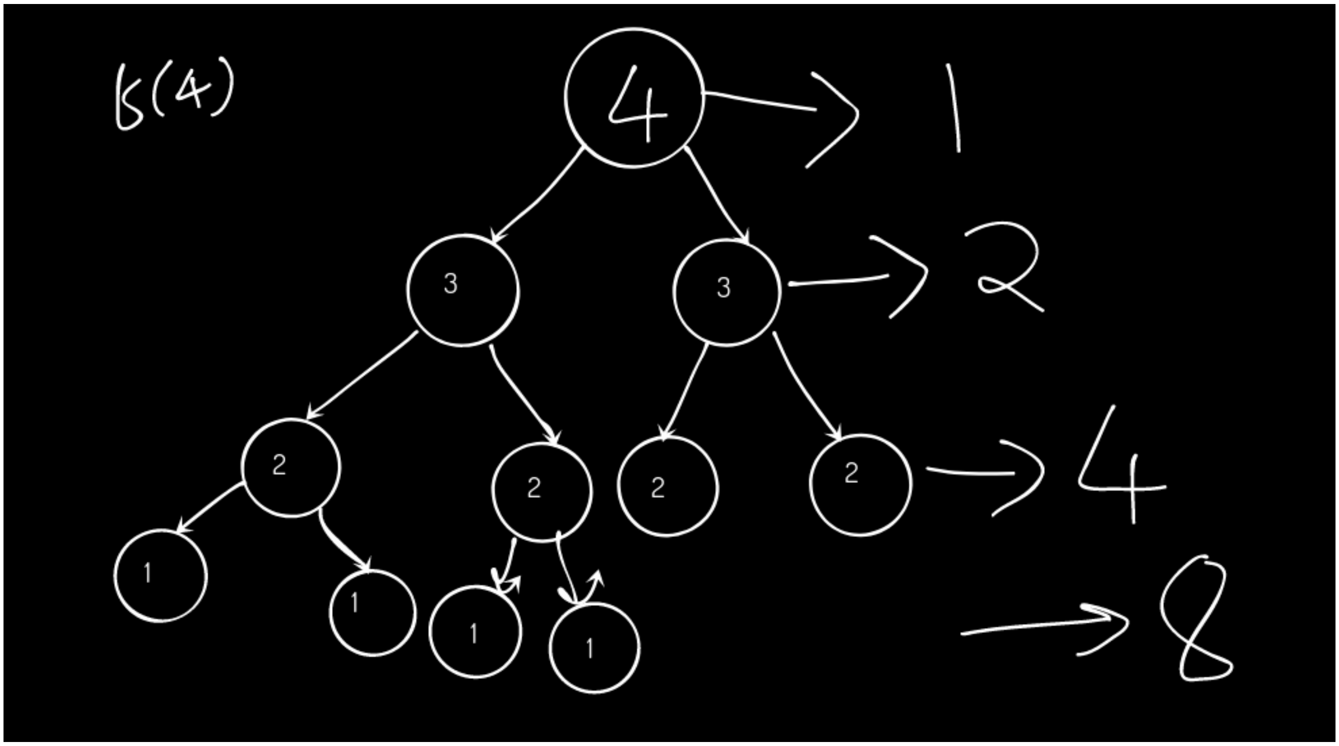
Introduction to Dynamic Programming

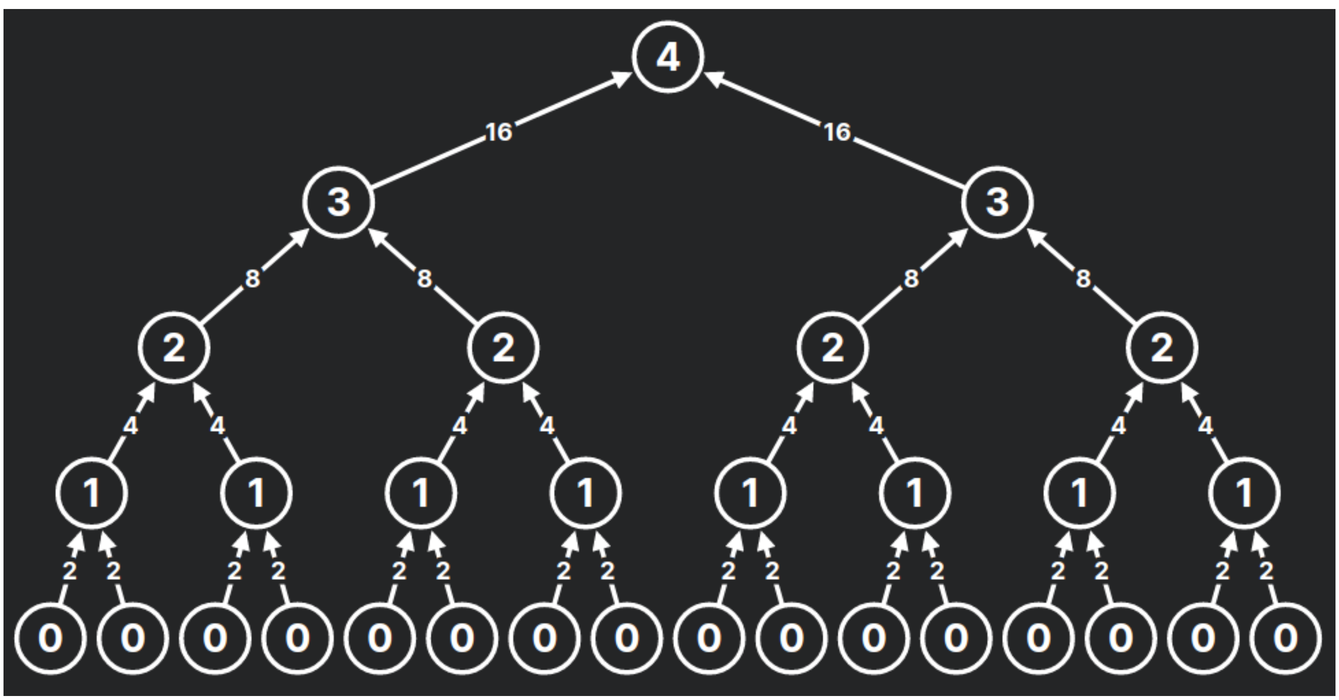
Finding time complexity of a recursive code

What will be the time complexity of this code ?

```
int f(int x)
{
    if(x==0)
    {
        return 2;
    }
    else
    {
        return f(x-1) + f(x-1);
    }
}
```

Let us take x = 4





Number of function calls $= 1 + 2 + 4 + 8 + 16$

$= 31$

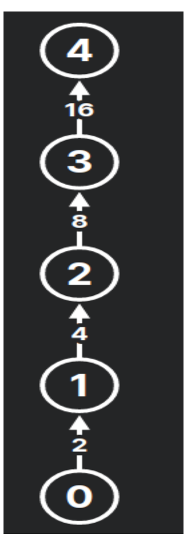
$= 2^5 - 1$

For if call f(n) , you will get $2^{(n+1)}$ - 1 function calls
So, the time complexity = O($2*2^n$ - 1) = O(2^n)

Now, let's change 1 line in the code

```
int f(int x)
{
    if(x==0)
    {
        return 2;
    }
    else
    {
        return f(x-1) * 2;
    }
}
```

What is the time complexity of this code?



Time complexity = O (n +1) = O(n)

You can try this website for visualising the recursion tree:
https://recursion.now.sh/

Intuition of Dynamic Programming

1 + 2 + 6 + 7 + 5 = ?
21

1 + 2 + 6 + 7 + 5 + 2 = ?
21 + 2 = 23

This is DP (Dynamic Programming). **Just remember the past answers and use it to compute your answer.**

Fibonacci Numbers

N   **:** 1, 2, 3, 4, 5, 6…...
F(N) : 0, 1, 1, 2, 3, 5….

Recurrence relation:

