# **BIG O**

# **INTRODUCTION**

* How much time does this algorithm need to complete?
* How much space does this algorithm need for computing?

We are trying to understand how quickly the runtime of algorithm grows if a size of inputs grows

**Definition1. Big O** - is the measure that describes a performance of algorithm in worst case.

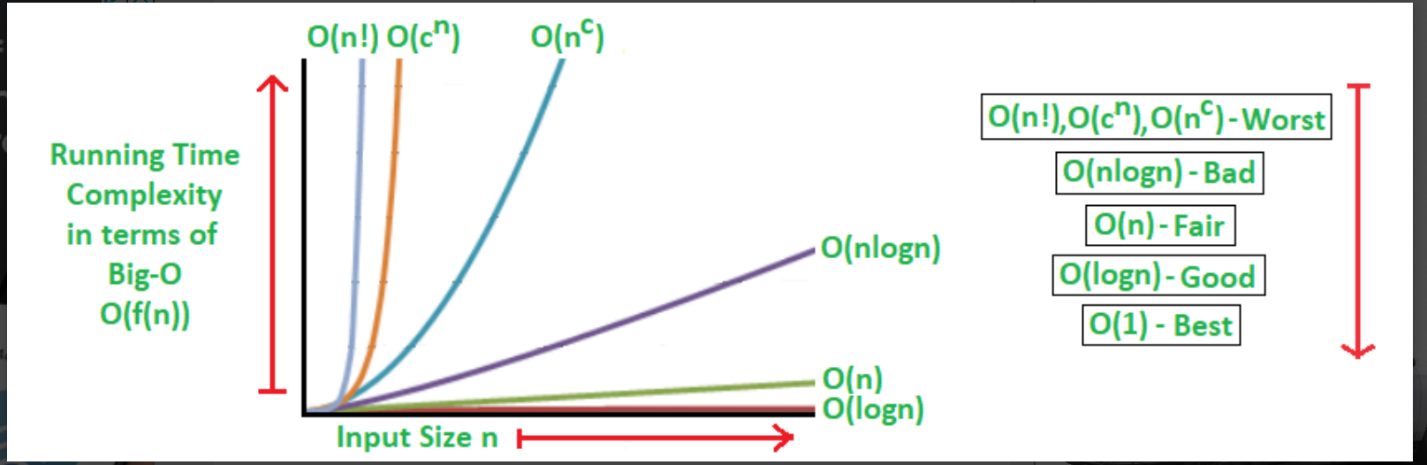
Definition2. Big O - is the measure that mean how long time takes to run an algorithm in the worst case

Definition3. Big O - notation is the measure of scalability of algorithm (code). It’s an upper measure

**Algorithm analysis** – is a study about algorithm performance or algorithm complexity.

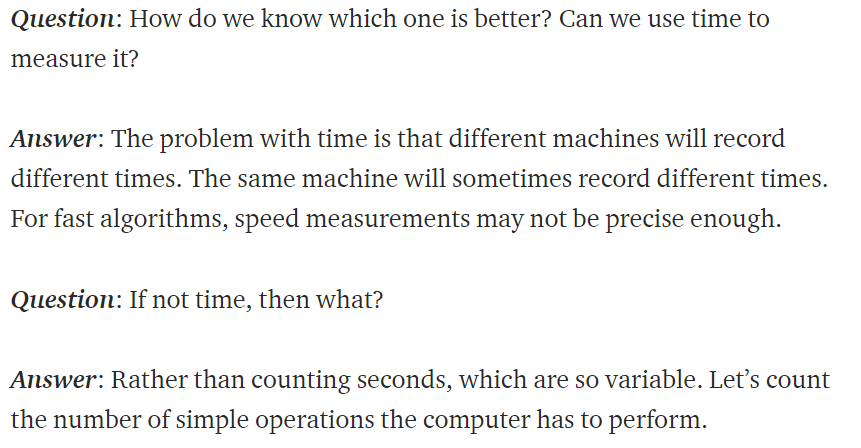
**Algorithm complexity** - is algorithm performance

* **Best case — Big Omega — Ω(n)**.
* **Average case —Big Theta — Θ(n)**.
* **Worst case —Big O Notation — O(n)**.



# **SPEED MEASUREMENT**

It is hard to pin down the exact runtime required by an algorithm, it depends on what processor you use, what other programs the computer is running. So instead of calculating that, we use a concept to see how quickly the runtime grows



