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

9 780615 459813 >

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

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

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).

What is an Algorithm?

Just to understand better, let us consider the problem of preparing an omelet. For preparing omelet, general steps which 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 a given problem.

One important note to remember while writing the algorithms is: we do not have to prove each step of the algorithm.

Why Analysis of Algorithms?

If we want to go from city "*A*" to city "*B*". 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, let us define some *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., $f(n)$). We can compare these different functions corresponding to running times and this kind of comparison is independent of machine time, programming style, etc..

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).

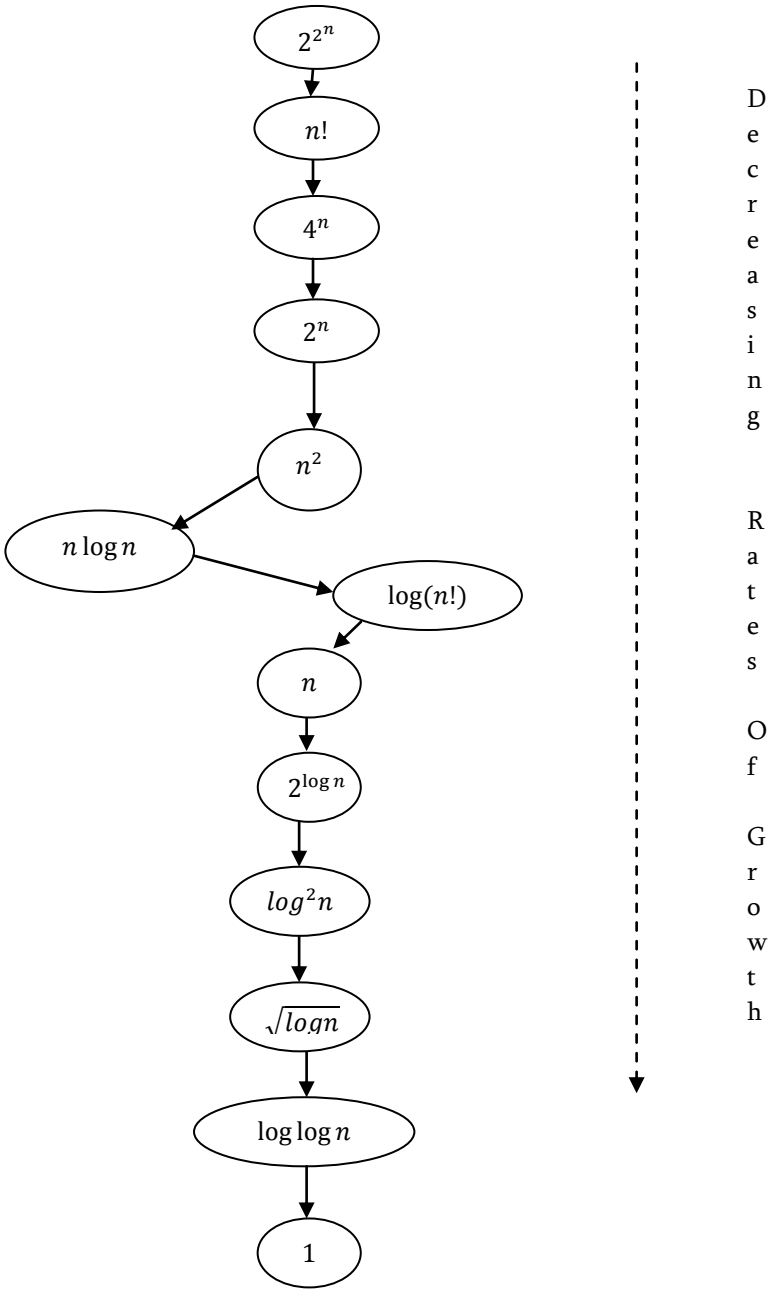
Total Cost = cost_of_car + cost_of_cycle
Total Cost ≈ cost_of_car (approximation)

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, $2n^2$, $100n$ 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.



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

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:

- **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

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

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?

