Big O Notation

Contents

[Big O Notation 2](#_Toc94815723)

[Five Basic Rules 3](#_Toc94815724)

[GROWTH HIERARCHY 5](#_Toc94815725)

[O(1) - Constant 5](#_Toc94815726)

[O(LOGN) - Logarithmic 5](#_Toc94815727)

[O(N) - Linear 5](#_Toc94815728)

[O(N LogN) – Log Linear 6](#_Toc94815729)

[O(N2) – Quadratic 6](#_Toc94815730)

[O(2N) – Exponential 7](#_Toc94815731)

[O(N!) – Factorial 7](#_Toc94815732)

[ORDERS OF GROWTH 8](#_Toc94815733)

[O(1) – Constant 8](#_Toc94815734)

[O(log n) – Logarithmic 8](#_Toc94815735)

[O(n) – Linear 8](#_Toc94815736)

[O(n log n) – Linearithmic 8](#_Toc94815737)

[O(n2) – Quadratic 8](#_Toc94815738)

[O(n3) – Cubic 8](#_Toc94815739)

[O(2n) – Exponential 8](#_Toc94815740)

[O(n!) – Factorial 8](#_Toc94815741)

# Big O Notation

What is Big O notation?  
Big O notation is used to analyze the efficiency of an algorithm as its input approaches infinity.

As the size of the INPUT to the algorithm grows, how drastically does the space or time requirements grow with it.

O(n) – suggests that an operation takes the same time per item (n) to process regardless of how many items (n) there are.

Asymptotic

O(n)2

|  |  |
| --- | --- |
|  |  |
| A picture containing graphical user interface  Description automatically generated | Time required to process is approximately the square of the amount of records / bytes / bits / etc.  “… the processing time required by the Algorithm will be the square of the data input.”  Time = O | ( records = n ) 2 |

# Five Basic Rules

|  |  |
| --- | --- |
| 1. Rule 1:  If an algorithm performs a certain sequence of steps ***f(N)*** times for a mathematical function ***f***, then it takes ***O(f(N))*** steps. | ***f(N) -> O(f(N))***  The algorithm provided examines each of the N items in the array once, so it has O(N). |
| Code example:   |  | | --- | | Integer: FindLargest (Integer: array[])      Integer: largest = array[0]      For i = 1 to <largest index>          If (array[i] > largest) then largest = array[i]      Next i      Return largest  End FindLargest | | |
| 1. Rule 2: If an algorithm performs an operation that takes ***O(f(N))*** steps and then performs a second operation that takes ***O(g(N))*** steps for functions ***f*** and ***g***, then the algorithm’s total performance is ***O(f(N)) + O(g(N))***. | ***O(1 + N 1) -> O(2 + N)*** |
| Code example:   |  | | --- | | Integer: FindLargest (Integer: array[])      Integer: largest = array[0]                             // O(1)      For i = 1 to <largest index>                            // O(N)          If (array[i] > largest) then largest = array[i]      Next i      Return largest                                          // O(1)  End FindLargest | | |
| 1. Rule 3: If an algorithm performs an operation that takes ***O(f(N)) + g(N)*** time and the function ***f(N)*** is greater than ***g(N)*** for large ***N***, then the algorithm’s performance can be simplified to ***O(f(N))***. |  |
| |  | | --- | | // Rule 3: No code provided | | |
| 1. Rule 4:   If an algorithm performs an operation that takes ***O(f(N))*** steps, and for every step in that operation if performs another ***O(g(N))*** steps, the algorithm’s total performance is ***O(f(N)) x O(g(N))***. | ***O(f(N)) x O(g(N))*** |
| Code example:  Boolean: ContainsDuplicates (Integer array[])      // Loop over all of the array's items      For i = 0 To <largest index>          For j = 0 T0 <largest index>              // See if these two items are duplicates              If (i != j) then                  If(array[i] == array[j]) Then Return True              End If          Next j      Next index      // If we get to this point, there are no duplicates      Return False  End ContainsDuplicates | |
| 1. Rule 5: Ignore constant multiples. If C is a constant, ***O(C x f(N))*** is the same as ***O(f(N))***, and ***O(f(C x N))*** is the same as ***O(f(N))***. |  |
|  | |

**Big O(n) Function sample**

Graphical user interface, text

Description automatically generated

Graphical user interface

Description automatically generated with medium confidence

Ignore the Constants

What is a Constant?

A Constant is any step that doesn’t scale with the INPUT to the function. For example, the time to evaluate this expression does not change with the INPUT. Below: 100 and 1000 are Constants.

**Big O (1) – Constants / Constants Algorithm Function Sample**

Text

Description automatically generated

# GROWTH HIERARCHY

|  |  |
| --- | --- |
| O(1) - Constant | |
| This algorithm takes the same number of steps to execute regardless of how much data is pass in. E.g.: **Access i-th element of an array.** |  |
| Graphical user interface, text, application  Description automatically generated with medium confidence    Code example:   |  | | --- | | // Constant - O(1) function  function constantFunc(arr){      console.log(1000 \* 100000);                         // O(1)  }  // Constant - Big O(1) Example  function linearFunc(arr){      for (let i = 0; i < arr.length; i++){               // O(n)          console.log(1000 \* 100000);                     // O(1)          let something = (2000000000 \* 2000000000) / 2;  // O(1)          console.log(something)                          // O(1)      }  }  Second example: All of the O(1) get cancelled out because the O(n) is the worst performing part of the function. This is why we ignore CONSTANTS – eliminating the non-dominant items. As a functions INPUT moves towards infinity, the CONSTANTS become less important. | | |

|  |  |
| --- | --- |
| O(LOGN) - Logarithmic | |
| This algorithm’s number of operations increases by one, each time the data is double. E.g.: **Dictionary loop** |  |
|  | |

|  |  |
| --- | --- |
| O(N) - Linear | |
| This algorithm takes as many steps as there are elements of data. E.g.: **Traverse an array**. |  |
| Graphical user interface, text, application  Description automatically generated | |

|  |  |
| --- | --- |
| O(N LogN) – Log Linear | |
| This algorithm is doing log(N) works N times - most of the **sorting** **algorithms** fall into this class. |  |
| Graphical user interface, text  Description automatically generated with medium confidence | |

|  |  |
| --- | --- |
| O(N2) – Quadratic | |
| This algorithm’s performance is proportional to the square of the size of the input elements. E.g.: **Traverse nested arrays.** |  |
| Graphical user interface, text, application  Description automatically generated | |

|  |  |
| --- | --- |
| O(2N) – Exponential | |
| This algorithm takes twice as long for every new element added. E.g.: **Find all subsets of a dataset.** |  |
| Graphical user interface, text, application  Description automatically generated | |

|  |  |
| --- | --- |
| O(N!) – Factorial | |
| This algorithm’s run time is proportional to the factorial of the input size. E.g.: Find all the different permutations in a dataset. |  |
|  | |

# ORDERS OF GROWTH

|  |  |
| --- | --- |
| O(1) – Constant | Constant |
| O(log n) – Logarithmic | Logarithmic |
| O(n) – Linear | Linear |
| O(n log n) – Linearithmic | Linearithmic |
| O(n2) – Quadratic | Quadratic |
| O(n3) – Cubic | Cubic |
| O(2n) – Exponential | Exponential |
| O(n!) – Factorial | Factorial |

Source: <https://www.bigocheatsheet.com/>

Source: <https://www.youtube.com/watch?v=V6mKVRU1evU&list=PLGLfVvz_LVvReUrWr94U-ZMgjYTQ538nT>