Data Structures and Algorithms
University of New Brunswick
Fredericton, New Brunswick, Canada

Spring 2023

Instructor:	Syed Eqbal Alam	**Time:**	T Th 1:00pm – 2:20pm
Email:	syed.eqbal@unb.ca	**Place:**	GWC 127, Gillin Hallway

Algorithm Analysis

1 Introduction

- Through algorithm analysis, we should be able to find out how much computational resource (CPU time, memory, storage or network bandwidth, etc.) an algorithm may need to complete the task, also referred to as *computational complexity*.

  We can analyze:

  - Running time or execution time (Time complexity)
  - Space usage (Space complexity)
  - Network bandwidth usage (Communication complexity)

- We can find out which algorithm performs better by implementing them and then calculating the difference between the Start Time and End Time of the programs. Better algorithm will have lesser execution (running) time given the same input.

- Furthermore, we can also find how much memory or storage an algorithm is taking, better Algorithm should take less space given the same input.

Example 1.1. *If Algorithm A takes five millisecond and Algorithm B takes 20 millisecond to execute, Then, we say that Algorithm A is better than Algorithm B in terms of running time.*

The main issues with the empirical (experimental) analysis are as follows:

- We need to implement the Algorithms, which could be difficult or time consuming to do.
- The running time is hardware and software dependent, latest computers will run a program faster than the older generation computers.
- It is not possible to check the algorithms on all possible inputs.

- Therefore, we need ways to analyze algorithms without their real implementation. Furthermore, the analysis should be independent of hardware and software environments, and considers all possible inputs.
  - Generally speaking, the analysis should be done on a high-level description of the algorithms.

- We can calculate approximate number of basic or primitive operations of algorithms to analyze the algorithms. We assume that basic operations take approximately the same execution time. Moreover, the total number of basic operations are proportional to the running time of an algorithm.

2 Asymptotic Analysis

We find the relationship between the input size and the number of primitive or basic operations required to complete a task.

- Notice that we are not interested in the exact running time; however, we want to find out approximate value and say how an algorithm performs when the input size increases or decreases. Such type of analysis is called *Asymptotic analysis*.

The following are some of the basic or primitive operations:

- Value assignment to a variable.
- Arithmetic operations (addition, subtraction, multiplication, division, increment, decrement, modulus, etc.).
- Comparison operations (equal, less than, less than equal, greater than, greater than equal, not equal, etc.)
- Logical operations (Logical or ||, Logical and &&, Logical not !).
- Calling a method.
- Returning a value from a method.
- Accessing an element of an array.

Operations as a function of input size:
We can write an algorithm of input size N as a function of basic operations.

Algorithm 1: Average value of all numbers in the array $arrayA$.

```
1 Input: N, arrayA;
2 Output: Average value of all numbers in the array arrayA.
3 int total = 0;
4 double avgVal;
5 for (i = 0; i < N; i++)
6       total = total + arrayA[i];
7 avgVal = total/N;
8 return avgVal;
```

Example 2.1. *Let us consider Algorithm 1, which calculates the average value of all values of an array.*

Line 3 takes one operation to execute—it is an assignment operation.

Line 6 repeats N times because of the for loop in Line 5; each iteration takes one operation to execute. So total, N operations are required to execute Line 6.

Lines 7 and 8 will take one operation each.

Therefore, the total number of operations required to execute Algorithm 1 will approximately be $(1 + 1 + N + 1 + 1) = N + 4$.

Let $\mathbb{N}_+ = \{1, 2, \ldots\}$ *denote the set of natural numbers starting from* 1*, and let* $\mathbb{R}_+$ *denote the set of positive real numbers. Furthermore, let* $f : \mathbb{N}_+ \to \mathbb{R}_+$ *denote a map between the given input size of the Algorithm and the total number of operations required to complete the task.*

Thus for Algorithm 1, we formulate $f(N) = N + 4$.

We study the following bounds on the computational complexity of an algorithm.

- Upper bound (Big Oh O)
- Average bound (Big Theta Θ)
- Lower bound (Big Omega Ω)

2.1 Upper bound, Big Oh

The upper bound of an Algorithm tells us that for a given input size or problem size, the algorithm will not take more operations (or time or space, etc.) than the upper bound to complete the task.

■ Think it as, if an algorithm performs better in the worst case then it will be perform better on all inputs than other algorithms.

Let $\mathbb{N}_+$ denote the set of natural numbers starting from 1 and $\mathbb{R}_+$ denote the set of positive real numbers. Furthermore, let $f : \mathbb{N}_+ \to \mathbf{R}_+$ and $g : \mathbb{N}_+ \to \mathbf{R}_+$ be functions mapped from input size to the number of operations required.

