

sli.do/197346

DoCSoc Interview Series: An Introduction to Space & Time Complexity

By Arjun Banerjee

sli.do/197346

About Me

- Computing Final Year MEng.
- Software Engineer Intern @ Microsoft.
- Previously interned at a fintech startup and a large oil & gas service provider.
- I have done a lot of interviews.
- Your Well-being Dep Rep :).

Technical Interview: Coding Section

Code Review

1. Code snippet is shown to you:

```
def f(x: list[str]) -> int:
    count = 0

    for s in x:
        if g(s):
            count += 1

    return count
```


1. Questions about what it does.
2. Questions about the performance of the code.
3. Questions about what improvements could be made. (Maybe even implementation)

Coding Problem

1. You are given a problem statement:


"Given an array of integers `nums` and an integer `target`, return *indices of the two numbers such that they add up to* `target`."

1. Discuss and implement a solution to the problem.
2. Questions about the performance of the code.
3. Iterate between step 2 and 3 until an optimal solution is reached or you run out of time.



sli.do/197346

How do you measure the
performance of code?

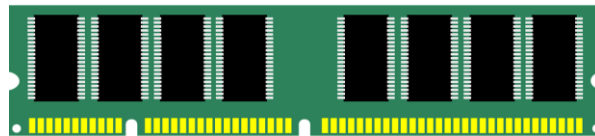


Code Performance




Time

- How long does it take to run on an arbitrary input?
What happens as the size of the input increases?



Space

- What is the memory footprint of the execution on an arbitrary input? What happens as the size of the input increases?



sli.do/197346

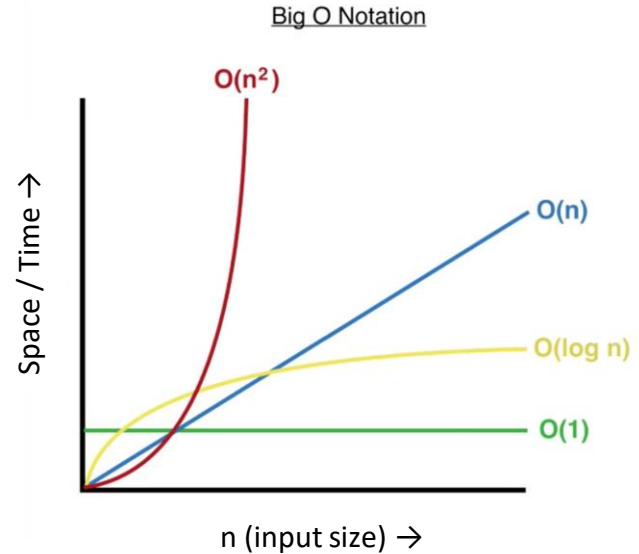
How do you describe the
performance of code in an
interview?



Big O Notation

A standard way to describe the **asymptotic upper bound** of a function. More on this in Algorithms II :)

- Why the upper bound?
 - Be able to handle the worst case.
- What is the input size?
 - Length of a list.
 - Depth of a tree.
 - How large a number is.



