SALIENT FEATURES OF BOOK

- All code written in C
- Enumeration of possible solutions for each problem
- Covers all topics for competitive exams
- Covers interview questions on data structures and algorithms
- Reference Manual for working people
- Campus Preparation
- Degree/Masters Course Preparation
- Big Job Hunters: Microsoft, Google, Amazon, Yahoo, Oracle, Facebook and many more

ABOUT THE AUTHOR

Narasimha Karumanchi is the Senior Software Developer at **Amazon Corporation**, **India**. Most recently he worked for **IBM Labs**, **Hyderabad** and prior to that he served for **Mentor Graphics** and **Microsoft**, **Hyderabad**. He received his B-TECH. in Computer Science from **JNT University** and his M-Tech. in Computer Science from **IIT Bombay.**

He has experience in teaching data structures and algorithms at various training centers and colleges. He was born and bought up in Kambhampadu, Macherla (Palnadu), Guntur, Andhra Pradesh.

ISBN 978-0-615-45981-3

CareerMonk Publications

DATA STRUCTURES AND ALGORITHMS MADE EASY

DATA STRUCTURES AND ALGORITHMS MADE EASY

Success Key for:

- Campus Preparation
- Degree/Masters Course Preparation
- Instructor's
- GATE Preparation
- Big Job hunters: Microsoft, Google, Amazon, Yahoo, Facebook, Adobe and many more
- Reference Manual for Working People

Multiple smart solutions with different complexities

Narasimha Karumanchi

M-Tech, IIT Bombay

Founder of CareerMonk.com

CareerMonk Publications

To My Parents
~Laxmi and Modaiah

To My Family Members

To My Friends

To IIT Bombay

To All Hard Workers

Designed by Narasimha Karumanchi

Printed in India

Acknowledgements

I would like to express my gratitude to the many people who saw me through this book, to all those who provided support, talked things over, read, wrote, offered comments, allowed me to quote their remarks and assisted in the editing, proofreading and design. In particular, I would like to thank the following individuals.

I would like to thank *Ram Mohan Mullapudi* for encouraging me when I was at IIT Bombay. He is the first person who taught me the importance of *algorithms* and its *design*. From that day I keep on updating myself.

I would like to thank *Vamshi Krishna* [*Mentor Graphics*] and *Kalyani Tummala* [*Xilinx*] for spending time in reviewing this book and providing me the valuable suggestions almost every day.

I would like to thank *Sobhan* [*Professor IIT, Hyderabad*] for spending his valuable time in reviewing the book and suggestions. His review gave me the confidence in the quality of the book.

I would like to thank *Kiran* and *Laxmi* [Founder's of *TheGATEMATE.com*] for approaching me for teaching Data Structures and Algorithms at their training centers. They are the primary reason for initiation of this book.

My *friends* and *colleagues* have contributed greatly to the quality of this book. I thank all of you for your help and suggestions.

Special thanks should go to my wife, *Sailaja* for her encouragement and help during writing of this book.

Last but not least, I would like to thank Director's of *Guntur Vikas College*, *Gopala Krishna Murthy* [Director of *ACE Engineering Academy*], *TRC Bose* [Former Director of *APTransco*] and *Venkateswara Rao* [*VNR Vignanajyothi Engineering College*, *Hyderabad*] for helping me and my family during our studies.

-Narasimha Karumanchi
M-Tech, IIT Bombay
Founder of CareerMonk.com

Preface

Dear Reader,

Please Hold on! I know many people do not read preface. But I would like to strongly recommend reading preface of this book at least. This preface has *something different* from regular prefaces.

As a *job seeker* if you read complete book with good understanding, I am sure you will challenge the interviewer's and that is the objective of this book.

If you read as an *instructor*, you will give better lectures with easy go approach and as a result your students will feel proud for selecting Computer Science / Information Technology as their degree.

This book is very much useful for the *students* of *Engineering* and *Masters* during their academic preparations. All the chapters of this book contain theory and their related problems as many as possible. There a total of approximately 700 algorithmic puzzles and all of them are with solutions.

