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

Nowadays, with all these data we consume and generate every single day, algorithms must be good enough to handle operations in large volumes of data.

Computational Complexity

Computational complexity is a field from computer science which **analyzes algorithms based on the amount resources required for running it**. The amount of required resources varies based on the input size, so the complexity is generally expressed as a function of n, where n is the size of the input.

It is important to note that when analyzing an algorithm we can consider the **time complexity** and **space complexity**. The space complexity is basically the amount of memory space required to

solve a problem in relation to the input size. Even though the space complexity is important when analyzing an algorithm, in this story we will focus only on the time complexity.

Time Complexity

As you're reading this story right now, you may have an idea about what is time complexity, but to make sure we're all on the same page, let's start understanding what time complexity means with a short description from [Wikipedia](#).

In computer science, the time complexity is the computational complexity that describes the amount of time it takes to run an algorithm. Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, supposing that each elementary operation takes a fixed amount of time to perform.

When analyzing the time complexity of an algorithm we may find three cases: **best-case**, **average-case** and **worst-case**. Let's understand what it means.

Suppose we have the following unsorted list **[1, 5, 3, 9, 2, 4, 6, 7, 8]** and we need to find the index of a value in this list using [linear search](#).

- **best-case**: this is the complexity of solving the problem for the best input. In our example, the best case would be to search for the value 1. Since this is the first value of the list, it would be found in the first iteration.

- **average-case**: this is the average complexity of solving the problem. This complexity is defined with respect to the distribution of the values in the input data. Maybe this is not the best example but, based on our sample, we could say that the average-case would be when we're searching for some value in the "middle" of the list, for example, the value 2.

- **worst-case**: this is the complexity of solving the problem for the worst input of size n. In our example, the worst-case would be to search for the value 8, which is the last element from the list.

Usually, when describing the time complexity of an algorithm, we are talking about the worst-case.

We use a mathematical notation called Big-O.

Big-O Notation

Big-O notation, sometimes called "asymptotic notation", **is a mathematical notation that describes the limiting behavior of a function** when the argument tends towards a particular value or infinity.

In computer science, Big-O notation is used to classify algorithms according to how their running time or space requirements grow as the input size (*n*) grows. This notation characterizes functions according to their growth rates: different functions with the same growth rate may be represented using the same O notation.

Let's see some common time complexities described in the Big-O notation.

Table of common time complexities

These are the most common time complexities expressed using the Big-O notation:

Name	Time Complexity
Constant Time	O(1)
Logarithmic Time	O(log n)
Linear Time	O(n)
Quasilinear Time	O(n log n)
Quadratic Time	O(n^2)
Exponential Time	O(2^n)
Factorial Time	O(n!)

As already said, we generally use the Big-O notation to describe the time complexity of algorithms. There's a lot of math involved in the formal definition of the notation, but informally we can assume that the Big-O notation gives us the algorithm's approximate run time in the worst case. When using the Big-O notation, we describe the algorithm's efficiency based on the increasing size of the input data (n). For example, if the input is a string, the n will be the length of the string. If it is a list, the n will be the length of the list and so on.

Time Complexities

Constant Time — O(1)

An algorithm is said to have a constant time when it is not dependent on the input data (n). No matter the size of the input data, the running time will always be the same.

```
if a > b:
    return True
else:
    return False
```

```
def get_first(data):
return data[0]

if __name__ == '__main__':
data = [1, 2, 9, 8, 3, 4, 7, 6, 5]
print(get_first(data))
```

Independently of the input data size, it will always have the same running time since it only gets the first value from the list.

An algorithm with constant time complexity is excellent since we don't need to worry about the input size.

Logarithmic Time — O(log n)

An algorithm is said to have a logarithmic time complexity when it reduces the size of the input data in each step (it don't need to look at all values of the input data), for example:

```
for index in range(0, len(data), 3):
    print(data[index])
```

Algorithms with logarithmic time complexity are commonly found in operations on binary trees or when using binary search. Let's take a look at the example of a binary search, where we need to find the position of an element in a sorted list

Linear Time — O(n)

An algorithm is said to have a linear time complexity when the running time increases at most linearly with the size of the input data. This is the best possible time complexity when the algorithm must examine all values in the input data. For example:

```
for value in data:
    print(value)
```

Let's take a look at the example of a linear search, where we need to find the position of an element in an unsorted list

Quasilinear Time — O(n log n)

An algorithm is said to have a quasilinear time complexity when each operation in the input data have a logarithm time complexity. It is commonly seen in sorting algorithms (e.g. mergesort, timsort, heapsort).

