CHAPTER 2: ANALYSIS OF ALGORITHMS

Introduction

The objective of this chapter is to explain the importance of analysis of algorithms, their notations, relationships and solving as many problems as possible. We first concentrate on understanding the importance of analysis and then slowly move towards analyzing the algorithms with different notations and finally, the problems. After completion of this chapter you should be able to find the complexity of any given algorithm (especially recursive functions).

2.1 Why Analysis of Algorithms?

If we want to go from city " " to city " ". There can be many ways of doing this: by flight, by bus, by train and also by cycle. Depending on the availability and convenience we choose the one which suits us. Similarly, in computer science there can be multiple algorithms exist for solving the same problem (for example, sorting problem has lot of algorithms like insertion sort, selection sort, quick sort and many more). Algorithm analysis helps us determining which of them is efficient in terms of time and space consumed.

Goal of Analysis of Algorithms?

The goal of **Analysis of Algorithms** is to compare algorithms (or solutions) mainly in terms of running time but also in terms of other factors (e.g., memory, developer's effort etc.)

What is Running Time Analysis?

It is the process of determining how processing time increases as the size of the problem (input size) increases. Input size is number of

elements in the input and depending on the problem type the input may be of different types. In general, we encounter the following types of inputs.

- Size of an array
- Polynomial degree
- Number of elements in a matrix
- Number of bits in binary representation of the input
- Vertices and edges in a graph

How to Compare Algorithms?

To compare algorithms, we need objective measures.

• Execution times?

Not reliable, because they depend on the specific machine (CPU, RAM, etc.).

• Number of statements executed?

Also not reliable, because it depends on the programming language and the style of the programmer.

✓ Ideal solution:

Express the running time as a function of input size n, written as f(n).

By comparing these functions, we can see which algorithm is more efficient.

This comparison does not depend on hardware or programming style, making it fair and universal.

What is Rate of Growth?

The rate at which the running time increases as a function of input is called Rate of growth . Let us assume that you went to a shop for buying a car and a cycle. If your friend sees you there and asks what you are buying then in general we say buying acar . This is because cost of car is too big compared to cost of cycle (approximating the cost of cycle to cost of car).

Total Cost = cost_of_car + cost_of_cycle

Total Cost ≈ cost_of_car (approximation)

For the above example, we can represent the cost of car and cost of cycle in terms of function and for a given function we ignore the low order terms that are relatively insignificant (for large value of input size, n). As an example in the below case n^4, $2n^2$, 100 n , and 500 are the individual costs of some function and we approximate it to . n^4 Since, n^4 is the highest rate of growth.

$$n^4 + 2n^2 + 100n + 500 \approx n^4$$

Commonly used Rate of Growths

Below diagram shows the relationship between different rates of growth.

