Big O and Its Calculation

Asymptotic Analysis measures the efficiency of an algorithm. It is an estimating technique.

Big O notation, written as "*O()"* is a mathematical notation that describes how an algorithm's time or space requirements increase as the size of its input, n increases.

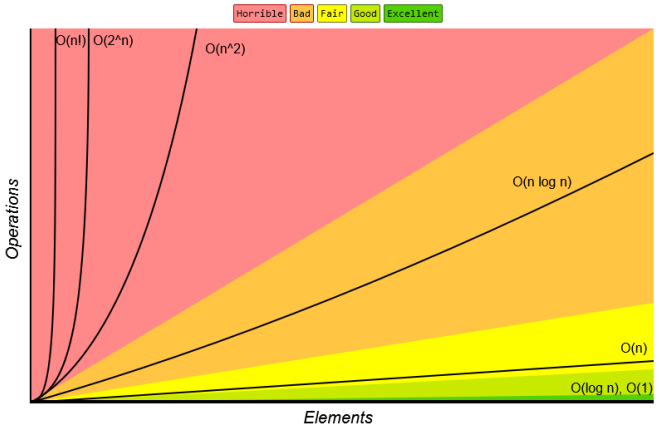
There are two kinds of complexities, Time and Space.

Time complexity describes how runtime (the duration for an algorithm to finish) will grow relative to the input as the input gets arbitrarily large.

Space complexity describes how memory (the amount of space needed for an algorithm to finish) will increase relative to the input as the input gets arbitrarily large.

The typical Big O runtime categories in terms of fastest to slowest are;

- O(1) - Constant Time
- O(log n) - Logarithmic Time
- O(n) - Linear Time
- O(n log n) - N Log N Time
- O(n^p) - Quadratic Time
- O(k^n) - Exponential Time
- O(n!) - Factorial Time



Constant Time: O(1)

When there is no dependence on the input size n, an algorithm is said to have a constant time of order O(1). Ex.

- Accesses (i.e. accessing value from an array) -> array[i]
- Arithmetic -> x + 5
- Assignments -> x = 5
- Comparisons -> x < 5
- Returns -> return x
- Increment/decrement -> ++1/- -1

Logarithm Time: O(log n)

When the size of the input data decreases in each step by a certain factor, an algorithm will have logarithmic time complexity. This means as the input size grows, the number of operations that need to be executed grows comparatively much slower. When an algorithm's steps are reduced by a factor, the resulting complexity is likely O(log n) Ex.

- Loops that divide the size in half each step

Linear Time: O(n)

Linear time is achieved when the running time of an algorithm increases linearly with the length of the input. This means that when a function runs for or iterates over an input size of n, it is said to have a time complexity of order O(n). Ex.

- Regular loops

Log- Linear Time: O(n log n)

Log-linear algorithms often divide a data set into smaller parts and process each piece independently. Ex.

- Sorting algorithms like Quicksort, Merge Sort, and Heapsort are examples

Quadratic Time: O(n^2)

Algorithms or operations that have quadratic time are identified as having to perform a linear time operation for *each value* in an input, not just for the input itself. Ex.

- Nested Loops

Exponential Time: O(2^n)

With each addition to the input (n), the growth rate doubles, and the algorithm iterates across all subsets of the input elements. When an input unit is increased by one, the number of operations executed is doubled.

Common Time and Space Complexities in DSA

For interviews we are mainly talking about worst-case and average cases. This means that for the worst case at worst our algorithms will run at the specified classification.

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)

Search Algorithms

Search Algorithms	Space Complexity	Time Complexity		
		Best Case	Average Case	Worst Case
Linear Search	O(1)	O(1)	O(n)	O(n)
Binary Search	O(1)	O(1)	O(log n)	O(log n)

Sorting Algorithms

Sorting Algorithms	Space Complexity	Time Complexity		
		Best Case	Average Case	Worst Case
Selection Sort	O(1)	O(n^2)	O(n^2)	O(n^2)
Insertion Sort	O(1)	O(n)	O(n^2)	O(n^2)
Bubble Sort	O(1)	O(n)	O(n^2)	O(n^2)
Quick Sort	O(log n)	O(log n)	O(n log n)	O(n log n)
Merge Sort	O(n)	O(n)	O(n log n)	O(n log n)
Heap Sort	O(1)	O(1)	O(n log n)	O(n log n)

How to Calculate Big O

To calculate Big O, there are five steps you should follow:

1. Break your algorithm/function into individual operations
2. Calculate the Big O of each operation
3. Add up the Big O of each operation together
4. Remove the constants
5. Find the highest order term - this will be what we consider the Big O of our algorithm/function and is the big O which is the slowest

```
def Example():
   l = []
   for n in range(10000):
      l = l + [n]
```

The big of Example is O(n) because O(n) is the highet order term.

Space Complexity

Many times we are also concerned with how much memory/space an algorithm uses. The notation of space complexity is the same, but instead of checking the time of operations, we check the size of the allocation of memory.

```
def spaceExample(n=10):
        '''
        Prints "hello world!" n times
        '''
        for x in range(n):
        print('Hello World!')

spaceExample()
```

spaceExample has O(1) space complexity since we only assign the 'hello world!' variable once, not every time we print (and an O(n) time complexity.)