# **Fibonacci Sequence**

* One of the simplest examples of dynamic programming is computing the nth Fibonacci number.

## **Problem Statement:**

* Write a function **fibonacci(n)** that takes in a number, n, as an argument, and returns the nth number of the Fibonacci sequence.

## **The Fibonacci Sequence Defined**

**Formal Definition**

* A Fibonacci Sequence Fn is a sequence of numbers, in which each number is the sum of the two preceding numbers in the sequence.
* We can define the nth number in the sequence as Fn = Fn-1 + Fn-2

**Base Cases**

* We have two base case numbers, 0 and 1.
  + F0 = 0
  + F1 = 1

**Recurrence Relation**

* By combing the formal definition and the base cases, we can come up with a recurrence relation.

Text

Description automatically generated with low confidence

**Number Sequence**

* Here are the first 10 numbers of the Fibonacci Sequence to show you how it looks:

fib(n): 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, and so on

n: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, and so on

## **Fibonacci Sequence Implementation**

* Here is a classic recursive implementation to compute the nth Fibonacci number

public int **fibonacci(int n)**

{

if (n == 0) return 0;  
if (n == 1) return 1;  
return fibonacci(n - 1) + fibonacci(n - 2);

}

* What is the runtime of this function? Think for a second before you answer.
* If you said O(n) or O(n2) (as many people do), think again.
* Drawing the code paths as a tree (that is, the recursion tree) is useful on this and many recursive problems.

## **Fibonacci Sequence Recursion Tree**

* Here is a recursion tree to compute the 5th number in the Fibonacci sequence (F5).

Diagram

Description automatically generated

* This follows what we know so far about Fibonacci:

1. We have two base cases for n, 0 and 1, where if they are encountered, we can simply return their value of n.

fibonacci(0) = 0

and

fibonacci(1) = 1

1. This also follows the mathematical definition of the Fibonacci Sequence we defined.

fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)

This is at its core a recursive problem.

We break this problem into subproblems that continues to diminish the problem size until we reach a base case.

## **Analyzing the Runtime**

* The **total number of nodes** in the tree will represent the runtime, since each call only does O(1) work outside of its recursive calls. Therefore, **the number of calls is the runtime**.
* So how many nodes are in the tree?
  + The root node has two children.
  + Each node branches out twice (has two children) except for base cases (leaves).
  + Each of those children has two children (so four children total in the "grand­ children" level).
  + Each of those grandchildren has two children, and so on.
* With each level we go down the tree, the number of nodes doubles.
* If we double the number of nodes n times, we'll have roughly O(2n) nodes.
* Another way to look at it is that we can calculate the number of nodes at a level in the tree by calculating 2h where h is the height of the tree.
* In this problem, the height of the tree is represented by the given input ***n***.
* This gives us a runtime of roughly O(2n).

**Note**:

This algorithm is actually it's slightly better than O(2n).

The right subtree of any node is always smaller than the left subtree.

If they were the same size, we'd have an O(2n) runtime. But since the right and left subtrees are not the same size, the true runtime is closer to O(1.6n).

Saying O(2n) is still technically correct though as it describes an **upper bound** on the runtime.)

## **Optimizing the Solution**

**Overlapping Subproblems**

* If you look closely at the tree, you will see a pattern.
* We perform repeated work in different parts of the recursion execution tree.
* Identical nodes represent overlapping subproblems in which we repeat work.
  + fibonacci(3) appears twice
  + fibonacci(2) appears three times
* Why should we have to recompute these calls from scratch each time we encounter it?
* It would be better if we somehow cached the results so we don’t have to repeat work for overlapping subproblems.
* In fact, when we call fibonacci(n), we shouldn't have to do much more than O(n) calls, since there's only O(n) possible values we can throw at the function.
* Each time we compute fibonacci(i), we should just cache this result and use it later.

**Caching**

* Caching is often called **memorization**.
* With just a small modification, we can tweak this function to run in O(n) time.
* We simply cache the results of fibonacci(i) between calls.
* Here is the adjusted code:

public int **fibonacci**(int n)

{

return **fibHelp**(n, new int[n+1]);

}

public int **fibHelp**(int n, int[] memo)

{

if(i == 0 || i == 1)

return n;

if(memo[i] == 0)

memo[i] = **fibHelp**(i-1, memo) + **fibHelp**(i-2, memo);

return memo[i];

}

* Now if we look at the recursion tree, we have eliminated the repeated work.
* The black boxes represent the calls we cached that returned immediately:

Diagram

Description automatically generated

* How many nodes are in this tree now?
* We might notice that the tree now just shoots straight down, to a depth of roughly n.
* Each node of those nodes has one other child, resulting in roughly 2n children in the tree. This gives us a runtime of O(n).