Given an input or problem size n, we call $f(n)$ is $O(g(n))$ if for any constant $c > 0$, we have

$$f(n) \leq cg(n), \quad n \geq n_0. \tag{1}$$

- We also say that $f(n)$ is the order of $O(g(n))$ or $f(n) \in O(g(n))$.

Example 2.2. *Show that Algorithm 1 is of the order of $O(n)$.*

Solution: *For Algorithm 1, we have $f(n) = n + 4$, for input size n.*

We obtain that

$$f(n) = n + 4 \leq n + 4n = 5n, \quad \text{for } n \geq 1.$$

Let $c = 5$ and $g(n) = n$. Thus, from Equation (1), we obtain that $f(n)$ is $O(g(n))$. As $g(n) = n$; hence, we say that $f(n)$ is $O(n)$.

Example 2.3. *Let for any input $n \geq 1$, Algorithm X takes a total of 50 operations to complete the task, then the Algorithm X is $O(1)$.*

Solution: Given that

$$f(n) = 50, \text{ for } n \geq 1.$$

We obtain

$$f(n) = 50 \leq 50 \times 1, \text{ for } n \geq 1.$$

Let $c = 50$ and $g(n) = 1$. Thus, from Equation (1), we obtain that $f(n)$ is $O(g(n))$. As $g(n) = 1$; hence, we say that $f(n)$ is $O(1)$.

Remark 2.4. *O(1) has a constant growth rate—the growth of the function $f(n)$ is independent of the input size n. No matter how large the input size n, the execution time or running time will always be constant.*

Example 2.5. *Let for the given input size $n \geq 2$ if an Algorithm takes $2\log_2 n + 3$ operations to complete the task. Then the Algorithm grows asymptotically as $O(\log_2 n)$.*

Solution: Given

$$f(n) = 2\log_2 n + 3, \text{ for } n \geq 1.$$

As for $n \geq 2$, we have $\log_2) \geq 1$. We obtain

$$f(n) = 3 + 2\log_2 n \leq 3\log_2 n + 2\log_2 n = 5\log_2 n, \text{ for } n \geq 2.$$

Let $c = 5$ and $g(n) = \log_2 n$, then we obtain

$$f(n) \leq c \cdot g(n), \text{ for } n \geq 2.$$

Thus, from Equation (1), we obtain that $f(n)$ is $O(g(n))$, where $g(n) = \log_2 n$. Hence, we say that $f(n)$ is $O(\log_2 n)$.

Remark 2.6. *$O(\log n)$ has a logarithmic growth rate—when the input size n increases, the growth of the function $f(n)$ increases in logarithmic terms; that is, the running time also increases in logarithmic terms.*

Example 2.7. *For $n \geq 1$, if $f(n) = 3n + 2$. Then $f(n)$ is $O(n)$.*

Solution: Given

$$f(n) = 3n + 2, \text{ for } n \geq 1.$$

We obtain

$$f(n) = 3n + 2 \leq 3n + 2n = 5n, \text{ for } n \geq 1.$$

Let $c = 5$ and $g(n) = n$, then we obtain

$$f(n) \leq c \cdot g(n), \text{ for } n \geq 1.$$

Thus, from Equation (1), we obtain that $f(n)$ is $O(g(n))$, where $g(n) = n$. Hence, we say that $f(n)$ is $O(n)$.

Example 2.8. *For $n \geq 1$, if $f(n) = 3n + 2\log_2(n)$. Then $f(n)$ is $O(n)$.*

Solution: Given

$$f(n) = 3n + 2\log_2(n), \text{ for } n \geq 1.$$

As for $n \geq 1$, we have $n \geq \log_2(n)$. We obtain

$$f(n) = 3n + 2\log_2(n) \leq 3n + 2n = 5n, \text{ for } n \geq 1.$$

Let $c = 5$ and $g(n) = n$, then we obtain

$$f(n) \leq c \cdot g(n), \text{ for } n \geq 1.$$

Thus, from Equation (1), we obtain that $f(n)$ is $O(g(n))$, where $g(n) = n$. Hence, we say that $f(n)$ is $O(n)$.

Remark 2.9. *$O(n)$ has a linear growth rate that is when the input size n increases, the growth of the function $f(n)$ increases linearly. For example, if the input size increases to $2n$ then the running time also doubles.*

Example 2.10. *For $n \geq 2$, if $f(n) = 3n + 2n\log_2 n$. Then $f(n)$ is $O(n\log_2 n)$.*

Solution: Given

$$f(n) = 3n + 2\log_2 n, \text{ for } n \geq 2.$$

As for $n \geq 2$, we have $n\log_2 n \geq n$. We obtain

$$f(n) = 3n + 2n\log_2 n \leq 3n\log_2 n + 2n\log_2 n = 5n\log_2 n, \text{ for } n \geq 2.$$

Let $c = 5$ and $g(n) = n\log_2 n$, then we obtain

$$f(n) \leq c \cdot g(n), \text{ for } n \geq 2.$$

Thus, from Equation (1), we obtain that $f(n)$ is $O(g(n))$, where $g(n) = n\log_2 n$. Hence, we say that $f(n)$ is $O(n\log_2 n)$.

