DYNAMIC PROGRAMMING(DP)

It is method used in Programming and Mathematics to Solve Complex Computational Problems Fastly, by breaking them down into smaller sub Problems .

By Breaking Down the Problems into smaller sub Problems , We store the result of these sub-Problems into specific Data Structures depending On Use - case of the Problem and We avoid Unnecessary and Redundant Computations and Retrieve the Already Computed Information From Data Structures and Use to Solve the Problems.

- It's Basically The Optimized Technique Over Recursion i.e.. In Some Problems while we solve them Their exists Multiple Overlapping Sub Problems , and we Compute Multiple Times. So for them if we compute Once It's enough and store the result in Data Structure and Retrieve Them When Needed. The Above process decreases The Computing (Time Complexity) and solve problem fastly.

- Dp Problems Involve Recursive patterns where solution is build up from the very base case to all the way up to the top.Therefors it's pretty intensive in trms of Time and space complexity . But There are few techniques which are used to optimize the solution so the algorithm takes less time and memory to execute. Most Of ones Them are:
  1) Memoization.
  2) Tabulation.

By using DP we can solve complex problems faster and some of the standard algorithms include:

Bellman-Ford algorithm for single-source shortest path, Floyd warshall algorithm for all pairs shortest algorithm, 0-1 knapsack Problem etc…

- Recursion:

  Lot of sub-problems are repetitive but solved again because of the nature of the recursive algorithms.

  For example 1) : Lets consider the fibonocci sequence 0, 1 , 1, 2 , 3, 5, 8 , 13 , 21 ,..etc... Find fib(n)??

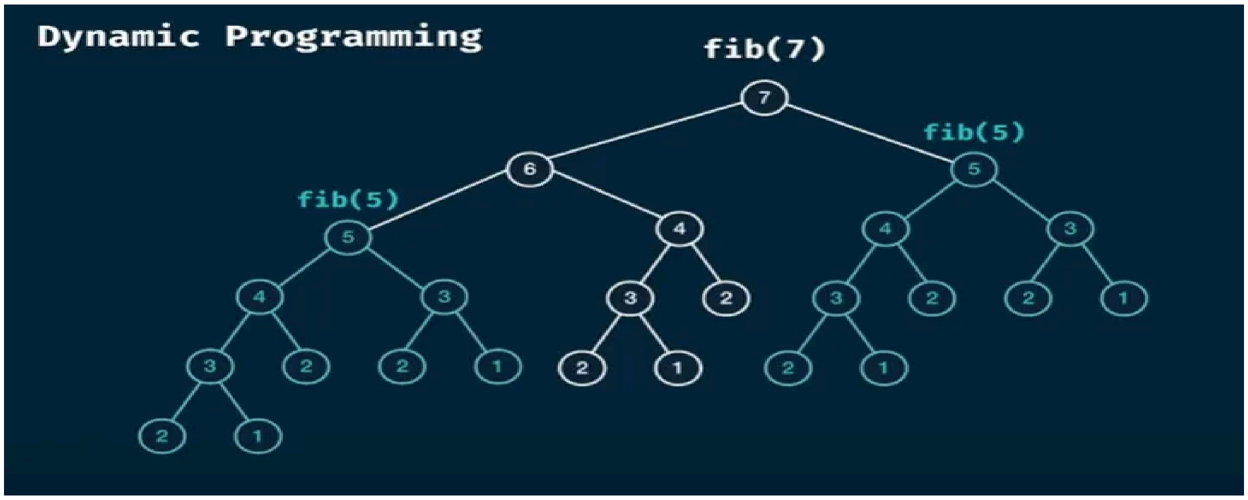
  Solution:

  We Know that Fib(n) = Fib(n-1) + Fib(n-2).
  Let us look at the code for solving above problem.

```
Fib(n){
         if(n<= 1) return n;

        return Fib(n-1) + Fib(n-2);

cout<<Fib(5)<<endl;
```



From above tree diagram fib(7) is calculated by Fib(6) + fib(5) and fib(6) by fib(4) and Fib(3) and this cycle goes on...

