

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 brought up in Kambhampadu, Macherla (Palnadu), Guntur, Andhra Pradesh.

CareerMonk Publications

ISBN 978-0-615-45981-3



9 780615 459813 >

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

Copyright ©2010 by CareerMonk.com

All rights reserved.

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 Vignanjyothi 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

Chapter 1	Introduction	29
	Variables	29
	Data types	29
	System defined data types (Primitive data types)	30
	User defined data types.....	30
	Data Structure	30
	Abstract Data Types (ADT's).....	31
	Memory and Variables	31
	Size of a Variable.....	32
	Address of a Variable	32
	Pointers.....	33
	Declaration of Pointers.....	33
	Pointers Usage.....	33
	Pointer Manipulation	34
	Arrays and Pointers	35
	Dynamic Memory Allocation.....	36
	Function Pointers.....	36
	Parameter Passing Techniques.....	37
	Actual and Formal Parameters.....	37
	Semantics of Parameter Passing.....	38
	Language Support for Parameter Passing Techniques	38
	Pass by Value.....	38
	Pass by Result.....	39
	Pass by Value-Result.....	40
	Pass by Reference (aliasing)	41
	Pass by Name.....	42
	Binding	43
	Binding Times	43
	Static Binding (Early binding).....	43
	Dynamic Binding (Late binding).....	43
	Scope.....	44

Static Scope.....	44
Dynamic Scope.....	45
Storage Classes.....	46
Auto Storage Class.....	46
Extern storage class.....	47
Register Storage Class	52
Static Storage Class.....	52
Storage Organization	53
Static Segment.....	54
Stack Segment	54
Heap Segment	56
Shallow Copy versus Deep Copy.....	57
Chapter 2 Analysis of Algorithms	58
Introduction	58
What is an Algorithm?	58
Why Analysis of Algorithms?	58
Goal of Analysis of Algorithms?.....	59
What is Running Time Analysis?.....	59
How to Compare Algorithms?	59
What is Rate of Growth?	60
Commonly used Rate of Growths.....	60
Types of Analysis	61
Asymptotic Notation?.....	62
Big-O Notation.....	62
Big-O Visualization.....	63
Big-O Examples.....	63
No Uniqueness?.....	64
Omega- Ω Notation.....	64
Ω Examples.....	65
Theta- θ Notation.....	65
Θ Examples.....	66
Important Notes.....	66
Why is it called Asymptotic Analysis?	67

Guidelines for Asymptotic Analysis?	67
Properties of Notations	69
Commonly used Logarithms and Summations	70
Master Theorem for Divide and Conquer	70
Problems Divide and Conquer Master Theorem	71
Master Theorem for Subtract and Conquer Recurrences	73
Variant of subtraction and conquer master theorem	73
Problems on Algorithms Analysis	73
Chapter 3 Recursion and Backtracking	88
Introduction	88
What is Recursion?	88
Why Recursion?	88
Format of a Recursive Function	88
Recursion and Memory (Visualization)	89
Recursion versus Iteration	90
Recursion	90
Iteration	91
Notes on Recursion	91
Example Algorithms of Recursion	91
Problems on Recursion	91
What is Backtracking?	92
Example Algorithms Of Backtracking	93
Problems On Backtracking	93
Chapter 4 Linked Lists	95
What is a Linked List?	95
Linked Lists ADT	95
Why Linked Lists?	95
Arrays Overview	96
Why Constant Time for Accessing Array Elements?	96
Advantages of Arrays	96
Disadvantages of Arrays	96
Dynamic Arrays	96
Advantages of Linked Lists	97

