**Big-O Notation**

**Big O notation** is used to classify algorithms according to how their running time or space requirements grow as the input size grows.

To analyze the performance of an algorithm, we use Big O Notation.

Big O Notation can give us a high level understanding of the time or space complexity of an algorithm.

Big O Notation doesn’t care about precision, only about general trends (linear? Quadratic? Constant?)

The time or space complexity ( as measure by Big O) depends only on the algorithm, not the hardware used to run the algorithm.

**Example – 1 : //Write a function that calculates the sum of all numbers from 1 up to (and including) some number n.**

**//1st approach**function addUPTo(n){ ***//Function Declaration***  
 let total = 0; //Execute: 1 time  
 for(let i= 1; i<=n; i++){ //Execute: n time, n is the input size  
 total+=i; //Execute: n time, n is the input size  
 }

return console.log(total); //Execute: 1 time  
}

***//now() method of performance object*** **used to get the execution time interval of a program**   
let t1 = performance.now(); ***//Time Start***

addUPTo(1000000); ***//Function Call***addUPTo(100000000); ***//Function Call***addUPTo(10000000000); ***//Function Call***

let t2 = performance.now(); ***//Time End***  
console.log(`Time Elapsed: ${(t2-t1)/100} seconds`);

**Output: // Time Increasing, On Increasing Input size**500000500000   
Time Elapsed **0.07**seconds *//For Input: 1000000*  
  
5000000050000000   
Time Elapsed **1.97** seconds *//For Input: 100000000*  
  
50000000000067860000   
Time Elapsed **239.85** seconds *//For Input: 10000000000*

**Example – 1 : //Same Question with Same size of Inputs but different 2nd Approach**

**//2nd approach**function addUPTo(n){ ***//Function Declaration***  
 return console.log(n \* (n + 1) / 2); //Execute: 1 time  
}

***//now() method of performance Object*** **used to get the execution time interval of a program**   
let t1 = performance.now(); ***//Time Start***

addUPTo(1000000); ***//Function Call***addUPTo(100000000); ***//Function Call***addUPTo(10000000000); ***//Function Call***

let t2 = performance.now(); ***//Time End***  
console.log(`Time Elapsed: ${(t2-t1)/100} seconds`);

**Output: // Output Same and Time is Constant on Increasing Input Size**500000500000   
Time Elapsed **0.002**seconds *//For Input: 1000000*  
  
5000000050000000   
Time Elapsed **0.002** seconds *//For Input: 100000000*  
  
50000000000067860000   
Time Elapsed **0.002** seconds *//For Input: 10000000000*

*Conclusion:*As per the 1st approach, Time would be like this -  
C1 + n + n + C2 = C + n

So, **Time Complexity would be O(n), which is called Linear Time Complexity**.   
where, C1 & C2 is constant (1) and n is the input size.   
Since, C is constant so it will be neglected.  
Best Case: O(1)  
Worst Case: O(n)

As per the 2nd approach, Time would be C.

So, Time Complexity would be **O(1), which is called Constant Time Complexity**.   
where, C is constant (1) for n input size.   
Best Case: O(1)  
Worst Case: O(1)

Hence, **2nd approach would be best** in order to solve this problem. Because in this case, Time complexity is O(1) in worst case which is constant time complexity whereas Time complexity is O(n) in the worst case for the 1st approach which is Linear Time Complexity.

**Here,** We got the best algorithm by comparing two of them. Not exactly because of time.   
Because the problem with time is what,

* Different machines will record different times.
* The same machine will record different times!
* For fast algorithms, speed measurement may not be precise enough.

**So, if not time then What?**

Rather than counting seconds, which are so variable.

We counted the number of simple operations the computer has to perform, in order to get the best approach / algorithm to solve the problem. Because this is what that remains constant no matter what computer we’re on.  
Here, the times will always be determined by the number of operations.

That’s why we can use that rather than the exact time, We can just focus on the number of simple operations the computer has to perform.

Now, We shouldn’t get bogged down in the detains of counting every minute operation, Because all that matters is the **general terms** of things (a very very fuzzy overview). We only care abou the general trend.  
And this is what we’re going to see a lot with Big O, we’re focused on the big picture.

