Dynamic Programming

"Those who forget the past are condemned to repeat it."

Approaches:

- Top Down: Recursion + Memoization
- Bottom Up (Tabulation)

Understand patterns of DP! [Overlapping Subproblems]

Motivation: Compute the fibonacci number. f(n) = f(n – 1) + f(n – 2)

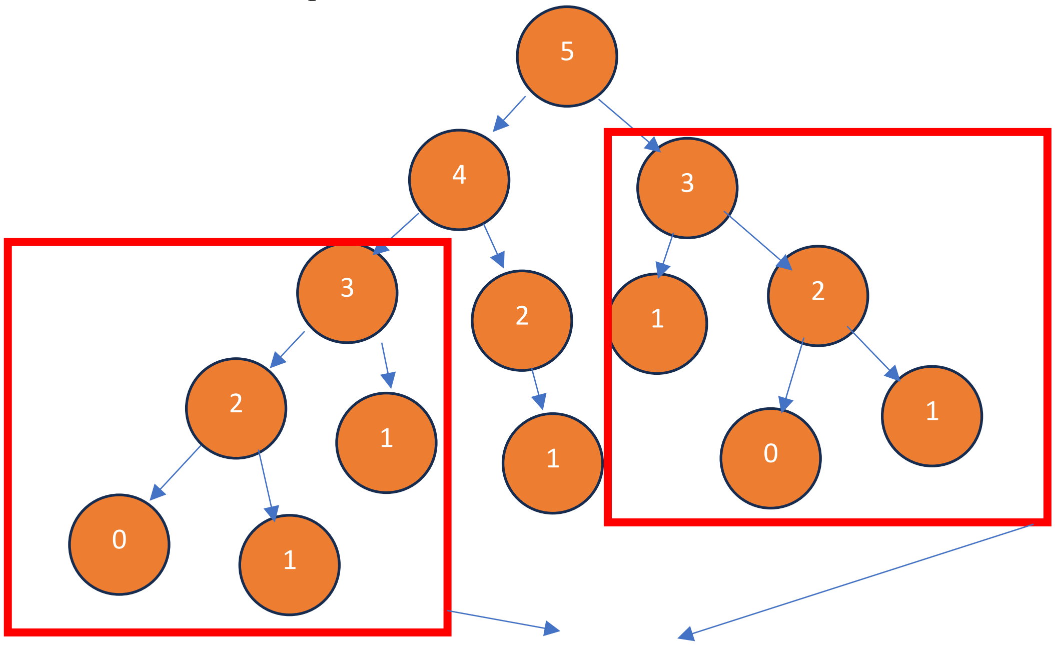
Consider this:

```
int fib(int n) {

        if( n == 1 || n == 0)
        {
                return n;
        }

        return fib(n - 1) + fib(n - 2);
}
```

Observe the State Space Tree: n = 5



Computed twice, waste of time!

Recursion and DP go very well together to remember whatever we've already computed. Any sort of thing being recomputed can be avoid using concepts of Dynamic Programming.

Observe this memoization abiding code for computing fibonacci:

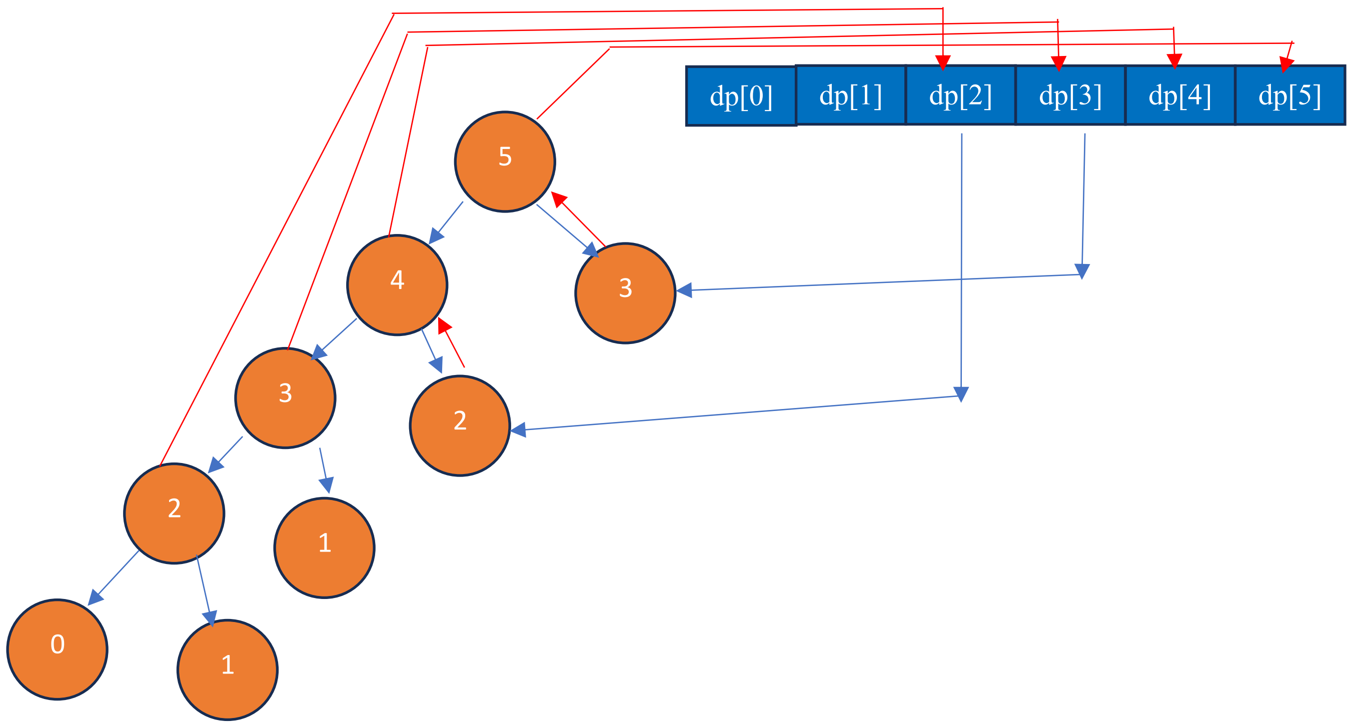
```
int fib(int n)
{
    vector<int> dp(n + 1, -1);
    return fib_helper(n, dp);
}

int fib_helper(int n, vector<int>& dp)
{
    // base case
    if(n <= 1)
        return n;

    if(dp[n] != -1)
    {
        return dp[n];
    }

    return dp[n] = fib_helper(n - 1, dp) + fib_helper(n - 2, dp);
}
```