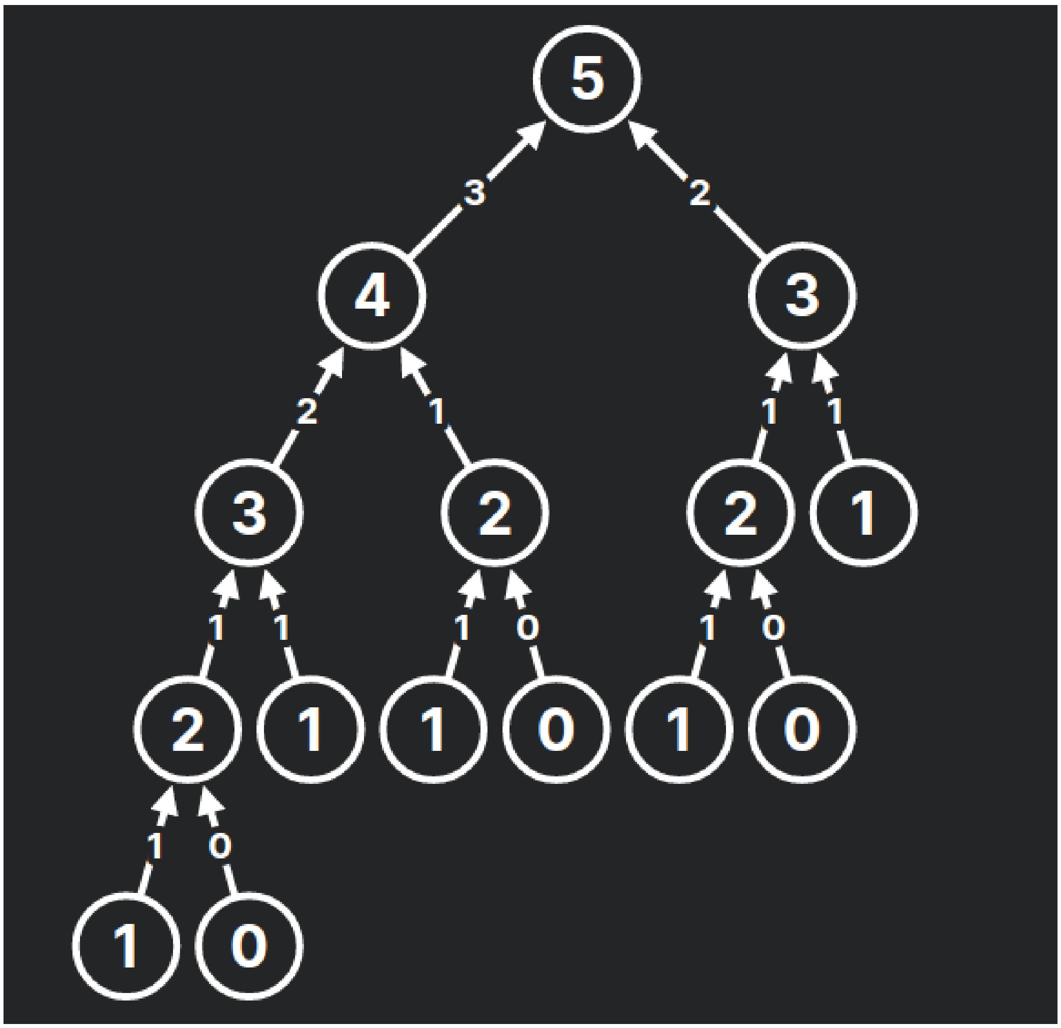
F(N) = F(N-1) + F(N-2)

Recursive code for find Nth Fibonacci number (Without DP)

```
int fib( int n)
{
if(n==1)
    return 0;
if(n==2)
    return 1;
return fib(n-1) + fib(n-2);
```

```
}
```

Time complexity: O (2 ^n)

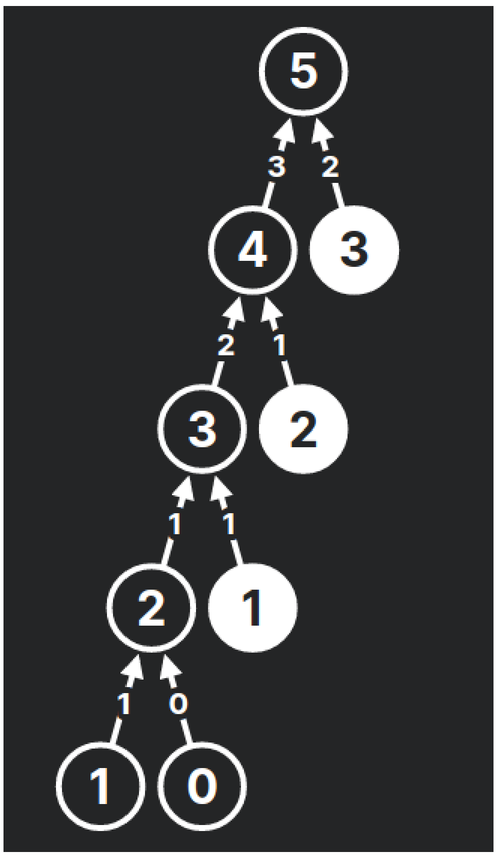


DP = Recursion + Memoization

```cpp
#include <bits/stdc++.h>

using namespace std;

const int MAX = 100000+1;

int dp[MAX]; // dp[i] = i-th fibonacci number

int fib(int n)
{
if(n==1)
    return 0;
if(n==2)
    return 1;
if(dp[n] != -1)
{
    return dp[n];
}
return dp[n]=fib(n-1) + fib(n-2); // Memoization
}


int main() {

  for(int i=0; i<MAX; i++)
  {  dp[i]=-1; // no values are computed at the beginning
  }
```

```
    int n;
    cin>>n;
    cout<<fib(n);
    return 0;
}
```

