Big(O) Notation and Asymptotic Time and Space Analysis

Chidera Anichebe
www.ieee.org
dera@ieee.org

What is Big O Notation?

- **According to Wikipedia: "Big *O* notation** is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity.

In computer science, big O notation is used to classify algorithms according to how their run time or space requirements grow as the input size grows. In analytic number theory, big O notation is often used to express a bound on the difference between an arithmetical function and a better understood approximation; a famous example of such a difference is the remainder term in the prime number theorem. Big O notation is also used in many other fields to provide similar estimates."

Why is it useful?

- Big-O tells you the complexity of an algorithm in terms of the size of its inputs. This is **essential** if you want to know how algorithms will scale as its input becomes very large.
- If you're designing a big website and you have a lot of users, the time it takes you to handle those requests is important. If you have lots of data and you want to store it in a structure, you need to know how to do that efficiently if you're going to write something that doesn't take a million years to run.

Why does it matter?

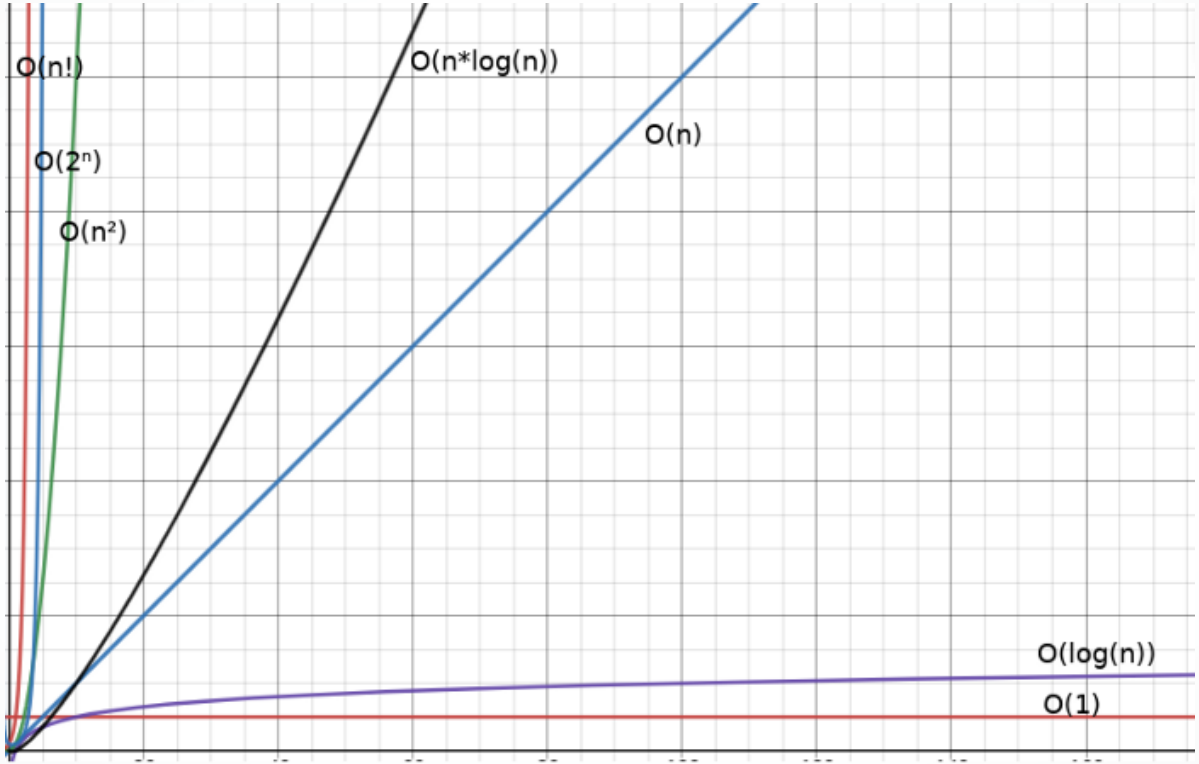
- Big-O tells you the complexity of an algorithm in terms of the size of its inputs. This is **essential** if you want to know how algorithms will scale as its input becomes very large.
- If you're designing a big website and you have a lot of users, the time it takes you to handle those requests is important. If you have lots of data and you want to store it in a structure, you need to know how to do that efficiently if you're going to write something that doesn't take a million years to run.

Why does it matter?

