Big-O Notation

Time Complexity Analysis
Algorithmic Efficiency

Code Platoon
Code Camp for Veterans

How your algorithm's performance is affected with a change in the size of an input.

Algorthimically

- Execution Speed
- Resource optimization (e.g. memory usage)

Big-O Notation

O(n)

Size of an input (ex: size of an array)

Also called Landau's symbol, is a symbolism used in complexity theory, computer science, and mathematics to describe the asymptotic behavior of functions. Basically, it tells you how fast a function grows or declines.

Big-O Notation

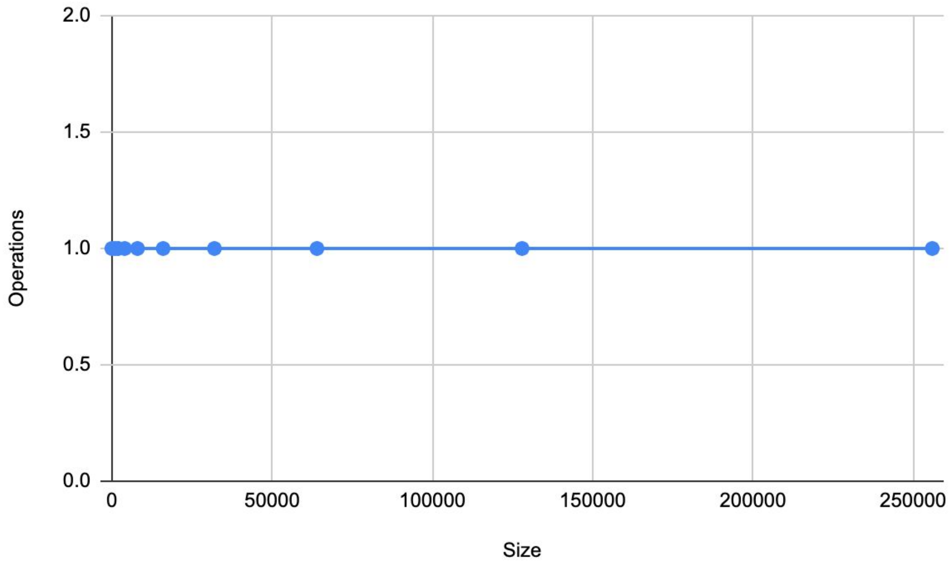
Common Complexities

O(1)	Constant Time	**Faster Algorithm**
O(log n)	Logarithmic Time	
O(n)	Linear Time	
O(n log n)	Log-Linear Time	
O(n^2)	Quadratic Time	
O(2^n)	Exponential Time	**Slower Algorithm**

Big-O Notation (constant time)

O(1)

```
arr = [1,2,3,4,5,6,7,8,9,10]
arr[5]
```



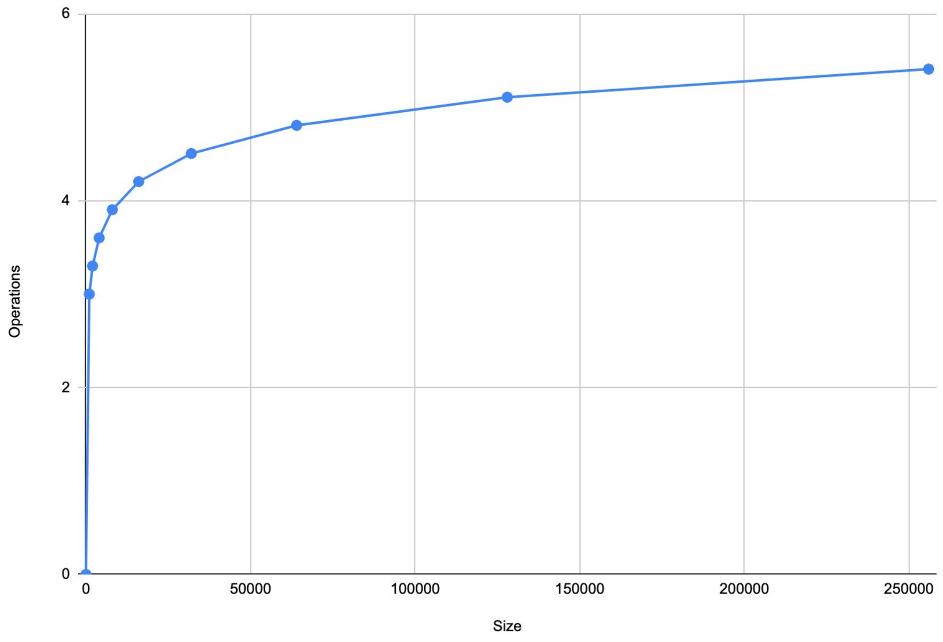
Ex: Accessing single element in an array