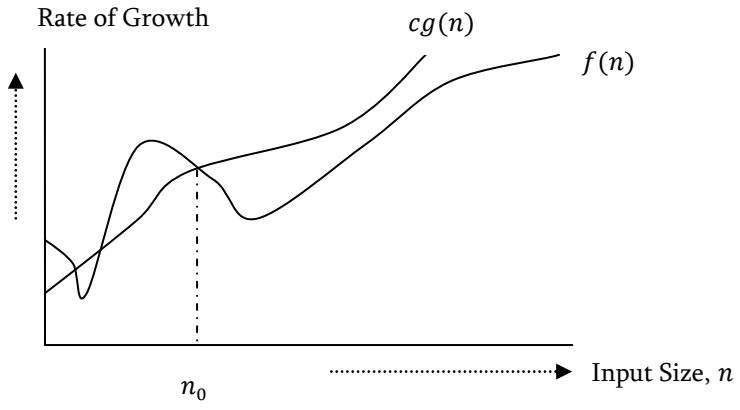
Having the expressions for best case, average case and worst case, for all the three cases we need to identify the upper bound, lower bounds. In order to represent these upper bound and lower bounds we need some syntax and that is the subject of following discussion. Let us assume that the given algorithm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

Big-O Notation

This notation gives the *tight* upper bound of the given function. Generally we represent it 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.

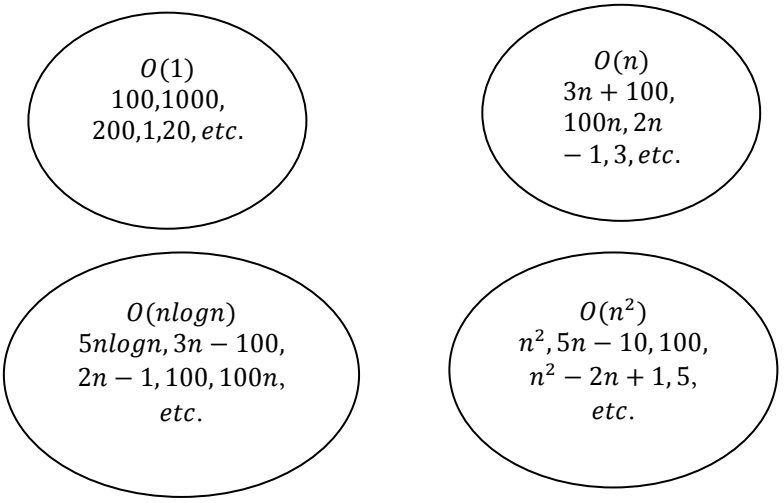
Let us see the O −notation with little more detail. O −notation defined as $O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$. $g(n)$ is an asymptotic tight upper bound for $f(n)$. Our objective is to give smallest rate of growth $g(n)$ 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 $n \geq 1$
$\therefore n^4 + 100n^2 + 50 = O(n^4)$ with $c = 2$ and $n_0 = 100$

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 is 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.

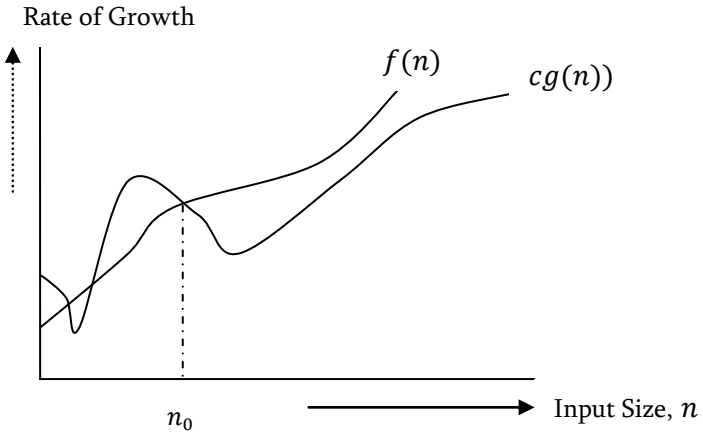
Solution2: $100n + 5 \leq 100n + 5n = 105n \leq 105n^2$ for all $n \geq 1, n_0 = 1$ and $c = 105$ is also a solution.

