22-Big-O Notation

Author: Vincent Lau

Learning Objectives

> Understand what complexity means for an algorithm.

> Understand how an algorithm works.

> Understand what Asymptotic Analysis means for Big-O Notation.

> Be able to determine the Big-O notation of an algorithm.

Overview

What is Algorithm?

- An algorithm is a finte sequence of well-deinfed computer-implementable instructions, typically to solve a class of problems or to perform a computation.

Complexity Analysis

- The process of determining how efficient an algorithm is.
- Complexity analysis usually involves finding both the time complexity and the space complexity of an algorithm.
- Both types of complexity describe how an algorithm performs as its input size increases.

Time Complexity

- A measure of how fast an algorithm runs, time complexity is a central concept in the field of algorithms.
- It is expressed using the Big-O notation.

Space Complexity

- A measure of how much auxiliary memory (輔助記憶裝置) that an algorithm takes up, space complexity is also a central concept in the filed of algorithms.
- It is expressed using the Big-O notation.

Memory

- Memory is a bounded canvas and has a finite number of memory slots.
- A computer stores variables and arrays in continuous memory slots.
- For example, an integer in Java takes up 4 bytes (or 32 bits) in memory, an array of four integers will take up at least 4 x 4 bytes in memory contiguously (plus the memory overhead of the array itself).
    - Learn more: https://www.educative.io/edpresso/what-is-integermaxvalue
- The less memory an algorithm takes, the better it performs.

Big-O Notation

- The notation O is used to describe the time complexity and space complexity of algorithms.
- Variables used in Big-O notation denote the sizes of inputs to algorithms.

  For example:

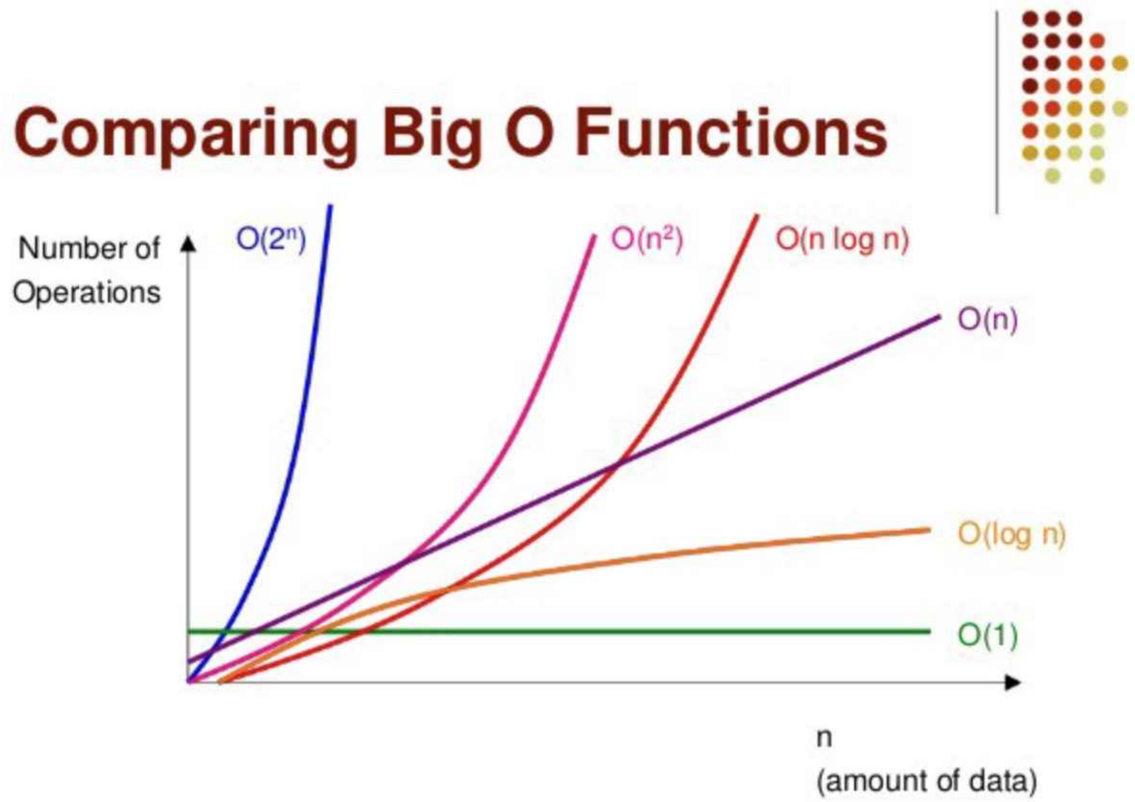
    - `O(n)` might be the time complexity of an algorithm that traverses through an array of length `n`.
    - `O(n + m)` might be the time complexity of an algorithm that traverses through an array of length `n` and through a string of length `m`.