Issues with Linked Lists (Disadvantages).....	97
Comparison of Linked Lists with Arrays and Dynamic Arrays.....	97
Singly Linked Lists.....	98
Basic Operations on a List	98
Traversing the Linked List.....	98
Singly Linked List Insertion	99
Inserting a Node in Singly Linked List at the Beginning.....	99
Inserting a Node in Singly Linked List at the Ending.....	100
Inserting a Node in Singly Linked List in the Middle.....	100
Singly Linked List Deletion	102
Deleting the First Node in Singly Linked List.....	102
Deleting the last node in Singly Linked List	102
Deleting an Intermediate Node in Singly Linked List	103
Deleting Singly Linked List	104
Doubly Linked Lists.....	105
Doubly Linked List Insertion	105
Inserting a Node in Doubly Linked List at the Beginning.....	106
Inserting a Node in Doubly Linked List at the Ending.....	106
Inserting a Node in Doubly Linked List in the Middle.....	106
Doubly Linked List Deletion	108
Deleting the First Node in Doubly Linked List.....	108
Deleting the Last Node in Doubly Linked List.....	109
Deleting an Intermediate Node in Doubly Linked List	110
Circular Linked Lists.....	111
Counting Nodes in a Circular List.....	112
Printing the contents of a circular list	112
Inserting a Node at the End of a Circular Linked List	113
Inserting a Node at Front of a Circular Linked List	114
Deleting the Last Node in a Circular List	116
Deleting the First Node in a Circular List.....	117
Applications of Circular List.....	118
A Memory-Efficient Doubly Linked List	119
Problems on Linked Lists	120

Chapter 5	Stacks	143
What is a Stack?		143
How are Stacks Used?		143
Stack ADT		144
Main stack operations		144
Auxiliary stack operations		144
Exceptions		144
Applications		144
Implementation		145
Simple Array Implementation		145
Dynamic Array Implementation		147
Performance		150
Linked List Implementation		150
Performance		152
Comparison of Implementations		152
Comparing Incremental Strategy and Doubling Strategy		152
Comparing Array Implementation and Linked List Implementation		153
Problems on Stacks		153
Chapter 6	Queues	176
What is a Queue?		176
How are Queues Used?		176
Queue ADT		177
Main queue operations		177
Auxiliary queue operations		177
Exceptions		177
Applications		177
Direct applications		177
Indirect applications		177
Implementation		178
Why Circular Arrays?		178
Simple Circular Array Implementation		178
Performance & Limitations		181
Dynamic Circular Array Implementation		181

Performance	184
Linked List Implementation	184
Performance	186
Comparison of Implementations.....	186
Problems on Queues	186
Chapter 7 Trees.....	190
What is a Tree?	190
Glossary	190
Binary Trees	191
Types of Binary Trees	192
Properties of Binary Trees	193
Structure of Binary Trees.....	194
Operations on Binary Trees.....	194
Applications of Binary Trees	195
Binary Tree Traversals.....	195
Traversal Possibilities	195
Classifying the Traversals	196
PreOrder Traversal	196
InOrder Traversal	198
PostOrder Traversal.....	199
Level Order Traversal	201
Problems on Binary Trees	202
Generic Trees (N-ary Trees).....	226
Representation of Generic Trees.....	227
Problems on Generic Trees	228
Threaded Binary Tree Traversals [Stack or Queue less Traversals]	234
Issues with Regular Binary Trees	234
Motivation for Threaded Binary Trees	235
Classifying Threaded Binary Trees	235
Types of Threaded Binary Trees	236
Threaded Binary Tree structure.....	236
Difference between Binary Tree and Threaded Binary Tree Structures	236
Finding Inorder Successor in Inorder Threaded Binary Tree	238

Inorder Traversal in Inorder Threaded Binary Tree.....	238
Finding PreOrder Successor in InOrder Threaded Binary Tree	239
PreOrder Traversal of InOrder Threaded Binary Tree	239
Insertion of Nodes in InOrder Threaded Binary Trees.....	240
Problems on Threaded binary Trees.....	241
Expression Trees	243
Algorithm for Building Expression Tree from Postfix Expression.....	243
Example	244
XOR Trees	246
Binary Search Trees (BSTs)	247
Why Binary Search Trees?	247
Binary Search Tree Property	247
Binary Search Tree Declaration	248
Operations on Binary Search Trees.....	248
Important Notes on Binary Search Trees	248
Finding an Element in Binary Search Trees.....	249
Finding an Minimum Element in Binary Search Trees	250
Finding an Maximum Element in Binary Search Trees.....	251
Where is Inorder Predecessor and Successor?	252
Inserting an Element from Binary Search Tree.....	252
Deleting an Element from Binary Search Tree	253
Problems on Binary Search Trees	255
Balanced Binary Search Trees	265
Complete Balanced Binary Search Trees	266
AVL (Adelson-Velskii and Landis) trees	266
Properties of AVL Trees	266
Minimum/Maximum Number of Nodes in AVL Tree	267
AVL Tree Declaration.....	267
Finding Height of an AVL tree	268
Rotations.....	268
Observation	268
Types of Violations	269
Single Rotations	269

