Big O

You will see Big O referred to as asymptotic runtime, or asymptotic computational complexity.

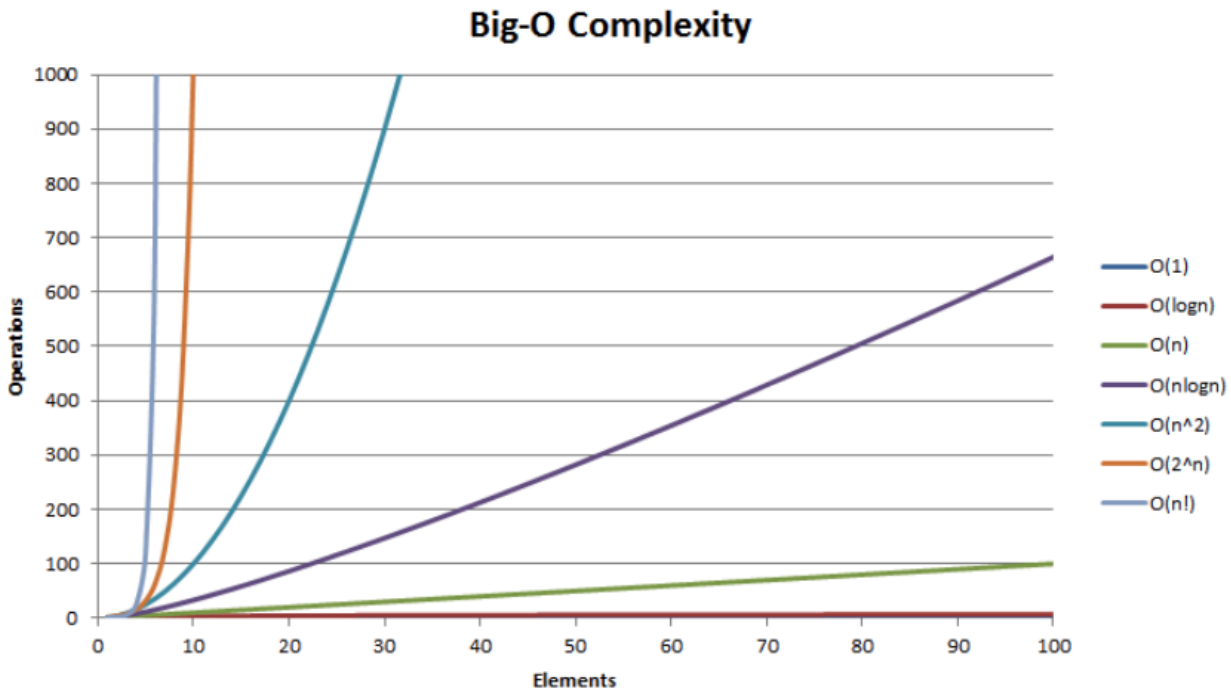
This is a fancy way of describing the **limits of a function**. The “limit” is known as the value that a function approaches as the input increases (or decreases)

Highlights:

- The O is short for “Order of” (aka. growth rate)
- O is the Greek Omicron character
- Used as a precise vocabulary to talk about how code performs.
- Useful for discussing trade-offs between different algo approaches.
- Mathematically describes the complexity of an algorithm in terms of time and space.
- When your code slows down or crashes, identifying parts of the code that are inefficient can help us find pain points in our applications
- Allows to talk formally about how the runtime of an algorithm grows as the inputs grow.
- We want to think about our algorithms in the abstract, not in terms of a specific implementation.

- Looks at **worst case** scenarios (aka. upper bound)
- Described in **classifications** or **runtimes** - O(1), O(log(n)),O(n), $O(n^2)$, O(2^n).
- Cannot have a great time and space complexity, so need to choose which is more important.
- Only cares about order of magnitude (simplifying classification). We forget the details (**drop the constant terms**)
- We don't measure the speed of an algorithm in seconds, we measure the **number of operations** it takes to complete.
- Helps us answer the question, "Can we do better?"

