iOS Course

Sixth class - Big O notation

Algorithm

In mathematics and computer science, an algorithm is a finite sequence of well-defined, computer-implementable instructions, typically to solve a class of problems or to perform a computation.

Does the same program run at the same speed on different computers?

How can we say an algorithm is good?

Speed

Memory used

Resources used

Number of operations

Big O

Used for...

In computer science, **Big O notation** is used to classify algorithms according to how their run time or space requirements grow as the input size grows.

Different functions with the same growth rate may be represented using the same notation

- **Big O notation** is the language we use to articulate how long it takes an algorithm to execute

- **Big O notation** describe the shape of a curve according to the quantity of information to process

- **Big O notation** will always refer to the worst possible scenario

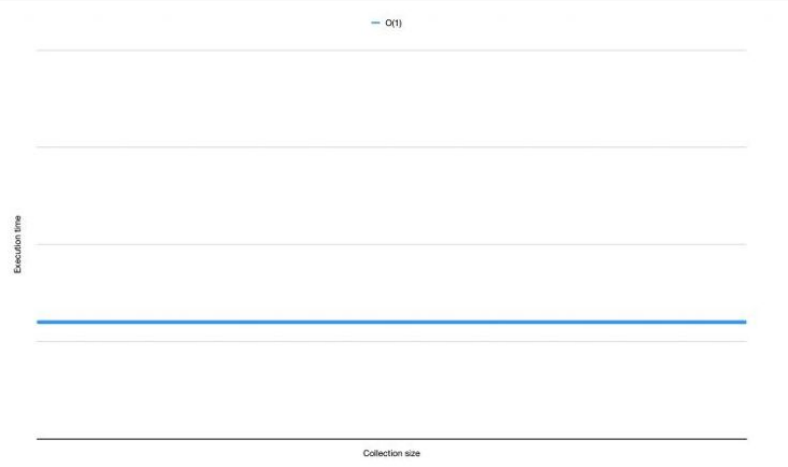
- **Big O notation** will always approach the nearest discrete curve

```
let someValues: [Int] = [1, 2, 3, 4, 5]
someValues[0]
```

O(1)

The performance of an algorithm that is O(1) is not tied to the size of the data set it's applied to.

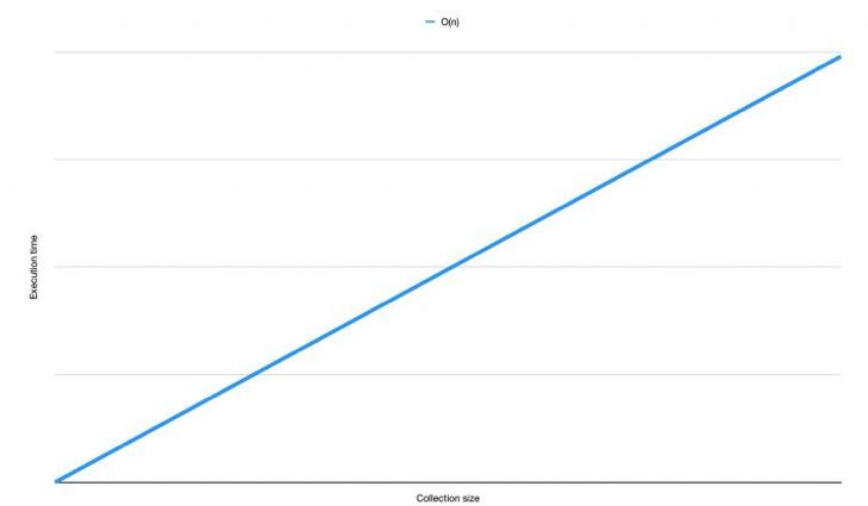
It doesn't matter if you're working with a data set that has 10, 20 or 10,000 elements in it. The algorithm's performance should stay the same at all times.



```swift
let cities: [String] = ["New York", "San Francisco", "Chicago", "Denver"]

func itemInList(_ list: [String], item: String) -> Bool {
    for value in list {
        if value == item {
            return true
        }
    }

    return false
}

itemInList(cities, item: "Denver")
```

O(n)

The algorithm's execution time or performance degrades linearly with the size of the data set.