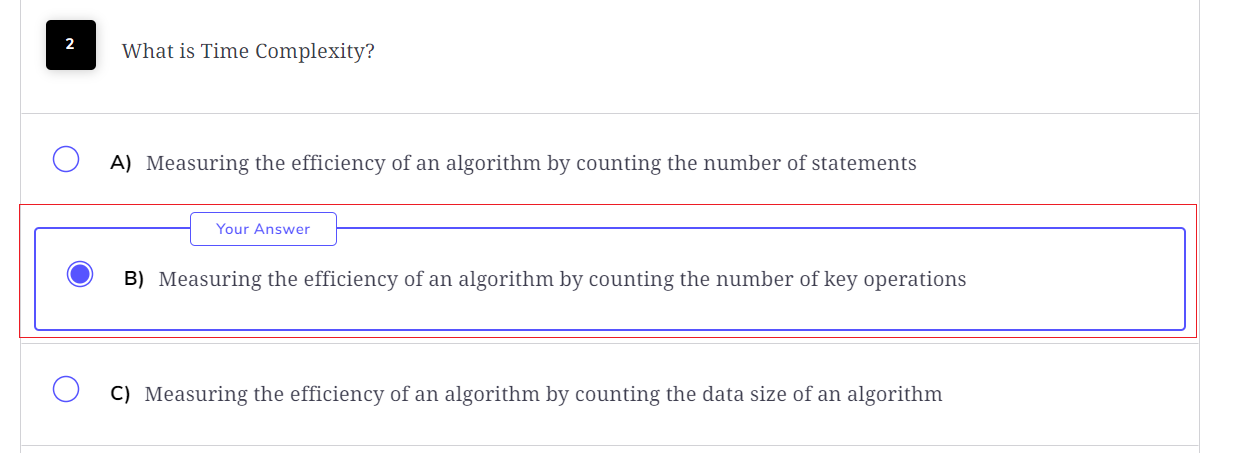
# The study of the performance of algorithms — or algorithmic complexity — falls into the field of [algorithm analysis](https://en.wikipedia.org/wiki/Analysis_of_algorithms).

# **TWO TYPES OF COMPLEXITY**

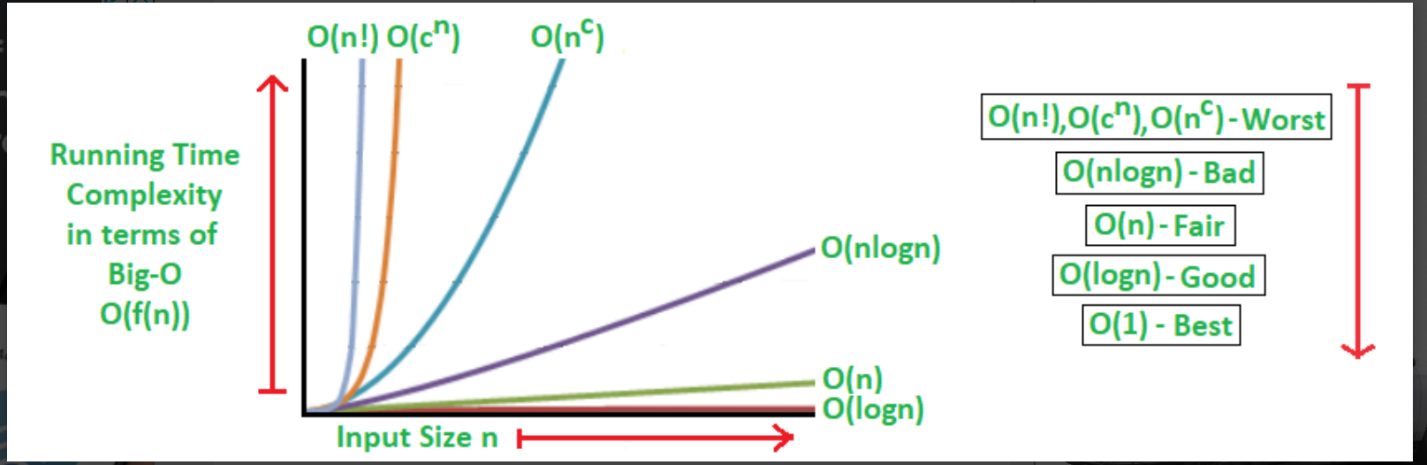
There are 2 types complexity:

* Time complexity – amount of time taken to run an algorithm
* Space complexity – amount of resources(usually space) taken to run an algorithm

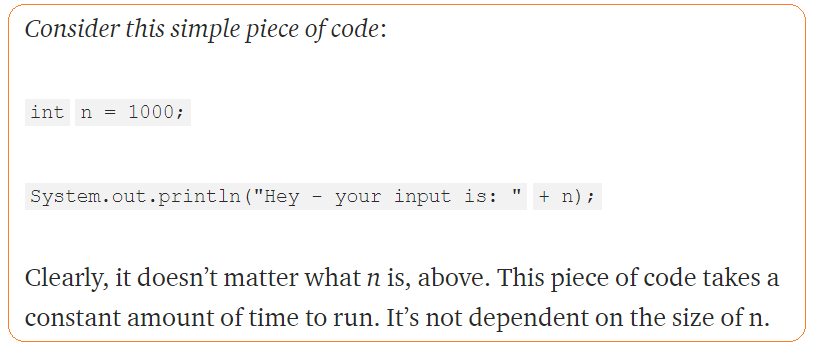
Note: space and time complexity must be equal, or space complexity must be less



# **COMPLEXITY**



1. **O(1) Constant time** - no loops. It does not depends on a size of input



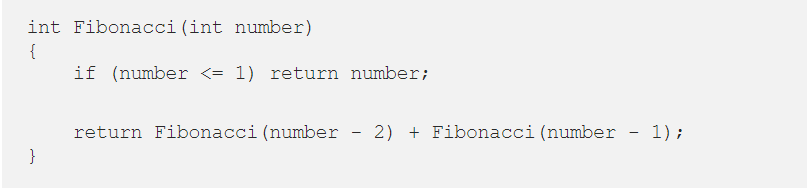
1. **O(log N) Logarithmic time**- normally searching algorithms have log n if the input is sorted (Binary Search)