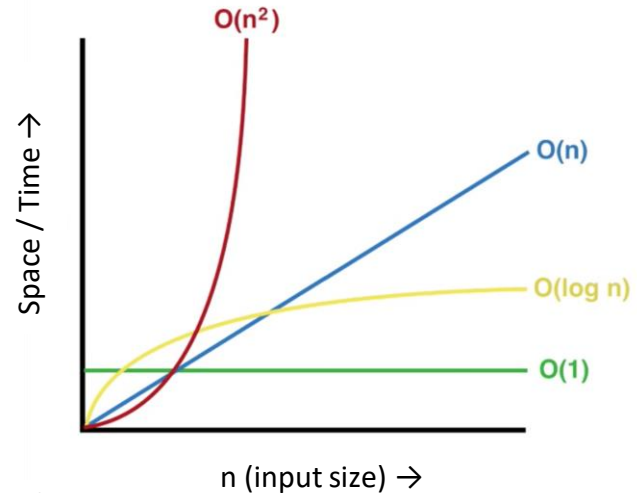
sli.do/197346

Big O Notation

Function	Big O	Name
$f(n) = 42$	$O(1)$	Constant
$f(n) = 10\log(n)$	$O(\log(n))$	Logarithmic
$f(n) = 60n + 2$	$O(n)$	Linear
$f(n) = 12n^2$	$O(n^2)$	
$f(n) = cn^x$	$O(n^x)$	Polynomial
$f(n) = 2^n$	$O(2^n)$	Exponential
$f(n) = 3n\log(n)$		
$f(m, n) = mn + m\log(n) + 3m + 2n$		N/A

[1*yiyfZodqXNwMouC0-B0Wlq.png \(1272x1100\) \(medium.com\)](#)

Big O Notation



Rules:

1. Drop constants
2. Drop non-dominant terms
3. If you have multiple inputs all should appear in the result, apply 1, 2 to each input's terms.

Time Complexity

*“Time complexity is the amount of time taken by an algorithm to run, as a function of the length of the input. It measures the time taken to execute the statements of **code** in an algorithm.”*

- We can assume primitive operations (that make up statements) will take a constant amount of real time to execute.
- Therefore we can estimate the runtime by the number of statements executed given an input.

General Approach To Calculating Time Complexity

1. Unless given, clearly state variables you are going to use to describe your inputs e.g. n is the length/size of input1.
2. Walk through your code statement by statement and assign a time complexity to it in terms of your variables. Add comments to your code if you have to.
3. Group logical sections of code together (discussed after) to create the terms in the final function on the input.
4. Use Big O notation to describe the time complexity.

Single statement

Java:

```
public int f(int x, int y) {  
    return x + y;  
}
```

```
public int f(List<Integer> x) {  
    return x.size();  
}
```

- Primitive operations e.g. arithmetic, logical, bitwise etc. Can be assumed to run in constant time or $O(1)$
- Method calls can be a bit more complex. These may require deeper knowledge about the method e.g. `Arrays.binarySearch(...)` in Java has a time complexity of $O(\log_2(n))$.

Conditionals (If-else/Switch statements)

Java:

```
// n is the length of foo
public int f(int[] foo) {

    if(foo.length % 2 == 0) {
        return evenFunc(foo); // O(log(n))
    } else {
        return oddFunc(foo); // O(n)
    }
}
```

1. Calculate the time complexity of each branch, also including the condition evaluation (additive).
2. Assign the worst time complexity to the if-statement.

Loops

Java:

```
// n is the size of xs
public boolean f(int[] xs, int t) {
    ...
    for(int x : xs) {
        if(x == t) return true;
    }
    ...
}
```

1. First calculate the time complexity of the statements that are executed inside the loop.
2. Multiply this by the length of the loop.
3. For nested loops you can recursively follow the steps above. Starting at the inner-most loop moving outwards.

Loops

Java:

```
// n <- size of xs
public boolean f(int[] xs, int t) {
    ...

    for(int x : xs) {

        for(int i = 0; i < t; i++) {
            f(x,i); // O(1)
        }
    }
    ...
}
```

1. First calculate the time complexity of the statements that are executed inside the loop.
2. Multiply this by the length of the loop.
3. For nested loops you can recursively follow the steps above. Starting at the inner-most loop moving outwards.

Consecutive Statements

Java:

```
// n <- size of xs
public List<Integer[]> f(int[] xs, int t) {
    Arrays.sort(xs);
    List<Integer[]> pairs = new ArrayList<>();
    for(int i : xs) {
        for(int j : xs) {
            if(i * j == t)
                pairs.add(new Integer[]{i, j}); // O(1)**
        }
    }
    return pairs;
}
```

1. Calculate the time complexity for each statement / logical group of statements.
2. Add these together.
3. Simplify with Big O rules.