- Big-O notation is usually understood to describe the **worst-case** complexity of an algorithm.

Asymptotic Analysis

- ***Asymptotic Analysis*** (漸近分析) means we analyze the behavior of a function as its value tends to infinity (在數學分析中是一種描述函數在極限附近的行為的方法)
- We do the same for Big-O Notation and only care about the element that has the fastest growth. To do that, we drop the **coefficients** and **constant** terms. For example:
    - `O(10)` and `O(1)` is the same in terms of Asymptotic Analysis.
    - `O(2n)` is the same as `O(n)`.
    - `O(n^2 + n + 1)` is the same as `O(n^2)`.
    - `O(m + n)` remains unchanged because `m` and `n` are different inputs.

Common Complexities



Constant: O(1)

The below example is constant time. Even if it takes 3 times as long to run, it *doesn't depend on the size of the input, n.* We denote constant time algorithms as follows: *O(1)*. Note that *O(2)*, *O(3)* or even *O(1000)* would mean the same thing.

We don't care about exactly how long it takes to run, only that it takes constant time.

```
//int n = 100000;
// data -> n
int[] arr = arr[n];
System.out.println("Hey - your input is: " + n);
System.out.println("Hmm.. I'm doing more stuff with: " + n);
System.out.println("And more: " + n);
```

Logarithmic: O(log(n))

- **Logarithmic time is the next quickest.** Unfortunately, they're a bit trickier to imagine.
- The **running time grows in proportion to the logarithm of the input (in this case, log to the base 2)**:

```
for (int i = 1; i < n; i = i * 2) {
    System.out.println("Hey - I'm busy looking at: " + i);
}
/*
when n grow from 8 to 32, the running time just grow from 3 to 5.
where log base 2 (8) = 3, log base 2 (32) = 5

If n = 8, the output will be the following:
Hey - I'm busy looking at: 1
Hey - I'm busy looking at: 2
Hey - I'm busy looking at: 4

If n = 32, the output will be the following:
Hey - I'm busy looking at: 1
Hey - I'm busy looking at: 2
Hey - I'm busy looking at: 4
Hey - I'm busy looking at: 8
Hey - I'm busy looking at: 16
*/
```

Linear: O(n)

- **The simple algorithm presented below will grow linearly with the size of its input.**
- We don't know exactly how long it will take for this to run, and don't worry about that. We just want to estimate if the algorithm is O(n).

```java
// data -> n
// Example 1, O(n)
for (int i = 0; i < n; i++) {
    System.out.println("Hey - I'm busy looking at: " + i);
}

/*
when n = 2,
Hey - I'm busy looking at: 0
Hey - I'm busy looking at: 1
when n = 4,
Hey - I'm busy looking at: 0
Hey - I'm busy looking at: 1
Hey - I'm busy looking at: 2
Hey - I'm busy looking at: 3
*/

// Example 2, O(2n) -> O(n) -> Linear
// We don't care about the coefficients, still linear
for (int i = 0; i < n; i++) {
    System.out.println("Hey - I'm busy looking at: " + i);
    System.out.println("Hmm.. Let's have another look at: " + i);
    System.out.println("And another: " + i);
}
```

Log-linear: O(nlog(n))

- ***n log n*** **is the next slower of algorithms, slower than O(n) algorithms**

```java
// O(8 * log(8))
for (int i = 1; i <= n; i++){ // n = 8
    for(int j = 1; j < n; j = j * 2) { // log base 2 (8)
        System.out.println("Hey - I'm busy looking at: " + i + " and " + j);
    }
}
/*
if the n is 8, then this algorithm will run 8 * log base2(8) = 8 * 3 = 24 times
Hey - I'm busy looking at:  1 and 1
Hey - I'm busy looking at:  1 and 2
Hey - I'm busy looking at:  1 and 4
Hey - I'm busy looking at:  2 and 1
Hey - I'm busy looking at:  2 and 2
Hey - I'm busy looking at:  2 and 4
... ...
... ...
```

```
Hey - I'm busy looking at:  8 and 1
Hey - I'm busy looking at:  8 and 2
Hey - I'm busy looking at:  8 and 4
*/
```