Example 2.11. *For $n \geq 1$, if $f(n) = 3n^2 + 2n + 7$. Then $f(n)$ is $O(n^2)$.*

Solution: Given

$$f(n) = 3n^2 + 2n + 7, \text{ for } n \geq 1.$$

As for $n \geq 1$, we have $n^2 \geq n$. We obtain

$$f(n) = 3n^2 + 2n + 7 \leq 3n^2 + 2n^2 + 7n^2 = 12n^2, \text{ for } n \geq 1.$$

Let $c = 12$ and $g(n) = n^2$, then we obtain

$$f(n) \leq c \cdot g(n), \text{ for } n \geq 1.$$

Thus, from Equation (1), we obtain that $f(n)$ is $O(g(n))$, where $g(n) = n^2$. Hence, we say that $f(n)$ is $O(n^2)$.

Remark 2.12. *$O(n^2)$ has a quadratic growth rate; that is when the input size n increases, the growth of the function $f(n)$ increases in a quadratic way. For example, if the input size increases to $2n$, then the running time becomes 4 times that of $f(n)$.*

2.2 Commonly used growth functions:

(i) Constant functions; example, $f(n) = c$, where $c > 0$. Here, $f(n)$ is $O(1)$.

(ii) Logarithmic functions; example, $f(n) = \log n$. Here, $f(n)$ is $O(\log n)$.

(iii) Linear functions; example, $f(n) = n$. Here, $f(n)$ is $O(n)$.

(iv) $n\log n$ functions; example, $f(n) = n\log n$. Here, $f(n)$ is $O(n\log n)$.

(v) Quadratic functions; example, $f(n) = n^2$. Here, $f(n)$ is $O(n^2)$.

(vi) Cubic functions; example, $f(n) = n^3$. Here, $f(n)$ is $O(n^3)$.

(vii) Exponential functions; example, $f(n) = 2^n$. Here, $f(n)$ is $O(2^n)$.

Remark 2.13. *An algorithm of order $O(1)$ is* **asymptotically better** *than the Algorithm of the order of $O(n)$ or $O(\log_2 n)$. Similarly, we state for others, refer Table 1.*

Question 2.14. *For $n \geq 2$, if $f(n) = 3n + 2n\log_2(n)$. Then show that $f(n)$ is order of $O(n\log_2 n)$.*

Question 2.15. *For $n \geq 1$, if $f(n) = 3n^2 + 2n\log_2(n)$. Then show that $f(n)$ is $O(n^2)$.*

Question 2.16. *For $n \geq 1$, if $f(n) = 3n^3 + 2n^2 + 5n\log_2(n)$. Then show that $f(n)$ is $O(n^3)$.*

Question 2.17. *For $n \geq 1$, if $f(n) = 3n^3 + 2^n$. Then show that $f(n)$ is $O(2^n)$.*

Question 2.18. *For $n \geq 4$, if $f(n) = 2^n + n!$. Then show that $f(n)$ is $O(n!)$.*

Constant	Logarithmic	Linear	n log n	Quadratic	Cubic	Exponential
1	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n

Table 1: Growth rate of functions of input size n, for $n \geq 10$, in increasing order.

2.3 Big Omega (Lower bound)

Let $\mathbb{N}$ denote the set of natural numbers and $\mathbb{R}_+$ denote the set of positive real numbers. Furthermore, let $f : \mathbb{N} \to \mathbf{R}_+$ and $g : \mathbb{N} \to \mathbf{R}_+$ be functions.

Given an input or problem size n, we call $f(n)$ is $\Omega(g(n))$ if for any constant $c > 0$, we have

$$f(n) \geq c \cdot g(n), \quad n \geq n_0. \tag{2}$$

We also say that $f(n)$ is $\Omega(g(n))$ if $g(n)$ is $O(f(n))$.

Example 2.19. *For $n \geq 1$, if $f(n) = 3n + 2$. Then $f(n)$ is $\Omega(\log_2 n)$.*

Solution: Given

$$f(n) = 3n + 2, \text{ for } n \geq 1.$$

We obtain

$$f(n) = 3n + 2 \geq 3 \log_2 n + 2 \geq 3 \log_2 n, \text{ for } n \geq 1.$$

Let $c = 3$ and $g(n) = \log_2 n$, then we obtain

$$f(n) \geq c \cdot g(n), \text{ for } n \geq 1.$$

Thus, from Equation (2), we obtain that $f(n)$ is $\Omega(g(n))$, where $g(n) = \log_2 n$. Hence, we say that $f(n)$ is $\Omega(\log_2 n)$.