Double Rotations	271
Insertion into an AVL tree	273
Problems on AVL Trees.....	274
Other Variations in Trees.....	279
Red-Black Trees	279
Splay Trees	280
Augmented Trees.....	280
Interval Trees	281
Chapter 8 Priority Queue and Heaps	283
What is a Priority Queue?.....	283
Priority Queue ADT	283
Main Priority Queues Operations.....	283
Auxiliary Priority Queues Operations.....	284
Priority Queue Applications	284
Priority Queue Implementations.....	284
Unordered Array Implementation.....	284
Unordered List Implementation	284
Ordered Array Implementation.....	284
Ordered List Implementation.....	285
Binary Search Trees Implementation	285
Balanced Binary Search Trees Implementation	285
Binary Heap Implementation.....	285
Comparing Implementations.....	285
Heaps and Binary Heap	285
What is a Heap?	285
Types of Heaps?.....	286
Binary Heaps	287
Representing Heaps	287
Declaration of Heap.....	287
Creating Heap	287
Parent of a Node.....	288
Children of a Node.....	288
Getting the Maximum Element	288

Heapifying an Element	289
Deleting an Element	291
Inserting an Element	291
Destroying Heap	293
Heapifying the Array	293
Heapsort	294
Problems on Priority Queues [Heaps]	295
Chapter 9 Disjoint Sets ADT	307
Introduction	307
Equivalence Relations and Equivalence Classes	307
Disjoint Sets ADT	308
Applications	308
Tradeoffs in Implementing Disjoint Sets ADT	308
Fast FIND Implementation (Quick FIND)	309
Fast UNION Implementation (Quick UNION)	309
Fast UNION implementation (Slow FIND)	309
Fast UNION implementations (Quick FIND)	313
UNION by Size	313
UNION by Height (UNION by Rank)	314
Comparing UNION by Size and UNION by Height	315
Path Compression	316
Summary	317
Problems on Disjoint Sets	317
Chapter 10 Graph Algorithms	319
Introduction	319
Glossary	319
Applications of Graphs	322
Graph Representation	322
Adjacency Matrix	322
Adjacency List	324
Adjacency Set	326
Comparison of Graph Representations	326
Graph Traversals	327

Depth First Search [DFS].....	327
Breadth First Search [BFS].....	332
Comparing DFS and BFS	334
Topological Sort	335
Applications of Topological Sorting.....	336
Shortest Path Algorithms	337
Shortest Path in Unweighted Graph.....	337
Shortest path in Weighted Graph [Dijkstra's].....	339
Bellman-Ford Algorithm.....	343
Overview of Shortest Path Algorithms.....	344
Minimal Spanning Tree	344
Prim's Algorithm	344
Kruskal's Algorithm	345
Problems on Graph Algorithms	349
Chapter 11 Sorting	378
What is Sorting?.....	378
Why Sorting?	378
Classification	378
By Number of Comparisons	378
By Number of Swaps.....	378
By Memory Usage	378
By Recursion	379
By Stability	379
By Adaptability	379
Other Classifications.....	379
Internal Sort	379
External Sort.....	379
Bubble sort	379
Implementation.....	380
Performance	381
Selection Sort	381
Algorithm	381
Implementation.....	381

