



Ahmad Albarasy · Follow

Last edited by Ahmad Albarasy on Nov 28, 2025 

Contributed by



DsaA Lab 3 - Algorithm Analysis

The Islamic University of Gaza
Engineering Faculty
Department of Computer Engineering

Prepared by: Eng. Ahmad Albarasy

Under the supervision of: Prof. Ahmad Mahdi

In this lab, we'll learn how to analyze our algorithms in terms of time and space complexity, which is a crucial skill to have as an engineer. Understanding complexity helps us evaluate how an algorithm performs with different input sizes and compare it with others to determine which one is faster and more efficient.

Why do we focus on the Worst-Case Scenario?

You'll often find people studying the **worst-case scenario** or the **worst-case input**. But why is that you may ask?

Well, the main reason behind that is because the **average-case** is usually hard to determine, given the different set of inputs that our algorithm might get tested with, and their probability of occurrence.

For instance, take a look at this chart from our book:

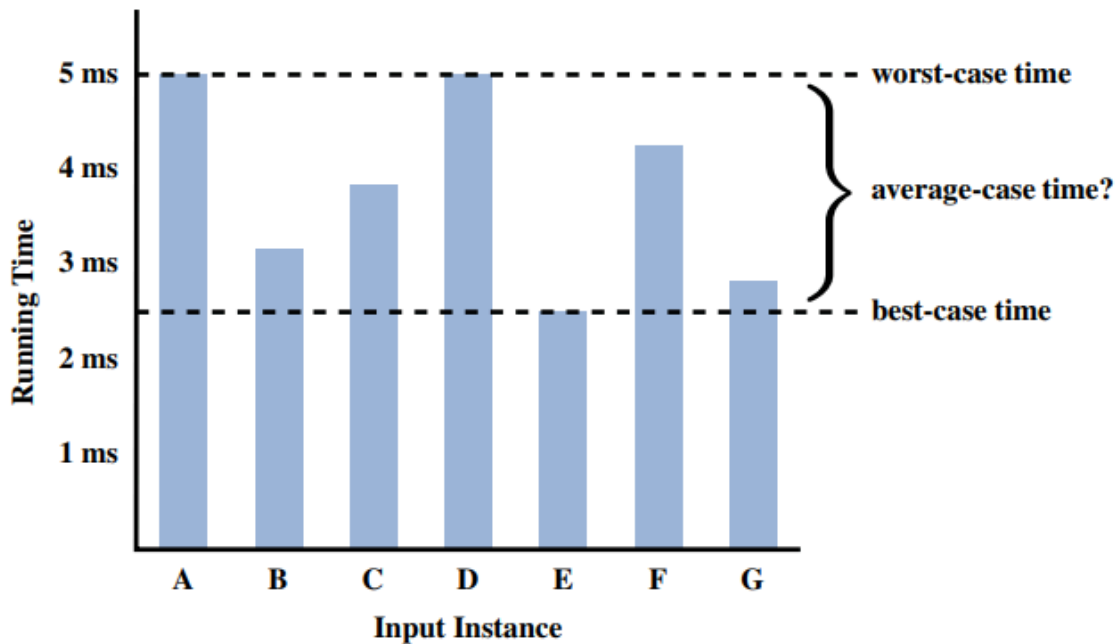


Figure 4.2: The difference between best-case and worst-case time. Each bar represents the running time of some algorithm on a different possible input.

Say that, for example, our inputs are really only of types `A` or `D`. In that case, we are far from accurately measuring the average-case.

An average-case analysis usually requires that we calculate expected running times based on a given input distribution, which usually involves sophisticated probability theory.

To give a quick glimpse on how we analyze average case, consider the following simple linear search algorithm that iterates through an array of integers `arr` and returns the index of our `target`. For simplicity, we will also assume that there is always an answer:

```

1  int linearSearch(int[] arr, int target) {
2      for (int i = 0; i < arr.length; i++) {
3          if (arr[i] == target) {
4              return i;
5          }
6      }
7      return -1; // won't happen based on our assumption
8  }
```

Intuitively, we know that the best case is finding our target at the first index, taking only one iteration. In contrast, the worst case is that our target is located at index `n-1` at the end of the array.

Now, to calculate the average case, we use simple mathematics:



$$E = \frac{1 + 2 + 3 + \dots + n}{n} = \frac{n + 1}{2}$$

But life isn't always that simple, because this result is based on the assumption that the probability distribution of each possible input is a uniform one :)

If that's not the case, we need to account for the probability of finding the target at each position. In other words, each position



i has its own probability P_i , and the expected number of comparisons becomes:



$$E = (1 \times P_1) + (2 \times P_2) + \dots + (n \times P_n)$$

When earlier elements are more likely to be the target, the expected value



E becomes smaller, meaning the algorithm performs better *on average* than in the uniform case.

For that reason, the average case analysis is rarely used in practice, and that comes in the favor of the Worst-case analysis, because it is much easier than average-case analysis, as it requires only the ability to identify the worst-case input, which is often simple.