If you read as a *student* preparing for competition exams for Computer Science/Information Technology], the content of this book covers *all* the *required* topics in full details. While writing the book, an intense care has been taken to help students who are preparing for these kinds of exams.

In all the chapters you will see more importance given to problems and analyzing them instead of concentrating more on theory. For each chapter, first you will see the basic required theory and then followed by problems.

For many of the problems, *multiple* solutions are provided with different complexities. We start with *brute force* solution and slowly move towards the *best solution* possible for that problem. For each problem we will try to understand how much time the algorithm is taking and how much memory the algorithm is taking.

It is *recommended* that, at least one complete reading of this book is required to get full understanding of all the topics. In the subsequent readings, readers can directly go to any chapter and refer. Even though, enough readings were given for correcting the errors, due to human tendency there could be some minor typos in the book. If any such typos found, they will be updated at *www.CareerMonk.com*. I request readers to constantly monitor this site for any corrections, new problems and solutions. Also, please provide your valuable suggestions at: *Info@CareerMonk.com*.

Wish you all the best. Have a nice reading.

-Narasimha Karumanchi
M-Tech, IIT Bombay
Founder of CareerMonk.com

Table of Contents

DATA STRUCTURES AND ALGORITHMS MADE EASY

INTRODUCTION

Chapter-1

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 basic elements of algorithms, 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).

1.1 Variables

Before going to the definition of variables, let us relate them to old mathematical equations. All of us have solved many mathematical equations since childhood. As an example, consider the below equation:

$$x^2 + 2y - 2 = 1$$

We don't have to worry about the use of above equation. The important thing that we need to understand is, the equation has some names (x and y) which hold values (data). That means, the *names* (x and y) are the place holders for representing data. Similarly, in computer science we need something for holding data and *variables* are the facility for doing that.

1.2 Data types

In the above equation, the variables x and y can take any values like integral numbers (10, 20 etc...), real numbers (0.23, 5.5 etc...) or just 0 and 1. To solve the equation, we need to relate them to kind of values they can take and *data type* is the name being used in computer science for this purpose.

A *data type* in a programming language is a set of data with values having predefined characteristics. Examples of data types are: integer, floating point unit number, character, string etc...

Computer memory is all filled with zeros and ones. If we have a problem and wanted to code it, it's very difficult to provide the solution in terms of zeros and ones. To help users, programming languages and compilers are providing the facility of data types.

For example, *integer* takes 2 bytes (actual value depends on compiler), *float* takes 4 bytes etc... This says that, in memory we are combining 2 bytes (16 bits) and calling it as *integer*. Similarly, combining 4 bytes (32 bits) and calling it as *float*. A data type reduces the coding effort. Basically, at the top level, there are two types of data types:

- System defined data types (also called *Primitive* data types)
- User defined data types

System defined data types (Primitive data types)

Data types which are defined by system are called *primitive* data types. The primitive data types which are provided by many programming languages are: int, float, char, double, bool, etc...

The number of bits allocated for each primitive data type depends on the programming languages, compiler and operating system. For the same primitive data type, different languages may use different sizes. Depending on the size of the data types the total available values (domain) will also changes. For example, "*int*" may take 2 bytes or 4 bytes. If it takes 2 bytes (16 bits) then the total possible values are -32,768 to +32,767 (-2^{15} *to* 2^{15}-1). If it takes, 4 bytes (32 bits), then the possible values are between −2,147,483,648 to +2,147,483,648 (-2^{31} *to* 2^{31}-1). Same is the case with remaining data types too.

User defined data types

If the system defined data types are not enough then most programming languages allows the users to define their own data types called as user defined data types. Good example of user defined data types are: structures in $C/C++$ and classes in *Java*.