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

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

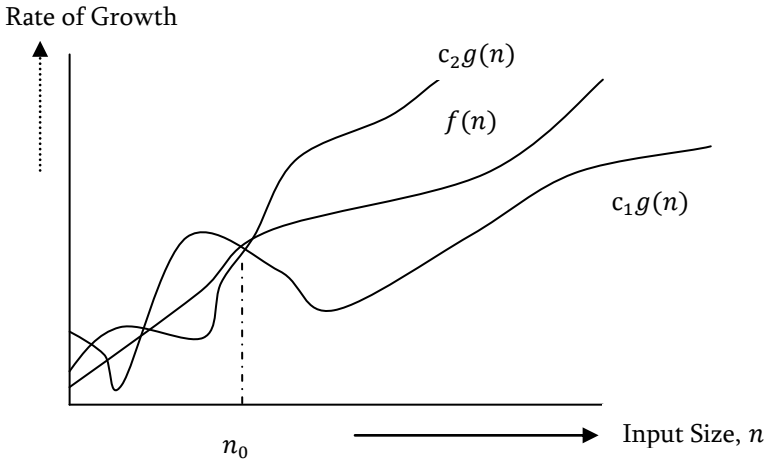
Solution: $\exists$ c, n_0 Such that: $0 \leq cn^2 \leq 100n + 5$
$100n + 5 \leq 100n + 5n\ (\forall n \geq 1) = 105n$

$cn^2 \leq 105n \Rightarrow n(cn - 105) \leq 0$
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 $n = \Omega(2n),\ n^3 = \Omega(n^2),\ n = \Omega(logn)$

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 $0 \leq c_1g(n) \leq f(n) \leq c_2g(n)$ 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 $g(n)$.

Θ 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 $c_1 = 1/5, c_1 = 1$ and $n_0 = 1$

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

Solution: $c_1 n^2 \leq n \leq c_2n^2 \Rightarrow$ 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 $n \neq \theta(logn)$

Solution: $c_1 logn \leq n \leq c_2 log\, n \Rightarrow c_2 \geq \frac{n}{\log n}, \forall n \geq n_0$ – 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.

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*.

Guidelines for Asymptotic Analysis?

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

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 $c \times n = cn = O(n)$.

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<=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 ( ) != otherStack. length ( ) )
{
    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(otherStack.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. That means at k^{th} step $2^i = n$ and we come out of loop. So, if we take logarithm on both sides,

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

So the total time = $O(logn)$.

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

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

Another example algorithm is binary search: finding a word in a dictionary of n pages

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

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

Commonly used Logarithms and Summations

Logarithms

$$\log x^y = y \log x$$
$$\log n = \log_e^n$$
$$\log xy = \log x + \log y$$
$$\log^k n = (\log n)^k$$
$$\log \log n = \log(\log n)$$
$$\log \frac{x}{y} = \log x - \log y$$
$$a^{\log_b^x} = x^{\log_b^a} \quad \log_b^x = \frac{\log_a^x}{\log_a^b}$$

Arithmetic series

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

Geometric series

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

Harmonic series

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

Other important formulae

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

Master Theorem for Divide and Conquer

In all divide and conquer algorithms we divide the problem into subproblems, each of which is some part of the original problem, and then perform some additional work to compute the final answer. As an example, if we consider merge sort [for details, refer *Sorting* chapter], it operates on two subproblems, each of which is half the size of the original, and then uses O(n) additional work for merging. This gives the running time equation:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

The following theorem can be used to determine the running time of divide and conquer algorithms. For a given program (algorithm), first we try to find the recurrence relation for the problem. If the recurrence is of the below form then we directly give the answer without fully solving it.

If the recurrence is of the form $T(n) = aT(\frac{n}{b}) + \Theta(n^k log^p n)$, where $a \geq 1, b > 1, k \geq 0$ and p is a real number, then:

1) If $a > b^k$, then $T(n) = \Theta\left(n^{log_b^a}\right)$
2) If $a = b^k$
   a. If $p > -1$, then $T(n) = \Theta\left(n^{log_b^a} log^{p+1} n\right)$
   b. If $p = -1$, then $T(n) = \Theta\left(n^{log_b^a} loglogn\right)$
   c. If $p < -1$, then $T(n) = \Theta\left(n^{log_b^a}\right)$
3) If $a < b^k$
   a. If $p \geq 0$, then $T(n) = \Theta\left(n^k log^p n\right)$
   b. If $p < 0$, then $T(n) = O\left(n^k\right)$

Problems Divide and Conquer Master Theorem

For each of the following recurrences, give an expression for the runtime $T(n)$ if the recurrence can be solved with the Master Theorem. Otherwise, indicate that the Master Theorem does not apply.

