



# Big-O Notation

# What is Big-O Notation ?

- Big-O Notation-
  - Allows to express the performance/complexity of algorithms in relation to their input.

Denotes about

- execution time required
- Space used (e.g. in memory or on disk)



Big-O specifically describes worst-case scenario

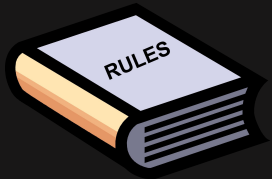
# Why understand Big-O Notation ?

- To be able to program
  - with best containers for storing data
  - with best performance algorithms
  - to design quality software products





# Big-O specifically describes worst-case scenario



1. Big-O deals with the dominant term.

- I.e

$$N^2 + 3N + 4 \sim N^2 \text{ when } N \text{ is very large}$$

$$\text{if } N = 1000, \text{ then } 1000^2 + 3 \cdot 1000 + 4 = 10,03,004 \sim 10,00,000$$

2. The constant coefficients of higher order terms are neglected

$$N + N + N = 3N \sim N \text{ when } N \text{ is infinity,}$$

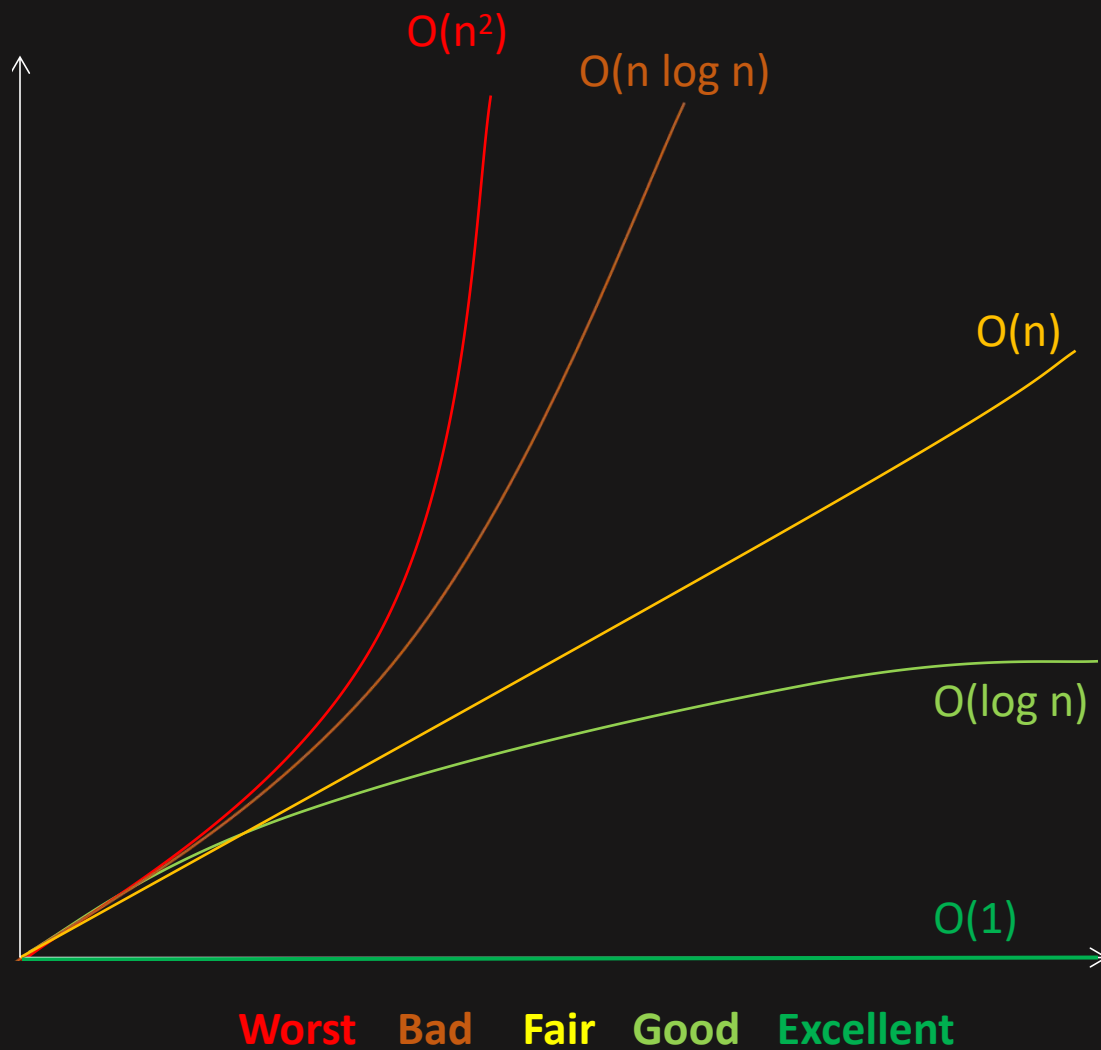
Three times infinity is also infinity.