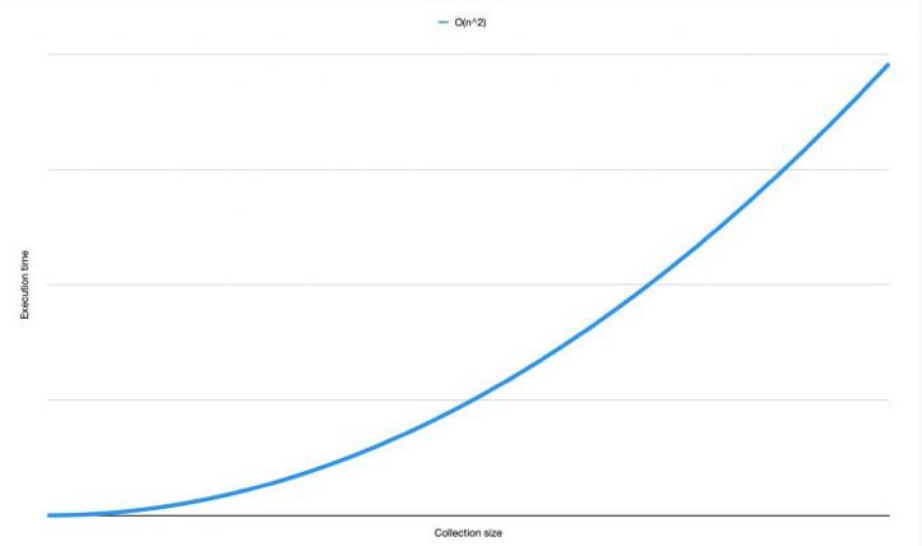


```swift
let items: [Int] = [1, 2, 3, 4, 5]

func allCombinations(of items: [Int]) -> [(Int, Int)] {
    var result: [(Int, Int)] = []

    for item in items {
        for nestedItem in items {
            result.append((item, nestedItem))
        }
    }

    return result
}

allCombinations(of: items)
```

O(n^2)

Quadratic performance

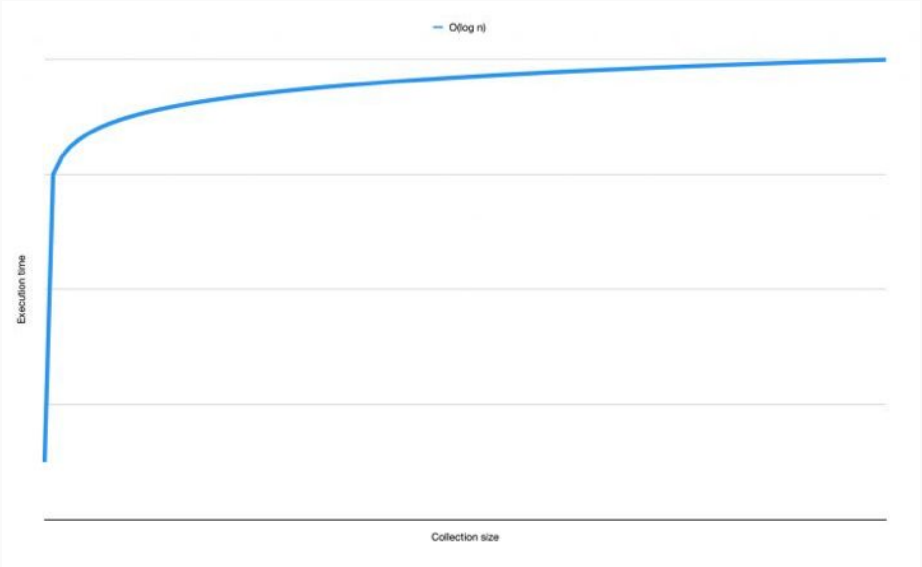
- Quadratic performance is common in some simple sorting algorithms like bubble sort



O(log n)

