Algorithms and Big O

What is an algorithm? It is defined as a well-defined **sequence of steps** for solving a computational problem.

A computational problem is a problem that a computer might be able to solve. For example, the problem of sorting a sequence of numbers in ascending order is a computational problem.

Problems must be well specified. That is, the statement of the problem must specify the inputs, outputs, and the relationship between the inputs and outputs.

There may be more than one way to solve that problem. An algorithm, then, is any sequence of steps that, if followed precisely, will solve the problem.

A note on notation

In the study of algorithms, the programming language does not matter. An algorithm may be implemented in any programming language. What is relevant is what **steps** are needed to complete the task.

For this reason, you can write an algorithm using pseudocode, is a form of structured English used for describing algorithms. It resembles programming code but isn't concerned with details such as semicolons.

Real world algorithms

Algorithms are everywhere. They impact your life in meaningful ways, even if you do not directly use them yourself.

Google's PageRank algorithm

If you have ever done a search on Google, you have used their PageRank algorithm. This algorithm attempts to measure the importance of a web page and its relevance to your search query.

Internet routing

The internet is made up of millions of nodes (computers and routers and other devices) that are all connected together in a vast array of connections. To make this all work, there are a series of different routing algorithms that attempt to find the fastest route from your computer to any other computer that you are connected to on the internet.

Cryptography

Encryption is used to secure communication on the internet. If you have ever used online banking, done a credit card transaction, or even used an online email service like Gmail, then you have used an encryption algorithm.

GPS

GPS works by synchronizing atomic clocks on a constellation of satellites orbiting the Earth with a ground-based control system and your phone. The algorithms involved are as complex as anything that humankind has ever created.

Big O

In order to measure the performance of an algorithm is through **algorithm complexity**.

Algorithm complexity can be broken down into time complexity and space complexity.

Time complexity => time to complete
Space complexity => memory occupied

To describe algorithm complexity, we need to use an **asymptotic notation** syntax. The three types of asymptotic notations are **big O**, **big Theta** and **big Omega**.

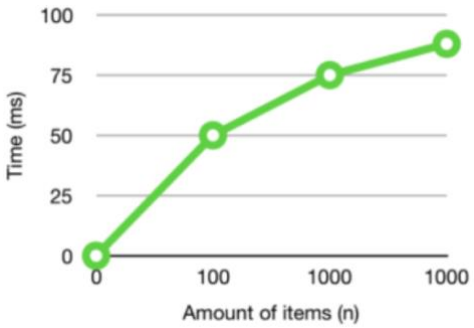
Big O is used for the worst-case running time of an algorithm, big Theta (Θ) is used when running time is the same for all cases, and big Omega (Ω) is used for the best-case running time.

Big O is the most commonly used notation because most of the time we only care about **worst-case** when analyzing performance of an algorithm.

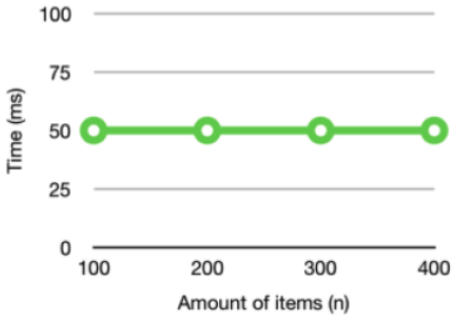
Common Runtimes

Input Size	O(1)	O(n)	O(n log n)	$O(n^2)$	$O(2^n)$	O(n!)
1	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.
10	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	4 seconds
10k	< 1 sec.	< 1 sec.	< 1 sec.	2 minutes	∞	∞
100k	< 1 sec.	< 1 sec.	1 second	3 hours	∞	∞
1M	< 1 sec.	1 second	20 seconds	12 days	∞	∞

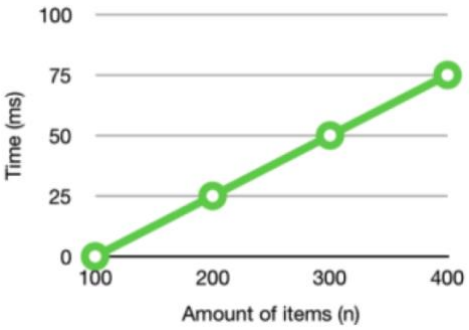
O(1). This is *constant* runtime. This is the runtime when a program will always do the same thing regardless of the input. Assignments, object accesses, and mathematical operation



O(log N). This is *logarithmic* runtime. The size of the input gets **split into half** with each iteration. This happens with different search algorithms such as [binary search](#).