Problem-1 $T(n) = 3T(n/2) + n^2$

Solution: $T(n) = 3T(n/2) + n^2 \Rightarrow T(n) = \Theta(n^2)$ (Master Theorem Case 3.a)

Problem-2 $T(n) = 4T(n/2) + n^2$

Solution: $T(n) = 4T(n/2) + n^2 \Rightarrow T(n) = \Theta(n^2 logn)$ (Master Theorem Case 2.a)

Problem-3 $T(n) = T(n/2) + 2^n$

Solution: $T(n) = T(n/2) + 2^n \Rightarrow \Theta(2^n)$ (Master Theorem Case 3.a)

Problem-4 $T(n) = 2^n T(n/2) + n^n$

Solution: $T(n) = 2^n T(n/2) + n^n$ => Does not apply (a is not constant)

Problem-5 $T(n) = 16T(n/4) + n$

Solution: $T(n) = 16T(n/4) + n \Rightarrow T(n) = \Theta(n^2)$ (Master Theorem Case 1)

Problem-6 $T(n) = 2T(n/2) + n \log n$

Solution: $T(n) = 2T(n/2) + n \log n \Rightarrow T(n) = \Theta(n \log^2 n)$ (Master Theorem Case 2.a)

Problem-7 $T(n) = 2T(n/2) + n/\log n$

Solution: $T(n) = 2T(n/2) + n/\log n \Rightarrow T(n) = \Theta(n \text{loglogn})$ (Master Theorem Case 2.b)

Problem-8 $T(n) = 2T(n/4) + n^{0.51}$

Solution: $T(n) = 2T(n/4) + n^{0.51} \Rightarrow T(n) = O(n^{0.51})$ (Master Theorem Case 3.b)

Problem-9 $T(n) = 0.5T(n/2) + 1/n$

Solution: $T(n) = 0.5T(n/2) + 1/n \Rightarrow$ Does not apply ($a < 1$)

Problem-10 $T(n) = 6T(n/3) + n^2 \log n$

Solution: $T(n) = 6T(n/3) + n^2 \log n \Rightarrow T(n) = \Theta(n^2 \log n)$ (Master Theorem Case 3.a)

Problem-11 $T(n) = 64T(n/8) - n^2 \log n$

Solution: $T(n) = 64T(n/8) - n^2 \log n \Rightarrow$ Does not apply (function is not positive)

Problem-12 $T(n) = 7T(n/3) + n^2$

Solution: $T(n) = 7T(n/3) + n^2 \Rightarrow T(n) = \Theta(n^2)$ (Master Theorem Case 3.as)

Problem-13 $T(n) = 4T(n/2) + \log n$

Solution: $T(n) = 4T(n/2) + \log n \Rightarrow T(n) = \Theta(n^2)$ (Master Theorem Case 1)

Problem-14 $T(n) = 16T(n/4) + n!$

Solution: $T(n) = 16T(n/4) + n! \Rightarrow T(n) = \Theta(n!)$ (Master Theorem Case 3.a)

Problem-15 $T(n) = \sqrt{2}T(n/2) + \log n$

Solution: $T(n) = \sqrt{2}T(n/2) + \log n \Rightarrow T(n) = \Theta(\sqrt{n})$ (Master Theorem Case 1)

Problem-16 $T(n) = 3T(n/2) + n$

Solution: $T(n) = 3T(n/2) + n \Rightarrow T(n) = \Theta(n^{\log 3})$ (Master Theorem Case 1)

Problem-17 $T(n) = 3T(n/3) + \sqrt{n}$

Solution: $T(n) = 3T(n/3) + \sqrt{n} \Rightarrow T(n) = \Theta(n)$ (Master Theorem Case 1)

Problem-18 $T(n) = 4T(n/2) + cn$

Solution: $T(n) = 4T(n/2) + cn \Rightarrow T(n) = \Theta(n^2)$ (Master Theorem Case 1)

Problem-19 $T(n) = 3T(n/4) + nlogn$

Solution: $T(n) = 3T(n/4) + nlogn \Rightarrow T(n) = \Theta(nlogn)$ (Master Theorem Case 3.a)

Problem-20 $T(n) = 3T(n/3) + n/2$

Solution: $T(n) = 3T(n/3) + n/2 \Rightarrow T(n) = \Theta(nlogn)$ (Master Theorem Case 2.a)