For example: for each value in the data1 (**O(n)**) use the binary search (**O(log n)**) to search the same value in data2

```
for value in data1:
result.append(binary_search(data2, value))
```

Another, more complex example, can be found in the Mergesort algorithm. Mergesort is an efficient, general-purpose, comparison-based sorting algorithm which has quasilinear time complexity, let's see an example
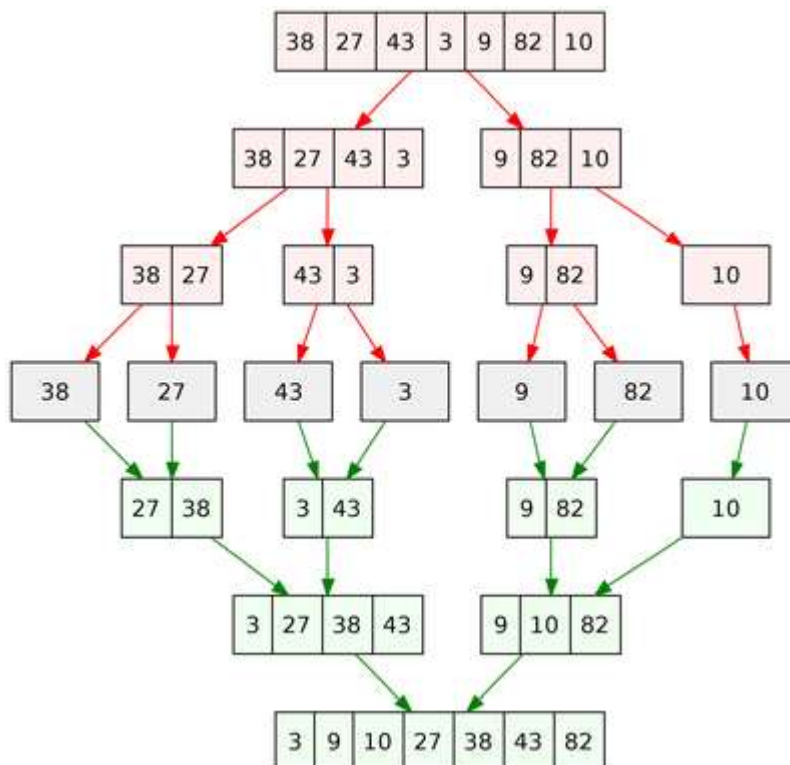
Quadratic Time — O(n²)

An algorithm is said to have a quadratic time complexity when it needs to perform a linear time operation for each value in the input data, for example:

```
for x in data:
    for y in data:
        print(x, y)
```

Bubble sort is a great example of quadratic time complexity since for each value it needs to compare to all other values in the list, let's see an example

Exponential Time — O(2^n)

An algorithm is said to have an exponential time complexity when the growth doubles with each addition to the input data set. This kind of time complexity is usually seen in brute-force algorithms.

In cryptography, a brute-force attack may systematically check all possible elements of a password by iterating through subsets. Using an exponential algorithm to do this, it becomes incredibly resource-expensive to brute-force crack a long password versus a shorter one. This is one reason that a long password is considered more secure than a shorter one

Another example of an exponential time algorithm is the recursive calculation of Fibonacci numbers

```
def fibonacci(n):
if n <= 1:
return n
return fibonacci(n-1) + fibonacci(n-2)
```