***So We can say that***, **Big O Notation** is a way to formalize fuzzy counting.

It allows us to talk formally about how the runtime of an algorithm grows as the inputs grow.

It’s a way of describing the relationship between the input to a function or as it grows and how that changes the runtime of that function.  
In short, The relationship between the input size anad then the time relative to that input.

We can 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.

So we’re describing the relationship between the input of function f(n) and then the run time

* f(n) could be linear (f(n) = n)  
  *Meaning*, As n scales (grows), the input, The runtime scales as well.
* f(n) could be quadratic (f(n) = n2 )  
  *Meaning*, As n scales (grows), the input, The runtime square is related to the square of n.
* f(n) could be constant (f(n) = 1)  
  *Meaning*, As n scales (grows), It doesn’t really have an impact because runtime is always constant, which we simplified down to 1.
* f(n) could be something entirely different!

As per the above mentioned question, the time complexity of 1st approach is O(1) where always having 3 operations (\*, +, /) for n input size.

Whereas in the 2nd approach, The Number of operations is (eventually) bounded by a multiple of n (say, 10n, 50n etc…). It doesn’t actually matter if it’s 1n, 5n, 10n or 50n, because at the end of the day, we simplify it down just to N.  
Hence, the time complexity of 2nd approach is O(n).

So when we see a big O and then in parentheses we see 1 or see N or N2 or nlogn or logn, telling us that as n grows as the input to this function grows, it will reflected in the runtime (It will not reflected in the runtime in the case of Constant time complexity).

**Note:**When we talk about Big ), we’re talking about the worst case scenario. So we’re talking about basically the upper bound for runtime.

**Example: - 2 //First Count Up then Down for n input**function countUpandDown(n){

console.log('Going Up');

for(let i=0; i<n; i++){ ***//Iterate n times***  
 console.log(i);  
 }

console.log("At the Top! \n Going Down...."); ***//Iterate n times***

for(let j=n-1;j>=0; j--){  
 console.log(j);  
 }

console.log("Back Down. Bye!");

}

countUpandDown(10); **//Input: n (as 10)**

**Time Complexity (Runtime) in Big O:**n + n => **O(n) //Linear Time Complexity**

**Example: - 3 //PrintAllPairs**function PrintAllPairs(n){

for(let i=0;i<n;i++){ ***//O(n) operation***

for(let j=0;j<n;j++){ ***//O(n) operation***  
 console.log(i,j);

}

}

}

PrintAllPairs(2); ***//Input: n (as 2)***

**Output:**0,0  
0,1  
1,0  
1,1

**Time Complexity (Runtime) in Big O:**n \* n => **O(n2)   
//Quadratic Time Complexity**O(n) operation inside of an O(n) operation.

**Example: - 4 //logAtLeast**function logAtLeast(n){

for(let i=1; i<=Math.max(5,n); i++){ ***//O(n) Operation***  
**//max() is the method of Math Object. Print from atleast 5 up to max n for any value of input n. (Go only with maximum Value or say loop iterate multiple times for max value)**

console.log(i);  
 }

}

logAtLeast(2); ***//Input: n (as 2)***

**Time Complexity (Runtime) in Big O:**n => **O(n) //Linear Time Complexity**

**Example: - 5 //logAtMost**function logAtMost(n){

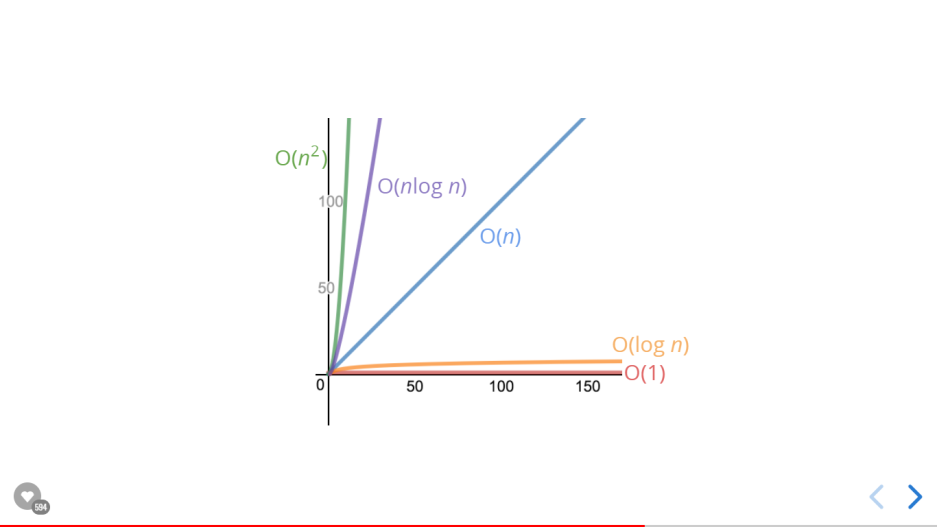
for(let i=1; i<=Math.min(5,n); i++){ ***//O(1) Operation***  
**//min() is the method of Math Object. Here, Print number from 1 to atmost 5 for any value of input n. (Go only with minimum Argument i.e. 5 or say loop iterate multiple times for min value). Note: [n>0]**