Master Theorem for Subtract and Conquer Recurrences

Let $T(n)$ be a function defined on positive n, and having the property

$$T(n) = \begin{cases} c, & \text{if } n \leq 1 \\ aT(n-b) + f(n), & \text{if } n > 1 \end{cases}$$

for some constants $c, a > 0, b > 0, k \geq 0$, and function $f(n)$. If $f(n)$ is in $O(n^k)$, then

$$T(n) = \begin{cases} O(n^k), & \text{if a} < 1 \\ O(n^{k+1}), & \text{if a} = 1 \\ O\left(n^k a^{\frac{n}{b}}\right), & \text{if a} > 1 \end{cases}$$

Variant of subtraction and conquer master theorem

The solution to the equation $T(n) = T(\alpha n) + T((1-\alpha)n) + \beta n$, where $0 < \alpha < 1$ and $\beta > 0$ are constants, is $O(nlogn)$.

Problems on Algorithms Analysis

Note: From the following problems, try to understand in what cases we get different complexities ($O(n), O(logn), O(loglogn)$ etc..).

Problem-21 Find the complexity of the below recurrence:

$$T(n) = \begin{cases} 3T(n-1), if\ n > 0, \\ 1, \quad otherwise \end{cases}$$

Solution: Let us try to solve this function with substitution.

$T(n) = 3T(n-1)$

$T(n) = 3(3T(n-2)) = 3^2T(n-2)$

$T(n) = 3^2(3T(n-3))$

.

.

$T(n) = 3^nT(n-n) = 3^nT(0) = 3^n$

This clearly shows that the complexity of this function is $O(3^n)$.

Note: We can use the *Subtraction and Conquer* master theorem for this problem.

Problem-22 Find the complexity of the below recurrence:

$$T(n) = \begin{cases} 2T(n-1) - 1, if\ n > 0, \\ 1, \qquad\qquad otherwise \end{cases}$$

Solution: Let us try to solve this function with substitution.

$$T(n) = 2T(n-1) - 1$$

$$T(n) = 2(2T(n-2) - 1) - 1 = 2^2T(n-2) - 2 - 1$$

$$T(n) = 2^2(2T(n-3) - 2 - 1) - 1 = 2^3T(n-4) - 2^2 - 2^1 - 2^0$$

$$T(n) = 2^nT(n-n) - 2^{n-1} - 2^{n-2} - 2^{n-3} \ldots . 2^2 - 2^1 - 2^0$$

$$T(n) = 2^n - 2^{n-1} - 2^{n-2} - 2^{n-3} \ldots . 2^2 - 2^1 - 2^0$$

$$T(n) = 2^n - (2^n - 1)\ [note: 2^{n-1} + 2^{n-2} + \cdots + 2^0 = 2^n]$$

$$T(n) = 1$$

∴ Complexity is $O(1)$. Note that while the recurrence relation looks exponential the solution to the recurrence relation here gives a different result.

Problem-23 What is the running time of the following function (specified as a function of the input value n)

```
void Function(int n)
{
        int i=1 ;
        int s=1 ;
        while( s <= n)
        {
                i++ ;
                s= s+i ;
                print(\*");
        }
}
```

Solution: Consider the comments in below function:

```
void Function (int n)
{
        int i=1 ;
        int s=1 ;
        // s is increasing not at rate 1 but i
        while( s <= n)
        {
                i++ ;
```

```
                    s= s+i ;
                    print("*");
                }
        }
```

We can define the terms '*s*' according to the relation $s_i = s_{i-1} + i$. The value of '*i*' increases by one for each iteration. So the value contained in '*s*' at the i^{th} iteration is the sum of the first '*i*' positive integers. If k is the total number of iterations taken by the program, then the while loop terminates once.

$$1 + 2 + \ldots + k = \frac{k(k+1)}{2} > n \Rightarrow k = O(\sqrt{n}).$$

Problem-24 Find the complexity of the function given below.

```
void Function(int n)
{
        int i, count =0;;
        for(i=1; i*i<=n; i++)
                count++;
}
```

Solution: Consider the comments in below function:

```
void Function(int n)
{
        int i, count =0;;
        for(i=1; i*i<=n; i++)
                count++;
}
```

In the above function the loop will end, if $i^2 \leq n \Rightarrow T(n) = O(\sqrt{n})$. The reasoning is same as that of Problem-23.