For example, in the below case, we are combining many system defined data types and called it as user defined data type with name "*newType*". This gives more flexibility and comfort in dealing with computer memory.

```
struct newType {
        int data1;
        float data 2;
        ...
        char data;
};
```

1.3 Data Structure

Based on the above discussion, once we have data in variables, we need some mechanism for manipulating that data to solve problems. *Data structure* is a particular way of storing and organizing data in a computer so that it can be used efficiently. That means, a *data structure* is a specialized format for organizing and storing data. General data structure types include arrays, files, linked lists, stacks, queues, trees, graphs and so on.

Depending on the organization of the elements, data structures are classified into two types:

1) *Linear data structures*: Elements are accessed in a sequential order but it is not compulsory to store all elements sequentially (say, Linked Lists). *Examples*: Linked Lists, Stacks and Queues.
2) *Non – linear data structures*: Elements of this data structure are stored/accessed in a non-linear order. *Examples*: Trees and graphs.

1.4 Abstract Data Types (ADTs)

Before defining abstract data types, let us consider the different view of system defined data types. We all know that, by default, all primitive data types (int, float, et..) supports basic operations like addition, subtraction etc... The system is providing the implementations for the primitive data types. For user defined data types also we need to define operations. The implementation for these operations can be done when we want to actually use them. That means, in general user defined data types are defined along with their operations.

To simplify the process of solving the problems, we generally combine the data structures along with their operations and are called *Abstract Data Types* (ADTs). An ADT consists of *two* parts:

1. Declaration of data
2. Declaration of operations

Commonly used ADTs *include*: Linked Lists, Stacks, Queues, Priority Queues, Binary Trees, Dictionaries, Disjoint Sets (Union and Find), Hash Tables, Graphs, and many other. For example, stack uses LIFO (Last-In-First-Out) mechanism while storing the data in data structures. The last element inserted into the stack is the first element that gets deleted. Common operations of it are: creating the stack, pushing an element onto the stack, popping an element from stack, finding the current top of the stack, finding number of elements in the stack etc...

While defining the ADTs do not care about implementation details. They come in to picture only when we want to use them. Different kinds of ADTs are suited to different kinds of applications, and some are highly specialized to specific tasks. By the end of this book, we will go through many of them and you will be in a position to relate the data structures to the kind of problems they solve.

1.5 What is an Algorithm?

Let us consider the problem of preparing an omelet. For preparing omelet, general steps we follow are:

1) Get the frying pan.
2) Get the oil.
    a. Do we have oil?
        i. If yes, put it in the pan.
        ii. If no, do we want to buy oil?
            1. If yes, then go out and buy.
            2. If no, we can terminate.
3) Turn on the stove, etc...

What we are doing is, for a given problem (preparing an omelet), giving step by step procedure for solving it. Formal definition of an algorithm can be given as:

An algorithm is the step-by-step instructions to solve a given problem.

Note: we do not have to prove each step of the algorithm.

1.6 Why Analysis of Algorithms?

To go from city "*A*" to city "*B*", there can be many ways of accomplishing 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 many 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.

1.7 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, developers effort etc.)

1.8 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

1.9 How to Compare Algorithms?

To compare algorithms, let us define few *objective measures*:
Execution times? *Not a good measure* as execution times are specific to a particular computer.

Number of statements executed? *Not a good measure*, since the number of statements varies with the programming language as well as the style of the individual programmer.

Ideal Solution? Let us assume that we expressed running time of given algorithm as a function of the input size n (i.e., $f(n)$) and compare these different functions corresponding to running times. This kind of comparison is independent of machine time, programming style, etc...

1.10 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 a car*. This is because, cost of car is too big compared to cost of cycle (approximating the cost of cycle to cost of car).

$$Total\ Cost\ =\ cost\_of\_car\ +\ cost\_of\_cycle$$
$$Total\ Cost\ \approx\ cost\_of\_car\ (approximation)$$

For the above example, we can represent the cost of car and cost of cycle in terms of function and for a given function 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, $2n^2$, $100n$ and 500 are the individual costs of some function and approximate it to n^4. Since, n^4 is the highest rate of growth.