It describes…

- Efficiency of an Algorithm
- Time factor of an Algorithm
- Space complexity of an Algorithm

Chart of common Big O notations



Order of Growth for common notations

Typical Big-O Functions

(in descending order of growth)

Function	Common Name
$n!$	Factorial
2^n	Exponential
n^d, $d > 3$	Polynomial
n^3	Cubic
n^2	Quadratic
$n\sqrt{n}$	n Square root n
$n \log n$	n log n
n	Linear
$\sqrt{n}$	Root - n
$\log n$	Logarithmic
1	Constant

```
Big-O       |  computations for 10 things |  computations for 100 things
-------------------------------------------------------------------------
O(1)        |   1                         |     1
O(log(n))   |   3                         |     7
O(n)        |  10                         |   100
O(n log(n)) |  30                         |   700
O(n^2)      | 100                         | 10000
```

Algorithm Analysis

Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation. They are the following:

- *A priori* analysis – This is a theoretical analysis of an algorithm. Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.
- *A posterior* analysis – This is an empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target's machine. In this analysis, actual statistics like running time and space required, are collected.
- **Big O notation deals with *a priori* analysis**

Algorithm analysis

Time Complexity

- The time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step takes constant time.
- For example, addition of two n-bit integers takes n steps. Consequently, the total computational time is T(n) = c * n where c is the time taken for the addition of two bits. Here, we observe that T(n) grows linearly as the input size increases

Algorithm analysis

Space Complexity

- Space complexity of an algorithm represents the amount of memory space required by an algorithm in its life cycle. The space required by an algorithm is equal to the sum of the following two components:
- A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size etc.
- A variable part is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack frames etc.
- Space complexity S(p) of any algorithm P is S(p) = C + S(i), where C is the fixed part and S(i) is the variable part of the algorithm, which depends on instance characteristics.

Asymptotic analysis

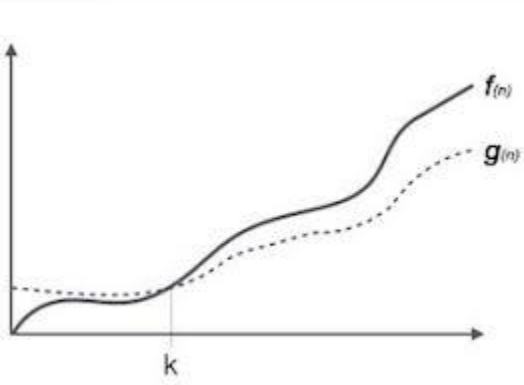
Best case, average case and worst case

- Asymptotic analysis of an algorithm refers to defining the mathematical boundaries of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case and worst case scenarios of an algorithm.
- Asymptotic analysis is input bound i.e. if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input", all other factors are considered constant.
- Usually, the time required by an algorithm falls under three types –

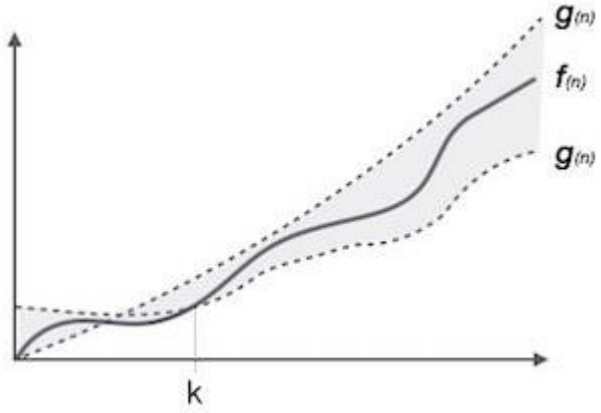
- Best case – Minimum time required for program execution
- Average case – Average time required for program execution
- Worst case – Maximum time required for program execution

Other forms of asymptotic notation

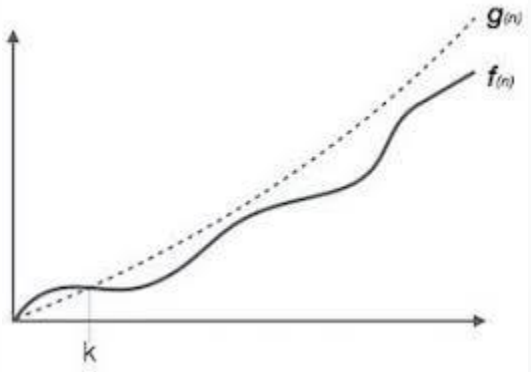
Big O, Omega and Theta notations



Omega (lower bound)