If you don't know what a [recursive function](#) is, let's clarify it quickly: a recursive function may be described as a function that calls itself in specific conditions. As you may have noticed, the time complexity of recursive functions is a little harder to define since it depends on how many times the function is called and the time complexity of a single function call
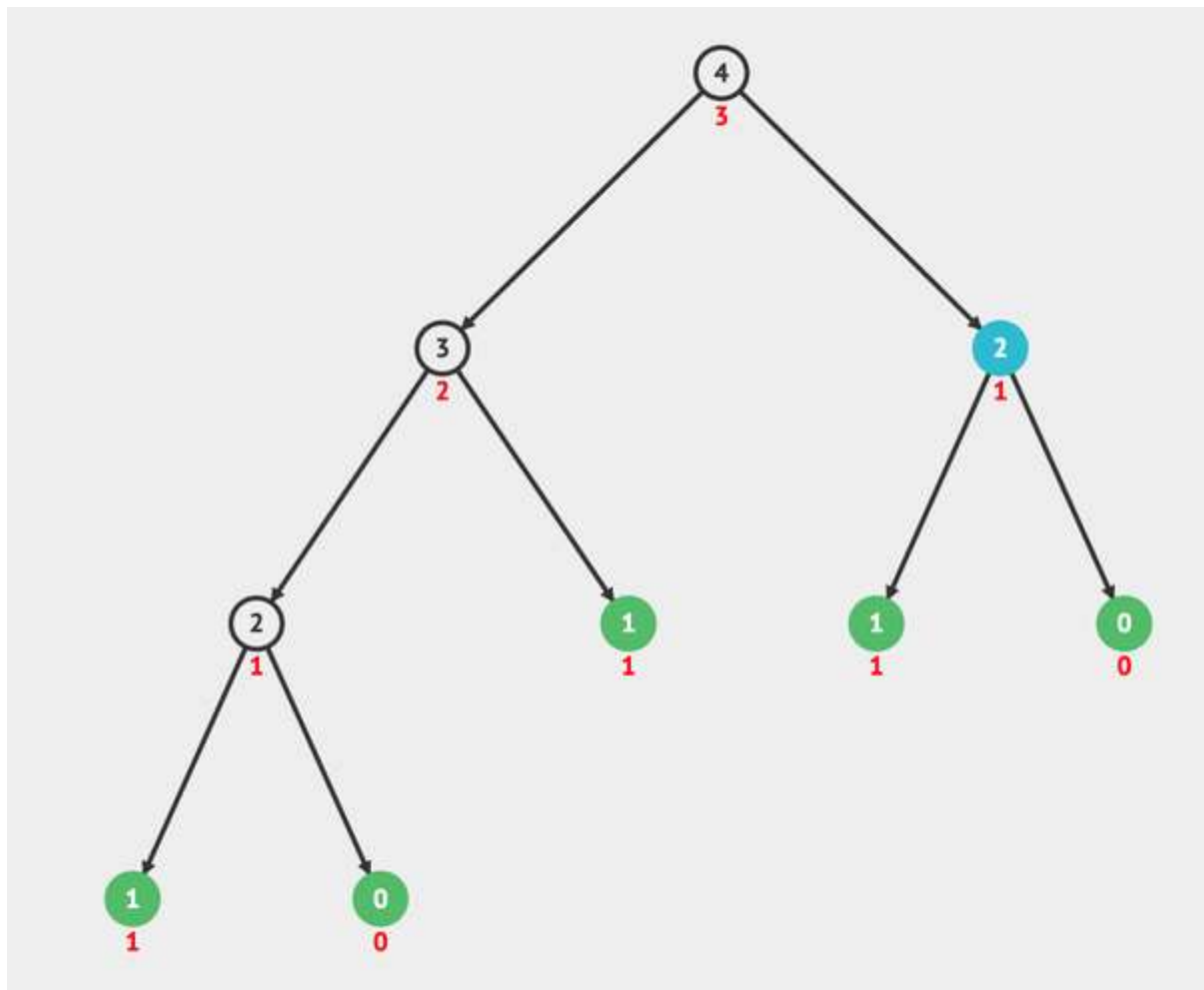
Note that it will call itself until it reaches the leaves. When reaching the leaves it returns the value itself

Factorial — O(n!)

An algorithm is said to have a factorial time complexity when it grows in a factorial way based on the size of the input data, for example:

```
2! = 2 x 1 = 2
3! = 3 x 2 x 1 = 6
4! = 4 x 3 x 2 x 1 = 24
5! = 5 x 4 x 3 x 2 x 1 = 120
6! = 6 x 5 x 4 x 3 x 2 x 1 = 720
7! = 7 x 6 x 5 x 4 x 3 x 2 x 1 = 5.040
8! = 8 x 7 x 6 x 5 x 4 x 3 x 2 x 1 = 40.320
```

A great example of an algorithm which has a factorial time complexity is the Heap's algorithm, which is used for generating all possible permutations of n objects

Note that it will grow in a factorial way, based on the size of the input data, so we can say the algorithm has factorial time complexity O(n!)

Important Notes

It is important to note that when analyzing the time complexity of an algorithm with several operations we need to describe the algorithm based on the largest complexity among all operations. For example:

```python
def my_function(data):
    first_element = data[0]

    for value in data:
        print(value)

    for x in data:
        for y in data:
            print(x, y)
```

Even that the operations in 'my_function' don't make sense we can see that it has multiple time complexities: O(1) + O(n) + O(n²). So, when increasing the size of the input data, the bottleneck of this algorithm will be the operation that takes O(n²). Based on this, we can describe the time complexity of this algorithm as O(n²).

Big-O Cheat Sheet

To make your life easier, here you can find a sheet with the time complexity of the operations in the most common data structures.

Common Data Structure Operations

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)

Here is another sheet with the time complexity of the most common sorting algorithms.

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)