CH 02 : ALGORITHMS ANALYSIS

By Dr. LOUNNAS Bilal

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation. They are the following:

1. A Priori Analysis: This is a theoretical analysis of an algorithm. Efficiency of an algorithm is measured by assuming that all other

factors, for example, processor speed, are constant and have no effect on the implementation.

2. A Posteriori Analysis: This is an empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.

We shall learn about a priori algorithm analysis. Algorithm analysis deals with the execution or running time of various operations involved. The running time of an operation can be defined as the number of computer instructions executed per operation.

2 WHAT IS ANALYSIS?

The analysis of an algorithm provides background information that gives us a general idea of how long an algorithm will take for a given problem set. For each algorithm considered, we will come up with an estimate of how long it will take to solve a problem that has a set of N input values. So, for example, we might determine how many comparisons a sorting algorithm does to put a list of N values into ascending order, or we might determine how many arithmetic operations it takes to multiply two matrices of size N x N.

There are a number of algorithms that will solve a problem. Studying the analysis of algorithms gives us the tools to choose between algorithms. For example, consider the following two algorithms 1 and 2 to find the largest of four values:

Algorithm 1

```
1: procedure GetLargeNumber1
2:     largest ← a
3:     if b > largest then largest = b
4:     if c > largest then largest = c
5:     if d > largest then largest = d
       return largest
```

If you examine these two algorithms, you will see that each one will do exactly three comparisons to find the answer. Even though the first is easier for us to read and understand, they are both of the same level of complexity for a computer to execute. In terms of time, these two algorithms are the same, but in terms of space, the first needs more because of the temporary variable called largest. This extra space is not significant if we are comparing numbers or characters, but it may be with other types of data. In many modern programming languages, we can define comparison operators for large and complex objects or records. For those cases, the amount of space needed for the temporary variable could be quite significant. When we are interested in the efficiency of algorithms, we will primarily be concerned with time is-

Algorithm 2

```
1: procedure GetLargeNumber2
2:     if a > b then
3:         if a > c then
4:             if a > d then return a
5:             elsereturn b
6:         else if c > d then return c
7:         elsereturn d
8:     else if b > c then
9:         if b > d then return b
10:        elsereturn d
11:    else if c > d then return c
12:    elsereturn d
```

sues, but when space may be an issue, it will also be discussed.

The purpose of determining these values is to then use them to compare how efficiently two different algorithms solve a problem. For this reason, we will never compare a sorting algorithm with a matrix multiplication algorithm, but rather we will compare two different sorting algorithms to each other.

3 THE PURPOSE OF ANALYSIS OF ALGORITHMS

The purpose of analysis of algorithms is not to give a formula that will tell us exactly how many seconds or computer cycles a particular algorithm will take. This is not useful information because we would then need to talk about the type of computer, whether it has one or many users at a time, what processor it has, how fast its clock is, whether it has a complex or reduced instruction set processor chip, and how well the compiler optimizes the executable code. All of those will have an impact on how fast a program for an algorithm will run. To talk about analysis in those terms would mean that by moving a program to a faster computer, the algorithm would become better because it now completes its job faster. Thats not true, so, we do our analysis without regard to any specific computers.

In the case of a small or simple routine it might be possible to count the exact number of operations performed as a function of N. Most of the time, however, this will not be useful. In fact, we will see later that the difference between an algorithm that does N + 5 operations and one that does N + 2500 operations becomes meaningless as N gets very large.

4 ALGORITHM COMPLEXITY

Suppose X is an algorithm and n is the size of input data, the time and space used by the algorithm X are the two main factors, which decide the efficiency of X.

1. Time Factor: Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
2. Space Factor: Space is measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm f(n) gives the running time and/or the storage space required by the algorithm in terms of n as the size of input data.

4.1 Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The space required by an algorithm is equal to the sum of the following two components:

1. A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.
2. A variable part is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.

Space complexity S(P) of any algorithm P is S(P) = C + SP(I), where C is the fixed part and S(I) is the variable part of the algorithm, which depends on instance characteristic I. Following is a simple example that tries to explain the concept:

```
Algorithm: SUM(A, B)
Step 1 -  START
Step 2 -  C ← A + B + 10
Step 3 -  Stop
```

Here we have three variables A, B, and C and one constant. Hence S(P) = 1+3. Now, space depends on data types of given variables and constant types and it will be multiplied accordingly.

4.2 Time Complexity

Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can

be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step consumes constant time.

For example, addition of two n-bit integers takes n steps. Consequently, the total computational time is T(n) = c*n, where c is the time taken for the addition of two bits. Here, we observe that T(n) grows linearly as the input size increases.