Example of Logarithmic time is binary search

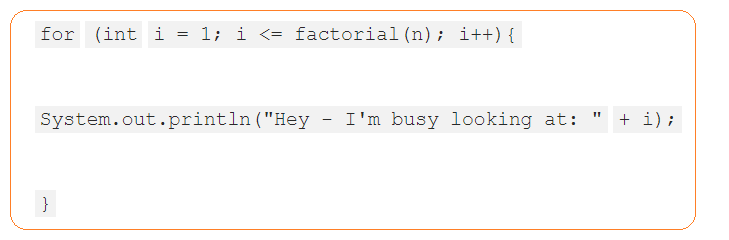
1. **O(n) Linear time** - for loops, while loops through n items

**grows linearly -**  means that it grows directly proportional to the size of its inputs

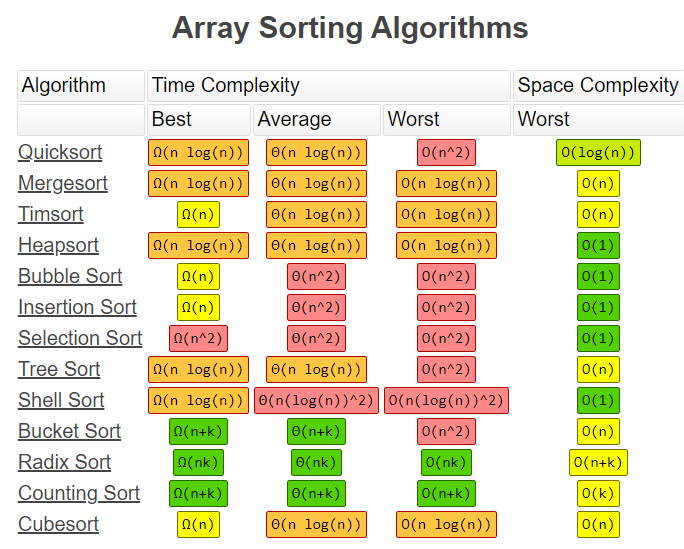
1. **O(n log(n)) Log Linear or super linear time** - running time grows in proportion to n log n of the input: usually sorting operations
2. **O(n²) Quadratic** - every element in a collection needs to be compared to every other element. Two nested loops
3. **O(nc) polynomial time** - common case of quadratic time complexity
4. **O(2n) Exponential** - recursive algorithms that solve a problem of size N. Example, Fibonacchi



O(n!) Factorial- we are adding a loop for every element



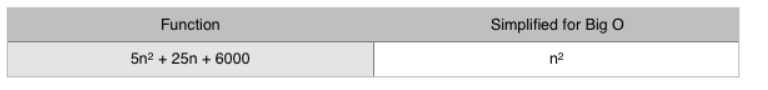
# 



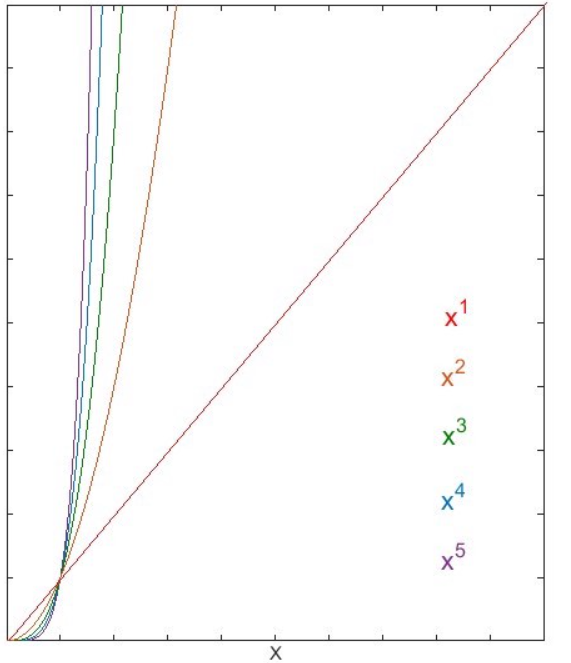
# **FIND THE BIG O COMPLEXITY OF AN ALGORITHM**