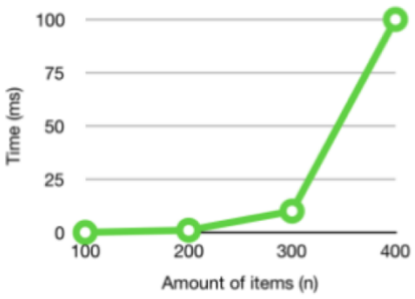


O(N). This is *linear* runtime. Algorithms with linear time complexity (O(n)) have running times that are directly proportional to the size (n) of the input.

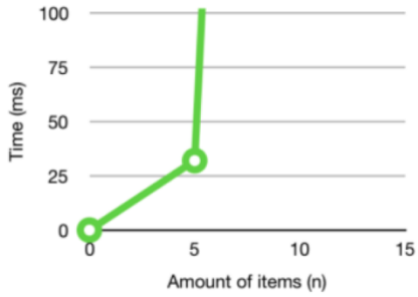


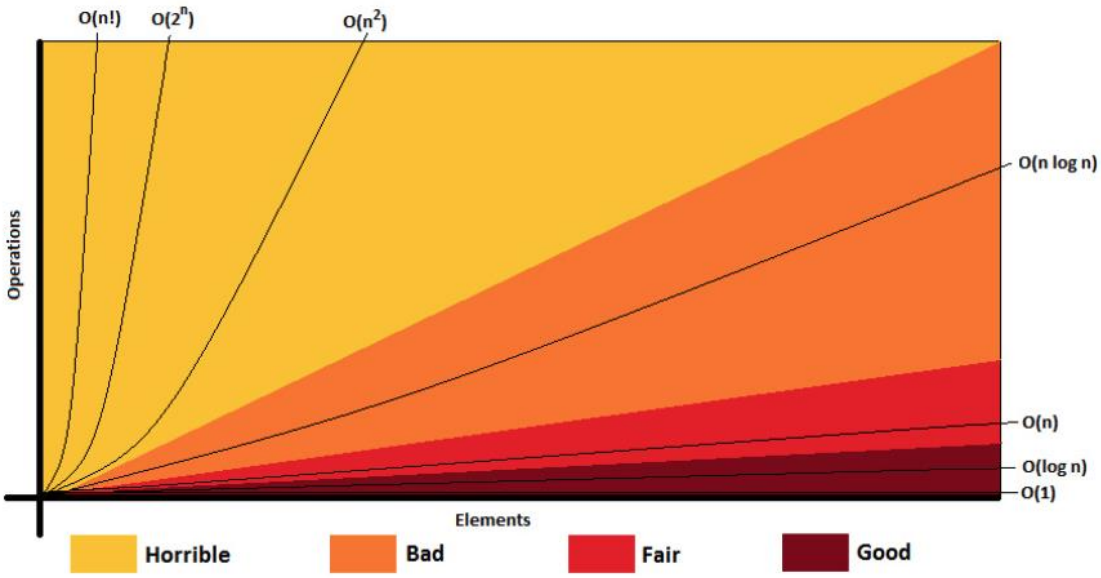
O(N*logN). You will see this runtime in sorting algorithms.

O(N^2). This is an example of a *quadratic* or *polynomial* runtime. You will see this runtime when you have to search through a two-dimensional dataset (like a matrix) or nested loops.



O(2^N). This is *exponential* runtime. Algorithms with exponential time complexity (O(2^n)) have running times that grow rapidly with an increase in the input size.





As you can see, not all O times perform the same. O(n!), O(2n), and O(n2) are considered horrible and we should strive to write algorithms that perform outside this area. O(n log n) is better than O(n!) but is still bad. O(n) is considered fair, while O(log n) and O(1) are good.

Space-Time complexity trade-offs

Space complexity is the measurement of the amount of working storage that an algorithm needs.

As with time complexity, you are mostly concerned with how the space needs to grow, in Big O terms, as the size of the input grows.

Remember that time and space complexity are like a seesaw (i.e., two separate poles). As you attempt to improve the efficiency of the time or space complexity of an algorithm, the other will get worse. For example, improving the time complexity from **O(n)** to **O(log n)** can lead to higher space complexity.