Problem-25 What is the complexity of the below program:

```
void function(int n)
{
        int i, j, k , count =0;
        for(i=n/2; i<=n; i++)
                for(j=1; j + n/2<=n; j= j++)
                        for(k=1; k<=n; k= k * 2)
                                count++;
}
```

Solution: Consider the comments in the following function.

```
void function(int n)
{
        int i, j, k , count =0;
```

```
        //outer loop execute n/2 times
        for(i=n/2; i<=n; i++)
                //Middle loop executes n/2 times
                for(j=1; j + n/2<=n; j= j++)
                        //outer loop execute logn times
                        for(k=1; k<=n; k= k * 2)
                                count++;
}
```

The complexity of the above function is $O(n^2 \log n)$.

Problem-26 What is the complexity of the below program:

```
void function(int n)
{
        int i, j, k , count =0;
        for(i=n/2; i<=n; i++)
                for(j=1; j<=n; j= 2 * j)
                        for(k=1; k<=n; k= k * 2)
                                count++;
}
```

Solution: Consider the comments in the following function.

```
void function(int n)
{
        int i, j, k , count =0;
        //outer loop execute n/2 times
        for(i=n/2; i<=n; i++)
                //Middle loop executes logn times
                for(j=1; j<=n; j= 2 * j)
                        //outer loop execute logn times
                        for(k=1; k<=n; k= k*2)
                                count++;
}
```

The complexity of the above function is $O(n \log^2 n)$.

Problem-27 Find the complexity of the below program.

```
    function( int n )
    {
        if (n == 1) return;
        for( i = 1 ; i <= n ; i + + )
        {
                for( j= 1 ; j <= n ; j + + )
                {
                        print("*" ) ;
```

```
                    break;
                }
            }
        }
```

Solution: Consider the comments in the following function.

```
function( int n )
{
    //constant time
    if ( n == 1 ) return;
    //outer loop execute n times
    for( i = 1 ; i <= n ; i + + )
    {
        // inner loop executes only time due to break statement.
        for( j= 1 ; j <= n ; j + + )
        {
            print("*" ) ;
            break;
        }
    }
}
```

The complexity of the above function is $O(n)$. Even though the inner loop is bounded by n, but due to the break statement it is executing only once.

Problem-28 Write a recursive function for the running time $T(n)$ of the function function, whose code is below. Prove using the iterative method that $T(n) = \Theta(n^2)$.

```
function( int n )
{
    if ( n == 1 ) return ;
    for( i = 1 ; i <= n ; i + + )
        for( j = 1 ; j <= n ; j + + )
            print("*" ) ;
    function( n-3 );
}
```

Solution: Consider the comments in below function:

```
function (int n)
{
    //constant time
    if ( n == 1 ) return ;

    //outer loop execute n times
    for( i = 1 ; i <= n ; i + + )
```

```
                    //inner loop executes n times
                    for( j = 1 ; j <= n ; j + + )
                            //constant time
                            print("*" ) ;
            function( n-3 );
    }
```

The recurrence for this code is clearly $T(n) = T(n-3) + cn^2$ for some constant $c > 0$ since each call prints out n^2 asterisks and calls itself recursively on n - 3. Using the iterative method we get:

$$T(n) = T(n-3) + cn^2$$

Using the *Subtraction and Conquer* master theorem, we get $T(n) = \Theta(n^3)$.

Problem-29 Determine Θ bounds for the recurrence relation: $T(n) = 2T\left(\frac{n}{2}\right) + nlogn$

Solution: Using Divide and Conquer master theorem, we get $O(n\ log^2 n)$.

Problem-30 Determine Θ bounds for the recurrence: $T(n) = T(n/2) + T(n/4) + T(n/8) + n$

Solution: Substituting in the recurrence equation, we get:

$$T(n) \leq c1 * \frac{n}{2} + c2 * \frac{n}{4} + c3 * \frac{n}{8} + cn$$
$$\leq k * n, \text{ where } k \text{ is a constant.}$$

Problem-31 Determine Θ bounds for the recurrence relation: $T(n) = T(\lceil n/2 \rceil) + 7$

Solution: Using Master Theorem we get $\Theta(logn)$.