Mainly focus on the maximum possible worst-case complexity (in time/space)

<https://youtu.be/ICAnn2tmt60>

# Big-O Complexity Chart



# $O(1)$

Excellent

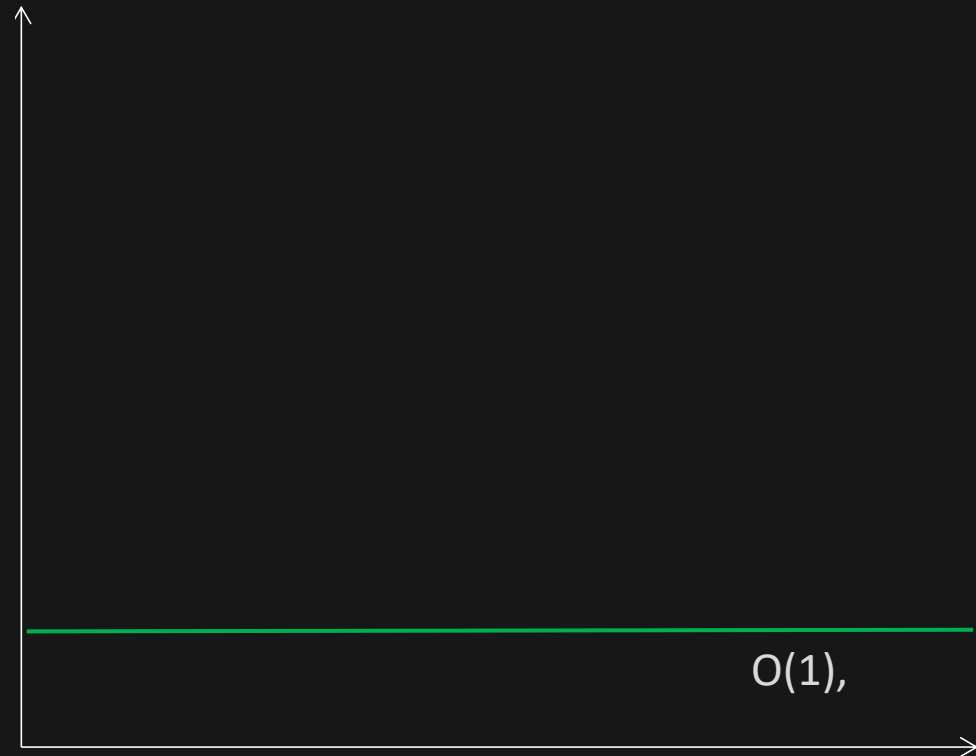
## Definition:

Time complexity is constant, regardless of how many/much items are present in the collection.

## Trend:

Time Taken to access  $N^{\text{th}}$  item:  $X$   
when Total Items present : 10  
or : 10000  
or : 100000  
or : infinity

Time Complexity :  $O(1)$



# $O(1)$

Excellent



## Real World Scenario:

In a Kitchen,

Number of chicken eggs available : 25

Number of Fry pan available : 1

Time taken to make a single Omlet is constant, no matter which of the 25 egg is used.

Here time complexity is  $O(1)$



# O(1)

Excellent

```
int GetNthElementOfArray( int arr[], int N )
{
    return arr[N];
}
```

## C++ STL Containers with O(1) lookup

- **std::array**  
<http://www.cplusplus.com/reference/array/array>
- **Hashtable (std::unordered\_map) for c++11**  
[http://www.cplusplus.com/reference/unordered\\_map/unordered\\_map](http://www.cplusplus.com/reference/unordered_map/unordered_map)

- **Constant Time**
- Irrespective of Array size (i.e, 1 or 1laks items) will only access 1 item
- Performance is constant, no matter item at which index is accessed.