Time complexity:

$$O(n \log(n) + 1 + n^2 + 1) = O(n^2)$$

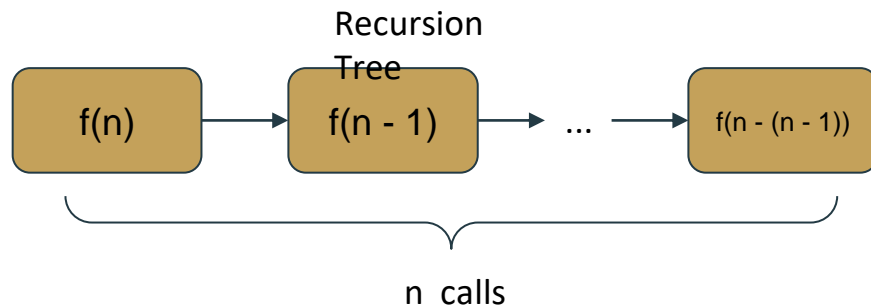
** Amortized time complexity

Recursion

Java:

```
// assume n >= 0
public int f(int n) {
    if(n <= 1) {
        return 1;
    }
    return n * f(n - 1);
}
```

1. Figure out worst time complexity of the method assuming you have the value of the recursive call.
2. Roughly calculate the number of recursive calls.
3. Multiply this by the number of recursive calls.
4. Simplify with Big O.

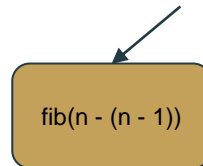
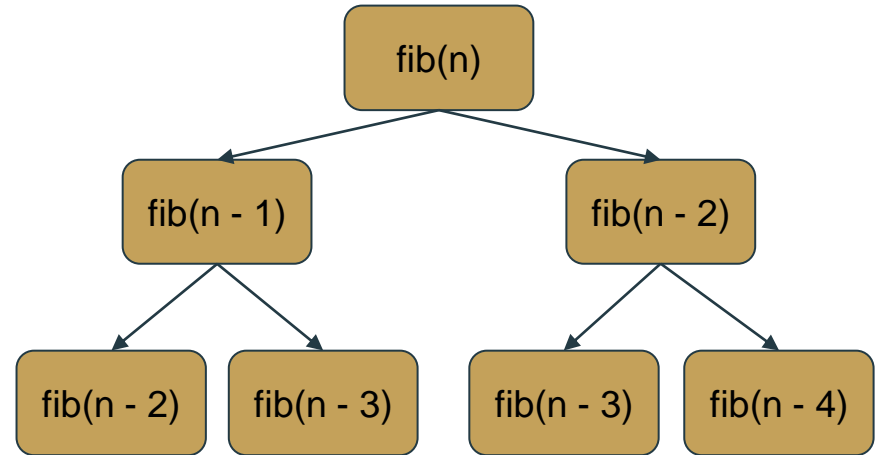


Time complexity: $O(1 * n) = O(n)$

Recursion

Java:

```
// assume n >= 0
public int fib(int n) {
    if(n <= 1) return n;
    return fib(n - 1) + fib(n - 2);
}
```



d (tree depth) = n

#nodes (assuming full tree) = $2^d - 1 = 2^n - 1$

$O(1 * \text{\#nodes}) = O(2^n - 1) = O(2^n)$

Space Complexity

“Space complexity is the amount of memory used by the algorithm (including the input values to the algorithm) to execute and produce the result.”

Space Complexity = Input Space + Auxiliary Space

- **Input Space**: Space allocated to store the input.
- **Auxiliary Space**: Temporary space allocated during the algorithm's execution.
- Generally, during interviews, the focus is on the **Auxiliary Space** used by your solution. So we will focus on that.
- We do not need to think in bytes, we can assume overhead is constant and primitive data type / static data structure sizes are constant too.