Time Complexity : O(2^n) as every call takes Two calls for each one.
Space complexity: O(n) for call stack.

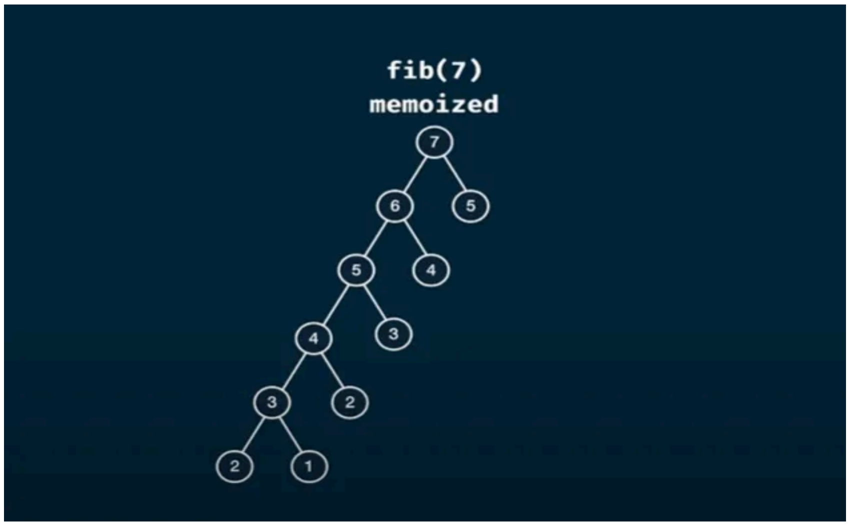
Memoization(Top-Down) : It is technique in which we store the results and use them when require instead of computing again to reduce time complexity.

```
#include<bits/stdc++.h>
Using namespace std;
unordered-map<int,int>memo;
int fib(n){
        if(memo.find(n) != memo.end()) return memo[n];

        if(n <= 1) return n;
        memo[n] = fib(n-1) + fib(n-2);

        return memo[n];
}
Int main(){
        int n = 7;

        cout<<fib(n)<<endl;
}
```



Here memo[2], memo[3], memo[4], memo[5]...are calculated once and stored for later use and it does faster.

Time complexity: O(n).
Space complexity : O(n).

TABULATION (Bottom-Up) (Iterative):

Using this technique, a dynamic programming problem is solved iteratively instead of recursion. Basically, after determining the relation between main problem and its sub-problems, we start from the very base case and solve up till the main problem. The results are stored in a data structure suitable to solve the problem.

Code:

```
int Fib(n){
        vector<int>vec(n+1,0);
        Fib[0] = 0;
        Fib[1] = 1;

        for(int i=2;i<=n;i++){
          Fib[i] = Fib[i-1] + Fib[i-2];
        } return Fib[n];
} int main(){
   cout<<Fib(7)<<endl;
}
```

Here we store the result in data structure, we store the result consecutively and return the result.

Here in Both Memoization and tabulation we store the results for further use and in memoization we use Recursion and in tabulation Iterative approach is used.

Example 2:

Given a set of non negative integers and a value sum, determine if their is a subset of given set whose sum is given sum.

Input: set[] = {3,34,4,12,5,2}
Sum = 9,
Output : True;

2) set[] = {3,34,4,12,5,2}; sum = 30;
Output : False;

M-1: Recursion:
For recursive approach we have two cases:

1) Consider last element and now hw have total sum= sum - last and total elements = total element - 1

2) Leave last element and total elements = total element - 1 and sum = sum

```
isSubset(set,n,sum)=
        isSubset(set,n-1,sum) ||
        isSubset(set,n-1,sum-set[n-1])
Basecase:
        isSubsetSum(set, n, sum) = false, if sum > 0 and n == 0
        isSubsetSum(set, n, sum) = true, if sum == 0
```

Tree simulation of Above Approach:

```
                    (4, 9)
              {True}
             /       \
       (3, 6)      (3, 9)

       /   \        /   \
    (2, 2)  (2, 6)   (2, 5)  (2, 9)
    {True}
    /   \
  (1, -3) (1, 2)
{False}  {True}
        /    \
      (0, 0)  (0, 2)
      {True} {False}
```