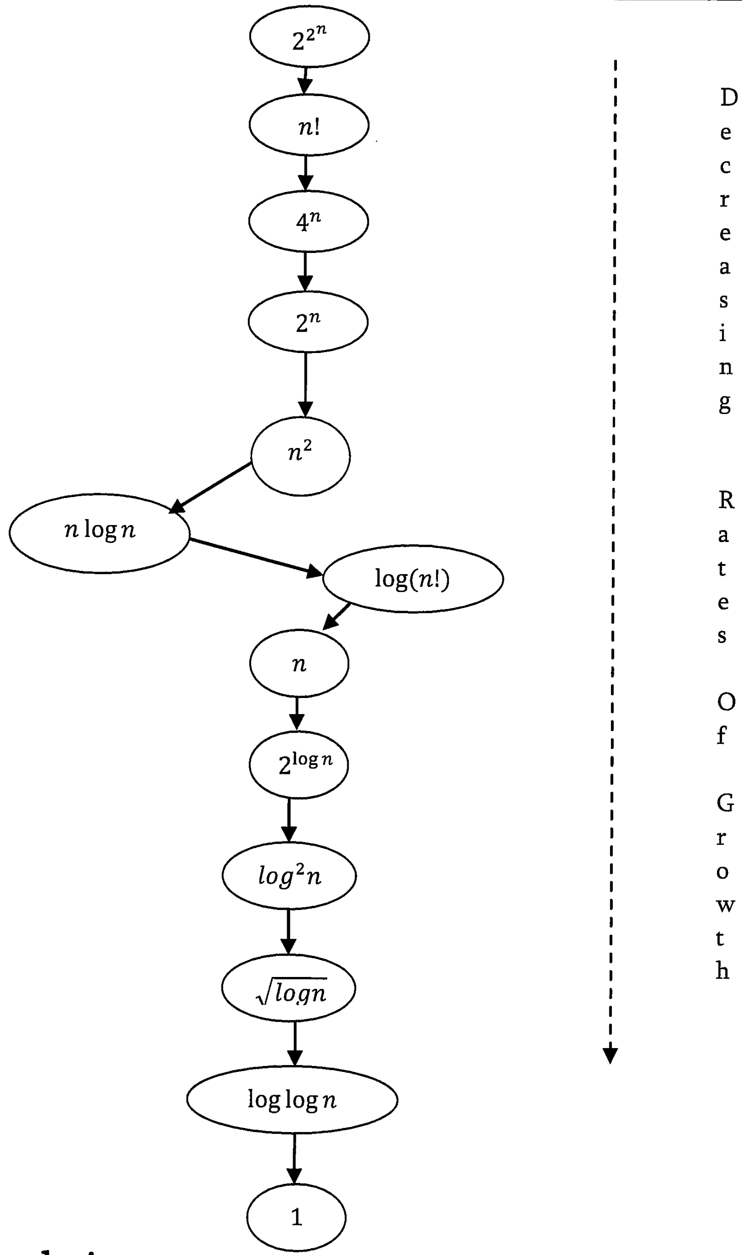
$$n^4\ +\ 2n^2\ +\ 100n\ +\ 500\ \approx\ n^4$$

1.11 Commonly used Rate of Growths

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

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

Below diagram shows the relationship between different rates of growth.



1.12 Types of Analysis

To analyze the given algorithm we need 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* of 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:

- Worst case
    - Defines the input for which the algorithm takes huge time.
    - Input is the one for which the algorithm runs the slower.
- Best case
    - Defines the input for which the algorithm takes lowest time.

  - Input is the one for which the algorithm runs the fastest.
- **Average case**
  - Provides a prediction about the running time of the algorithm
  - Assumes that the input is random

$$Lower\ Bound <= Average\ Time <= Upper\ Bound$$

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

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

Similarly, for average case too. The expression defines the inputs with which the algorithm takes the average running time (or memory).