General Approach To Calculating Space Complexity

Pretty much the same as time complexity.

1. Unless given, clearly state variables you are going to use to describe your inputs e.g. d is the depth of tree1.
2. Walk through your code statements look for ones that will allocate memory.
3. Figure out the maximum sizes for dynamic data structures (ones that can grow and shrink).
4. Sum all of these together.
5. Use Big O notation to describe the space complexity / auxiliary space used.

Auxiliary Space: Variable Declaration

Java:

```
public void f(int[] xs) {  
    int len = xs.length;  
    int[] arr1 = new int[len];  
    int[] arr2 = new int[26];  
    List<Integer> l = new ArrayList<>();  
}
```

1. Initialising primitive types e.g. int, char, bool, etc or static data types will have constant space ($O(1)$).
2. Initialising a dynamic data structures will usually have constant overhead.
3. Initialising an array:
 - a. Size known at compile time: constant
 - b. Size dependent on input (evaluated at runtime): a function of the input.

Auxiliary Space: Modifying Dynamic Data Structures

Java:

```
public List<Integer> f(int[] xs) {  
    List<Integer> l = new ArrayList<>();  
  
    for(int i = 0; i < xs.length; i++) {  
        if(xs[i] % 2 == 0) l.add(i);  
    }  
  
    return l;  
}
```

- Dynamic data structures can allocate and deallocate memory (grow or shrink) during runtime e.g. lists, maps / dicts, trees, graphs, heaps, etc.
- Look for methods that will modify their size.
- For each of these data structures calculate the maximum size (worst case) they can reach during execution and sum them.

Auxiliary Space = $O(1 * n) = O(n)$

Input size = $O(n)$

Space Complexity = $O(n + n) = O(n)$

Auxiliary Space: Recursion

Java:

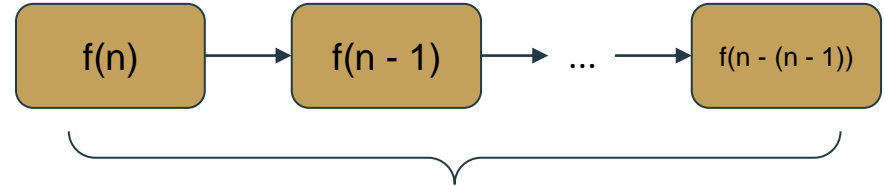
```
// assume n >= 0
public int f(int n) {
    if(n <= 1) {
        return 1;
    }
    return n * f(n - 1);
}
```

1. Think about how method calls are executed
 - a. Each method call pushes a **stack frame** onto the **call stack**.
 - b. Information about parameters (values/references), local variables, return addresses, etc are stored in this **stack frame**.
 - c. Recursion causes the **call stack** to grow until the base case is reached, then it will shrink to aggregate the results.
2. Again visualise the recursion tree and figure out the max depth. This is the maximum number of **stack frames** present for the method on the **call stack** at one time.

Auxiliary Space: Recursion

Java:

```
// assume n >= 0
public int f(int n) {
    if(n <= 1) {
        return 1;
    }
    return n * f(n - 1);
}
```