Quadratic: O(n^2)

- Slower than *n log n* algorithms
- The important message here is, ***O(n(2))* is faster than *O(n(3))* which is faster than *O(n(4))*, etc.**

```
// O(n^2), when n = 3, this algorithm will run 3^2 = 9 times.
for (int i = 1; i <= n; i++) {
    for(int j = 1; j <= n; j++) {
        System.out.println("Hey - I'm busy looking at: " + i + " and " + j);
    }
}
// One more nested loop
// O(n^3), when n = 3, this algorithm will run 3^3 = 27 times.
for (int i = 1; i <= n; i++) {
    for(int j = 1; j <= n; j++) {
        for(int k = 1; k <= n; k++) {
            System.out.println("Hey - I'm busy looking at: " + i + " and " + j + " and " + k);
    }
}
// Output
// ... Skip here, you can imagine
```

Other Complexities

Expontential: `O(2^n)`

```
//O(2^8), when n = 8, This algorithm will run 2^8 = 256 times
for (int i = 1; i <= Math.pow(2, n); i++){
    System.out.println("Hey - I'm busy looking at: " + i);
}
```

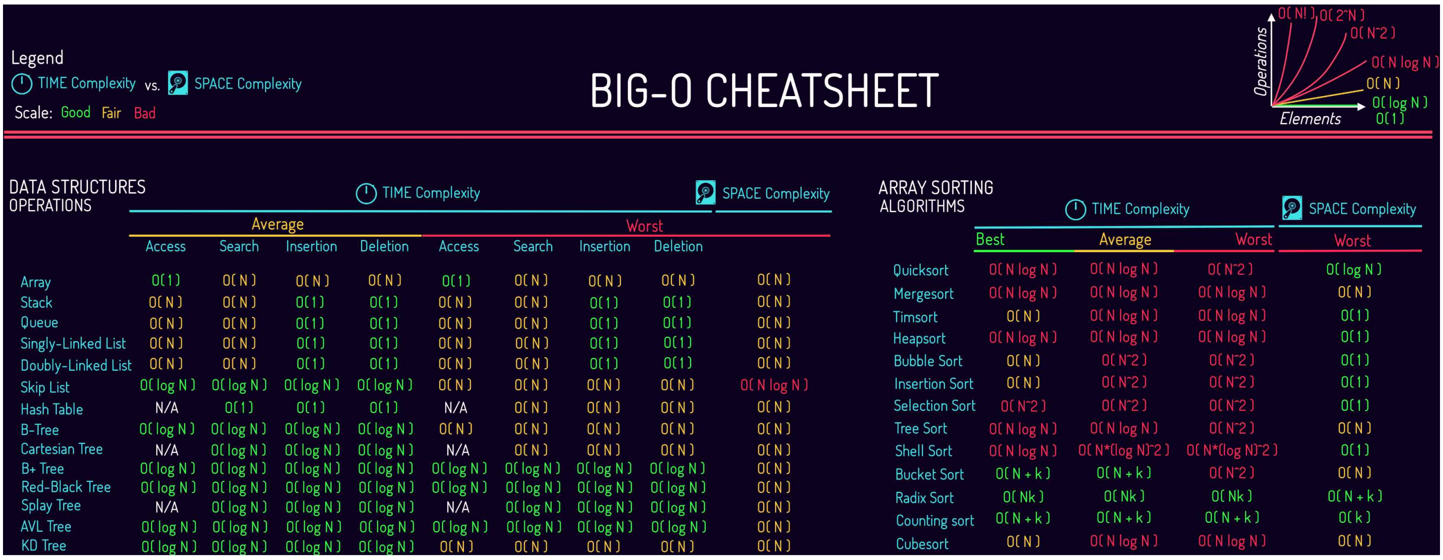
Factorial: `O(n!)`

```java
// factorial(n) simply calculates n!
// If n is 8, this algorithm will run 8! = 8*7*6*5*4*3*2*1 = 40320 times.
for (int i = 1; i <= factorial(n); i++){
    System.out.println("Hey - I'm busy looking at: " + i);
}

public static long factorial(int num) {
    if (num <= 1) {
        return num;
    }
    return num * factorial(num - 1);
}
```

BIG-O Cheat Sheet

- This course will not go through the following data structure or algorithm one by one (it is relatively not important at this moment).
- Provide an overview and feeling of how efficient these algorithms are.
- In this chapter, please try to understand the algorithm causes the complexity growth, in terms of BIG-O.