1.13 Asymptotic Notation

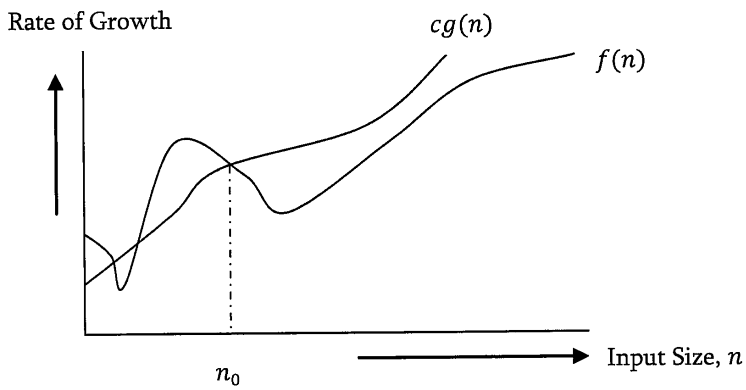
Having the expressions for best, average case and worst cases, for all the three cases we need to identify the upper and lower bounds. In order to represent these upper and lower bounds we need some kind syntax and that is the subject of following discussion. Let us assume that the given algorithm is represented in the form of function $f(n)$.

1.14 Big-O Notation

This notation gives the *tight* upper bound of the given function. Generally, it is represented as $f(n) = O(g(n))$. That means, at larger values of n, the upper bound of $f(n)$ is $g(n)$. For example, if $f(n) = n^4 + 100n^2 + 10n + 50$ is the given algorithm, then n^4 is $g(n)$. That means, $g(n)$ gives the maximum rate of growth for $f(n)$ at larger values of n.

Let us see the O−notation with little more detail. O−notation defined as $O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$. $g(n)$ is an asymptotic tight upper bound for $f(n)$. Our objective is to give smallest rate of growth $g(n)$ which is greater than or equal to given algorithms rate of growth $f(n)$.

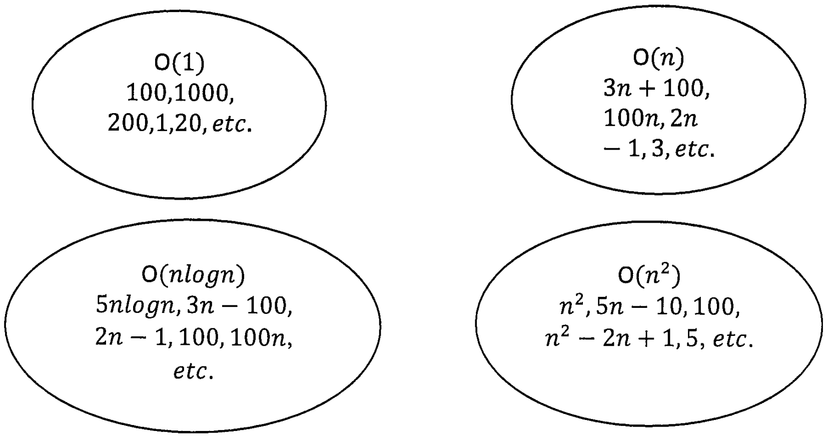
In general, we discard lower values of n. That means the rate of growth at lower values of n is not important. In the below figure, n_0 is the point from which we need to consider the rate of growths for a given algorithm. Below n_0 the rate of growths could be different.



Big-O Visualization

$O(g(n))$ is the set of functions with smaller or same order of growth as $g(n)$. For example, $O(n^2)$ includes $O(1)$, $O(n)$, $O(nlogn)$ etc..

Note: Analyze the algorithms at larger values of n only. What this means is, below n_0 we do not care for rate of growth.



