Algorithm: Big O Analysis

The Big O analysis is a way we can analyze the efficiency of our algorithm. This means checking to see how much time it takes for an algorithm to complete depending on the amount of input it receives. The Big O means we take the most important part of the code or the part that does most of the heavy lifting of the code. Big O is mostly concerned with when the time difference is large or significant.

Take for example in a mathematical approach to this, let’s take the equation 5x2+3x+1 and we assume x to be 5. When we substitute the x for 5 in the equation for the first term we get 125, the second term is 15 and the last term is 1. This made more prominent if x was let’s say 50,000 and we substitute this in the equation, the first term is 12,500,000,000, the second term becomes 150,000 and the last term is still 1. Therefore the part of the equation that does most of the work is the 5x2, in the Big O notation the x is represented as n and the coefficient in this case which is 5 is taken not used (this depends on the situation or problem at hand), therefore the Big O of the equation is n\*n i.e O(n2), the O vacuums of fluffs out the non-important parts.

Let’s explain this better with some code examples:  
function crossAdd(input) {

var answer = [];

for (var i = 0; i < input.length; i++) {

var goingUp = input[i];

var goingDown = input[input.length - 1 - i];

answer.push(goingUp + goingDown);

}

return answer;

}

Let’s breakdown the code shall we, the first part of the code is the **function** keyword and the function name (**crossAdd**) and the parameter, these are constant since they only run once no matter what amount of inputs is passed into the function. Next is the **answer** variable which can be considered as the coefficient like our 5 in the algebra example above. Next is our **For** loop which is our main part of the code because looking at the code it’s the only part that is dependent on the amount of input being passed into the code and does the most work. Lastly, our return is also a constant.

Since there is only one **For** loop this is an O(n) because we go through all inputs in the loop once.

function find(needle, haystack) {

for (var i = 0; i < haystack.length; i++) {

if (haystack[i] === needle) return true;

}

}

Same for the above it’s an O(n) loop because there is only one loop.

function makeTuples(input) {

var answer = [];

for (var i = 0; i < input.length; i++) {

for (var j = 0; j < input.length; j++) {

answer.push([input[i], input[j]]);

}

}

return answer;

}

This is an O(n2) because the we go through a full loop within another full loop. So the trick is to loop out for loops but note again this depends but in most cases it’s the surest way to get your Big O of an algorithm.

Also code like the one below are called **constant time (O(1)) since** there are no loops.

function getMiddleOfArray(array) {

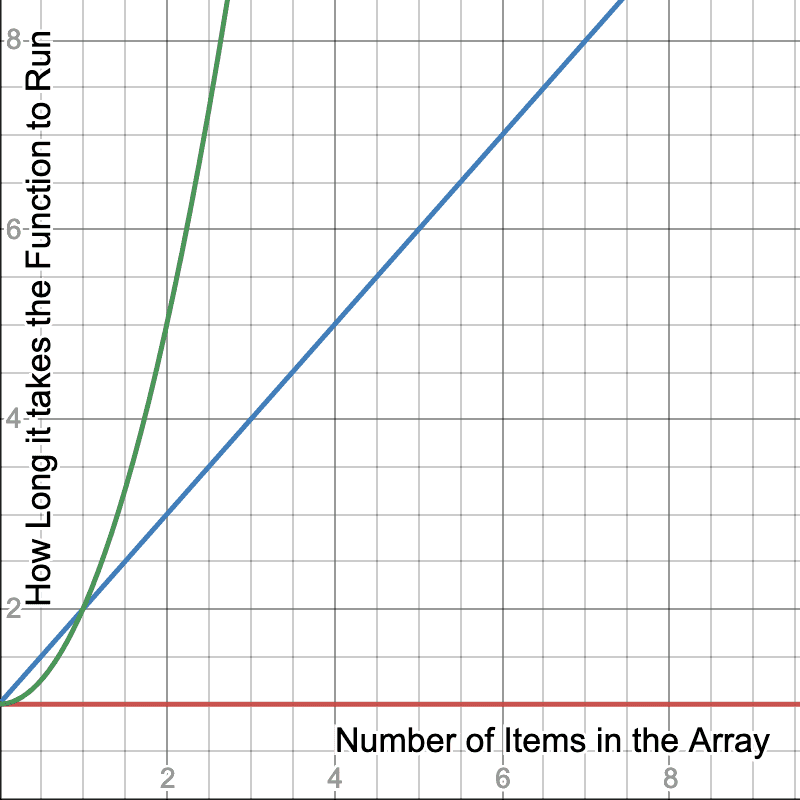
return array[Math.floor(array.length / 2)];

}

Note: Big O is just like a yard stick to make tradeoffs between different algorithms, it does not necessarily help to determine that one algorithm is better than the other just helps to you to make an appropriate choice based on the situation or problem placed before you.

Note: The more loops an algorithm has the more time it takes to complete depending on how many inputs is provided in it.

The graph below helps to explain it better:

[](https://btholt.github.io/complete-intro-to-computer-science/static/e331672c4244a2ae881a5123175c2c59/5a190/graph.png)

The red line represent a constant time (i.e O(1)) meaning no matter the amount of input the algorithm still completes in the same time.

The blue line represents the O(n), meaning the time it takes to complete is proportional to the amount of input.

The green line represent the O(n2), meaning the more input we add, it takes exponentially more time to complete.

Note: O(1) is also called **Constant Time,** O(n) is also called  **Linear Time** and O(n2) is also called **Quadratic Time**.

**Time Complexity vs Spatial Complexity**

So far what we have been looking at is time complexity or computational complexity which is how much time it takes for an algorithm to complete its task.

Spatial Complexity talks about how much space or RAM will be used by the algorithm in completing the task.

Like stated before Big O is just a yard stick to make a good tradeoff between different algorithms to find which best fit the problem at hand.

For some problem you might have to sacrifice speed for space and vice versa. We would be using array for out inputs here to make understanding spatial complexity easier.

Under Spatial complexity we have:

**Linear Spatial Complexity:** This means that the algorithm creates another array for every item in the original array input. Let’s say there are 5 items in the array, then the algorithm creates 5 new arrays for each item in the array on completion. This is not very viable as this just uses up too much space, especially in a scenario where RAM space is limited.

**Constant Spatial Complexity:** This means that the same number of arrays specified will be created irrespective of the number of items in the array. So if there 5 arrays were specified to be created, even if there are 5, 10, or even 10,000 items in the array only 5 new arrays will be created.

**Logarithmic Spatial Complexity**: This means we create a diminishing number of arrays for every item in the original array. For example, if we have 7 items in the array we create 5 arrays and so on.

**Quadratic Spatial Complexity**: This means that we create n2 the number of arrays for every input in the original array. This leads to significant resource or space use and is mostly considered a red flag and rarely every used unless in special cases.