Problem-32 Prove that the running time of the code below is $\Omega(logn)$.

```
    Read(int n);
    {
            int k = 1 ;
            while( k < n )
                    k = 3k;
    }
```

Solution: The while loop will terminate once the value of *'k'* is greater than or equal to the value of *'n'*. Since each loop the value of *'k'* is being multiplied by 3, if i is the number of iterations, then *'k'* has the value of 3i after i iterations. That is the loop is terminated upon reaching i iterations when $3^i \geq n \leftrightarrow i \geq \log_3 n$, which shows that $i = \Omega(logn)$.

Problem-33 Solve the following recurrence.

$$T(n) = \begin{cases} 1, & if\ n = 1 \\ T(n-1) + n(n-1), & if\ n \geq 2 \end{cases}$$

Solution: By iteration:

$$T(n) = T(n-2) + (n-1)(n-2) + n(n-1)$$
$$\dots$$
$$T(n) = T(1) + \sum_{i=1}^{n} i(i-1)$$
$$T(n) = T(1) + \sum_{i=1}^{n} i^2 - \sum_{i=1}^{n} i$$
$$T(n) = 1 + \frac{n((n+1)(2n+1)}{6} - \frac{n(n+1)}{2}$$
$$T(n) = \Theta(n^3)$$

Note: We can use the *Subtraction and Conquer* master theorem for this problem.

Problem-34 Consider the following program:

```
Fib[n]
if (n==0) then return 0
else if (n==1) then return 1
else return Fib[n-1]+Fib[n-2]
```

Solution: The recurrence relation for running time of this program is

$$T(n) = T(n-1) + T(n-2) + c.$$

Notice T(n) has two recurrence calls indicating a binary tree. Each step recursively calls the program for n reduced by 1 and 2, so the depth of the recurrence tree is $O(n)$. The number of leaves at depth n is 2^n since this is a full binary tree, and each leaf takes at least $O(1)$ computation for the constant factor. Running time is clearly exponential in n.

Problem-35 Running time of following program?

```
function(n)
{
        for( i = 1 ; i <= n ; i + + )
                for( j = 1 ; j <= n ; j+ = i )
                        print( "*" ) ;
}
```

Solution: Consider the comments in below function:

```
function (n)
{
        //this loop executes n times
        for( i = 1 ; i <= n ; i + + )
```

```
                //this loop executes j times with j increase by the rate of i
                for( j = 1 ; j <= n ; j+ = i )
                        print( "*" ) ;
        }
```

Its running time is $n \times (\sqrt{n}) = O(n^{\frac{3}{2}})$[since the inner loop is same as that of Problem-23].

Problem-36 What is the complexity of $\sum_{i=1}^{n} log\ i$?

Solution: Using the logarithmic property, $logxy = logx + logy$, we can see that this problem is equivalent to

$$\sum_{i=1}^{n} \log i = \log 1 + \log 2 + \cdots + \log n$$

$$= \log(1 \times 2 \times \ldots \times n)$$

$$= \log(n!)$$

$$\leq \log(n^n)$$

$$\leq n\log n$$

This shows that that the time complexity $= O(nlogn)$.

Problem-37 What is the running time of the following recursive function (specified as a function of the input value n)? First write the recurrence formula and then find its complexity.

```
function(int n)
{
        if (n <= 1)
                return ;

        for (int i=1 ; i <= 3; i++ )
                f(⌈n/3⌉);
}
```

Solution: Consider the comments in below function:

```
function (int n)
{
        //constant time
        if (n <= 1)
                return ;

        //this loop executes with recursive loop of n/3 value
        for (int i=1 ; i <= 3; i++ )
                f(⌈n/3⌉);
}
```

We can assume that for asymptotical analysis $k = \lceil k \rceil$ for every integer $k \geq 1$. The recurrence for this code is $T(n) = 3T(\frac{n}{3}) + \Theta(1)$.
Using master theorem, we get $T(n) = \Theta(n)$.