Big-O Examples

Example-1 Find upper bound for $f(n) = 3n + 8$
Solution: $3n + 8 \leq 4n$, for all $n \geq 1$
$\therefore 3n + 8 = O(n)$ with c = 4 and $n_0 = 8$

Example-2 Find upper bound for $f(n) = n^2 + 1$
Solution: $n^2 + 1 \leq 2n^2$, for all $n \geq 1$
$\therefore n^2 + 1 = O(n^2)$ with $c = 2$ and $n_0 = 1$

Example-3 Find upper bound for $f(n) = n^4 + 100n^2 + 50$
Solution: $n^4 + 100n^2 + 50 \leq 2n^4$, for all $n \geq 11$
$\therefore n^4 + 100n^2 + 50 = O(n^4)$ with $c = 2$ and $n_0 = 11$

Example-4 Find upper bound for $f(n) = 2n^3 - 2n^2$
Solution: $2n^3 - 2n^2 \leq 2n^3$, for all $n \geq 1$
$\therefore 2n^3 - 2n^2 = O(2n^3)$ with $c = 2$ and $n_0 = 1$

Example-5 Find upper bound for $f(n) = n$
Solution: $n \leq n^2$, for all $n \geq 1$
$\therefore n = O(n^2)$ with $c = 1$ and $n_0 = 1$

Example-6 Find upper bound for $f(n) = 410$
Solution: $410 \leq 410$, for all $n \geq 1$
$\therefore 100 = O(1)$ with $c = 1$ and $n_0 = 1$

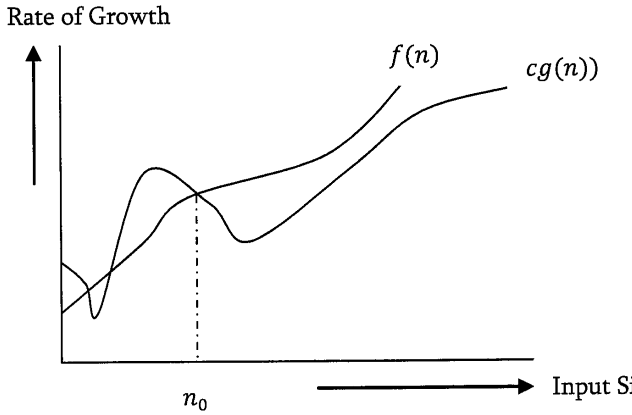
No Uniqueness?

There are no unique set of values for n_0 and c in proving the asymptotic bounds. Let us consider, $100n + 5 = O(n^2)$. For this function there are multiple n_0 and c values possible.

Solution1: $100n + 5 \leq 100n + n = 101n \leq 101n^2$ for all $n \geq 5$, $n_0 = 5$ and $c = 101$ is a solution.
Solution2: $100n + 5 \leq 100n + 5n = 105n \leq 105n^2$ for all $n \geq 1$, $n_0 = 1$ and $c = 105$ is also a solution.

1.15 Omega-Ω Notation

Similar to O discussion, this notation gives the tighter lower bound of the given algorithm and we represent it as $f(n) = \Omega(g(n))$. That means, at larger values of n, the tighter lower bound of $f(n)$ is $g(n)$. For example, if $f(n) = 100n^2 + 10n + 50$, $g(n)$ is $\Omega(n^2)$.



The Ω notation can be defined as $\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0\}$. $g(n)$ is an asymptotic tight lower bound for $f(n)$. Our objective is to give largest rate of growth $g(n)$ which is less than or equal to given algorithms rate of growth $f(n)$.

Ω Examples

Example-1 Find lower bound for $f(n) = 5n^2$

Solution: $\exists\ c,\ n_0$ Such that: $0 \leq cn \leq 5n^2 \Rightarrow cn \leq 5\,n^2 \Rightarrow c = 1$ and $n_0 = 1$
$\therefore\ 5n^2 = \Omega(n)$ with $c = 1$ and $n_0 = 1$

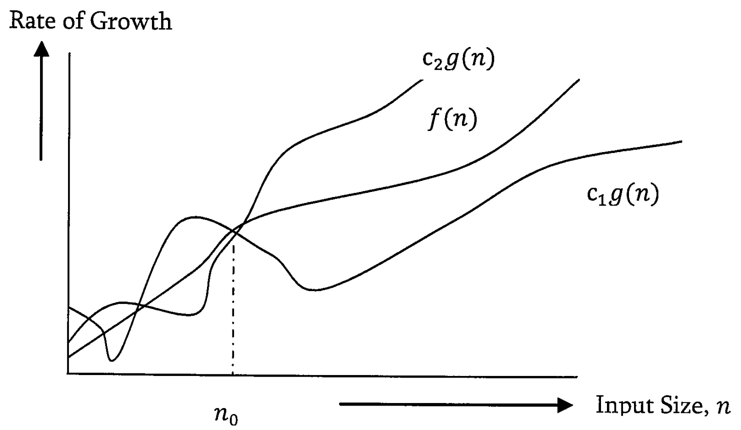
Example-2 Prove $f(n) = 100n + 5 \neq \Omega(n^2)$