Time complexity: O(n)



DP without recursion (iterative DP)

```
#include <bits/stdc++.h>
```

```cpp
using namespace std;

const int MAX = 100000+1;

int dp[MAX]; // dp[i] = i-th fibonacci number

int main() {

  int n;
  cin>>n;

  dp[1]=0;
  dp[2]=1;

  for(int i=3; i<=n; i++)
  {
    dp[i] = dp[i-1] + dp[i-2];
  }

  cout<<fib(n);
  return 0;
}
```

Time complexity: O(N)

Wherever we see a recursive solution that has repeated calls for the same inputs, we can optimize it using Dynamic Programming. The idea is to **simply store the results of subproblems**, so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial.

Bottom Up vs Top Down

Bottom Up Approach:

Analogy to understand:
I am going to learn to program. Then, I will start practising. Then, I will participate in coding contests. I will improve by solving those questions which I couldn't solve during every contest. I will be able to crack an internship at a good company.

In Bottom-up you start with the small solutions (base case) and build up.

Example: The without-recursion approach (iterative) for finding n-th fibonacci number shown above

Advantages :
1. Fast and uses less memory than top down.
2. Shorter Code

Top Down Approach:

Analogy to understand:
I will be able to crack an internship at a good company. How? I will improve by solving those questions which I couldn't solve during every

contest. How? I will participate in coding contests. How? I will start practicing? I am going to learn to program.

In Top-down you start building the big solution right away by explaining how you build it from smaller solutions.

Example: The recursion + memoization approach shown above

Advantages :
1. Easy to apply
2. Order doesn't matter.

Q: https://atcoder.jp/contests/dp/tasks/dp_a

Recursion Solution:-

```
#include<bits/stdc++.h>
using namespace std;
vector<int> h;
vector<int> Memo;

int minCost(int i){
// It will give me the minimum cost to reach at ith stone
    if(i==0) return 0;
    if(i==1) return abs(h[1]-h[0]);
    if(Memo[i]!=-1) return Memo[i];
    int lastCost = minCost(i-1) + abs(h[i]-h[i-1]);
    int lastlastCost = minCost(i-2) + abs(h[i]-h[i-2]);
    Memo[i]=min(lastCost,lastlastCost);
    return Memo[i];
```

```
}
int main(){
    int n;
    cin>>n;
    h.resize(n);
    Memo.resize(n);
    for(int i=0;i<n;i++) Memo[i]=-1;
    for(int i=0;i<n;i++) cin>>h[i];
    cout<<minCost(n-1);
    return 0;
}
Time Complexity without Memoization = O(2^n)
Time Complexity with Memoization = O(n)
```

- **H.W- Solve the last problem using iterative DP.**

Q.) You are climbing a staircase. It takes **n steps** to reach the top. Each time you can either **climb 1 or 2 steps.** In **how many distinct ways** can you climb to the top?

Iterative Solution:-

```
int climbStairs(int n) {
        vector<int> dp(n+1);
        //dp[i]-> number of ways to reach at i-th floor
        dp[0]=1;
        dp[1]=1;
        for(int i=2;i<=n;i++) dp[i]=dp[i-1]+dp[i-2];
        return dp[n];
    }
```

- **H.W- Solve the last problem using Recursive DP**

Practice Problems:
1. https://atcoder.jp/contests/dp/tasks/dp_b
2.
https://www.hackerearth.com/practice/algorithms/dynamic-programming/introduction-to-dynamic-programming-1/practice-problems/algorithm/jump-k-forward-250d464b/
3. https://atcoder.jp/contests/dp/tasks/dp_c
4. https://atcoder.jp/contests/abc129/tasks/abc129_c
5. https://codeforces.com/problemset/problem/1245/C
6. https://codeforces.com/problemset/problem/455/A%7C
7. https://codeforces.com/problemset/problem/1195/C
8. https://www.spoj.com/problems/ACODE/
9. https://codeforces.com/problemset/problem/189/A

Just Follow only these websites for practising in first year:
Codeforces, Atcoder, Codechef, Hackerrank, Hackearth, Spoj

On hackerrank, solve all the implementation problems:
(Very important for building the basics)
https://www.hackerrank.com/domains/algorithms?filters%5Bsubdomains%5D%5B%5D=implementation