Complexity that grows on a logarithmic

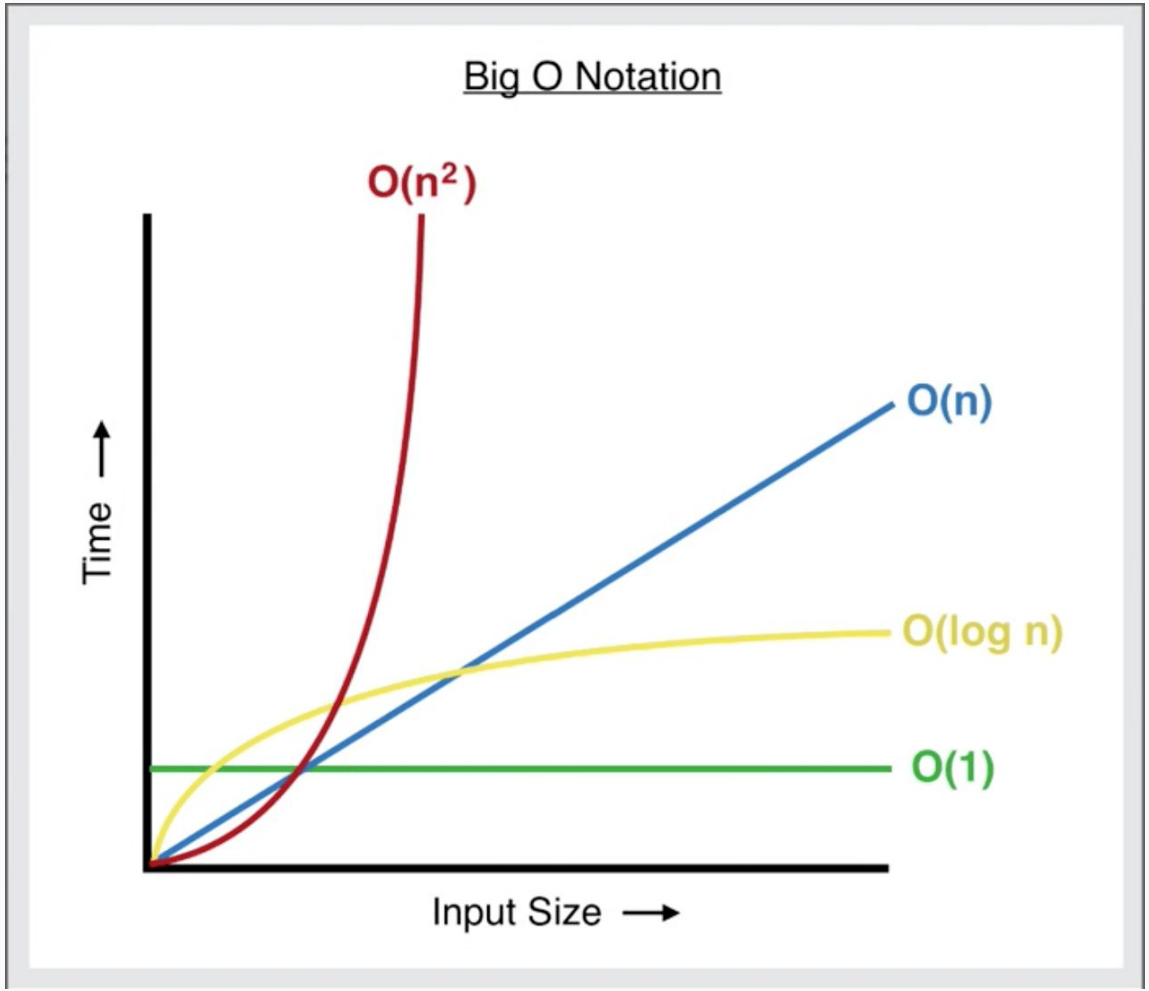
Complexity will often perform worse than some other algorithms for a smaller data set. However, as the data set grows and n approaches an infinite number, the algorithm's performance will degrade less and less.