Solution: $\exists$ c, n_0 Such that: $0 \leq cn^2 \leq 100n + 5$
$100n + 5 \leq 100n + 5n\ (\forall n \geq 1) = 105n$
$cn^2 \leq 105n \Rightarrow n(cn - 105) \leq 0$
Since n is positive $\Rightarrow cn - 105 \leq 0 \Rightarrow n \leq 105/c$
$\Rightarrow$ Contradiction: n cannot be smaller than a constant

Example-3 $n = \Omega(2n)$, $n^3 = \Omega(n^2)$, $n = \Omega(logn)$

1.16 Theta-Θ Notation

This notation decides whether the upper and lower bounds of a given function (algorithm) are same or not. The average running time of algorithm is always between lower bound and upper bound. If the upper bound (O) and lower bound (Ω) gives the same result then Θ notation will also have the same rate of growth. As an example, let us assume that $f(n) = 10n + n$ is the expression. Then, its tight upper bound $g(n)$ is $O(n)$. The rate of growth in best case is $g(n) = O(n)$. In this case, rate of growths in best case and worst are same. As a result, the average case will also be same. For a given function (algorithm), if the rate of growths (bounds) for O and Ω are not same then the rate of growth Θ case may not be same.



Now consider the definition of Θ notation. It is defined as $\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2 and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0\}$. $g(n)$ is an asymptotic tight bound for $f(n)$. $\Theta(g(n))$ is the set of functions with the same order of growth as $g(n)$.

Θ Examples

Example-1 Find Θ bound for $f(n) = \frac{n^2}{2} - \frac{n}{2}$

Solution: $\frac{n^2}{5} \le \frac{n^2}{2} - \frac{n}{2} \le n^2$, for all, $n \ge 1$

$\therefore \frac{n^2}{2} - \frac{n}{2} = \Theta(n^2)$ with $c_1 = 1/5, c_2 = 1$ and $n_0 = 1$

Example-2 Prove $n \neq \Theta(n^2)$

Solution: $c_1 n^2 \le n \le c_2 n^2 \Rightarrow$ only holds for: $n \le 1/c_1$

$\therefore n \neq \Theta(n^2)$

Example-3 Prove $6n^3 \neq \Theta(n^2)$

Solution: $c_1 n^2 \le 6n^3 \le c_2 n^2 \Rightarrow$ only holds for: $n \le c_2/6$

$\therefore 6n^3 \neq \Theta(n^2)$

Example-4 Prove $n \neq \Theta(logn)$

Solution: $c_1 logn \le n \le c_2 log\, n \Rightarrow c_2 \ge \frac{n}{\log n}, \forall\, n \ge n_0$ – Impossible

Important Notes

For analysis (best case, worst case and average) we try to give upper bound (O) and lower bound (Ω) and average running time (Θ). From the above examples, it should also be clear that, for a given function (algorithm) getting upper bound (O) and lower bound (Ω) and average running time (Θ) may not be possible always. For example, if we are discussing the best case of an algorithm, then we try to give upper bound (O) and lower bound (Ω) and average running time (Θ). In the remaining chapters we generally concentrate on upper bound (O) because knowing lower bound (Ω) of an algorithm is of no practical importance and we use Θ notation if upper bound (O) and lower bound (Ω) are same.