State Space Tree (*n* = 5)

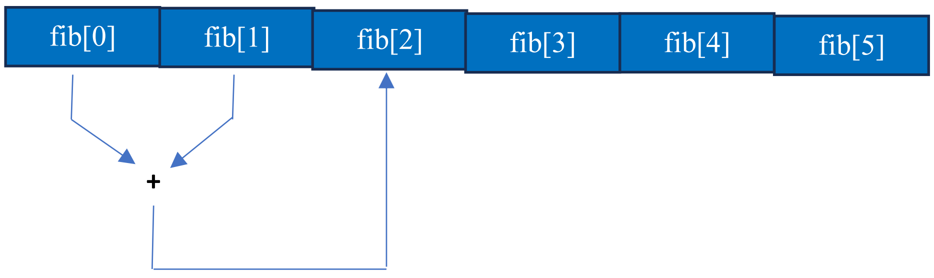


See how remembering the results we compute into a globalized array (respective to the recursive calls), helps us avoid recomputing values that have already been solved earlier. This is drastically faster than above code.

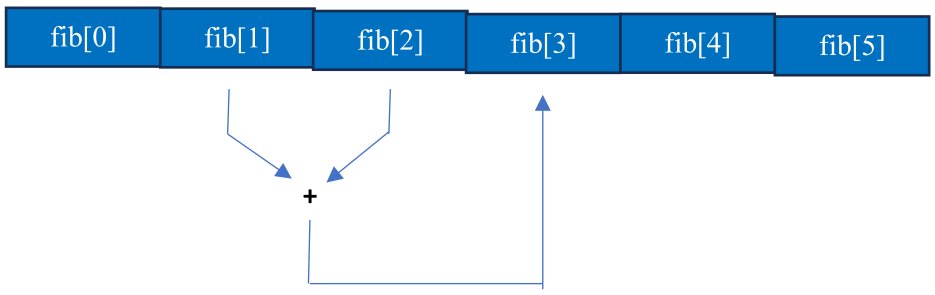
Next up, if we utilize our reasoning further, we can solve this from the bottom to top. ***From the base case up to required case.*** Such implementation gives way to iterative solutions and further reduce space complexity of call stack (involved with Recursive Memoization).

See, how the fibonacci can be modelled for '**tabulation**':

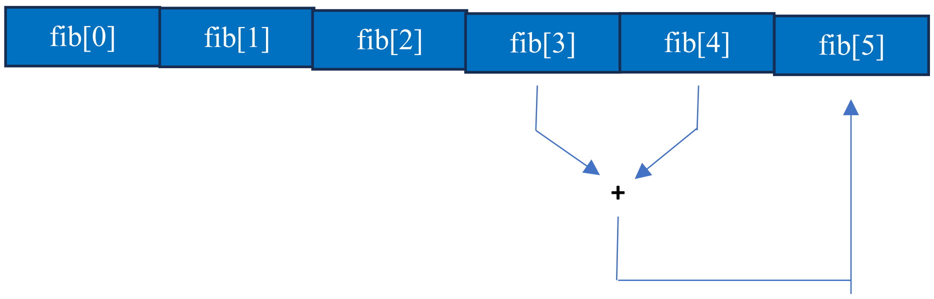
Step one: Find fib(2). Then find fib(3), then fib(4) and then finally we can resolve fib(5).