Legend
TIME Complexity vs. SPACE Complexity
Scale: Good Fair Bad

BIG-O CHEATSHEET

O(N!) O(2^N) O(N^2) O(N log N) O(N) O(log N) O(1)
Operations
Elements

DATA STRUCTURES OPERATIONS

	TIME Complexity								SPACE Complexity
	Average				Worst				
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	O(1)	O(N)	O(N)	O(N)	O(1)	O(N)	O(N)	O(N)	O(N)
Stack	O(N)	O(N)	O(1)	O(1)	O(N)	O(N)	O(1)	O(1)	O(N)
Queue	O(N)	O(N)	O(1)	O(1)	O(N)	O(N)	O(1)	O(1)	O(N)
Singly-Linked List	O(N)	O(N)	O(1)	O(1)	O(N)	O(N)	O(1)	O(1)	O(N)
Doubly-Linked List	O(N)	O(N)	O(1)	O(1)	O(N)	O(N)	O(1)	O(1)	O(N)
Skip List	O(log N)	O(log N)	O(log N)	O(log N)	O(N)	O(N)	O(N)	O(N)	O(N log N)
Hash Table	N/A	O(1)	O(1)	O(1)	N/A	O(N)	O(N)	O(N)	O(N)
B-Tree	O(log N)	O(log N)	O(log N)	O(log N)	O(N)	O(N)	O(N)	O(N)	O(N)
Cartesian Tree	N/A	O(log N)	O(log N)	O(log N)	N/A	O(N)	O(N)	O(N)	O(N)
B+ Tree	O(log N)	O(log N)	O(log N)	O(log N)	O(log N)	O(log N)	O(log N)	O(log N)	O(N)
Red-Black Tree	O(log N)	O(log N)	O(log N)	O(log N)	O(log N)	O(log N)	O(log N)	O(log N)	O(N)
Splay Tree	N/A	O(log N)	O(log N)	O(log N)	N/A	O(log N)	O(log N)	O(log N)	O(N)
AVL Tree	O(log N)	O(log N)	O(log N)	O(log N)	O(log N)	O(log N)	O(log N)	O(log N)	O(N)
KD Tree	O(log N)	O(log N)	O(log N)	O(log N)	O(N)	O(N)	O(N)	O(N)	O(N)

ARRAY SORTING ALGORITHMS

	TIME Complexity			SPACE Complexity
	Best	Average	Worst	Worst
Quicksort	O(N log N)	O(N log N)	O(N^2)	O(log N)
Mergesort	O(N log N)	O(N log N)	O(N log N)	O(N)
Timsort	O(N)	O(N log N)	O(N log N)	O(1)
Heapsort	O(N log N)	O(N log N)	O(N log N)	O(1)
Bubble Sort	O(N)	O(N^2)	O(N^2)	O(1)
Insertion Sort	O(N)	O(N^2)	O(N^2)	O(1)
Selection Sort	O(N^2)	O(N^2)	O(N^2)	O(1)
Tree Sort	O(N log N)	O(N log N)	O(N^2)	O(N)
Shell Sort	O(N log N)	O(N*(log N)^2)	O(N*(log N)^2)	O(1)
Bucket Sort	O(N + k)	O(N + k)	O(N^2)	O(N)
Radix Sort	O(Nk)	O(Nk)	O(Nk)	O(N + k)
Counting sort	O(N + k)	O(N + k)	O(N + k)	O(k)
Cubesort	O(N)	O(N log N)	O(N log N)	O(N)

Self-Learning

[Complexities & DSA]

Questions

- What is an algorithm?

- What does complexity mean for an algorithm?
- What are the two types of complexity?
- What does Asymptotic Analysis mean for Big-O Notation?

References

- *Big-O Cheat Sheet*. https://www.bigocheatsheet.com/.