This chart shows you the time complexities with relation to the input sizes:



This table also shows the number of operations required by different time complexities with inputs of size 10, 100, and 1000:

Big-O Notation	n = 10	n = 100	n = 1000
O(1)	1	1	1
O(log n)	3	6	9
O(n)	10	100	1000
O(n^2)	100	10000	1000000
O(2^n)	1024	2^100	2^1000

Big O Expressions (aka runtimes)

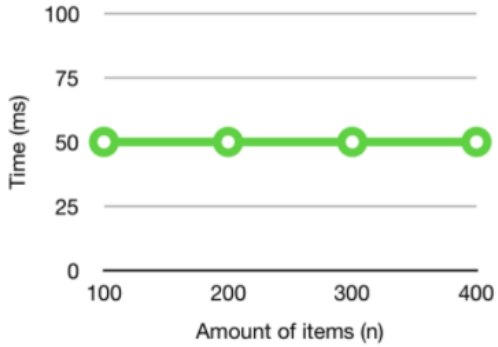
We say that an algorithm is **O(f(n))** if the number of simple operations the computer has to do is eventually less than a constant times **f(n)**, as **n** increases

Green and Yellow are good, Red is **bad**.

Constant	**O(1)**
Linear	**O(n)**
Logarithmic	**O(log n)**
Quadratic	**$O(n^2)$**
Exponential	**$O(2^n)$**

Constant time O(1)

Constant time complexity is the "holy grail".



No matter the size of your input, the algorithm will take the same amount of time to complete.

- declarations/assignments (let num = 0)
- comparisons (num > 0)
- object accesses (arr[0])
  simple mathematical operations (+, -, &, /)

```
const grader = score => {
   if (score < 60) {
       return "Fail!";
   } else if (score > 60) {
       return "Pass!";
   } else {
       return "Living on the edge!";
   };
}
```

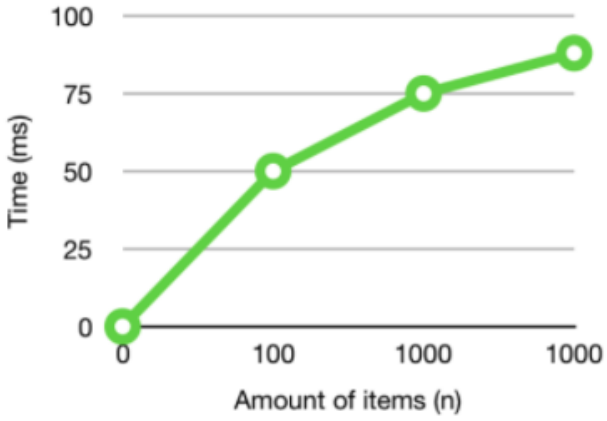
Even though we check multiple conditions before returning, the rate of growth is constant (aka constant order of growth).

We know the upper bound, or **worst-case scenario**, in advance, and we know it will not change.

Logarithmic time O(log(n))

While logarithmic time complexity algorithms do take longer with larger inputs, running time increases **slowly**.

If an algorithm has logarithmic time complexity, it means that the size of the input we are considering gets **split into half** with each iteration.



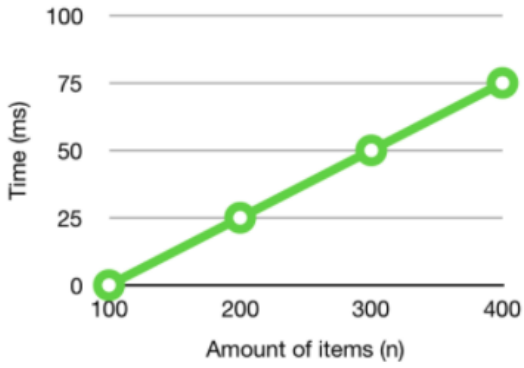
Let's say that we have a function that takes 1 second to execute if the input size is 100. With a logarithmic runtime, it would then take 2 seconds if the input size is 1000, and 3 seconds if the input size is 10,000. The bigger the input size gets, the smaller the difference in runtime!

Let's say that we're looking for the value of 7 in a sorted array. With a specific searching (e.g. binary search - sorted array) algorithm, we split the input array on every round, and only check the side of the half (displayed as the dotted line) where the value could potentially be.

Algorithms like a guessing game, which are similar to a binary search, splits the problem in half at each iteration, and generally have a logarithmic growth rate.

After every execution, the size of the array gets split (approximately) in half!

Linear time O(n)



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

Given input a and input b, where b is twice as large as a, it will take a linear algorithm 2 times as long to process b compared to a.

The rate of growth of an algorithm scales in **direct proportion to the input**.

```
const nums = [1,2,3,4,5,6,7,8,9,10];

const sumHarder = arr => {
   let sum = 0;
   for (let i = 0; i < arr.length; i++) {
       sum += arr[i];
   }
   return sum;
}

const result = sumHarder(nums);
```

Our function needs to perform one operation for every input, so the order of our algorithm is O(n) or linear time complexity.