**Big O only cares about the part where it grows the fastest**.

Rule 1. Get Rid of Constants and Coefficients

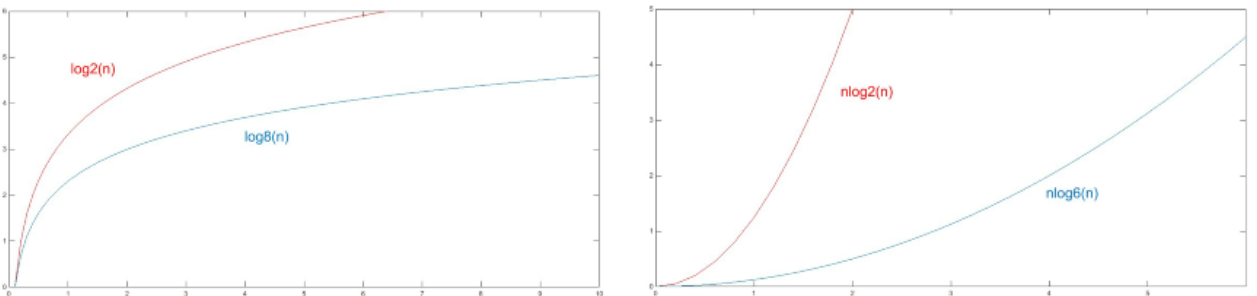


Rule 2. Combine Exponents and Know That The Larger Ones Scale Faster





Rule 3. Know That All Log Bases Other Than 2 Can Be Simplified to Log Base 2, Yet Larger Ones Do Scale Faster



Rule 4.

----

Rule 1: Worst case

Rule 2: Remove the leading constants

Rule 3: Different terms for inputs

Rule 4: Drop nondominant terms

# **BEST CASE SCENARIO, AVERAGE-CASE SCENARIO, AND WORST-CASE SCENARIO**

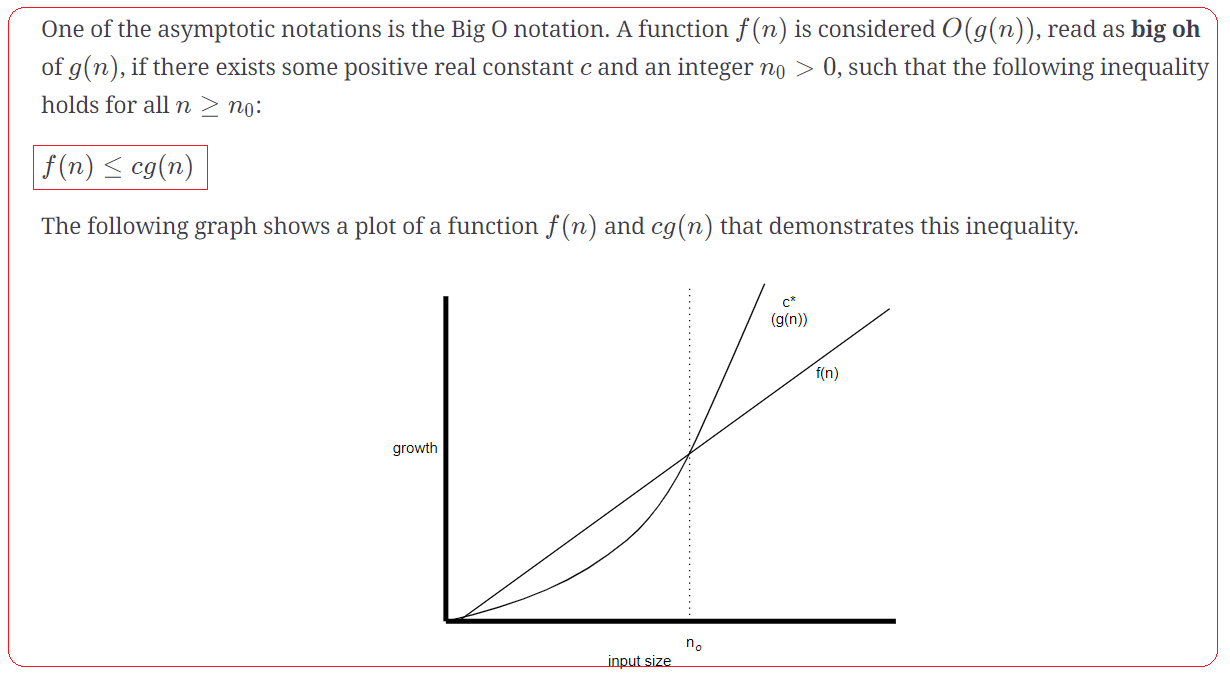
**Best case — Big Omega — Ω(n)**.

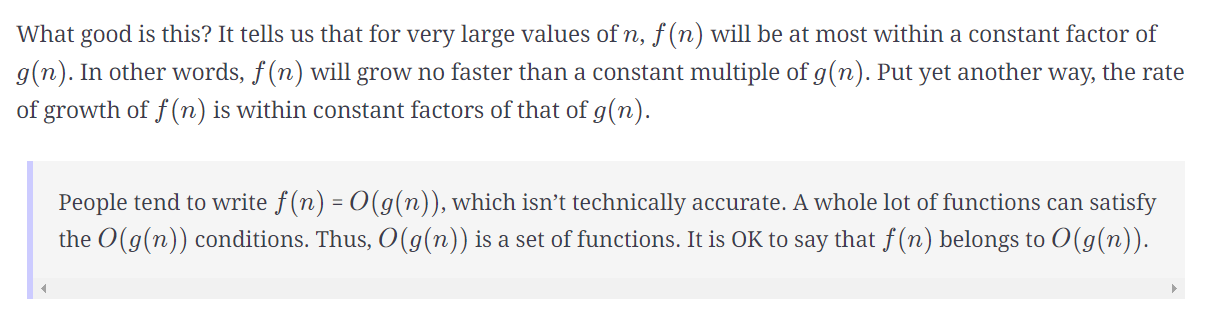
**Average case —Big Theta — Θ(n)**.

