Dynamic Programming

Dynamic Programming is the recursion optimizing technique. When recursion calls for the same value again and again the DP is implemented in order to optimize the code so that it can run only for a single time which result in the reducing the time taken by the program.

Two approaches of DP: -

1. Top-Down approach → Recursion + Memorization
2. Bottom-Up approach → Tabulation

Note: - After applying any one approach to your problem remember to optimize space complexity so that your code will be more efficient.

Recursion: - It is a mechenism in which a function calls itself directly or indirectly.

Memorization: - Remember the output of a sub problems so that there is not a single chance of recursion call with same parameters.

Example program of Fibonacci series: -

1. **Using recursion: -**

```
#include<iostream>

using namespace std;
int fib(int n){
    if(n==0){
        return 0;
    }
    if(n==1){
        return 1;
    }
    return fib(n-1)+fib(n-2);
}
int main()
```

```
15.{
16.    int n;
17.    cout<<"Enter value of n"<<endl;
18.    cin>>n;
19.    int ans=fib(n);
20.    cout<< ans;
21.
22.    return 0;
23.}
```

Using Top-Down approach (Recursion + Memorization)

In this approach a dp vector is used in order to store the values of the previous answers so that recursion does not called again and again.

Example: -

```
#include<iostream>

#include<vector>

using namespace std;
int fib(int n,vector<int> dp){
    // base case
    if(n==0){
        return 0;
    }
    if(n==1){
        return 1;
    }
    // check if ans is already available in dp or not
wh
        return dp[n];
    }
    // recursion call and save the answer to dp array
    dp[n]=fib(n-1,dp)+fib(n-2,dp);
    // return the resultion answer
    return dp[n];
}
int main()
{
    int n;
    cout<<"Enter value of n"<<endl;
    cin>>n;
```

```
    // step 1: -  create a data structure to memorise the answer of sub
process
    vector<int> dp(n+1);
    // Initializing this vector by -1
    for(int i=0;i<=n;i++){
        dp[i]=-1;
    }

    int ans=fib(n,dp);
    cout<< ans;

    return 0;
}
```

Using Bottom-Up approach (Tabulation approach)

we have to optimize the solution of the top down approach so that it can be run as well as compile more efficiently.

In top-down approach we use an array or vector and operate with its last index but in bottom-up approach we start operating the memory block with its first free block/index.

In tabulation the array or vector is initialized inside the recursive function.

Example: -

```
#include<iostream>
#include<vector>
using namespace std;
int fib(int n){
//Creating the vector
    vector<int> dp(n+1);
    // base case
    dp[0]=0;
    dp[1]=1;
//calculating the nth Fibonacci series
    for(int i=2;i<=n;i++){
        dp[i]=dp[i-1]+dp[i-2];
    }

    return dp[n];
}
```

```
int main()
{
    int n;
    cout<<"Enter value of n"<<endl;
    cin>>n;
    int ans=fib(n);
    cout<< ans;

    return 0;
}
```

Using only limited variables (Resulting or space optimised solution)

```
#include<iostream>

using namespace std;
int fib(int n){
    int prev1=1,prev2=0;
    for(int i=0;i<n;i++){
        int curr=prev1+prev2;
        prev2=prev1;
        prev1=curr;
    }
    return prev2;
}
int main()
{
    int n;
    cout<<"Enter value of n"<<endl;
    cin>>n;
    int ans=fib(n);
    cout<< ans;

    return 0;
}
```