Recursive Code:

```
#include <iostream>
using namespace std;

bool isSubsetSum(int set[], int n, int sum)
{
     if (sum == 0)   // if sum is zero
        return true;
   if (n == 0)
```

```
        return false;

    // If last element is greater than sum,
    // then ignore it
    if (set[n - 1] > sum)
        return isSubsetSum(set, n - 1, sum);

       return isSubsetSum(set, n - 1, sum)
             || isSubsetSum(set, n - 1, sum - set[n - 1]);    // exclude last or
include last element
}

         int main()
         {
             int set[] = { 3, 34, 4, 12, 5, 2 };
             int sum = 9;
             int n = sizeof(set) / sizeof(set[0]);
             if (isSubsetSum(set, n, sum) == true)
                  cout <<"Found a subset with given sum";
             else
                 cout <<"No subset with given sum";
             return 0;
         }

         OUTPUT: Found a subset of Given sum
```

By recursion we calculate if subset is present or not in the set by eliminating one element each or adding it to subset.
Time Complexity: O(2^n).
Space Complexity : O(n)

By Method 2:

To solve the problem in Pseudo-polynomial time use the Dynamic programming.

So we will create a 2D array of size (arr.size() + 1) * (target + 1) of type boolean. The state DP[i][j] will be true if there exists a subset of elements from A[0....i] with sum value = 'j'. The approach for the problem is:

```
if (A[i-1] > j)
DP[i][j] = DP[i-1][j]
else
DP[i][j] = DP[i-1][j] OR DP[i-1][j-A[i-1]]
```

This means that if current element has value greater than 'current sum value' we will copy the answer for previous cases
And if the current sum value is greater than the 'ith' element we will see if any of previous states have already experienced the sum='j' OR any previous states experienced a value 'j - A[i]' which will solve our purpose.
The below simulation will clarify the above approach:
Ex: 1)
set[]={3, 4, 5, 2}
target=6

	0	1	2	3	4	5	6
0	T	F	F	F	F	F	F
3	T	F	F	T	F	F	F
4	T	F	F	T	T	F	F
5	T	F	F	T	T	T	F
2	T	F	T	T	T	T	T

Code:

```
bool isSubsetSum(int set[], int n, int sum)
{
    // The value of subset[i][j] will be true if
    // there is a subset of set[0..j-1] with sum
    // equal to i
    bool subset[n + 1][sum + 1];

    // If sum is 0, then answer is true
    for (int i = 0; i <= n; i++)
        subset[i][0] = true;

    // If sum is not 0 and set is empty,
    // then answer is false
    for (int i = 1; i <= sum; i++)
        subset[0][i] = false;

    // Fill the subset table in bottom up manner
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= sum; j++) {
            if (j < set[i - 1])
                subset[i][j] = subset[i - 1][j];
            if (j >= set[i - 1])
                subset[i][j] = subset[i - 1][j]
                               || subset[i - 1][j - set[i - 1]];
        }
    }

        for (int i = 0; i <= n; i++)
     {
       for (int j = 0; j <= sum; j++)
          printf ("%4d", subset[i][j]);
       cout <<"\n";
     }
    return subset[n][sum];
}


int main()
{
    int set[] = { 3, 34, 4, 12, 5, 2 };
    int sum = 9;
    int n = sizeof(set) / sizeof(set[0]);
    if (isSubsetSum(set, n, sum) == true)
        cout <<"Found a subset with given sum";
    else
        cout <<"No subset with given sum";
    return 0;
}
```

Output:

```
        Found a subset with given sum
```

Complexity Analysis:

Time Complexity: **O(sum*n),** where sum is the 'target sum' and 'n' is the size of array.

Auxiliary Space: **O(sum*n),** as the size of 2-D array is sum*n. + **O(n)** for recursive stack space

Approach 3):
Memoization Technique for finding Subset Sum:

In this method, we also follow the recursive approach but In this method, we use another 2-D matrix in we first initialize with -1 or any negative value.
In this method, we avoid the few of the recursive call which is repeated itself that's why we use 2-D matrix. In this matrix we store the value of the previous call value.