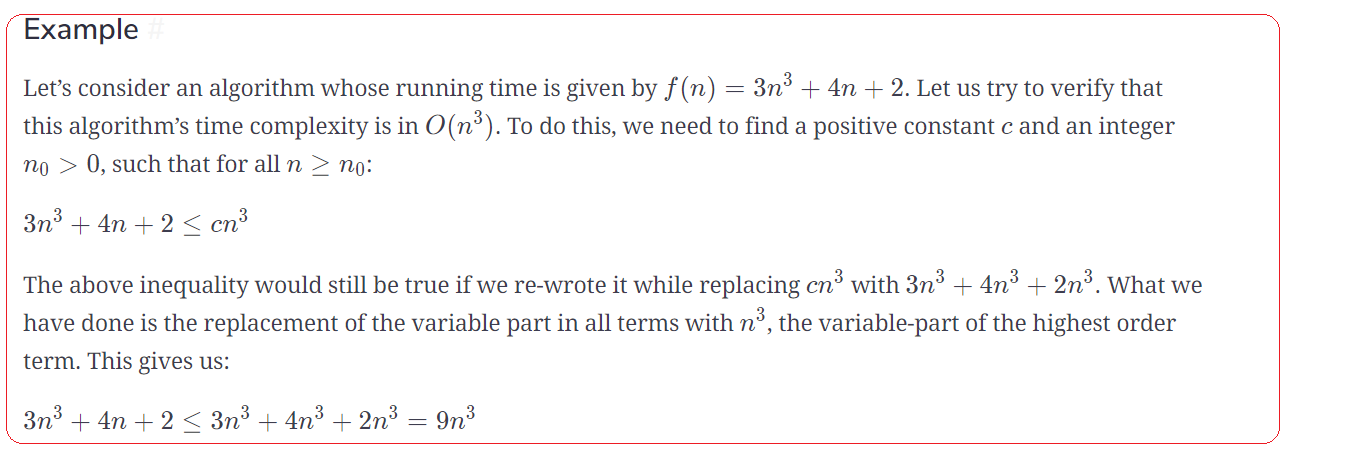
**Worst case —Big O Notation — O(n)**.

Note: programmers typically assess the worst case O(n)

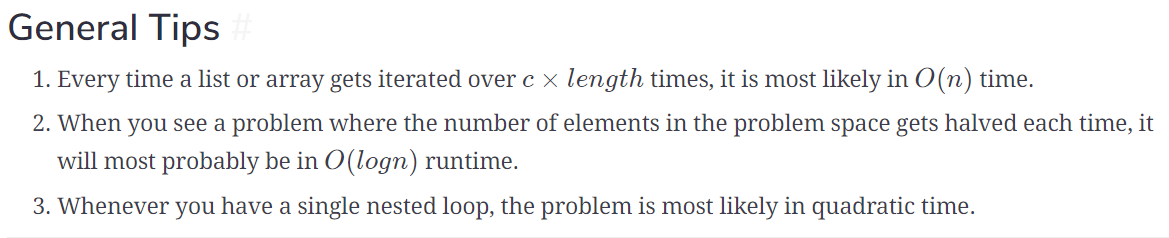
# **ASYMPTOTIC ANALYSIS**



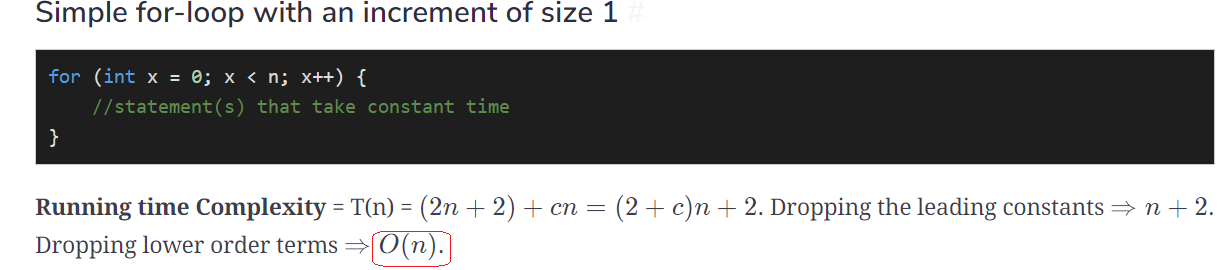




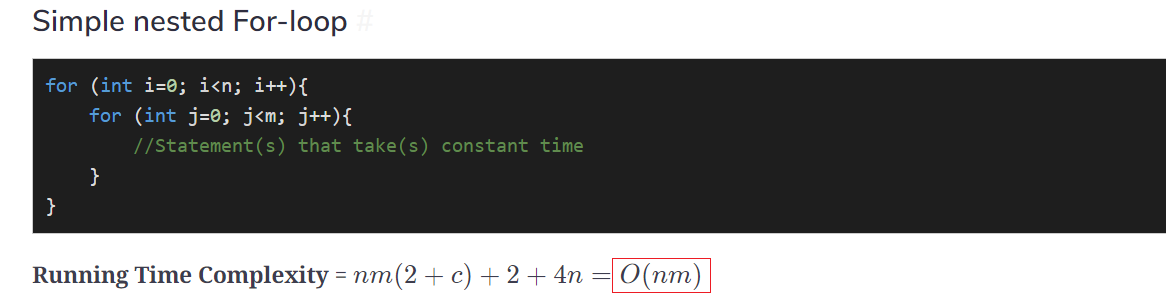
# **COMMON SCENARIOUS OF ANALYZING COMPLEXITY**