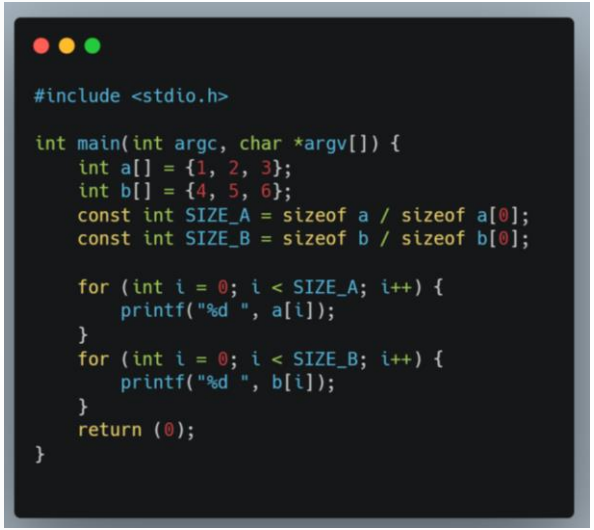


Theta (tight bound)



Big O (upper bound)

Common mistakes in Big O notation



```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int a[] = {1, 2, 3};
    int b[] = {4, 5, 6};
    const int SIZE_A = sizeof a / sizeof a[0];
    const int SIZE_B = sizeof b / sizeof b[0];

    for (int i = 0; i < SIZE_A; i++) {
        printf("%d ", a[i]);
    }
    for (int i = 0; i < SIZE_B; i++) {
        printf("%d ", b[i]);
    }
    return (0);
}
```

This is O(a+b) not O(n)

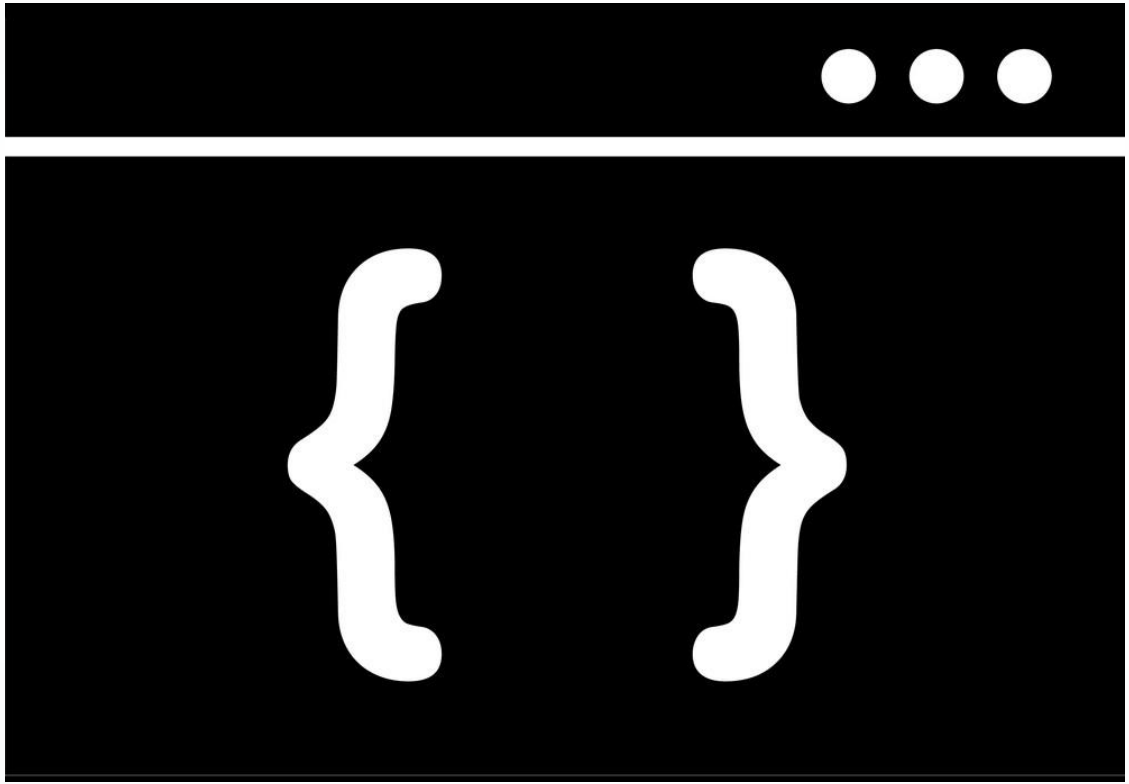
```python
a = [1, 2, 3]
b = [4, 5, 6]

for i in range(len(a)):
    for j in range(len(b)):
        print(a[i] * b[j])
```

This is O(ab) not O(n^2)

NOTE: Add the runtimes if you do x then y; Multiply the runtimes if for each x you do y

PRACTICAL SESSION…



Questions?

Thank you!

Let's connect

https://twitter.com/starlingvibe

https://linkedin.com/in/chideraanichebe

https://github.com/starlingvibes