Big-O Notation (logarithmic time)

O(log n)

```
arr = [1,2,...256000]
```

Ex: Binary Search

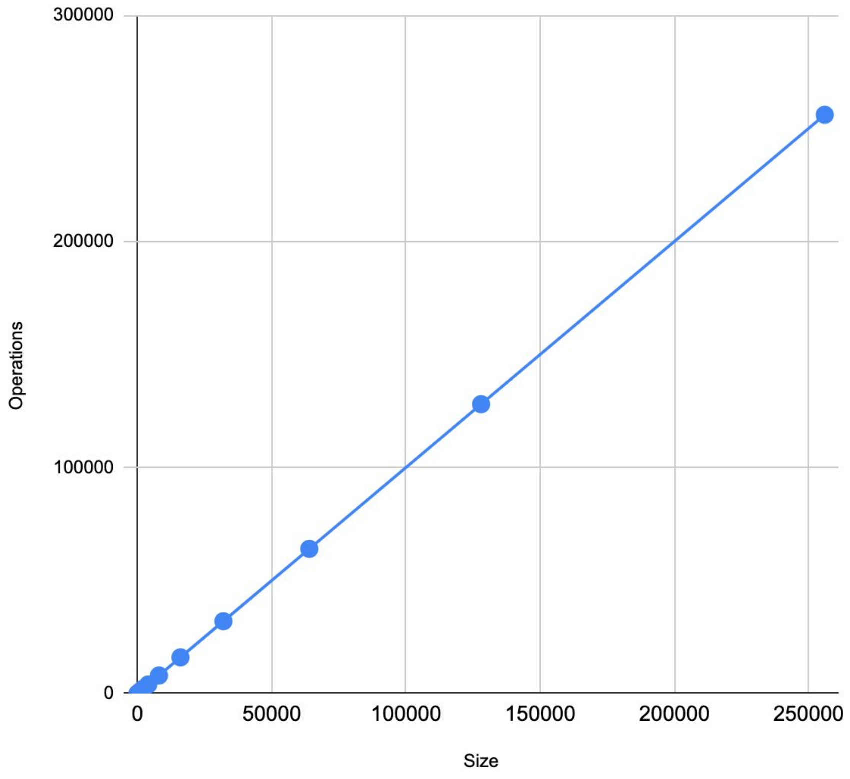


Big-O Notation (linear time)

O(n)

```
arr = [1,2,3,4,5,6,7,8,9,10]
```

Ex: Linear Search

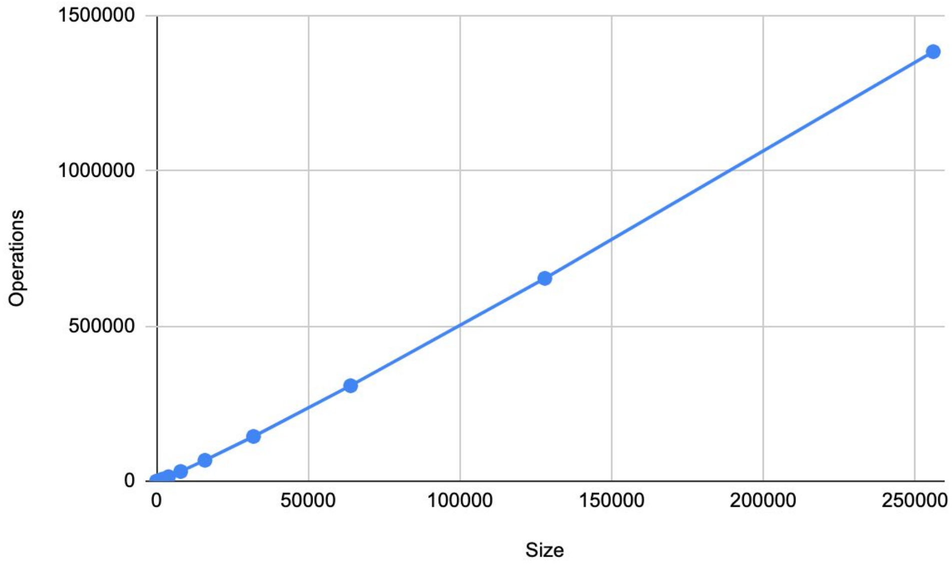


Big-O Notation (log linear time)