Step 2:



Step 4:



Fib(5) will contain the answer. Implementation in C++:

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

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

    return fibonacci[n];
}
```

Now, dynamic programming paradigm also says that most of these iterative solutions can be **further space optimized** if we apply more reasoning, producing the **most optimal solution possible within the DP paradigm.**

Notice that when we have computed *fib*[2], we no longer require the value of *fib*[0]. When we have computed the value of *fib*[3], we no longer require the value of *fib*[1]. Thus you see that at any step say *i*, you only need to know *fib*[*i* - 1] and *fib*[*i* - 2] if we build the solution bottom up using tabulation. Thus actually, all this can be done using just two variables! That's right. A time complexity of O(*n*) and Space Complexity of O(1).

```
int fib(int n)
{
    int previous_2 = 0;     // to mark fib[i - 2]  or here, fib[0]
    int previous_1 = 1;     // to mark fib[i - 1]  or here, fib[1]

    for(int i = 2; i <= n; i++)
    {
        int current = previous_1 + previous_2;
        previous_2 = previous_1;   // adjust for next computation i.e., fib[i + 1]
        previous_1 = current;      // adjust for next computation i.e., fib[i + 2]
    }

    return previous_1;
}
```

This is how we use Dynamic Programming.

The best you can achieve using Dyanmic Programming alone in this case is O(*n*) time.

But it's worth noting that the least time implementation of fibonacci is in *O(log(n))* time! That however involves the use of one more topic called "*Binary Exponentiation of Matrices*". While we're at computing fibonacci, let's also discuss that approach.

Keep the last solution we discussed in mind and also consider this matrix.

Matrix = $\begin{pmatrix} F(2) & F(1) \\ F(1) & F(0) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$

Now, compute the square of this matrix i.e., multiply it by itself.

Matrix = $\begin{pmatrix} F(2) & F(1) \\ F(1) & F(0) \end{pmatrix} * \begin{pmatrix} F(2) & F(1) \\ F(1) & F(0) \end{pmatrix}$

Matrix = $\begin{pmatrix} F(2) * F(2) + F(1) * F(1) & F(2) * F(1) + F(1) * F(0) \\ F(1) * F(2) + F(0) * F(1) & F(1) * F(1) + F(0) * F(0) \end{pmatrix}$

Matrix = $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} * \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$

Matrix = $\begin{pmatrix} 3 & 2 \\ 2 & 1 \end{pmatrix}$

Which is essentially, $\begin{pmatrix} F(3) & F(2) \\ F(2) & F(1) \end{pmatrix}$

Thus, it is easy to prove that $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F(n+1) & F(n) \\ F(n) & F(n-1) \end{pmatrix}$

And we know that we can compute a^n in log(n) time using binary exponentiation.

Binary Exponentiation:

We usually compute a^n like this in O(n) time: [assuming, $n > 0$ and $a > 0$]

```
int pow(int a, int n)
{
    if(n == 0)
    {
        return 1;
    }

    return a * pow(a, n - 1);
}
```

Now see this case,

We are to compute 3^8.

$3^8 = (3*3)^4 = 9^4$

$9^4 = (9*9)^2 = 81^2$

81^2 = ? is the answer

Instead of using 8 steps, we calculated it in 3 steps only! $\log_2(8) = 3$ *i.e., O(log(n))* time!

If n is odd at any point, it becomes $a^n = a*(a*a)^{n/2}$

Thus, we have

$$\begin{aligned} a^n &= (a*a)^{n/2} \quad \text{if } n \text{ is even} \\ &= a*(a*a)^{n/2} \quad \text{if } n \text{ is odd} \end{aligned}$$

At any recursive step.

So the C++ implementation would look like this:

```cpp
struct matrix
{
    long long mat[2][2];
    matrix friend operator *(const matrix &a, const matrix &b){
        matrix c;
        for (int i = 0; i < 2; i++)
        {
          for (int j = 0; j < 2; j++)
          {
              c.mat[i][j] = 0;
              for (int k = 0; k < 2; k++)
              {
                  c.mat[i][j] += a.mat[i][k] * b.mat[k][j];
              }
          }
        }
        return c;
    }
};

matrix matpow(matrix base, long long n)
{
    matrix ans{ {
      {1, 0},
      {0, 1}
    } };

    while (n)
    {
        if(n&1)
            ans = ans*base;
        base = base*base;
        n >>= 1;
    }
    return ans;
}


long long fib(int n)
{
    matrix base{ {
      {1, 1},
      {1, 0}
    } };
    return matpow(base, n).mat[0][1];
}
```

This is the most optimal solution created for fibonacci computation.