Call Stack



Auxiliary Space = $O(1 * n) = O(n)$

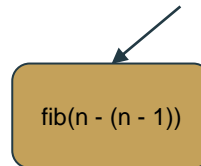
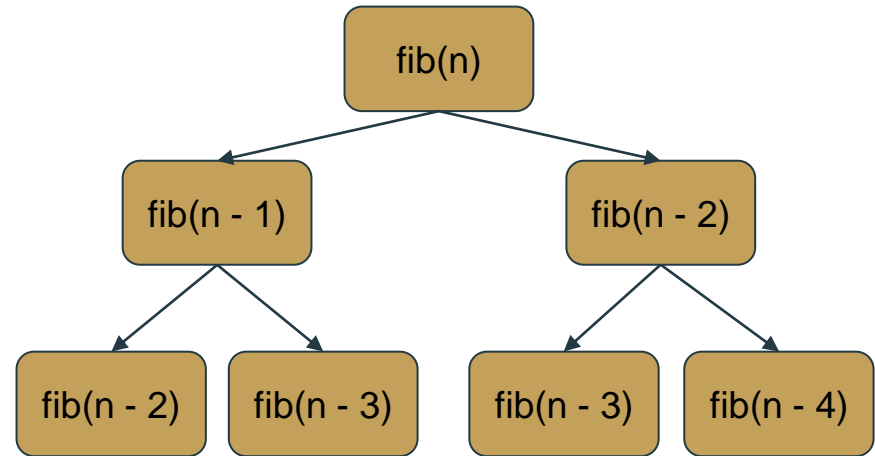
Input size = $O(1)$

Space Complexity = $O(1 + n) = O(n)$

Auxiliary Space: Recursion

Java:

```
// assume n >= 0
public int fib(int n) {
    if(n <= 1) return n;
    return fib(n - 1) + fib(n - 2);
}
```




d (max tree depth) = n
Auxiliary Space = $O(n)$
Input Size = $O(1)$
Space Complexity = $O(n)$

General Approach To Optimising A solution

After you have calculated the space and time complexity for your solution. You might wonder how you could improve it or whether it is possible to.

- Have an idea of the best conceivable runtime: If you were to solve the problem by hand what steps would you intuitively take?
- Look for bottlenecks: sections of code which produce the dominant terms in your worst case complexities.
- Space / Time trade-off: You can most likely sacrifice space to improve the time complexity e.g. caching.
- Optimise for the problem context. Are there constraints/patterns in the input, more reads than writes etc?



sli.do/197346

An Example



Calculate The n^{th} Fibonacci Number (0 indexed)

Java:

```
// assume n >= 0
public int fib(int n) {
    if(n <= 1) return n;
    return fib(n - 1) + fib(n - 2);
}
```

Time complexity: $O(2^n)$

Space complexity: $O(n)$

Where can we optimise?

The recursive branches causes an exponential number of method calls? Can we reduce this?

Overlapping subproblems! We are solving things we have solved before.

Memoisation / Caching

sli.do/197346

Java:

```
public int fib(int n) {
    if(n <= 1) return n;
    return helper(n, new HashMap<>());
}

public int helper(int n, Map<Integer, Integer> cache)
{
    if(n <= 1) return n;

    if(!cache.containsKey(n)) {
        cache.put(n, helper(n - 1, cache)
            + helper(n - 2, cache));
    }

    return cache.get(n);
}
```

Time complexity: $O(n)$

Space complexity: $O(n + n) = O(n)$

Where can we optimise?

We are still calculating in a top-down approach so the call stack is still growing quite large. This could lead to a stack overflow error.

Iterative approach: bottom-up

sli.do/197346

Java:

```
public int fib(int n) {  
    if(n <= 1) return n;  
    int[] cache = new int[n + 1];  
    cache[1] = 1;  
    for(int i = 2; i <= n; i++) {  
        cache[i] = cache[i - 1] + cache[i - 2];  
    }  
  
    return cache[n];  
}
```

Time complexity: $O(n)$

Space complexity: $O(n)$

Where can we optimise?

Do we ever reuse a cache entry more than 2 indices away from the current iteration?

Iterative approach: only cache needed values sli.do/197346

Java:

```
public int fib(int n) {  
    if(n <= 1) return n;  
  
    int prev = 0, total = 1, tmp; // tmp vars  
  
    for(int i = 2; i <= n; i++) {  
        tmp = total; // keep total to update prev  
        total = total + prev; // calc next value  
        prev = tmp; // update prev  
    }  
  
    return total;  
}
```

Time complexity: $O(n)$

Space complexity: $O(1)$

Where can we improve?

This solution is liable to cause an integer overflow.

Further work. Reimplement this to be able to handle large n .