Performance	382
Insertion sort	382
Advantages	382
Algorithm	383
Implementation.....	383
Example	383
Analysis	384
Performance	384
Comparisons to Other Sorting Algorithms.....	384
Shell sort	385
Implementation.....	385
Analysis	386
Performance	386
Merge sort	386
Important Notes	386
Implementation.....	387
Analysis	388
Performance	388
Heapsort	388
Performance	389
Quicksort	389
Algorithm	389
Implementation.....	389
Analysis	390
Performance	392
Randomized Quick sort	392
Tree Sort	393
Performance	393
Comparison of Sorting Algorithms	393
Linear Sorting Algorithms.....	394
Counting Sort	394
Bucket sort [or Bin Sort].....	395
Radix sort.....	396

Topological Sort	396
External Sorting	397
Problems on Sorting	398
Chapter 12 Searching	414
What is Searching?	414
Why Searching?	414
Types of Searching	414
Unordered Linear Search	414
Sorted/Ordered Linear Search	415
Binary Search	415
Comparing Basic Searching Algorithms	417
Symbol Tables and Hashing	417
String Searching Algorithms	417
Problems on Searching	417
Chapter 13 Selection Algorithms [Medians]	450
What are Selection Algorithms?	450
Selection by Sorting	450
Partition-based Selection Algorithm	450
Linear Selection algorithm - Median of Medians algorithm	450
Finding the k Smallest Elements in Sorted Order	451
Problems on Selection Algorithms	451
Chapter 14 Symbol Tables	464
Introduction	464
What are Symbol Tables?	464
Symbol Table Implementations	465
Unordered Array Implementation	465
Ordered [Sorted] Array Implementation	465
Unordered Linked List Implementation	465
Ordered Linked List Implementation	465
Binary Search Trees Implementation	465
Balanced Binary Search Trees Implementation	465
Ternary Search Implementation	466
Hashing Implementation	466

Comparison of Symbol Table Implementations.....	466
Chapter 15 Hashing.....	467
What is Hashing?	467
Why Hashing?.....	467
HashTable ADT.....	467
Understanding Hashing.....	467
If Arrays Are There Why Hashing?.....	468
Components in Hashing	469
Hash Table.....	469
Hash Function.....	470
How to Choose Hash Function?.....	470
Characteristics of Good Hash Functions.....	470
Load Factor.....	470
Collisions	470
Collision Resolution Techniques.....	471
Separate Chaining	471
Open Addressing.....	471
Linear Probing	472
Quadratic Probing.....	472
Double Hashing.....	473
Comparison of Collision Resolution Techniques	473
Comparisons: Linear Probing vs. Double Hashing.....	473
Comparisons: Open Addressing vs. Separate Chaining	474
Comparisons: Open Addressing methods.....	474
How Hashing Gets $O(1)$ Complexity?.....	474
Hashing Techniques	474
Static Hashing	475
Dynamic Hashing.....	475
Problems for which Hash Tables are not Suitable	475
Problems on Hashing.....	475
Chapter 16 String Algorithms.....	489
Introduction	489
String Matching Algorithms.....	489

Brute Force Method.....	490
Robin-Karp String Matching Algorithm	490
Selecting Hash Function.....	490
Step by Step explanation.....	492
String Matching with Finite Automata	492
Finite Automata	492
How Finite Automata Works?	492
Important Notes for Constructing the Finite Automata	493
Matching Algorithm	493
KMP Algorithm	493
Filling Prefix Table	494
Matching Algorithm	495
Boyce-Moore Algorithm	498
Data structures for Storing Strings.....	499
Hash Tables for Strings.....	499
Binary Search Trees for Strings.....	499
Issues with Binary Search Tree Representation	499
Tries	500
What is a Trie?	500
Why Tries?	500
Inserting a String in Trie	501
Searching a String in Trie	501
Issues with Tries Representation	502
Ternary Search Trees.....	502
Ternary Search Trees Declaration.....	502
Inserting strings in Ternary Search Tree	503
Searching in Ternary Search Tree.....	505
Displaying All Words of Ternary Search Tree	506
Finding Length of Largest Word in TST.....	507
Comparing BSTs, Tries and TSTs	507
Suffix Trees.....	507
Prefix and Suffix.....	507
Observation	508