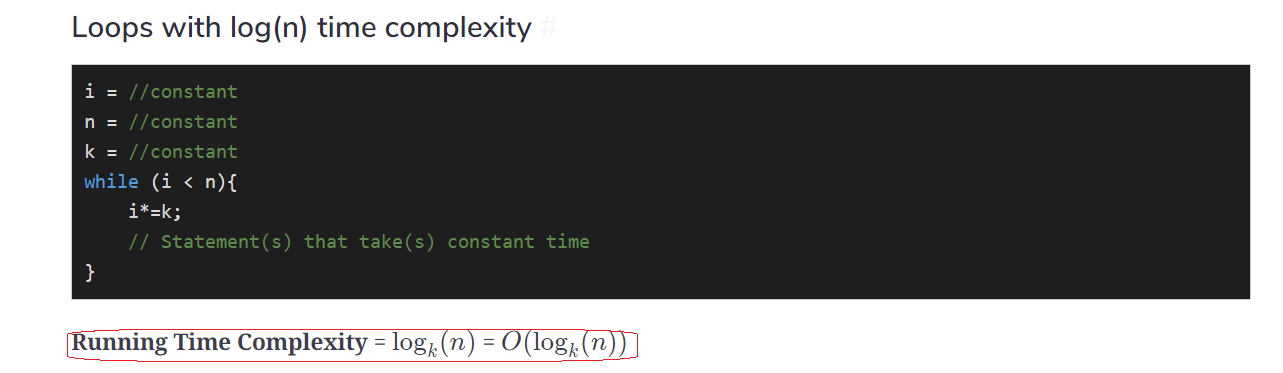
# **#1.SIMPLE FOR LOOP**

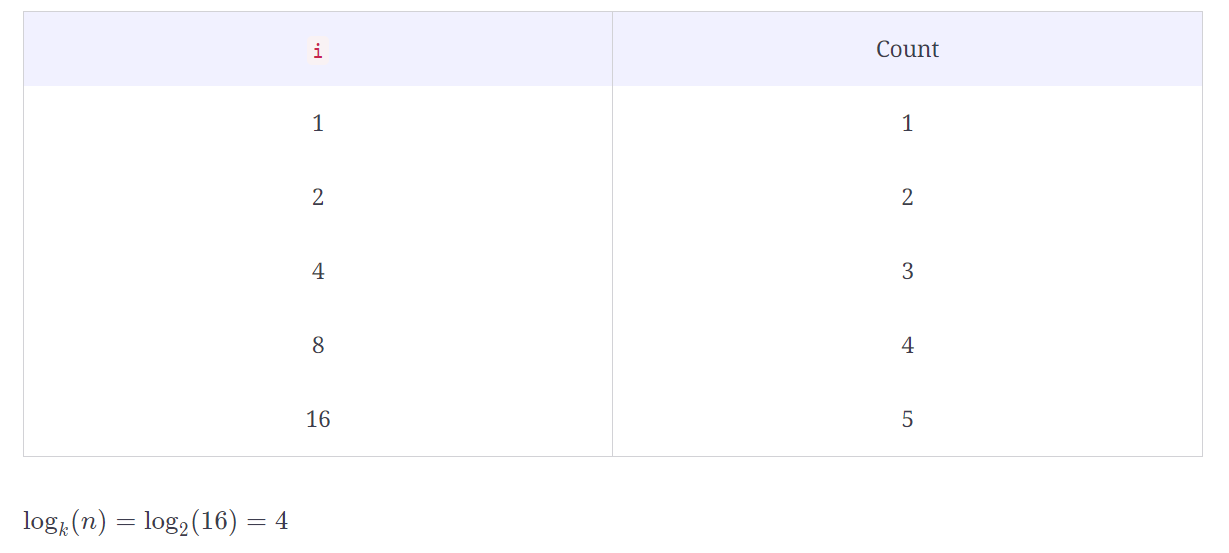


# **#2.** **SIMPLE NESTED FOR-LOOP**

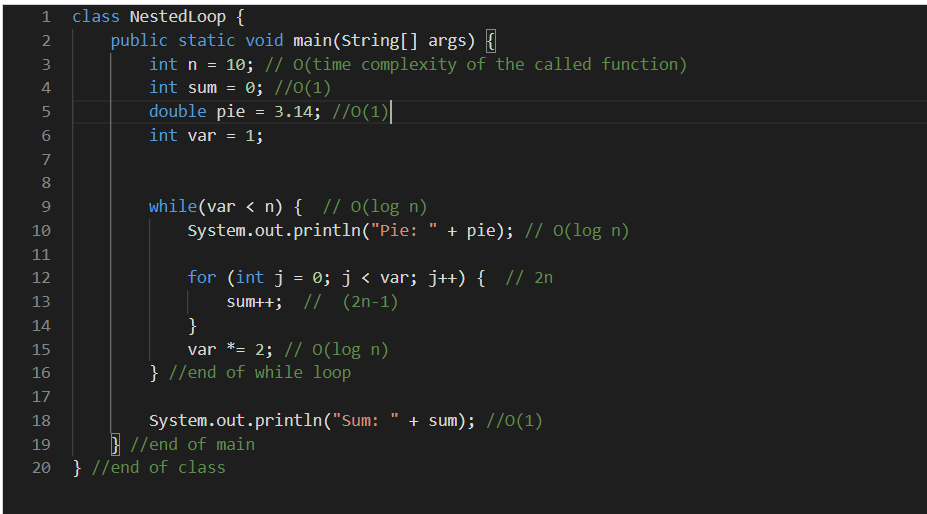


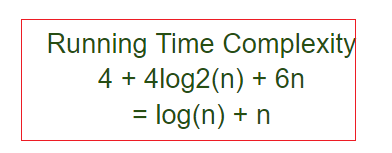
# **#4.** **LOOPS WITH LOG(N) TIME COMPLEXITY**