O(n log n)

```
arr = [1,2,3,4,5,6,7,8,9,10]
```

Ex: Merge Sort

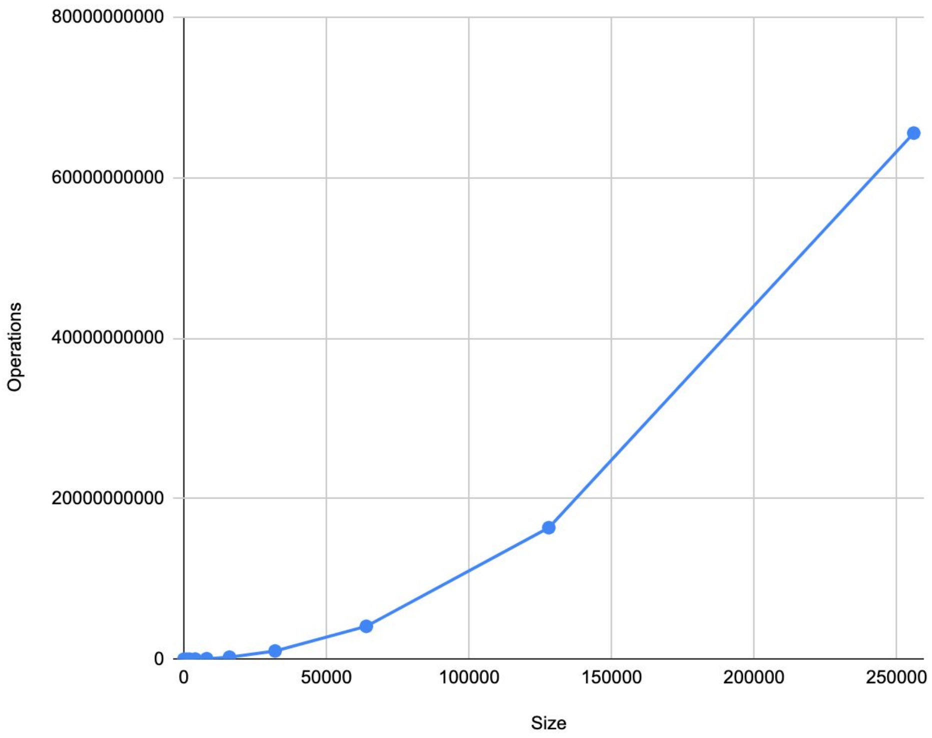


Big-O Notation (quadratic time)

O(n^2)

```
arr = [1,2,3,4,5,6,7,8,9,10]
```

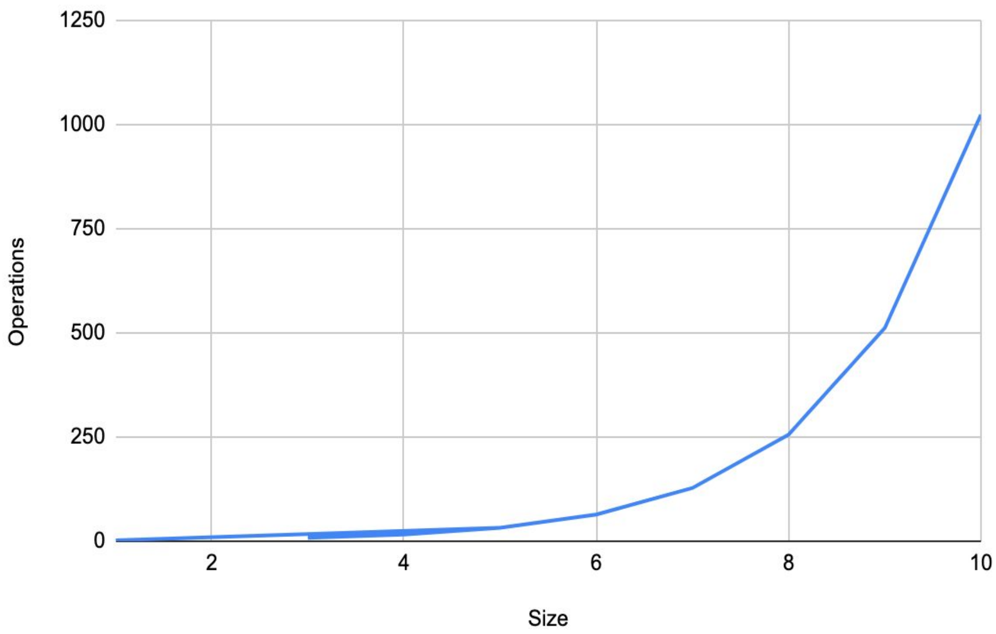
Ex: Finding Duplicate Values (nested for loop of same array)