What is a Suffix Tree?	508
The Construction of Suffix Trees	508
Applications of Suffix Trees.....	511
Problems on Strings	511
Chapter 17 Algorithms Design Techniques	520
Introduction	520
Classification	520
Classification by Implementation Method	520
Recursion or Iteration.....	520
Procedural or Declarative (Non-Procedural)	521
Serial or Parallel or Distributed.....	521
Deterministic or Non-Deterministic.....	521
Exact or Approximate	521
Classification by Design Method.....	521
Greedy Method	521
Divide and Conquer.....	522
Dynamic Programming	522
Linear Programming.....	522
Reduction [Transform and Conquer].....	522
Other Classifications	523
Classification by Research Area	523
Classification by Complexity	523
Randomized Algorithms.....	523
Branch and Bound Enumeration and Backtracking.....	523
Chapter 18 Greedy Algorithms.....	524
Introduction	524
Greedy strategy	524
Elements of Greedy Algorithms.....	524
Greedy choice property	524
Optimal substructure	524
Does Greedy Works Always?	525
Advantages and Disadvantages of Greedy Method	525
Greedy Applications	525

Understanding Greedy Technique	525
Huffman coding algorithm	525
Problems on greedy algorithms	529
Chapter 19 Divide and Conquer Algorithms	540
Introduction	540
What is Divide and Conquer Strategy?	540
Does Divide and Conquer Work Always?	540
Divide and Conquer Visualization	540
Understanding Divide and Conquer	541
Advantages of Divide and Conquer	542
Disadvantages of Divide and Conquer	542
Master Theorem	543
Divide and Conquer Applications	543
Problems on Divide and Conquer	543
Chapter 20 Dynamic Programming	560
Introduction	560
What is Dynamic Programming Strategy?	560
Can Dynamic Programming Solve Any Problem?	560
Dynamic Programming Approaches	560
Bottom-up Dynamic Programming	561
Top-down Dynamic Programming	561
Bottom-up versus Top-down Programming	561
Examples of Dynamic Programming Algorithms	561
Understanding Dynamic Programming	561
Fibonacci Series	562
Observations	564
Factorial of a Number	564
Problems on Dynamic Programming	566
Chapter 21 Complexity Classes	611
Introduction	611
Polynomial/exponential time	611
What is Decision Problem?	612
Decision Procedure	612

What is a Complexity Class?	612
Types of Complexity Classes	612
P Class.....	612
NP Class.....	612
Co-NP Class.....	613
Relationship between P, NP and Co-NP	613
NP-hard Class.....	613
NP-complete Class	614
Relationship between P, NP Co-NP, NP-Hard and NP-Complete	614
Does $P=NP$?	615
Reductions.....	615
Important NP-Complete Problems (Reductions).....	616
Problems on Complexity Classes.....	618
Chapter 22 Miscellaneous Concepts.....	621
Introduction	621
Hacks on Bitwise Programming.....	621
Bitwise AND.....	621
Bitwise OR.....	621
Bitwise Exclusive-OR	622
Bitwise Left Shift.....	622
Bitwise Right Shift	622
Bitwise Complement.....	622
Checking whether K-th bit is set or not.....	623
Setting K-th bit.....	623
Clearing K-th bit	623
Toggling K-th bit.....	623
Toggling Rightmost One bit	624
Isolating Rightmost One bit	624
Isolating Rightmost Zero bit	624
Checking Whether Number is Power of 2 not	625
Multiplying Number by Power of 2.....	625
Dividing Number by Power of 2.....	625
Finding Modulo of a Given Number.....	625

Reversing the Binary Number.....	626
Counting Number of One's in number	626
Creating for Mask for Trailing Zero's	628

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 change. 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 allow the users to define their own data types called as user defined data types. Good examples 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 calling 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, etc..) support 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).

$$\text{Total Cost} = \text{cost_of_car} + \text{cost_of_cycle}$$

$$\text{Total Cost} \approx \text{cost_of_car} \text{ (approximation)}$$

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.