We can improve the time complexity of our algorithm by re-writing it to this:

```
const sumSmarter = arr => arr.length * (arr.length + 1) / 2;
```

The order of the function now is O(1) because regardless of the length of the array, the function will always perform the same number of operations.

Quadratic time O(n^k) - aka Polynomial

An algorithm with quadratic time complexity has a running time that would be some input size n raised to some constant power k.

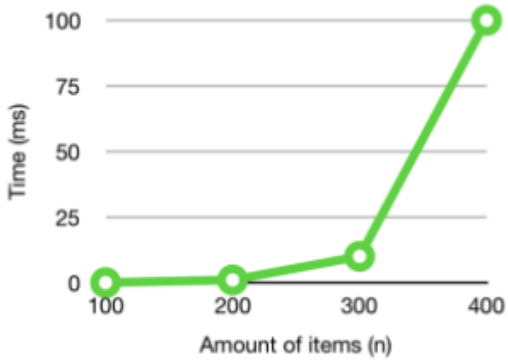
”Quadratic” is the fancy adjective used to describe squaring, or raising to the power of 2. It’s from the Latin ‘quadrus’, which means, you guessed it, square.

If an algorithm is in the order of O(n^2), the number of operations it performs scales in proportion to the square of the input.

A common example of an algorithm with quadratic time complexity is by looping over an array, and comparing the current element with all other elements in the array.

We have to loop over n elements, and for every element, we again have to loop over n elements.

For every element in the array, we loop over the array and compare their values. This ends up being n * n, which is n^2.



Example #1:

```
const fruit = [
    "strawberry",
    "pear",
    "orange",
    "banana",
    "pineapple",
    "peach",
    "apple",
    "orange",
    "raspberry"
];

const matcher = array => {
   for (let i = 0; i < array.length; i++){
       for (let j = 0; j < array.length; j++){
           if (i !==j && array[i] === array[j]){
               return `Match found at ${i} and ${j}`;
           }
       }
   }
   return 'No matches found';
}
```

Our function iterates over an array. For each element in the array, we then iterate over the array again. If the array indices are not the same and the elements at the indices are the same, then we return the locations of the matched fruit. If no matches are found, we return disappointment.

How many operations are performed?

Our outer loop is performing n iterations. Our inner loop also performs n iterations. But it performs n iterations for every iteration of the outer loop.

When i is 0, we iterate j 10 times. When i is 1, we iterate j 10 times.
This is O(n * n), or O(n^2).

Example #2:

```
const summer = (num, t = false) => {
   let sum = 0;

   if (t === 'quadratic'){
       for (i = 0; i <= num; i++){
           for (j = 0; j < num; j++){
               sum += j;
           }
       }
   } else if (t === 'linear') {
       for (i = 0; i <= num; i++){
           sum += i;
       }
   } else {
       return num * (num + 1) / 2;
   }
   return sum;
}
```

If our first condition is met, we run the nested iterations and the order of our function is **O(n^2)**.

If our second condition is met, we run one iteration and the order of our function is **O(n)**. If neither of the conditions are met, our function is **O(1)**. What is the actual order of our function?

We could calculate this as O(n^2 + n + 1). Why would we not do this?

When using Big O to measure the rate of growth of our algorithms, what do we really want to know? We want to know the **worst-case scenario**. Here, it's **n^2**.

In Big O, **we drop constant terms** and **drop the non-dominant term**, because it doesn't provide any meaningful additional information. Whether

or not n influences the rate of growth of n is irrelevant. With or without it, our algorithm is still quadratic.

What we really want to know is the order of our function, not the details of its specific implementation. So the example above is **O(n^2)**.

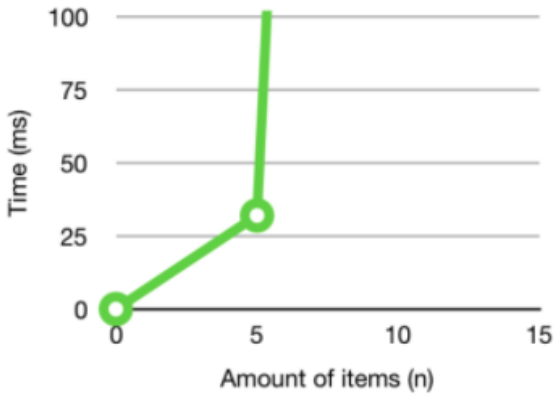
Exponential time O(2^n)

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

For an input of size 2, an exponential time algorithm will take 2^2 = 4 time. With an input of size 10, the same algorithm will take 2^10 = 1024 time, and with an input of size 100, it will take 2^100 = 1.26765060022823 * 1030 time. Yikes!