console.log(i);  
 }

}

logAtLeast(2); ***//Input: n (as 2)***

**Time Complexity (Runtime) in Big O:**n => **O(1) //Constant Time Complexity**

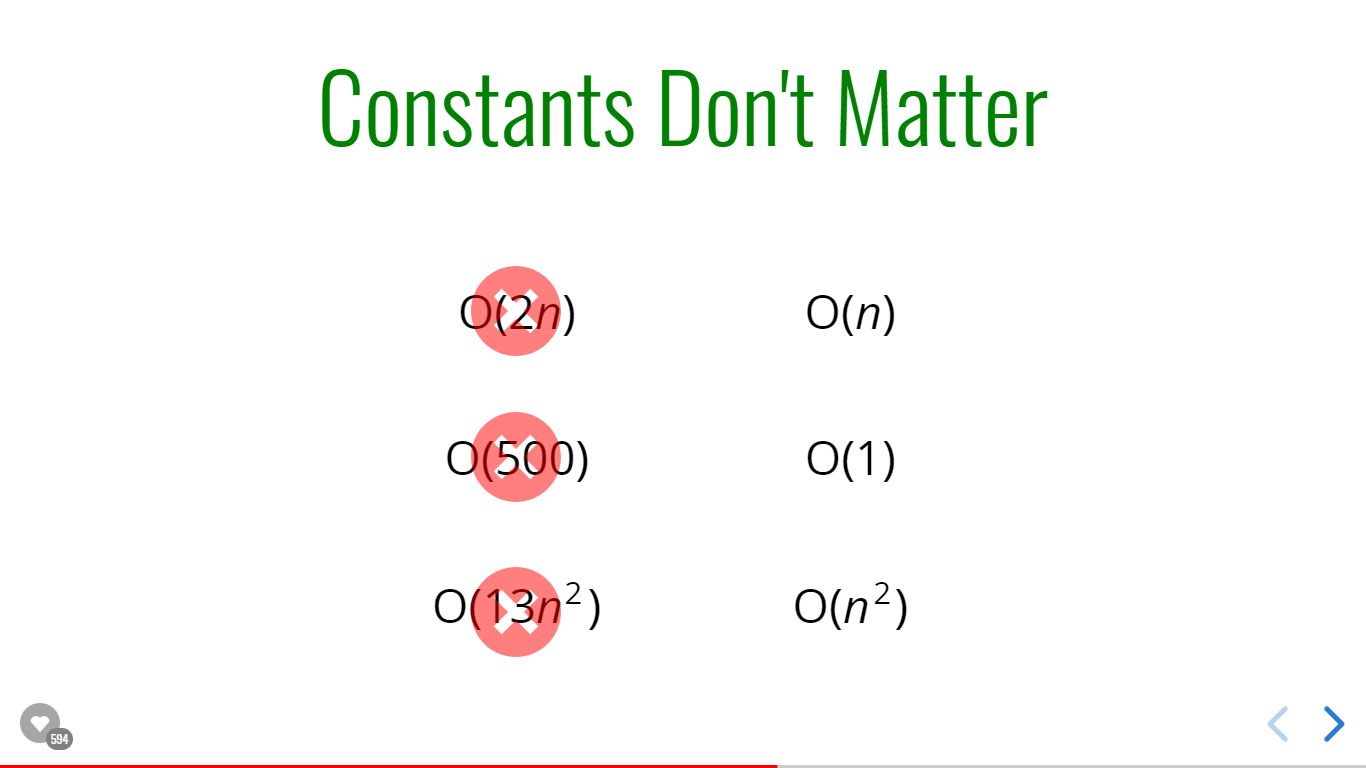
****

**Rules of Thumb for Time Complexity in Big O Notation / Expression:-**

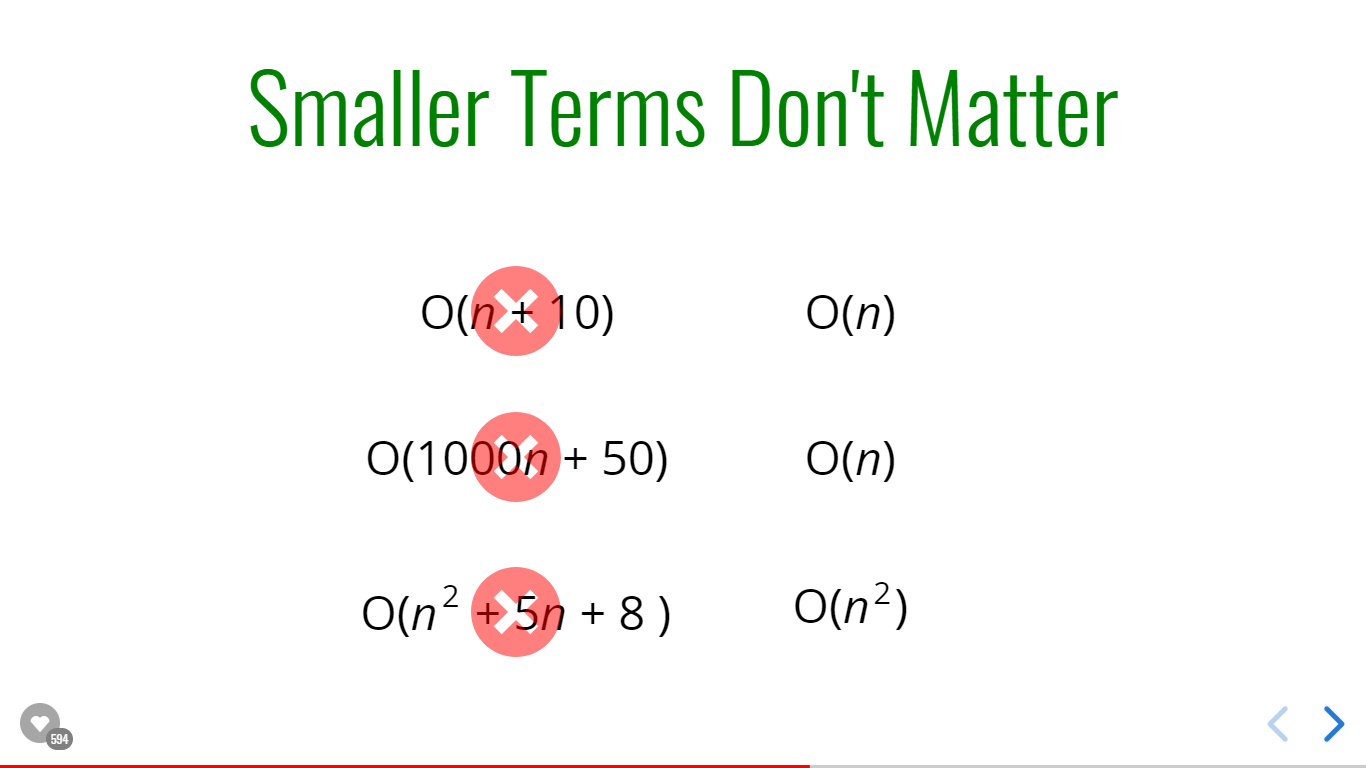
When determining the time complexity of an algorithm, there are some helpful rule of thumbs for big O expressions.

These rules of thumb are consequences of the definition of big O notation.

**Constant Doesn’t Matter:**

****

**Smaller Terms Don’t Matter:**

****

**Big O Shorthands:**

Analyzing complexity with big O Can get complicated.

There are several rules of thumb that can help

These rules won’t **ALWAYS** work. But are helpful starting point.