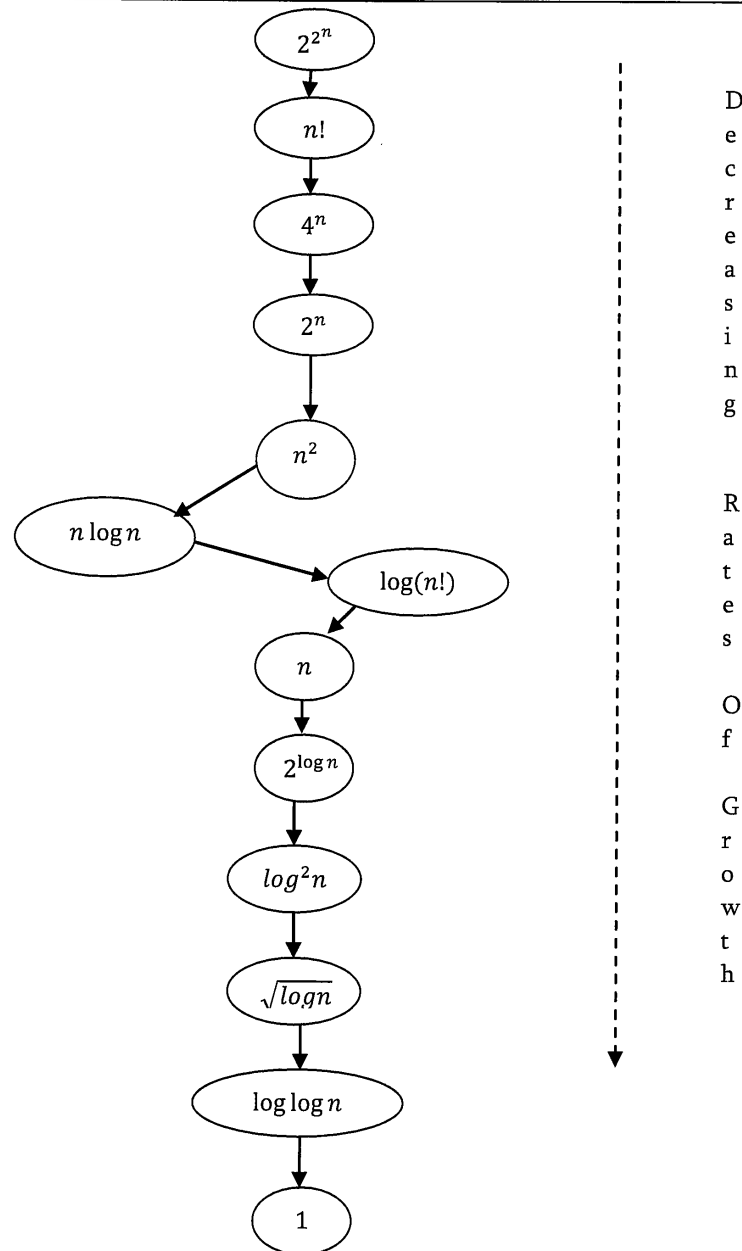
$$n^4 + 2n^2 + 100n + 500 \approx n^4$$

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
$\log n$	Logarithmic	Finding an element in a sorted array
n	Linear	Finding an element in an unsorted array
$n \log n$	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