Code:

```
        int tab[2000][2000];

int subsetSum(int a[], int n, int sum)
{

    if (sum == 0)
        return 1;

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

If the value is not -1 it means it already call the function with the same value. it will save our from the repetition.

```
    if (tab[n - 1][sum] != -1)
        return tab[n - 1][sum];
```

if the value of a[n-1] is greater than the sum. we call for the next value

```
    if (a[n - 1] > sum)
        return tab[n - 1][sum] = subsetSum(a, n - 1, sum);
    else
    {
```

Here we do two calls because we don't know which value is full-fill our criteria that's why we doing two calls

```
        return tab[n - 1][sum] = subsetSum(a, n - 1, sum) ||
                       subsetSum(a, n - 1, sum - a[n - 1]);
    }
```

```
}

int main()
{
   // Storing the value -1 to the matrix
   memset(tab, -1, sizeof(tab));
   int n = 5;
   int a[] = {1, 5, 3, 7, 4};
   int sum = 12;

   if (subsetSum(a, n, sum))
   {
      cout << "YES" << endl;
   }
   else
      cout << "NO" << endl;

}
```

Output

```
YES
```

Complexity Analysis:

Time Complexity: O(sum*n), where sum is the 'target sum' and 'n' is the size of array.
Auxiliary Space: O(sum*n) + O(n) -> O(sum*n) = the size of 2-D array is sum*n and O(n)=auxiliary stack space.

QUIZ:

Attempt the QUIZ and check whether you have understood the Concept or Not

1) **Which of the standard algorithm is not dynamic programming based?**

   1) Bellman-Ford algorithm for single-source shortest path.
   2) Floyd warshall algorithm for all pairs shortest algorithm.
   3) 0-1 knapsack Problem.
   4) 4) Prim’s Minimum spanning tree.

Answer:

4) It is Greedy Algorithm , remaining all other are Dynamic programming based.

2) We use Dynamic Programming Approach When?

1) We need an Optimal solution.
2) The solution has optimal substructure.
3) The given problem can be reduced to easy one.
4) Faster than greedy.

Answer:

We use Dp when solution has an optimal substructure.

3) Which of the following is not an characteristic of DP?

1) Memoization which involves storing results of expensive function calls and reusing them.
2) Breaking a problem into smaller sub problems.
3) Solving problem in sequential problem
4) Dp can be used for the problems where solution has optimal substructure.

Answer:

Option 1) In Dp we breakdown the problem into smaller subproblems to solve it if we see the problem is solved here itself return saved answer else if it hasn;t been solved solve it and save the return answerThis is usually easy to think and very intuitive and it is Memoization.

2) It is big hint for solving DP problems where we breakdown problems into smaller problems and solve them.

3) Dp can never be solve dby sequential process.

4) Dp can be used for the problems where it has optimal substructure.

4) what is memoization in context of DP?

1) A Technique to write memory efficient program.
2) A way to avoid to solving subproblems and storing results and reusing.
3) A process of converting recursive algorithms to iterative
4) A method of analyzing the time complexity of algorithms.

Explanation: Memoization is fundamental technique in DP where e store the computed result in data structure and next we don't compute result instead we use the same result and decrease execution time.

5) Consider a sequence a f00 defined as:
Foo (0) = 1
Foo(1) = 1
Foo(n) = 10*Foo(n-1)+100

Foo(n-2) for n>=2
Then what shall be the set of values for the sequence F00?

1) **1,110,1200** 2) **1,110,600,1200**
3) **1,2,55,66,77** 4) **1,55,23,445**

Answer:
Foo(0) = 1 and Foo(1) = 1
Foo(n) = 10*Foo(n-1)+100
Foo(2) = 10*Foo(1) + 100
F(2) = 110

Similarly
Foo(3) = F(2)*10+100
Foo(3) = 1200

The sequence will be 1,110,1200.
So option 1 is correct.

Here is The video Link to Understand Basics About Dynamic Programming (Dp):
Dynamic Programming | Introduction

Reference Link to understand More About the Concept :
Dynamic Programming or DP - GeeksforGeeks