Determining Time Complexity

In general, you can determine the time complexity by analyzing the program's statements. However, you have to be mindful of how are the statements arranged. Suppose they are inside a loop or have function calls or even recursion. All these factors affect the runtime of your code.

Sequential Statements

If we have statements with basic operations like conditionals, assignments, reading a variable. We can assume they take **constant time O(n).**

Conditional Statements

Statements that are conditional (if, switch) still have a worst-case scenario of **constant time O(n).**

Loop Statements

For any loop, we find out the runtime of the block inside them and multiply it by the number of times the program will repeat the loop.

```
for (let i = 0; i < array.length; i++) {
   statement 1;
  statement 2;
}
```

All loops that grow proportionally to the input size have a **linear time complexity O(n).** If you loop through only half of the array, that's still O(n).

Remember that we drop the constants so 1/2 n => O(n).

However, if a constant number bounds the loop, let's say 4 (or even 4000). Then, the runtime is **constant**.
O(4000) => **O(1)**

```
for (let i = 0; i < 4000; i++) {
  statement 1;
  statement 2;
}
```

That code is O(1) because it no longer depends on the input size.

Nested loop statements

Sometimes you might need to visit all the elements on a 2D array (grid/table). For such cases, you might find yourself with two nested loops.

```
for (let i = 0; i < n; i++) {
   statement 1;
  for (let j = 0; j < m; j++) {
     statement 2;
    statement 3;
  }
}
```

Assuming the statements from 1 to 3 are O(1), we would have a runtime of O(n * m). If instead of m, you had to iterate on n again, then it would be **quadratic O(n^2).**

Another typical case is having a function inside a loop.

Function call statements

When you calculate your algorithms' time complexity and invoke a function, you need to be aware of its runtime. If you created the function, that might be a simple inspection of the implementation. However, you might infer it from the language/library documentation if you use a 3rd party function.

```
for (let i = 0; i < n; i++) {
  fn1();
  for (let j = 0; j < n; j++) {
    fn2();
    for (let k = 0; k < n; k++) {
      fn3();
    }
  }
}
```

Depending on the runtime of fn1, fn2, and fn3, you would have different runtimes.

Rules of Big O

Big O notation only cares about the “biggest” terms in the time/space complexity.

Big O is not about counting the code statements (e.g., how many if statements, etc.). Its goal is to express the **runtime growth** for input sizes and express how the **runtime scales**. In short, Big O just describes the runtime rate of increase.

Moreover, don't fall into the trap of thinking that because your code has 2 separate loops the Big O is O(2n). Simply remove 2 from 2n since 2 is a **constant**.

The same works for two loops where one loop has an inner loop. You may think the Big O is O(n + n2). The rate of increase is given by n2, while n is a **non-dominant** term. If the size of the array is increased, then n2 affects the rate of increase much more than n, and so n is not relevant.

<u>Drop constants and non-dominant terms</u>

O(2n) => O(n)
O(5) => O(1)
O(n + log n) => O(n)
O(13n2) => O(n2)
O(n + n + n + n) => O(n) => n being the same input
O(n + 5) => O(n)

Different inputs mean different variables

If you have two separate loops but both loops iterate through the **same input**, the Big O will be **O(n).**

```
for (let i=0; i<a.length; i++) {}

for (let i=0; i<a.length; i++) {}
```

If you have two separate loops and each loop iterates through **separate inputs,** the Big O will be the sum of the two O(n) runtimes which will be **O(a + b).**

```
for (let i=0; i<a.length; i++) {}

for (let i=0; i<b.length; i++) {}
```

Different steps are summed or multiplied

We cannot sum up the runtimes for a nested loop, so in each case of a[i], the code loops the b array, and so the Big O is **O(a * b)**.

```
for (let i=0; i<a.length; i++) {
   for (let j=0; i<b.length; j++) {
       …
   }
}
```

Don’t get this confused with a nested loop that both loops iterate through the **same input**:

```
for (let i=0; i<a.length; i++) {
   for (let j=0; i<a.length; j++) {
        …
   }
}
```

The Big O would be **O(n^2)** because the number of iterations are squared as the input grows. So, for each iteration in the outer loop, the inner loop iterates through **all** elements of the same input in the outer loop.