$$\text{Lower Bound} \leq \text{Average Time} \leq \text{Upper Bound}$$

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

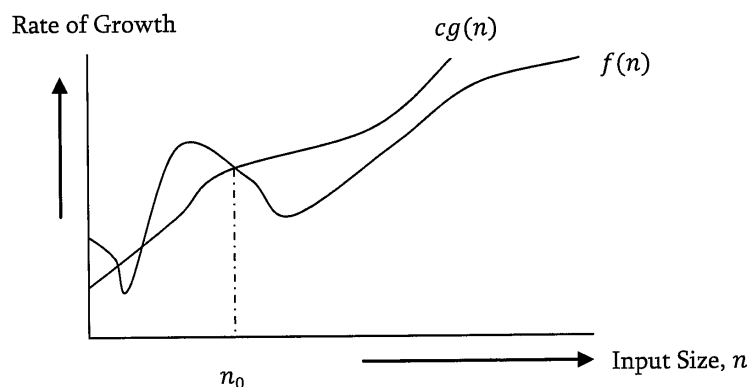
Having the expressions for best, average case and worst cases, for all the three cases we need to identify the upper and lower bounds. In order to represent these upper and lower bounds we need some kind syntax and that is the subject of following discussion. Let us assume that the given algorithm is 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 $O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$. $g(n)$ 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 $f(n)$.

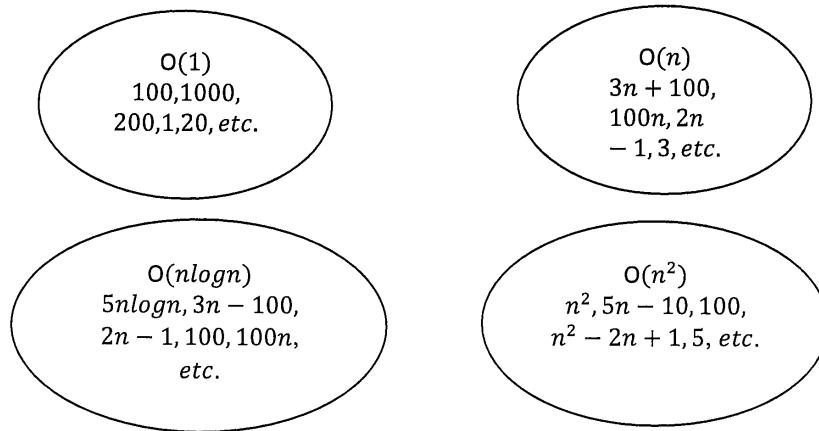
In general, we discard lower values of n . That means the rate of growth at lower values of n is not important. In the below figure, n_0 is the point from which we need to consider the rate of growths for a given algorithm. Below n_0 the rate of growths could be different.



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 $O(1)$, $O(n)$, $O(n \log n)$ 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?

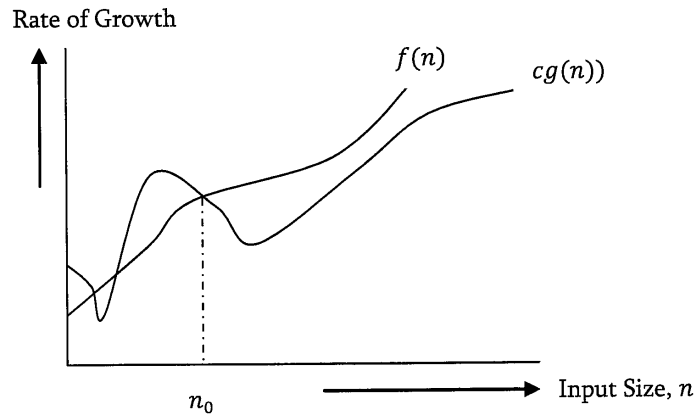
There are no unique set of values for n_0 and c in proving the asymptotic bounds. Let us consider, $100n + 5 = O(n^2)$. For this function there are multiple n_0 and c values possible.

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 $\Omega(g(n)) = \{f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$. $g(n)$ 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: $0 \leq cn \leq 5n^2 \Rightarrow cn \leq 5n^2 \Rightarrow c = 1$ and $n_0 = 1$
 $\therefore 5n^2 = \Omega(n)$ with $c = 1$ and $n_0 = 1$