5 ASYMPTOTIC ANALYSIS

Asymptotic analysis of an algorithm refers to defining the mathematical boundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.

Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.

Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, the running time of one operation is computed as f(n) and may be for another operation it is computed as g(n2). This means the first operation running time will increase linearly with the increase in n and the running time of the second operation will increase exponentially when n increases. Similarly, the running time of both operations will be nearly the same if n is significantly small.

Usually, the time required by an algorithm falls under three types:

1. Best Case: Minimum time required for program execution.
2. Average Case: Average time required for program execution.
3. Worst Case: Maximum time required for program execution.

5.1 Asymptotic notations

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

1. $\mathcal{O}$ Notation
2. Ω Notation
3. Θ Notation

5.2 Big Oh Notation, $\mathcal{O}$

The notation $\mathcal{O}(n)$ is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

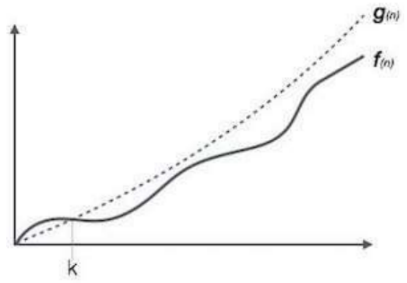

Figure 1: Big O Notation

For example, for a function f(n):

```
O(f(n)) = { g(n) : there exists c > 0 and n0 such that g(n) ≥ c.f(n) for all n
> n0. }
```

5.3 Omega Notation, Ω

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

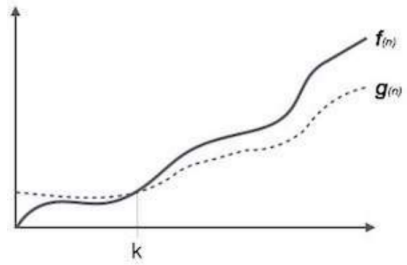

Figure 2: Big Ω Notation

For example, for a function f(n):

```
Ω(f(n)) ≥ { g(n) : there exists c > 0 and n0 such that g(n) ≤ c.f(n) for all n
> n0. }
```

5.4 Theta Notation, Θ

The notation Θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows

For example, for a function f(n):

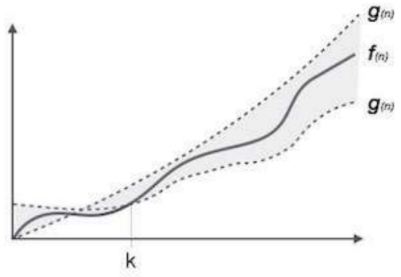

Figure 3: Big Ω Notation

```
θ(f(n)) = { g(n) if and only if g(n) =  O(f(n)) and g(n) = Ω(f(n)) for all n >
n0. }
```

6 COMMON ASYMPTOTIC NOTATIONS

Following is a list of some common asymptotic notations:

Complexity	Nomination
$\mathcal{O}(1)$	Constant
$\mathcal{O}(\log n)$	Logarithmic
$\mathcal{O}(n)$	Linear
$\mathcal{O}(n \log n)$	n log n
$\mathcal{O}(n^2)$	Quadratic
$\mathcal{O}(n^3)$	Cubic
$\mathcal{O}(n^{\mathcal{O}(1)})$	Polynomial
$\mathcal{O}(2^{\mathcal{O}(n)})$	Exponential

Table 1: list of some common asymptotic notations

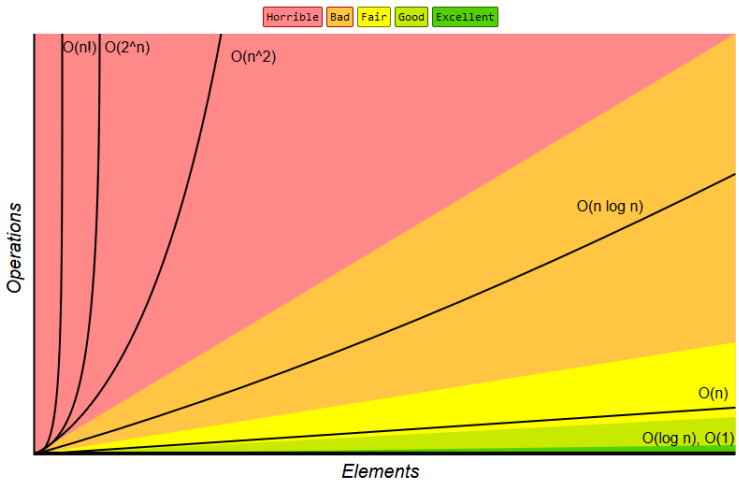

Figure 4: Big-O Complexity Chart