1.17 Why is it called Asymptotic Analysis?

From the above discussion (for all the three notations: worst case, best case and average case), we can easily understand that, in every case for a given function $f(n)$ we are trying to find other function $g(n)$ which approximates $f(n)$ at higher values of n. That means, $g(n)$ is also a curve which approximates $f(n)$ at higher values of n. In mathematics we call such curve as *asymptotic curve*. In other terms, $g(n)$ is the asymptotic curve for $f(n)$. For this reason, we call algorithm analysis as *asymptotic analysis*.

1.18 Guidelines for Asymptotic Analysis

There are some general rules to help us in determining the running time of an algorithm.

1) **Loops**: The running time of a loop is, at most, the running time of the statements inside the loop (including tests) multiplied by the number of iterations.

```
// executes n times
for (i=1; i<=n; i++)
    m = m + 2; // constant time, c
```

Total time = a constant $c \times n = c\,n = O(n)$.

2) **Nested loops:** Analyze from inside out. Total running time is the product of the sizes of all the loops.

```
//outer loop executed n times
for (i=1; i<=n; i++) {
    // inner loop executed n times
    for (j=1; j<=n; j++)
        k = k+1; //constant time
}
```

Total time = $c \times n \times n = cn^2 = O(n^2)$.

3) **Consecutive statements:** Add the time complexities of each statement.

```
x = x +1; //constant time
// executed n times
for (i=1; i<=n; i++)
    m = m + 2; //constant time
//outer loop executed n times
for (i=1; i<=n; i++) {
    //inner loop executed n times
    for (j=1; j<=n; j++)
        k = k+1; //constant time
}
```

Total time = $c_0 + c_1 n + c_2 n^2 = O(n^2)$.

4) **If-then-else statements:** Worst-case running time: the test, plus *either* the *then* part *or* the *else* part (whichever is the larger).

```
//test: constant
if(length( ) == 0 ) {
    return false; //then part: constant
}
else {// else part: (constant + constant) * n
    for (int n = 0; n < length( ); n++) {
        // another if : constant + constant (no else part)
        if(!list[n].equals(otherList.list[n]))
            //constant
            return false;
    }
}
```

Total time = $c_0 + c_1 + (c_2 + c_3) * n = O(n)$.

5) **Logarithmic complexity:** An algorithm is O($logn$) if it takes a constant time to cut the problem size by a fraction (usually by ½). As an example let us consider the following program:

```
for (i=1; i<=n;)
        i = i*2;
```

If we observe carefully, the value of i is doubling every time. Initially $i = 1$, in next step $i = 2$, and in subsequent steps $i = 4, 8$ and so on. Let us assume that the loop is executing some k times. At k^{th} step $2^i = n$ and we come out of loop. Taking logarithm on both sides, gives

$log(2^i) = logn$
$ilog2 = logn$
$i = logn$ //if we assume base-2

Total time = O($logn$).

Note: Similarly, for the below case also, worst case rate of growth is O($logn$). The same discussion holds good for decreasing sequence as well.

```
for (i=n; i<=1;)
    i = i/2;
```

Another example: binary search (finding a word in a dictionary of n pages)

- Look at the center point in the dictionary
- Is word towards left or right of center?
- Repeat process with left or right part of dictionary until the word is found

1.19 Properties of Notations

- Transitivity: $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$. Valid for O and Ω as well.
- Reflexivity: $f(n) = \Theta(f(n))$. Valid for O and Ω also.
- Symmetry: $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$.
- Transpose symmetry: $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.

1.20 Commonly used Logarithms and Summations

Logarithms

$log\, x^y = y\, log\, x$ $logn = log_{10}^n$
$log\, xy = logx + logy$ $log^k n = (logn)^k$
$log\, logn = log(logn)$ $log\frac{x}{y} = logx - logy$
$a^{log_b^x} = x^{log_b^a}$ $log_b^x = \frac{log_a^x}{log_a^b}$

Arithmetic series

$$\sum_{K=1}^{n} k = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

Geometric series

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 \ldots + x^n = \frac{x^{n+1} - 1}{x - 1} (x \neq 1)$$

Harmonic series

$$\sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \ldots + \frac{1}{n} \approx log\, n$$