## Examples

1. Accessing Array Index (int a = arr[5];)
2. Inserting a node in Linked List
3. Pushing and Popping on Stack
4. Insertion and Removal from Queue





# $O(\log n)$

Good

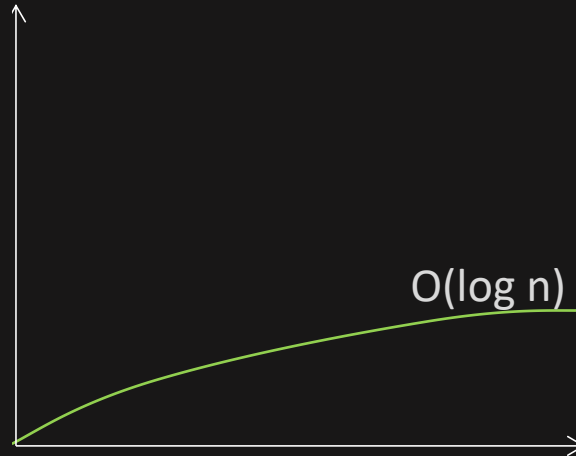
## Definition

$O(\log n)$  means that the running time grows in proportion to the logarithm of the input size.

## Trend,

If 10 items takes time :  $X$ ,  
100 items takes time :  $2X$ ,  
...  
...  
10,000 items takes time :  $4X$

Time Complexity :  $O(\log n)$



# $O(\log n)$

Good

## Real World Example:

Word's meaning search in Dictionary.

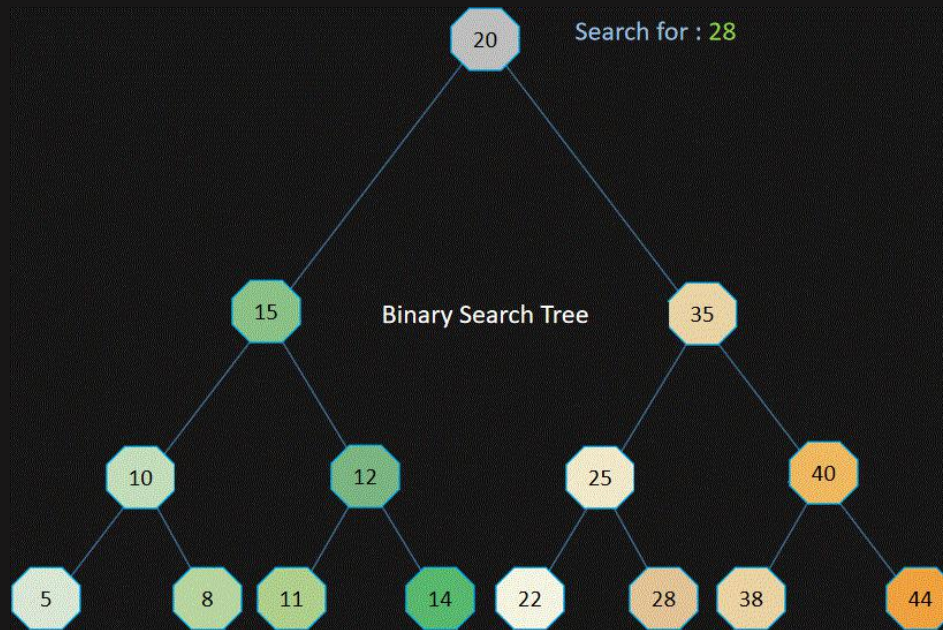
## How Search works

1. Open a random page
2. if the opened page has higher letter than the First Letter of search word, open random page towards left side. Else open random page towards right. Thus an entire portion is skipped from search, which makes it logarithmic search.
3. Continue this until the searching word is found.



# $O(\log n)$

Good



- **Logarithmic Time**
- If 'n' is the array size, then, it takes  $\log(n)$  iteration to complete the operation.

## Examples

1. Binary Search
2. Calculating Fibonacci Numbers

# $O(n)$

Fair

## Definition:

Time complexity is linear, i.e. larger the number of items, the more the running time.

## Trend,

say 1 item takes time :  $X$ ,  
2 items takes time :  $2X$ ,  
4 items takes time :  $4X$ ,  
...  
...  
'n' items takes time :  $n \cdot X$

Time Complexity :  $O(n)$





# $O(n)$

Fair

## Real World Scenario:

Customers waiting in queue in a store's billing counter.

- Max time taken by store employee to complete one billing : 5 min
- Number of customer in Queue = 5
- Max time taken for a customer currently at 5th Queue position =  $5 * 5 \text{ min} = 25 \text{ min}$

## Trend,

Time taken for each queue position

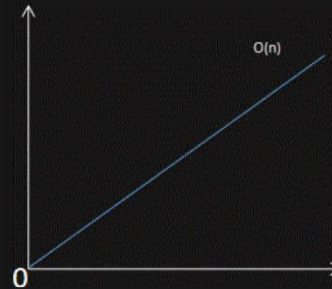
Customer 1 :  $1 * 5 \text{ min} = 5 \text{ min}$

Customer 2 :  $2 * 5 \text{ min} = 10 \text{ min}$

...