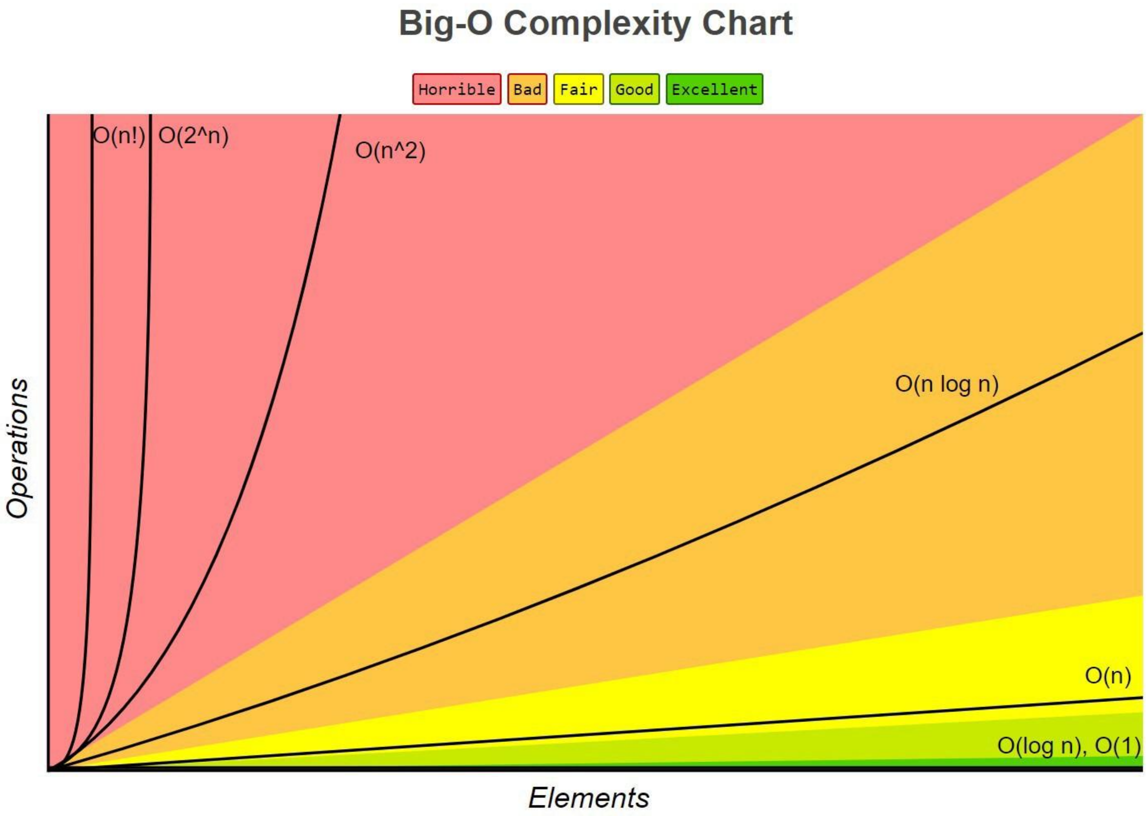
Big-O Notation (exponential time)

O(2^n)

Ex: Finding the nth value in the fibonacci sequence



Big-O Notation



Data Structures

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Things to remember

- Drop Constants (two separate for loops of same array)
    - O(n + n) = O(2n) => O(n)
- Use Worst Case Scenario
    - 1 for loop O(n)
    - 1 nested for loop O(n^2) <- worst case
    - O(n^2)
- Searching through dictionairies or objects are faster than searching through lists or arrays because objects don't keep track of indices
- Wherever possible, avoid nested loops
- There's usually a way to avoid poor Big O