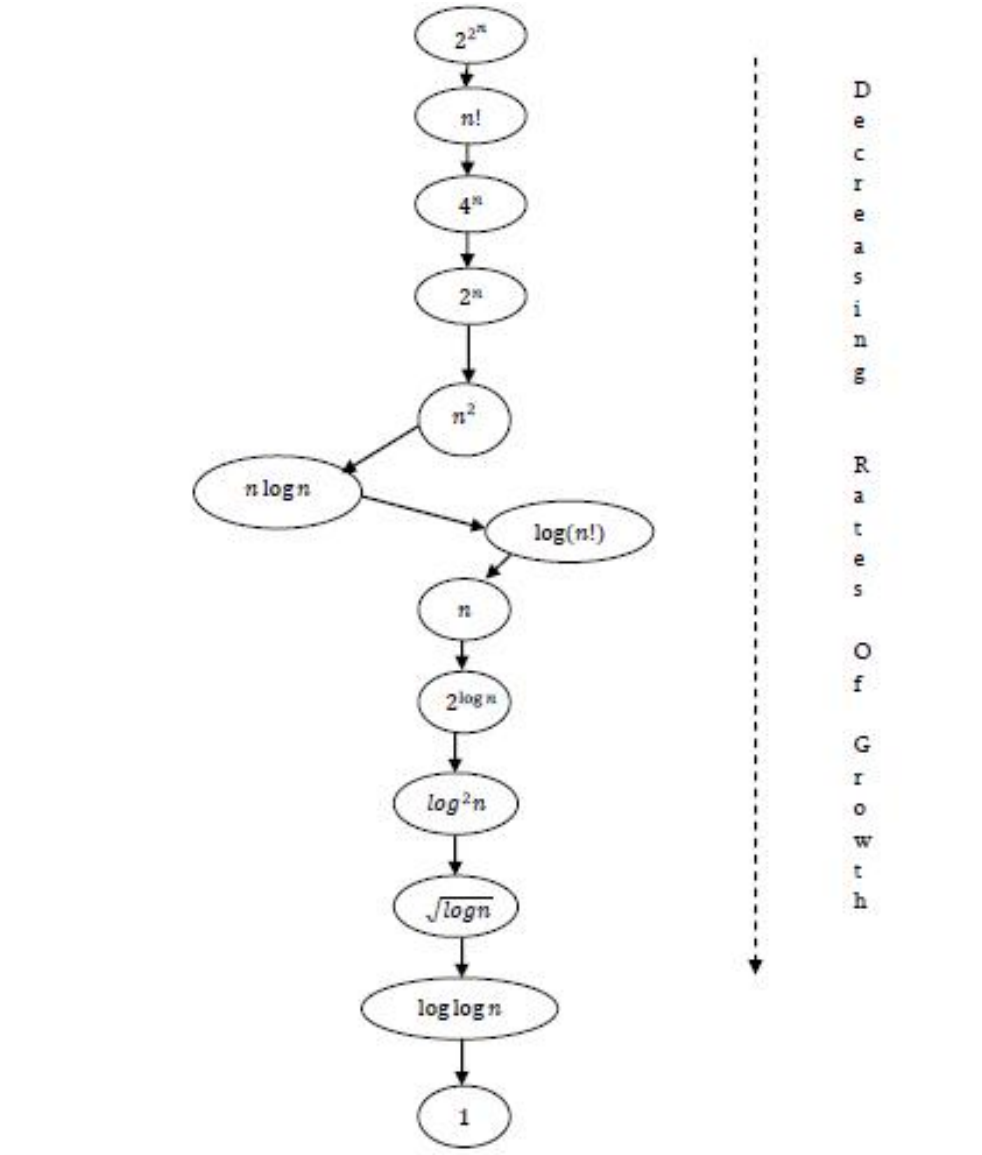

Figure 1. the relationship between different rates of growth

Below is the list of rate of growths which come across in remaining chapters.

Table 1. rate of growths

Time complexity	Name	Example
1	Constant	Adding an element to the front of a linked list
$logn$	Logarithmic	Finding an element in a sorted array
n	Linear	Finding an element in an unsorted array
$nlogn$	Linear Logarithmic	Sorting n items by 'divide-and-conquer'-Mergesort
n^2	Quadratic	Shortest path between two nodes in a graph
n^3	Cubic	Matrix Multiplication
2^n	Exponential	The Towers of Hanoi problem

2.2 Types of Analysis

If we have an algorithm for a problem and want to know on what inputs the algorithm is taking less time

(performing well) and on what inputs the algorithm is taking huge time.

We have already seen that an algorithm can be represented in the form of an expression. That means we

represent the algorithm with multiple expressions: one for case where it is taking the less time and other for case where it is taking the more time. In general the first case is called the best case and second case is

called the worst case for the algorithm.

To analyze an algorithm we need some kind of syntax and that forms the base for asymptotic

analysis/notation. There are three types of analysis:

1. **Worst case**

o Defines the input for which the algorithm takes huge time.

o Input is the one for which the algorithm runs the slower.

2. **Best case**

o Defines the input for which the algorithm takes lowest time.

o Input is the one for which the algorithm runs the fastest.

3. **Average case**

o Provides a prediction about the running time of the algorithm

o Assumes that the input is random

$$\text{Lower Bound} <= \text{Average Time} <= \text{Upper Bound}$$

For a given algorithm, we can represent best case, worst case, and average case analysis in the form of expressions. As an example, let $f(n)$ be the function which represents the given algorithm.

$f(n) = n^2 + 500$, for worst case
$f(n) = n + 100n + 500$, for best case

Similarly, for average case too. The expression defines the inputs with which the algorithm takes the

average running time (or memory).

Asymptotic Notation?

Having the expressions for best case, average case and worst case, for all the three cases we need to identify

the upper bound, lower bounds. In order to represent these upper bound and lower bounds we need some

syntax and that is the subject of following discussion. Let us assume that the given algorithm is represented in the form of function *f(n)*.