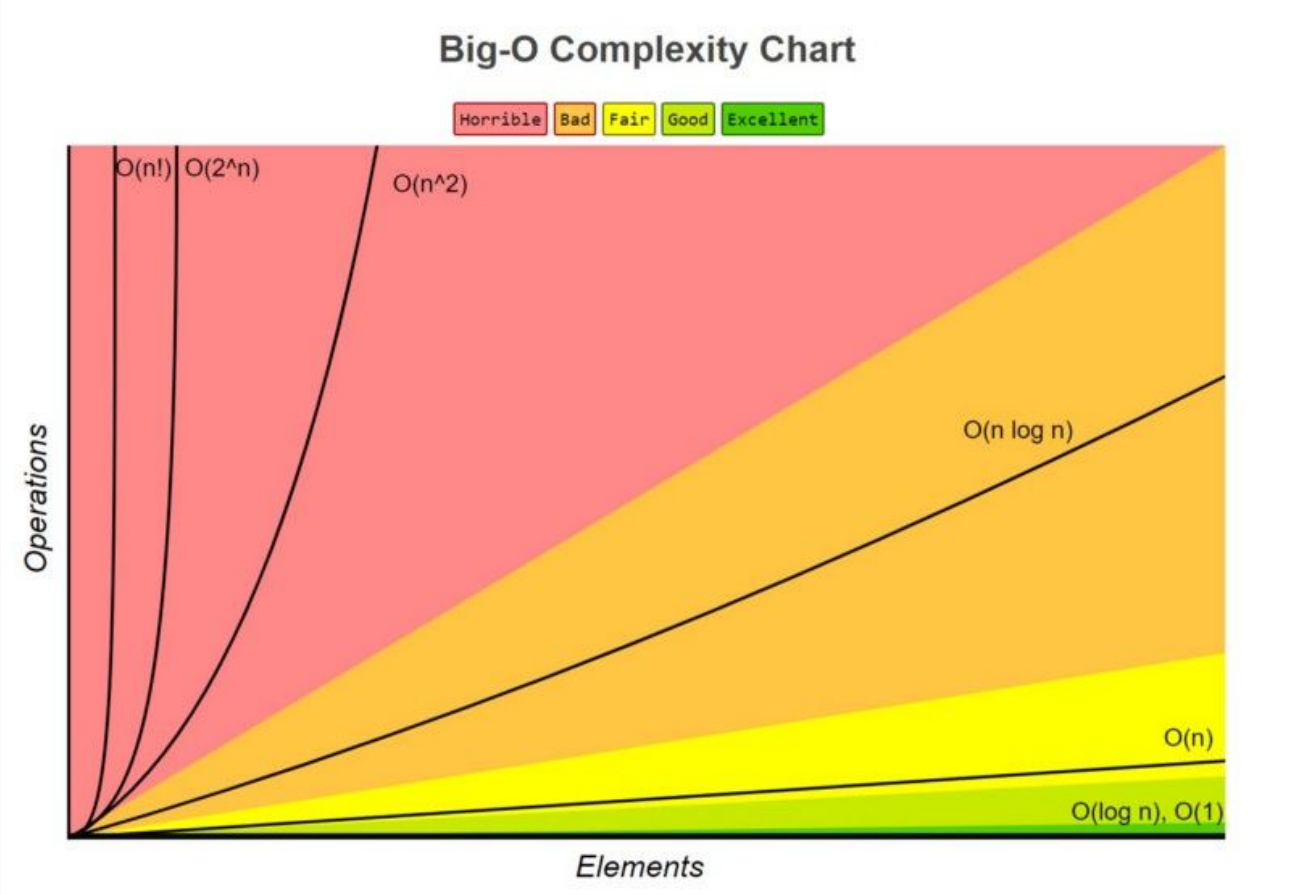


Notation

- O(1)
- O(n)
- O(n^2)
- O(log n)
- O(n log n)

Big O graphs




Big-O Complexity Chart
Horrible
Bad
Fair
Good
Excellent
O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(log n), O(1)
Operations
Elements

Determining the Big O notation of your code

```swift
let grades: [Int] = [1, 2, 3, 4, 5, 6, 7, 8]

func printAll(from items: [Int]) {
    for item in items {
        print(item)
    }
}

printAll(from: grades)
```

O(n) → 8

```swift
let cities: [String] = ["New York", "San Francisco", "Chicago", "Denver"]

func printFirst(from items: [String]) {
    if items.count > 0 {
        print(items[0])
    }
}

printFirst(from: cities)
```

O(1)

```swift
let cities: [String] = ["New York", "San Francisco", "Chicago", "Denver"]

func doublePrint(items: [String]) {
    for item in items {
        print("- \(item)")
    }

    for item in items {
        print("* \(item)")
    }
}
doublePrint(items: cities)
```

O(2n) → 8

```swift
let cities: [String] = ["New York", "San Francisco", "Chicago", "Denver"]

func doublePrint(items: [String]) {
    for item in items {
        print("- \(item)")
    }

    for item in items {
        print("* \(item)")
    }
}
doublePrint(items: cities)
```

O(2n) → 8

Challenges

Completa la función 'products_from()', la cual recibe 1 arreglo de números enteros llamado numbers y retorna un nuevo arreglo de números enteros de la misma dimensión donde cada elemento del arreglo sea el producto de todos los números del arreglo excepto el elemento que se encuentra en ese indice.

- *Ten cuidado con las posibles errores*
- *Tu solución debe de ser en O(n)*

Entrada: [1, 2, 6, 5, 9]

Salida: [540, 270, 90, 108, 60]

DESERTOR

Completa la función 'find_safe_place', la cual recibe el número de soldados alineados en un circulo y retorna el lugar del ultimo soldado sobreviviente.Ten cuidado con las posibles errores

- *Cada soldado matara al soldado inmediato vivo un turno a la vez*
- *Para soluciones en O(n^2) menos 50 puntos del total*
- *Para solución en O(n) nada*
- *Para solución en O(log(n)) una participación*
- *Para solución en O(1) dos participaciones*

Entrada: 12

Salida: 9

Entrada: 47

Salida: 31

Entrada: 1000

Salida: 977

Desertor

DESERTOR

Soldados	Sobreviviente
1	1
2	1
3	3
4	1
5	3
6	5
7	7
8	1
9	3
10	5
11	7
12	9
13	11
14	13
15	15
16	1