1. Arithmetic Operations are constant.
2. Variable assignment is constant.
3. Accessing elements in an array(by index) or object (by key) is constant .
4. In a loop, the complexity is the length of the loop times the complexity of whatever happens inside of the loop.

**Space Complexity using Big-O Notation**

Till now, we’ve been worrying about time, about how fast algorithms run with the runtime. That’s called time complexity. We’ve been analyzing the runtime of an algorithm as the size of the input increases.

Now, let’s talk about what happens to the space that an algorithm takes up as the size of the input increases.

We can also use big O notation to analyze **space complexity:**How much additional memory do we need to allocate in order to run the code in our algorithm?

**Auxiliary Space Complexity,** refers to space required by the algorithm, *not including space taken up by the inputs already*.

We don’t care about the included space. Here, we’re are going to say that as N grows, we assume that the inputs n is going to grow.

Unless otherwise noted, when we talk about space complexity, technically we’ll be talking about auxiliary space complexity.

**Rules of Thumb for Space Complexity in Big O Notation / Expression:-**

* Most primitives (Booleans, numbers, undefined, null) in JavaScript are constant space.  
  So, it doesn’t matter what the size of the input is, if the number is 1 or 1000, we can consider it constant space. It doesn’t matter if a Boolean is true or false takes up the same amount of space.
* Strings require O(n) space (where n is the string length).  
  Let the string we’ve of 50 characters, The string takes up 50 times more space than a single character string.
* Reference types are generally O(n), where n is the length (for arrays) or the number of keys (for objects.  
  There’s not really a length technically, but so if N is the length of an array is 4 compared to another one i.e. 2, it takes about twice as much space as the shorter array.

**Example: - 1 //Determine Space Complexity**function sum(arr) {

let total = 0;

for (let i = 0; i < arr.length; i++) {

total += arr[i];

}

return total;

}

**Space Complexity: O(1)**Since, We’ve to concern about space complexity. Here, no matter what the array length is, we have one variable called total (One Number). And then we’ve a second declaration inside the for loop with variable called i (Another Number). So that’s space.  
No matter what is the size of the array of N. In this case, as arr (array) grows, it might 1000 items, or Millions items, It doesn’t have an impact on the space that’s taken up because we only have these two variables and they exist no matter what. We’re not adding new variables based off of the length. We’re adding to the total variable, but we’re not making a new variable.

So that really just means we have constant space **O(1).** It’s always the same no matter the size of the input.

**Note:**As per **Auxiliary Space Complexity,** We’re not going to consider the space already taken up the array (arr), In the above example of space complexity.

**Example: - 2 //Determine Space Complexity**function double(arr) {

let newArr = [];

for (let i = 0; i < arr.length; i++) {

newArr.push(2 \* arr[i]);

}

return newArr; **//O(n)**

}

double([1,2,3]);

**Output:**[2,4,6]

**Space Complexity: O(n)**Here, it’s making a new array. As the array length grows, as the input approaches *infinity* (just for assumption), this new array getting longer and longer and longer directly in proportional to the length of the input. So if the array is 10 items here, we’re storing 10 items in a new array. If this is 50 items we’re storing 50 items here in a new array and returning that.

So the space that’s taken up by new array (newArr) is directly proportionate to and to the input array.

So n numbers (n size), we got **O(n)** space

Time Complexity in **Logarithms** in Big O Notations

We've encountered some of the most common complexities: O(1), O(n), O(n  )

2

Sometimes big O expressions involve more complex mathematical expressions

One that appears more often than you might like is the logarithm!

**What is a log?**(Log) is the short form of Logarithm. A logarithm is the inverse of exponentiation.

So just like division and multiplication are pair logarithms and exponents (exponentiation) are a pair.

**Ex:**log2 (8) = 3 (Read as, log base2 of 8, equals to 3) 🡪 23  = 8

Syntax:  
log2 (value) = exponent 🡪 2exponent = value

They are the inverse exponentiation and logarithms.