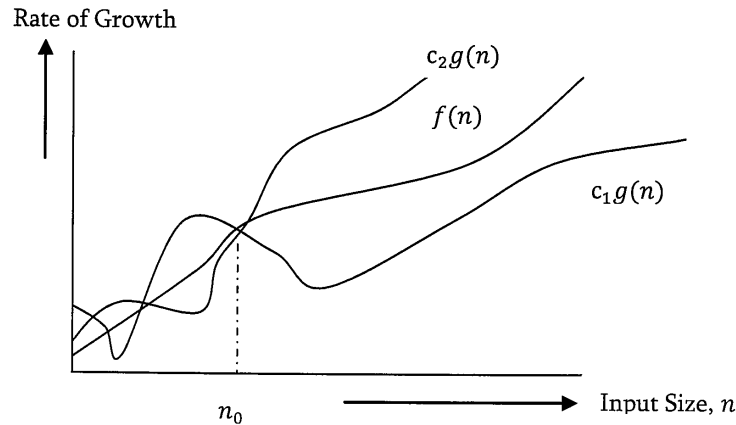
Example-2 Prove $f(n) = 100n + 5 \neq \Omega(n^2)$

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 $n = \Omega(2n)$, $n^3 = \Omega(n^2)$, $n = \Omega(\log n)$

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 $\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ for all } n \geq n_0\}$. $g(n)$ is an asymptotic tight bound for $f(n)$. $\Theta(g(n))$ is the set of functions with the same order of growth as $g(n)$.

⊖ Examples

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

Solution: $\frac{n^2}{5} \leq \frac{n^2}{2} - \frac{n}{2} \leq n^2$, for all, $n \geq 1$
 $\therefore \frac{n^2}{2} - \frac{n}{2} = \Theta(n^2)$ with $c_1 = 1/5, c_2 = 1$ and $n_0 = 1$

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

Solution: $c_1 n^2 \leq n \leq c_2 n^2 \Rightarrow$ only holds for: $n \leq 1/c_1$
 $\therefore n \neq \Theta(n^2)$

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

Solution: $c_1 n^2 \leq 6n^3 \leq c_2 n^2 \Rightarrow$ only holds for: $n \leq c_2/6$
 $\therefore 6n^3 \neq \Theta(n^2)$

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

Solution: $c_1 \log n \leq n \leq c_2 \log n \Rightarrow c_2 \geq \frac{n}{\log n}, \forall n \geq n_0$ - 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 ($i=1$; $i \leq n$; $i++$)

$m = m + 2$; // constant time, c

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

- 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 ($i=1$; $i \leq n$; $i++$) {

 // inner loop executed n times

 for ($j=1$; $j \leq n$; $j++$)

$k = k+1$; //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 ($i=1$; $i \leq n$; $i++$)

$m = m + 2$; //constant time

//outer loop executed n times

for ($i=1$; $i \leq n$; $i++$) {

 //inner loop executed n times

 for ($j=1$; $j \leq n$; $j++$)

$k = k+1$; //constant time

}

Total time = $c_0 + c_1n + c_2n^2 = O(n^2)$.

- 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($\text{length}() == 0$) {

 return false; //then part: constant

}

else { // else part: (constant + constant) * n

 for (int $n = 0$; $n < \text{length}()$; $n++$) {

 // another if : constant + constant (no else part)

 if(! $\text{list}[n].\text{equals}(\text{otherList.list}[n])$)

 //constant

 return false;

 }

}

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

- 5) **Logarithmic complexity:** An algorithm is $O(\log n)$ if it takes a constant time to cut the problem size by a fraction (usually by $\frac{1}{2}$). 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

$$\begin{aligned}\log(2^i) &= \log n \\ i \log 2 &= \log n \\ i &= \log n \quad // \text{if we assume base-2}\end{aligned}$$

Total time = $O(\log n)$.

Note: Similarly, for the below case also, worst case rate of growth is $O(\log n)$. 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

$$\begin{aligned}\log x^y &= y \log x & \log n &= \log_{10} n \\ \log xy &= \log x + \log y & \log^k n &= (\log n)^k \\ \log \log n &= \log(\log n) & \log \frac{x}{y} &= \log x - \log y \\ a^{\log_b x} &= x^{\log_b a} & \log_b x &= \frac{\log_a x}{\log_a b}\end{aligned}$$

Arithmetic series

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

Geometric series

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

Harmonic series

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