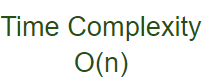




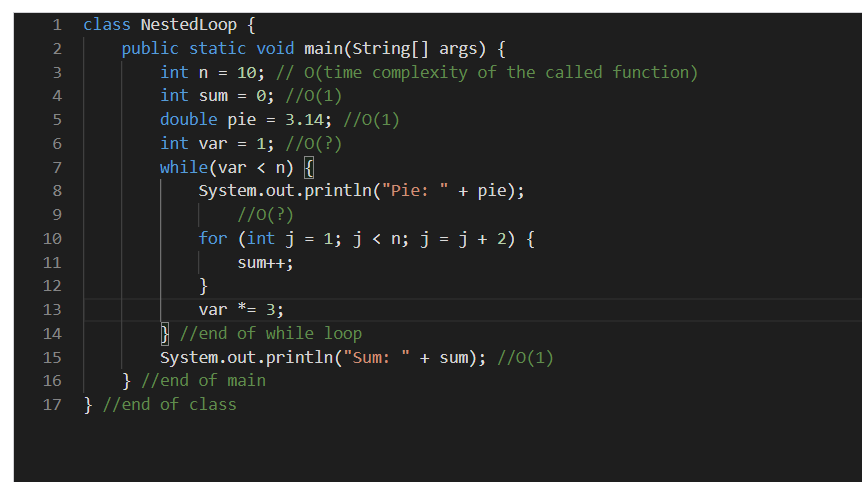
# **#5.** **BIG O OF NESTED LOOP WITH MULTIPLICATION -TRICKY**

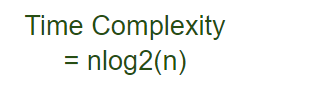
****



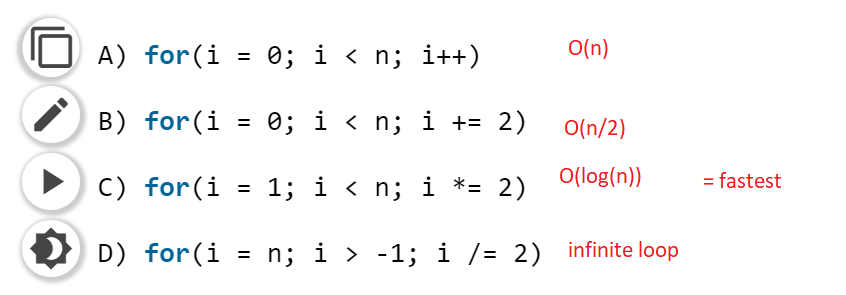


# **#6. NESTED LOOP WITH MULTIPLICATION(2)**

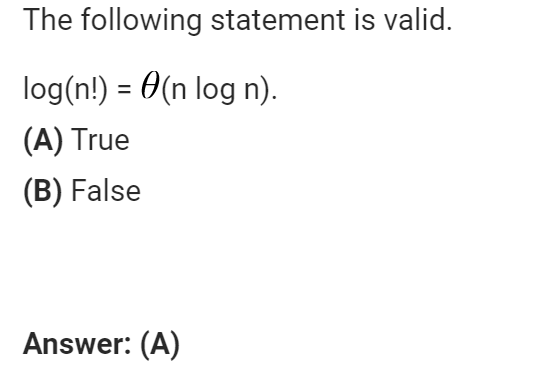




**#7.WHICH ONE IS FASTES?**



# **#8. O(n\*log(n)) = O(log(n!))**



# **#2.**

# **#2.**

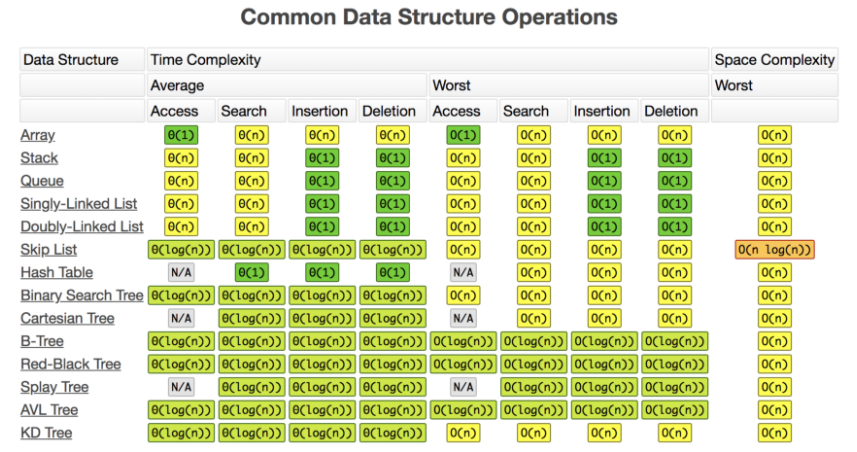
# **#2.**

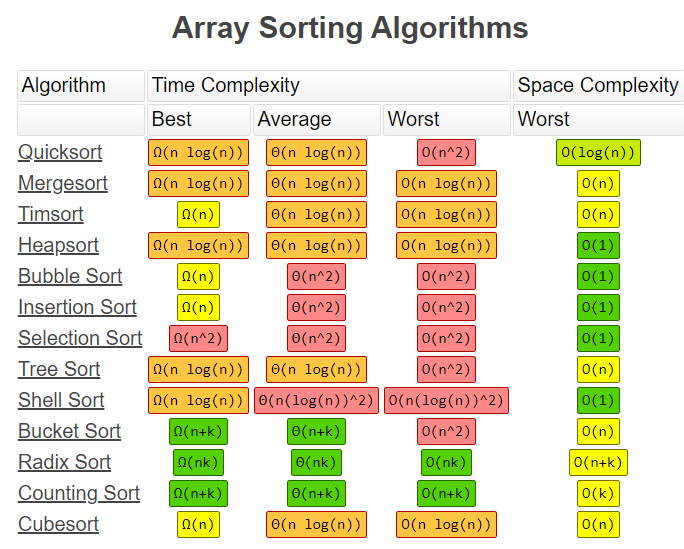
# **#2.**

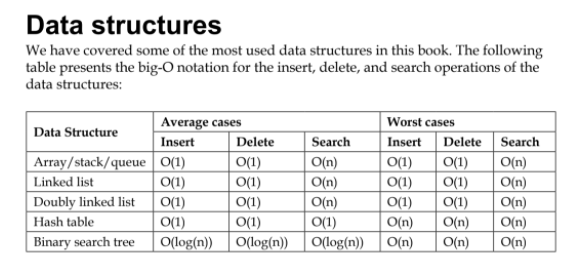
# **#2.**

|  |  |  |  |
| --- | --- | --- | --- |
| ID | DATA STRUCTURE | TIME COMPLEXITY | SPACE COMPLEXITY |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