**Note:**

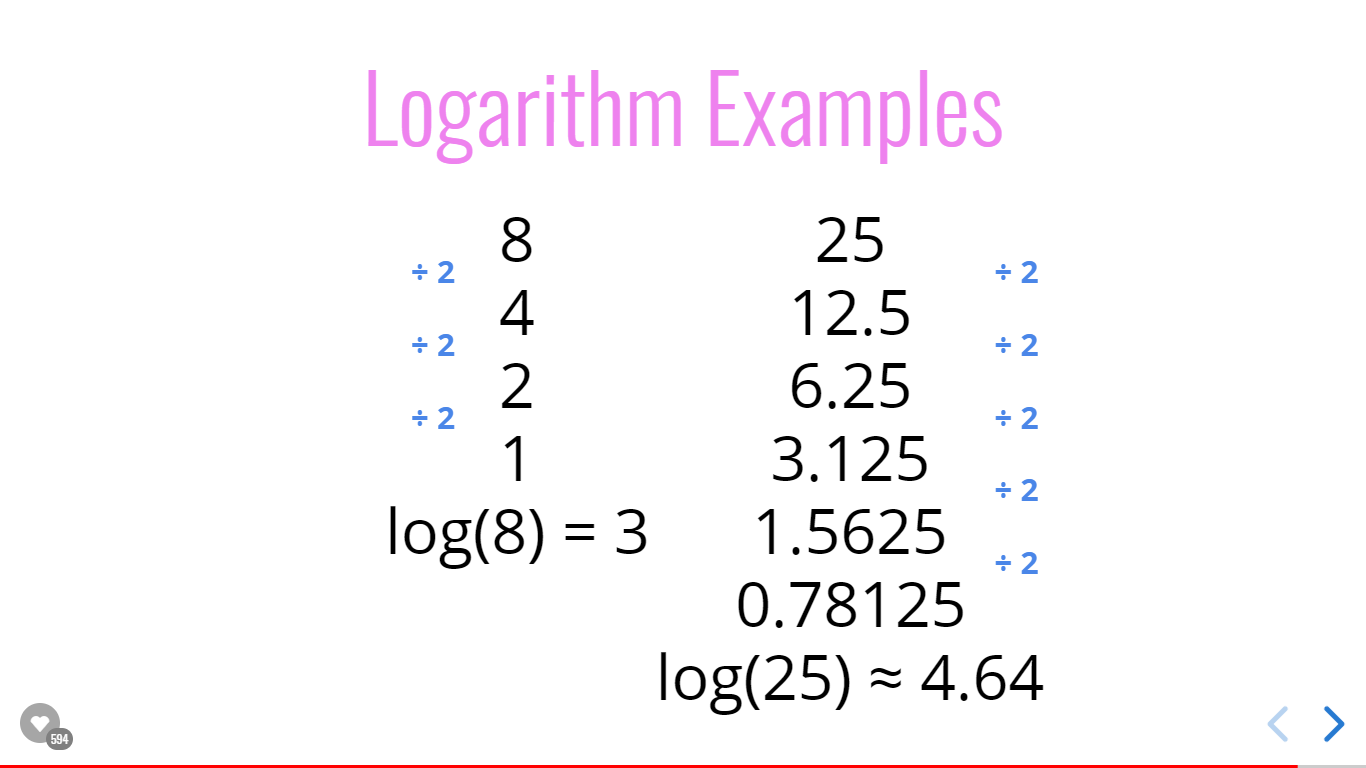
* Now, logarithms aren’t always working with base2. The most common ones, though, are the binary logarithms, which is log base 2 and base 10.
* The subscript of log i.e. base, doesn’t really matter at the end of the day because if we’re comparing the graph of a constant time (1) and quadratic time (n2) time and log(n) time, it doesn’t really matter if it’s log base 2, log base 3 or log base 10 etc.   
  It’s going to be the general trend that we care about.
* We’ll omit the 2.  
  log === log2

**Rules of Thumb for Logarithm in Big O Notation:-**

The logarithm of a number roughly measures the number of times you can divide that number by 2 **before you get a value that’s less than or equal to 1.**

**Rule of thumb**: Just to calculate roughly the binary logarithm of number, roughly measures the number of times you can divide the number by 2 before you get a value that’s less than or equal to 1.

Example:

**  
Where Majorly comes Logarithm comes into play?**Certain Searching algorithms have logarithmic time complexity.

Efficient sorting algorithms involve logarithms.

Recursion sometimes involves logarithmic space complexity.