Example 2.20. *For $n \geq 1$, if $f(n) = 3n^2 - 2n$. Then $f(n)$ is $\Omega(n \log_2 n)$.*

Solution: Given

$$f(n) = 3n^2 - 2n, \text{ for } n \geq 1.$$

We obtain

$$f(n) = 3n^2 - 2n \geq n \log_2 n, \text{ for } n \geq 1.$$

Let $c = 1$ and $g(n) = n \log_2 n$, then we obtain

$$f(n) \geq c \cdot g(n), \text{ for } n \geq 1.$$

Thus, from Equation (2), we obtain that $f(n)$ is $\Omega(g(n))$, where $g(n) = n \log_2 n$. Hence, we say that $f(n)$ is $\Omega(n \log_2 n)$.

2.4 Big Theta (Average bound)

Let $\mathbb{N}$ denote the set of natural numbers and $\mathbb{R}_+$ denote the set of positive real numbers. Furthermore, let $f : \mathbb{N} \to \mathbf{R}_+$ and $g : \mathbb{N} \to \mathbf{R}_+$ be functions.

Given an input size n, we call $f(n)$ is $\Theta(g(n))$ if for any constants $c_1 > 0$ and $c_2 > 0$, we have

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n), \quad n \geq n_0. \tag{3}$$

We also say that $f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(f(n))$ and $\Omega(g(n))$.

Remark 2.21. *Functions $f(n)$ and $g(n)$ grow at the same rate up to constant growth factors.*

Example 2.22. *For $n \geq 2$, if $f(n) = 3n \log n + 2n$. Then $f(n)$ is $\Theta(n \log_2 n)$.*

Solution: Given

$$f(n) = 3n \log n + 2n, \text{ for } n \geq 2.$$

We obtain

$$3n \log n \leq f(n) = 3n \log n + 2n \leq 3n \log n + 2n \log n, \text{ for } n \geq 2.$$

Furthermore, we get

$$3n \log n \leq f(n) = 3n \log n + 2n \leq 5n \log n, \text{ for } n \geq 2.$$

Let $c_1 = 3$ and $c_2 = 5$, and let $g(n) = n \log_2 n$, then we obtain

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n), \text{ for } n \geq 2.$$

Thus, from Equation (3), we obtain that $f(n)$ is $\Theta(g(n))$, where $g(n) = n \log_2 n$. Hence, we say that $f(n)$ is $\Theta(n \log_2 n)$.

Remark 2.23. *Unless otherwise specified, by* log *we mean* log *of base* 2.

2.5 Try these questions

Question 2.24. *Show that the continuous integer division of a number until it reaches* 1 *is $O(\log n)$.*

Hint:
for(int $i = num; i > 1; i = i/2$)
System.out.println(i + " ");

———
For num = 32;
Output:
32
16
8
4
2

Question 2.25. *Show that the Algorithm to find a number is even or odd is $O(1)$.*

Question 2.26. *Show that the Algorithm to find the maximum value of an array is $O(n)$.*

Question 2.27. *Show that the Binary search algorithm is $O(\log n)$.*

Question 2.28. *Show that calculating the minimum value of an array is $O(n)$.*

Question 2.29. *Show that bubble sort is* $O(n^2)$.

Question 2.30. *Show that Insertion sort is* $O(n^2)$.

Question 2.31. *Show that Merge sort is* $O(n \log n)$.

Question 2.32. *Find out the time complexity of Quick sort (the lower bound, average bound, and the upper bound).*

Question 2.33. *Show that matrix addition is* $O(n^2)$.

Question 2.34. *Show that matrix multiplication is* $O(n^3)$.

Question 2.35. *If* $f(n) = \sum_{i=1}^{n} i$, *then show that* $f(n)$ *is* $O(n^2)$.

Question 2.36. *If* $f(n) = \sum_{i=1}^{n} i^2$, *then show that* $f(n)$ *is* $O(n^3)$.

2.5.1 Stack and Queue methods' growth rates (see Chapter 6 of the reference book [1])

Method	size	isEmpty	top	push	pop
Running time	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$

Table 2: Growth rate of array-based Stack implementation.

Method	size	isEmpty	first	enqueue	dequeue
Running time	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$

Table 3: Growth rate of array-based Queue implementation.

Remark 2.37. *In array-based Stack and Queue implementations, space utilization is* $O(N)$.

Reference book: [1]

References

[1] Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser. *Data Structures and Algorithms in Java*. Wiley, 6th edition, 2014. 8