So the runtime of this type of algorithm gets doubled after every addition in the input.

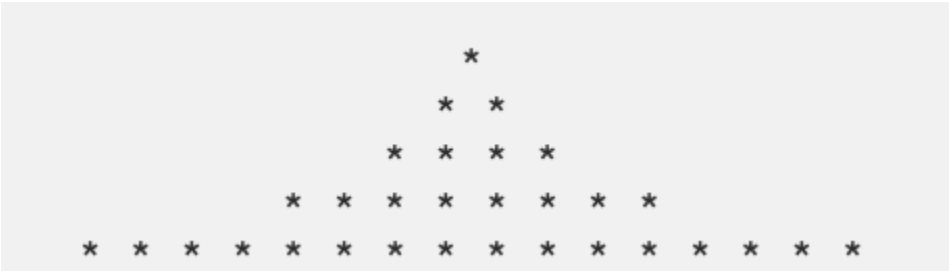
If 5 items took 30 seconds, 6 items would take 60 seconds.



```
function countTriangle(layers) {
    let ticks = 1;
    let count = 0; // the number of dots we've counted so
    let layer = 0; // the current layer we're on
    let lastLayer = 1; // the number of dots we counted i
    while (layer < layers) {
        ticks++;
        let newLayer = 0; // the number of dots we've cou
        for (let i = 0; i < lastLayer; i++) {
            ticks++;
            newLayer += 2;
        }
        lastLayer = newLayer;
        count += lastLayer;
        layer++;
    }
    return {
        result: count,
        ticks: ticks
    };
}

countTriangle(2);
countTriangle(4);
countTriangle(16);
```

The following countTriangle is meant to count the number of points in a triangle that looks like this:



In this function, we count the number of dots in a triangle with a given number of layers. We start at the top layer of the triangle, which is the 0th layer and has 1 dot (or you can think of it as 2^0 = 1).

As you move to the next layer, the number of dots increases by a power of 2. So, in the 1st layer, the dots you will count will be 2^1 = 2.

In the 2nd layer, the number of dots will be 2^2 = 4. By the time you're at the nth layer, the number of dots would be 2^n. Therefore, as the input size increases, the number of operations to count the dots increases exponentially making the running time of this algorithm O(2^n).

Space Complexity

Space complexity is a measure of the amount of **working storage** that an algorithm needs. It answers the following question: in the worst case, how much memory is needed at any point in the algorithm?

Space complexity includes both **auxiliary space** and the space used by the input. Auxiliary space is the temporary or extra space used by the algorithm while it is being executed.

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. Below are a few examples of expressing space complexity using big O notation, starting from slowest space growth (best) to fastest (worst):

- O(1): Constant complexity means that the algorithm takes the same amount of space regardless of the input size.

- O(log n): Logarithmic complexity means that the algorithm takes space proportional to the log of the input size.

- O(n): Linear complexity means that the algorithm takes space directly proportional to the input size.

- O(n log n) Log-linear or quasilinear complexity (also called linearithmic) means that the space complexity grows proportionally to the input size and a logarithmic factor.

- $O(n^2)$: Quadratic complexity means that the space complexity grows proportionally to the square of the input size.

Constant space (O(1))

```
function sum(left, right) {
  return left + right;
}
```

In this function there are variables used in memory:

- left
- right
- return value

In JavaScript, a single number occupies 8 (eight) bytes of memory.

In the above function, there are three variables assumed to be numbers. Therefore, this algorithm always takes 24 bytes of memory to complete (3 * 8 bytes). As a result, the space complexity is constant, so it can be expressed in big O notation as O(1).

Logarithmic space (O(log n))

Logarithmic space complexity (O(log n)) is the next best thing after constant space. Although logarithmic space complexity algorithms do require more space with larger inputs, space usage increases slowly.

For example, imagine that you have a function, quickSort(), which takes 128 bytes to process an input of size 10. When you increase the input by 10 times, to 100, the space required only grows to 256 bytes. When you increase the input size to 1,000, the space only grows to 384 bytes.

Linear space (O(n))

Algorithms with linear space complexity (O(n)) use an amount of space directly proportional to the size of the input. Every time that you double the size of the input, the algorithms will require twice as much memory.

```
function sum(numbers) {
  let total = 0;
  for (const number of numbers) {
    total += number;
  }
  return total;
}
```

In this function there are variables used in memory:

- numbers: The space taken by the array is equal to **8n** bytes, where **n** is the length of the array
- number: An 8-byte number
- total: An 8-byte number

The total space needed for this algorithm is **8n+8+8** bytes. The highest order of n in this equation is just n. Thus, the space complexity of the above is O(n).