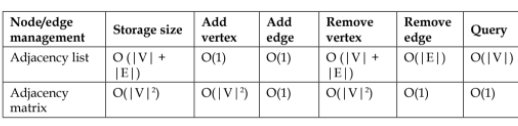
|  |  |  |  |
| --- | --- | --- | --- |
| ID | ALGHORITHMS | TIME COMPLEXITY | SPACE COMPLEXITY |
|  |  |  |  |
|  | BINARY SEARCH | O(log(N)) |  |
|  | SLIDING WINDOW | O(N) | O(1) |
|  | QUICKSORT | O(N\*log(N)) |  |
|  |  |  |  |
|  | FIBONACCHI | O(2n) |  |
|  | Tower’s of Hanoi puzzle | O(2n) |  |
|  | <https://en.wikipedia.org/wiki/Travelling_salesman_problem> | N! |  |

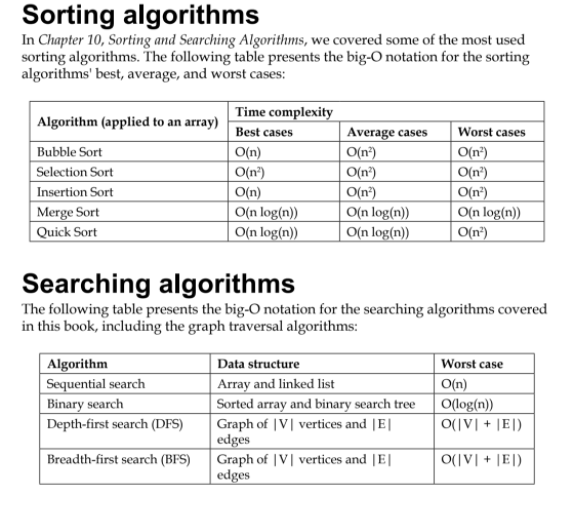












* Give an example of an algorithm whose best case is equal to its worst case?

**Counting Sort and Radix Sort are two algorithms which have the same best, worst and average case complexities**.