if(w1[m-1] == w2[n-1])

    return editDistance(w1, w2, m-1, n-1);

else

   int insertChar = editDistance(w1, w2, m, n-1);

   int deleteChar = editDistance(w1, w2, m-1, n);

   int replaceChar = editDistance(w1, w2, m-1, n-1);
```

Time complexity: **O(min((3^N, 3^M))**

Auxiliary Space: **O(1)**

Memoization:

We will create a ans matrix of size [n][m] and initialize the matrix to -1. For finding the answer of particular parameters (say f(n,m)), we first check whether the answer is already calculated using the ans matrix(i.e ans[i][j]!= -1). If yes, simply return the value of ans[ind][ind2]. If not, then we are finding the answer for the given value for the first time, we will use the recursive relation as usual.

Time complexity: **O(N*M)**

Auxiliary Space: **O(N*M)**

Pattern Matching - 1

We are given two strings S and P. S consists of only lower case alphabets and P is a pattern that can include the characters '?' and '*' along with lower case alphabets. Where,

'?' – matches any single character

'*' – Matches any sequence of characters (including the empty sequence)

Find if both strings can match or not?

Input: S= "abbac", P= "ab*"

Output: True

Approach:

We will process all characters one by one from right to left. Suppose we have two strings S[0…n] and P[0…m]. We can have three different possibilities:

1. if(P[m]=='*') Here two cases arise:
   1. We can ignore '*' character and move to next character in the Pattern.
   2. '*' character matches with one or more characters. Here we will move to the next character in the string S.
2. if(P[m]=='?'): ignore current characters of both strings and check if P[0…m-1] matches S[0…n-1].
3. If the current character in the pattern is not a special character, it should match the current character in the input string.

Base conditions:

1. if(n==-1 && m==-1) i.e both the strings S and P reach their end, return true.
2. if(m==-1) i.e. only string P reaches its end, return false.
3. if(n==-1) i.e. only the string S reaches its end, return true only if the remaining characters in string P are all '*'.

Brute Force - A simple method that is a direct recursive implementation.

```
if (n < 0 && m < 0)

        return 1;

    if (m < 0)

        return 0;

    if (n < 0){

        while (m >= 0) {

            if (P[m] != '*')

                    return 0;

                m--;

            }

            return 1;

    }

    if (p[m] == '*'){

            return isvalid(s, p, n - 1, m) || isvalid(s, p, n, m - 1);

    }

    else if (p[m] == '?'){

            return isvalid(s, p, n - 1, m-1);

    }

    else{

            return P[m]==S[n] && isvalid(s, p, n - 1, m-1);

   }

Time complexity: O(2^N)

Auxiliary Space: O(1)
```