Customer 5 :  $5 * 5 \text{ min} = 25 \text{ min}$

Time complexity :  $O(n)$



# O(n)

Fair



```
int SumOfArrayElements( int arr[], int size )
{
    int nSum = 0;
    for( int i = 0; i < size; ++i )
    {
        // sequence of statements of O(1)
        nSum += arr[i]
    }
    return nSum;
}
```

} O(n)

- Linear Time
- If 'n' is the array size, then, it takes 'n' iteration to complete the operation.
- If Array size is '1 million' then iterate '1 million' times.
- Performance is inversly propotional to size.

## Examples

1. Traversing an array
2. Traversing a linked list
3. Linear Search
4. Comparing two strings
5. Checking for Palindrome

# $O(n^2)$

Worst

## Definition:

Time complexity is Quadratic, here number of operations is proportional to the size of the task squared.

## Trend,

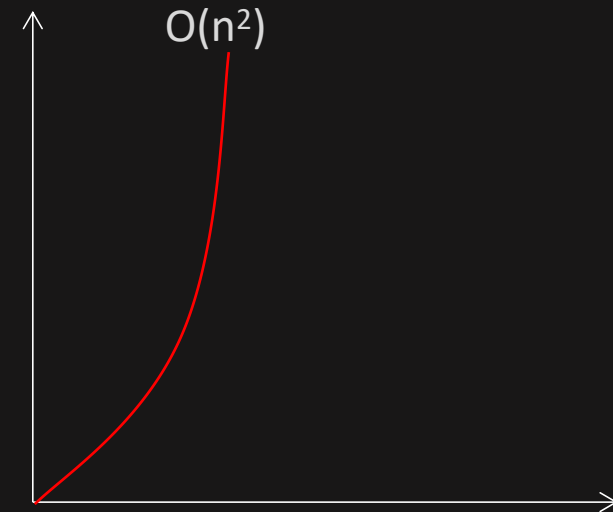
If input is 2, perform 4 operation  
input is 4, perform 16 operation,  
input is 8, perform 64 operation

...

...

'n' items takes time :  $n^2$

Time Complexity :  $O(n^2)$



# $O(n^2)$

Worst

## Real World Scenario:

- Imagine that, you have 10 blank note books, each of with 10 pages. The task is to manually write the page number in each page.
- Number of page numbers wirtten in 1 book = 10,
- Number of page numbers wirtten in 10 book =  $(10 * 10) = 10^2 = 100$

Trend,

Number of Operation = square of (size of Task)

Size of Task : 2 then number of Operations :  $2^2 = 4$

Size of Task : 8 then number of Operations :  $8^2 = 64$

...

Size of Task : 'n' then number of Operations :  $n^2$


Time complexity :  $O(n^2)$





# $O(n^2)$

Worst



```
bool CheckDuplicates( int arr1[], int size , )
{
    int nSum = 0;
    for( int i = 0; i < size; ++i )
    {
        for( int j = 0; j < size; ++j )
        {
            if( i == j ) continue;
            // sequence of statements of O(1)
            if( arr[i] == arr[j] )
            {
                return true;
            }
        }
    }
    return false;
}
```

$O(n^2)$

✓ Quadratic Time

## Examples

1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Traversing a simple 2D array

# Complexities of common data structures in Big-O Notations



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

# Complexities of Array Sorting Algorithms in Big-O Notations



Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
<u>Quicksort</u>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(\log(n))$
<u>Mergesort</u>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
<u>Timsort</u>	$\Omega(n)$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
<u>Heapsort</u>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(1)$
<u>Bubble Sort</u>	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
<u>Insertion Sort</u>	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
<u>Selection Sort</u>	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
<u>Tree Sort</u>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(n)$
<u>Shell Sort</u>	$\Omega(n \log(n))$	$\Theta(n(\log(n))^2)$	$O(n(\log(n))^2)$	$O(1)$
<u>Bucket Sort</u>	$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$	$O(n)$
<u>Radix Sort</u>	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$	$O(n+k)$
<u>Counting Sort</u>	$\Omega(n+k)$	$\Theta(n+k)$	$O(n+k)$	$O(k)$
<u>Cubesort</u>	$\Omega(n)$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$



Thank You

