# **Recursion and Big O**

## **Time Complexity**

* Time Complexity is the **number of operations** an algorithm performs to complete its task.
* We usually represent the total number of operations as ***n***.
* The time complexity of a **recursive function** is determined by
  + loops
  + the total number of function calls
* To determine the number of recursive function calls, we must analyze
  + how the algorithm breaks the problem into subproblems
    - does the function call itself once or multiple times in the same line?
    - how much space do parameters take up & what do we pass in as arguments?
  + how long it takes to reach a base case
    - how many total recursive function calls are there?

## **Space Complexity**

* The space complexity of recursive algorithm is proportional to **maximum depth of the recursion** **tree generated**.
* Each recursive call a function makes to itself is one unit of space.
* There can be up to n frames on the call stack at a given time.
  + This would be to the equivalent of the max depth of a recursive tree.
* Therefore, the space taken JUST for recursive function calls can be quantified in terms of the total **number of recursive function calls**.
* However, the maximum amount of stack frames that are generated in memory at any given time in the recursive call sequence is only **one part** of the space.
* If a single function call, say, needed to create O(m) worth of space for a parameter of length m, and if the maximum depth of recursion tree is 'n' then space complexity of recursive algorithm would be O(nm).

## **Linear Recursive Algorithm Big O**

Here we outline several types of common linear recursive call patterns, along with their associated time and space complexity.

**Algorithm 1 --------------------------------------------------------------------------------------------------------------**

public void **foo1** (int n) {

if (n <= 1) return;

**foo (n - 1);**

}

**Time Complexity: O(n)**

We make n function calls.

Let’s say we pass in 5. The function foo1 would call itself 5 times, with the n parameter going from 5, to 4, to 3, to 2, until we reach the base case at 1.

Therefore, this algorithm will take O(n) time.

**Space Complexity: O(n)**

We will have up to n stack frames on the class stack when we reach the base case.

Therefore, this algorithm requires O(n) space.

Icon

Description automatically generated

**Algorithm 2 --------------------------------------------------------------------------------------------------------------**

public void **foo2** (int n) {

if (n <= 1) return;

**foo (n - 2);**

}

**Time Complexity: O(n/2) = O(n)**

This function deducts 2 from n each time it recursively calls itself, halving the number of function calls compared to foo1.

Therefore, this algorithm will take O(n/2) time.

However, with big O, we ignore multiplicative/division constants, we it is actually O(n) time.

**Space Complexity: O(n/2) = O(n)**

Using the same logic, we will have up to O(n/2) = O(n) stack frames on the class stack when we reach the base case.

Therefore, this algorithm requires O(n) space.

Icon

Description automatically generated

**Algorithm 3 --------------------------------------------------------------------------------------------------------------**

public void **foo3** (int n) {

if (n <= 1) return;

**foo (n / 2);**

}

**Time Complexity: O(logn)**

As opposed to deducting 2 from n each time the function recursively calls itself, it **divides** the input by 2 on each recursive call.

Therefore, this algorithm will take O(logn) time.

**Space Complexity: O(logn)**

We will have up to O(logn) stack frames on the class stack when we reach the base case.

Therefore, this algorithm requires O(logn) space.

**Algorithm 4 --------------------------------------------------------------------------------------------------------------**

public void **foo4** (int n) {

if (n <= 1) return;

for (i = 0; i < n; i += 2)

// do something O(1)

return **foo4(n - 5);**

}

**Time Complexity: O(n/2) \* O(n/5) = O(n)**

The for loop takes n/2 time since we’re increasing by 2, and the recursion takes n/5 since we subtract n by 5 each recursive call.

Since the for loop that takes n/2 time is called recursively n/5 times, we multiply the two

· =

Due to Asymptotic behavior and worst-case scenario considerations or the upper bound that big O is striving for, we are only interested in the largest term so O(n2).

**Space Complexity**

As far as we can see, no extra space is taken up by the loop, so the space complexity is only based on the number of stack calls/frames.

Therefore, this algorithm requires O(n) space.

## **Tree Recursive Algorithm Big O**

Here we outline several types of common recursive call patterns, where there are **multiple recursive calls in every function call**.

**Algorithm 5 --------------------------------------------------------------------------------------------------------------**

public void **foo5** (int n) {

if (n <= 1) return;

**foo5(n - 1);**

**foo5(n - 1);**

}

When a function has this form of two recursive function calls in its body, with one immediately following the other, we can track the recursive call stack in a tree like structure.

This actually builds on our previous linear examples.

The left side of the tree is the first path we take and is identical to the linear stack trace.

The total number of function calls can be found by finding the **total number of nodes in the tree**. We identify that each level in the tree progressively has twice as many nodes.

Level 1 has 1 nodes

Level 2 has 2 nodes

Level 3 has 4 nodes

…

Therefore, if a tree has a **height of n** (we know this because the left side of the tree is identical to our linear problem), and we multiply each level by 2, then we are essentially multiplying by 2 n times.

Our time complexity is therefore 2n

Our space complexity is not 2n, however

The amount of space taken up by recursive calls can be found by finding the **maximum number of stack frames that are present on the stack at the same time**.

When we reach the base case, a frame is popped from the stack. Therefore, at any point in time, the max number of frames on the stack is the **height of the tree**, or n stack frames.

We can do this by

**Time Complexity = O(2n)**

<https://www.youtube.com/watch?v=oBt53YbR9Kk&t=769s>

14:57 – 19:57

**Space Complexity = O(n)**

<https://www.youtube.com/watch?v=oBt53YbR9Kk&t=769s>

14:57 – 19:57

**Algorithm 6 --------------------------------------------------------------------------------------------------------------**

public void **foo6** (int n) {

if (n <= 1) return;

**foo6(n - 2);**

**foo6(n - 2);**

}

**Time Complexity = O(2n/2) = O(2n)**

<https://www.youtube.com/watch?v=oBt53YbR9Kk&t=769s>

19:57 – 21:14

**Space Complexity = O(n/2) = O(n)**

<https://www.youtube.com/watch?v=oBt53YbR9Kk&t=769s>

19:57 – 21:14

## **Fibonacci**

After looking at how multiple recursive calls in a function can affect time and space complexity, how would this apply to calculating the Fibonacci sequence?

The fib function would actually be right in between the two functions dib and lib, it takes a piece from each.

public void **dib**(int n) {

if (n <= 1) return;

**dib(n - 1);**

**dib(n - 1);**

}

public void **fib**(int n) {

if (n <= 1) return;

**fib(n - 1) + fib(n - 2);**

}

public void **lib**(int n) {

if (n <= 1) return;

**lib(n - 2);**

**lib(n - 2);**

}

We can say the time complexity is

O(lib) O(fib) O(dib)

O(**2n**) O(?) O(**2n**)

O(**2n**) O(**2n**) O(**2n**)

The Fibonacci function runs in O(**2n**) time.

This is obviously not an ideal time complexity and is the source of our main **bottleneck**.

If we wanted to pass in 50 to fib(int n), it would take

**250** steps = 1.12e+15 steps (over one quadrillion steps

This is interesting, because we ask our Fibonacci function for something relatively small yet takes extremely long to do.

**How Can We Reduce the Number of Recursive Calls?**

Look for patterns!

Shape, arrow, polygon

Description automatically generated