Space-time complexity tradeoff

An algorithm's efficiency is a combination of its time and space complexity. So an efficient algorithm is one that is fast and that takes the least amount of memory possible.

Space and time complexity are often linked. Usually, increasing speed leads to greater memory consumption, and vice versa. However, this isn't always the case; sometimes, space and time complexity aren't correlated.

Tips on simplifying Big O expressions

- Constants don't matter
  $O(2n) \Rightarrow O(n)$
  $O(5) \Rightarrow O(1)$
  $O(13n^2) \Rightarrow O(n^2)$

- Smaller terms don't matter
  $O(n + 5) \Rightarrow O(n)$
  $O(1000n + 50) \Rightarrow O(n)$
  $O(n^2 + 5n + 8) \Rightarrow O(n^2)$

- Arithmetic operations and variable assignments are considered constant => O(1)

- Accessing elements in an array (by index) or object (by key) is constant => O(1)

- In a loop, the complexity is the length of the loop times the complexity of whatever happens inside the loop

Practice:

Simply the following Big O expressions:

1. O(n + 10) - O(n)
2. O(n + n + n + n) - O(n)
3. O(25) - O(1)
4. O(n^2 + n^3) - O(n^3)
5. O(1000 * log(n) + n) - O(n)
6. O(1000 * n * log(n) + n) - O(n)
7. O(2^n + n^2) - O(2^n)
8. O(5 + 3 + 1) - O(1)

Determine the Big O **time and space complexity** of these algorithms:

#1 - time: O(1), space: O(1)

```
function isEven(value) {
    if (value % 2 === 0) {
        return true;
    }
    else {
        return false;
    }
}
```

#2 - time: O(n^2), space: O(1)

```
function areYouHere(arr1, arr2) {
    for (let i = 0; i < arr1.length; i++) {
        const el1 = arr1[i];
        for (let j = 0; j < arr2.length; j++) {
            const el2 = arr2[j];
            if (el1 === el2) return true;
        }
    }
    return false;
}
```

#3 time - O(n), space - O(1)

```
function doubleArrayValues(array) {
    for (let i = 0; i < array.length; i++) {
        array[i] *= 2;
    }
    return array;
}
```

#4 time - O(n), space - O(1)

```
function naiveSearch(array, item) {
    for (let i = 0; i < array.length; i++) {
        if (array[i] === item) {
            return i;
        }
    }
}
```

#5 time - O(n^2), space O(1)

```
function createPairs(arr) {
    for (let i = 0; i < arr.length; i++) {
        for(let j = i + 1; j < arr.length; j++) {
            console.log(arr[i] + ", " +  arr[j] );
        }
    }
}
```

#6 time - O(n), space O(n)

```
function compute(num) {
    let result = [];
    for (let i = 1; i <= num; i++) {

        if (i === 1) {
            result.push(0);
        }
        else if (i === 2) {
            result.push(1);
        }
        else {
            result.push(result[i - 2] + result[i - 3]);
        }
    }
    return result;
}
```

#7 time - O(2^n), space - O(n)

```
function efficientSearch(array, item) {
    let minIndex = 0;
    let maxIndex = array.length - 1;
    let currentIndex;
    let currentElement;

    while (minIndex <= maxIndex) {
        currentIndex = Math.floor((minIndex + maxIndex) / 2);
        currentElement = array[currentIndex];

        if (currentElement < item) {
            minIndex = currentIndex + 1;
        }
        else if (currentElement > item) {
            maxIndex = currentIndex - 1;
        }
        else {
            return currentIndex;
        }
    }
    return -1;
}
```

#8 time - O(1), space - O(1)

```
  function findRandomElement(arr) {
      return arr[Math.floor(Math.random() * arr.length)];
  }
```

#9 time - O(n), space - O(1)

```
  function isWhat(n) {
      if (n < 2 || n % 1 !== 0) {
          return false;
      }
      for (let i = 2; i < n; ++i) {
          if (n % i === 0) return false;
      }
      return true;
  }
```

Resources

Big O in 5 minutes:

https://www.youtube.com/watch?v=__vX2sjlpXU

Big O parts 1 & 2:

https://www.youtube.com/watch?v=HfIH3czXc-8&t=1108s

https://www.youtube.com/watch?v=zo7YFqw5hNw&t=691s

Simplifying Big O notation:

- Product rule (drop constants)
- Sum rule (keep the largest term)

https://www.youtube.com/watch?v=sTtOu10mW5c