Also, this approach typically leads to better algorithms, making the standard of success for an algorithm to perform well in the worst case necessarily requires that it will do well on every input. That is, designing for the worst case leads to stronger algorithmic “muscles,” much like a track star who always practices by running up an incline.

Measuring an Algorithm Efficiency

As mentioned in the introduction, our goal is to analyze and evaluate the efficiency of algorithms. We have two ways of doing so:

1. Experimental Studies
2. Theoretical Analysis

Experimental Studies

One way to study the efficiency of an algorithm is to implement it and experiment by running the program on various test inputs while recording the time spent during each execution. A simple mechanism for collecting such running times in Java is based

on use of the `currentTimeMillis` method of the `System` class. That method reports the number of milliseconds that have passed since a benchmark time known as the epoch (January 1, 1970 UTC):

```
1 | long startTime = System.currentTimeMillis(); // record the starting time
2 | /* (run the algorithm) */
3 | long endTime = System.currentTimeMillis(); // record the ending time
4 | long elapsed = endTime - startTime; // compute the elapsed time
```

For extremely quick operations, Java provides a method `nanoTime` that measures in nanoseconds rather than milliseconds.

Experimenting with algorithms in this way can be helpful for getting a general idea of how one algorithm performs compared to another. However, the measured times reported by both methods `currentTimeMillis` and `nanoTime` will vary greatly from machine to machine, and may likely vary from trial to trial, even on the same machine. This is because many processes share use of a computer's central processing unit (or CPU) and memory system; therefore, the elapsed time will depend on what other processes are running on the computer when a test is performed. While the precise running time may not be dependable, experiments are quite useful when comparing the efficiency of two or more algorithms, so long as they gathered under similar circumstances.

Theoretical Analysis

Our goal is to develop an approach to analyzing the efficiency of algorithms that:

1. Allows us to evaluate the relative efficiency of any two algorithms in a way that is independent of the hardware and software environment.
2. Is performed by studying a high-level description of the algorithm without need for implementation.
3. Takes into account all possible inputs.

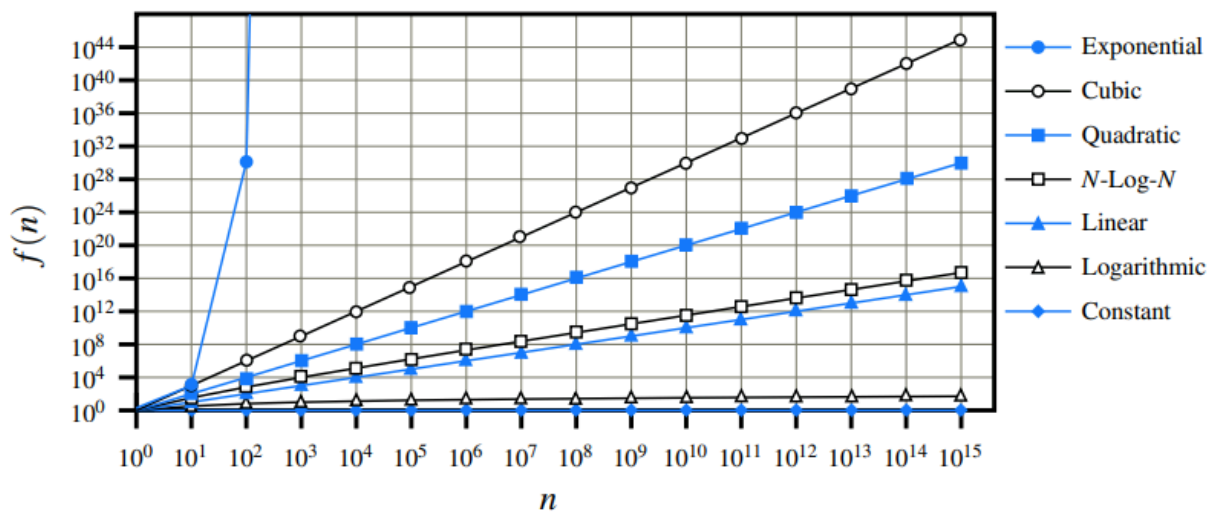
To achieve this goal, we use theoretical analysis, where a high-level description of the algorithm is sufficient for evaluation, eliminating the need for full implementation.

Random Access Machine Model (RAM)

The RAM model is an idealized model of computation used to analyze algorithms in a way that reflects how real computers work but in a simplified form.

Think of it like an imaginary computer that has a CPU and almost an infinite amount of memory, where each memory access takes the same amount of time.

The Seven Functions used for Theoretical Analysis



Fun fact

The Travelling Salesman Problem (TSP) is one of the most famous problems in computer science. It asks for the shortest route that visits every city once and returns to the starting point. Even though it sounds simple, no one has ever found an efficient (polynomial-time) solution for it. The best exact algorithms still take exponential time, making it a classic example of a problem that resists optimization to this day.

Defining Primitive Operations

When we analyze an algorithm's efficiency, we don't measure real time on a computer. Instead, we count primitive operations. Primitive Operations are the basic computations that a computer can perform in a single step, such as:

1. Evaluating an expression: $(x + y) * z$
2. Assigning a value to a variable: $x = 10$
3. Following an object reference: `arr[2]` , `list.length`
4. Calling a method: `sort(arr)`
5. returning from a method: `return false`

Asymptotic Analysis

Asymptotic analysis studies how the running time or space requirements of an algorithm grow as the input size n increases toward infinity.

- It ignores constant factors and lower-order terms that become insignificant for large inputs.

- It provides a high-level estimate of an algorithm's efficiency.

The Big Oh Notation

Let



$f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers. We say that $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that:



$$f(n) \leq c \cdot g(n), \quad \text{for } n \geq n_0$$

This definition is often referred to as the “big-Oh” notation, for it is sometimes pronounced as “ $f(n)$ is



big – Oh of $g(n)$.”

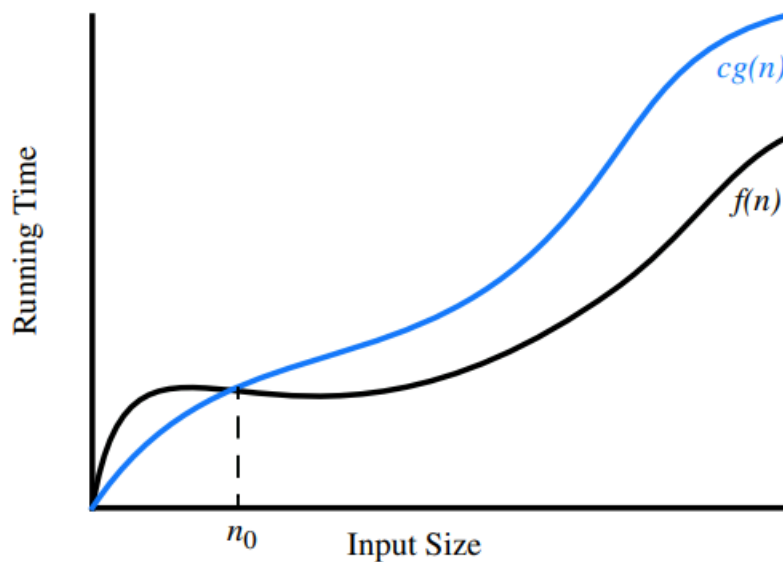


Figure 4.5: Illustrating the “big-Oh” notation. The function $f(n)$ is $O(g(n))$, since $f(n) \leq c \cdot g(n)$ when $n \geq n_0$.

Tasks

Task 1 (5 points)

You are given the following algorithm that generates all 2-element combinations from an array of unique integers:

```
1  public class FindCombinations {
2      public static int findCombinations(int[] arr) {
3
4          int n = arr.length;
5          int total = n * (n - 1) / 2;
6          int[][] result = new int[total][2];
7
8          int index = 0;
9          for (int i = 0; i < n; i++) {
10             for (int j = i + 1; j < n; j++) {
11                 result[index][0] = arr[i];
12                 result[index][1] = arr[j];
13                 index++;
14             }
15         }
16         return total;
17     }
18 }
```

Analyze the algorithm's running time and determine its order of growth using the Big-Oh notation.

Task 2 (5 points)

You are given the following algorithm that returns the middle node of a Singly Linked List:

```
1  /*
2  * class Node {
3  *   int data;
4  *   Node next;
5  *
6  *   Node(int data) {
7  *     this.data = data;
8  *     this.next = null;
9  *   }
10 * }
11 */
12
13 public class FindMiddle {
14
15     public static Node findMiddle(Node head) {
16         int length = 0;
17         Node temp = head;
18
19         while (temp != null) {
20             length++;
21             temp = temp.next;
22         }
23
24         int middleIndex = length / 2;
25         temp = head;
26         for (int i = 0; i < middleIndex; i++) {
27             temp = temp.next;
28         }
29         return temp;
30     }
31 }
32
```

1. Analyze the algorithm's running time and determine its order of growth using the Big-Oh notation.
2. Rewrite the algorithm to find the middle in one pass.

Happy Coding 😊

Last changed by

Ahmad Albarasy · Follow

0 9

Add a comment

Read more

Data Structures and Algorithms Lab 2 - Linked Lists

In this lab, we are going to learn about Linked Lists. What they are, how can we implement them, their key...

Nov 22, 2025

Data Structures and Algorithms Lab 1 - Arrays

In this lab, we are going to dive into Arrays, one of the most basic yet fundamental data structures that we us...

Nov 15, 2025

Data Structures and Algorithms Lab 0 - Introduction

Welcome one and all to the Data Structures and Algorithms lab! This lab is one of the most important...

Nov 8, 2025

Read more from Ahmad Albarasy

Published on  HackMD