Problem-38 What is the running time of the following recursive function (specified as a function of the input value n)? First write a recurrence formula, and show its solution using induction.

```
function(int n)
{
        if (n <= 1)
                return;
        for (i=1 ; i <= 3 ; i++ )
                function (n – 1).
}
```

Solution: Consider the comments in below function:

```
function (int n)
{
        //constant time
        if (n <= 1)
                return;
        //this loop executes 3 times with recursive call of n-1 value
        for (i=1 ; i <= 3 ; i++ )
                function (n – 1).
}
```

The $if\ statement$ requires constant time ($O(1)$). With the $for\ loop$, we neglect the loop overhead and only count the three times that the function is called recursively. This implies a time complexity recurrence:

$T(n) = c, if\ n \leq 1;$
$= c + 3T(n - 1), if\ n > 1.$

Now we use repeated substitution to guess at the solution when we substitute k times:

$$T(n) = c + 3T(n - 1)$$

Using the $Subtraction\ and\ Conquer$ master theorem, we get $T(n) = \Theta(3^n)$.

Problem-39 Write a recursion formula for the running time $T(n)$ of the function f, whose code is below. What is the running time of $function$, as a function of n?

```
function (int n)
{
        if (n <= 1)
                return;
```

```
        int i = 1 ;
        for(i = 1; i < n; i + +)
                print("*");
        function ( 0.8n ) ;
}
```

Solution: Consider the comments in below function:

```
function (int n)
{
        //constant time
        if (n <= 1)
                return;
        //constant time
        int i = 1 ;
        // this loop executes n times with constant time loop
        for(i = 1; i < n; i + +)
                print("*");

        //recursive call with 0.8n
        function ( 0.8n ) ;
}
```

The recurrence for this piece of code is $T(n) = T(.8n) + O(n)$

$$T(n) = T(\frac{4}{5}n) + \Theta(n)$$

$$T(n) = \frac{4}{5}T(n) + \Theta(n)$$

Applying master theorem, we get $T(n) = O(n)$.

Problem-40 Find the complexity of the recurrence: $T(n) = 2T(\sqrt{n}) + logn$

Solution: The given recurrence is not in the master theorem form. Let try to convert this master theorem format. For that let use assume that $n = 2^m$.

Applying logarithm on both sides gives, $logn = mlog2 \Rightarrow m = logn$
Now, the given function becomes,

$$T(n) = T(2^m) = 2T\left(\sqrt{2^m}\right) + m = 2T\left(2^{\frac{m}{2}}\right) + m.$$

To make it simple we assume $S(m) = T(2^m) \Rightarrow S(\frac{m}{2}) = T(2^{\frac{m}{2}}) \Rightarrow S(m) = 2S\left(\frac{m}{2}\right) + m$
Applying the master theorem would result $S(m) = O(mlogm)$
If we substitute $m = logn$ back, $T(n) = S(logn) = O((logn)\, loglogn)$.

Problem-41 Find the complexity of the recurrence: $T(n) = T(\sqrt{n}) + 1$

Solution: We apply the same logic as that of Problem-40 and we get

$$S(m) = S\left(\frac{m}{2}\right) + 1$$

Applying the master theorem would result $S(m) = O(logm)$.
Substituting $m = logn$, gives $T(n) = S(log\ n) = O(loglogn)$.

Problem-42 Find the complexity of the recurrence: $T(n) = 2T(\sqrt{n}) + 1$

Solution: Applying the logic of Problem-40, gives:

$$S(m) = 2S\left(\frac{m}{2}\right) + 1$$

Using the master theorem results $S(m) = O\left(m^{log_2^2}\right) = O(m)$.
Substituting $m = logn$ gives $T(n) = O(logn)$.

Problem-43 Find the complexity of the below function.

```
int Function (int n)
{
        if (n <= 2)
                return 1;
        else
                return (Function (floor(sqrt(n))) + 1);
}
```

Solution: Consider the comments in below function:

```
int Function (int n)
{
        //constant time
        if (n <= 2)
                return 1;
        else
                // executes √n + 1 times
                return (Function (floor(sqrt(n))) + 1);
}
```

For the above function, the recurrence function can be given as: $T(n) = T(\sqrt{n}) + 1$. And, this is same as that of Problem-41.

Problem-44 Analyze the running time of the following recursive procedure as a function of n.

```
void function(int n)
        {
                if ( n < 2 )
                        return;
                else
```

```
                counter = 0;
        for i = 1 to 8 do
                function (n/2);
        for I =1 to n^3 do
                counter = counter + 1;
}
```

Solution: Consider the comments in below function and let us refer to the running time of function (n) as $T(n)$.

```
void function(int n)
{
        //constant time
        if ( n < 2 )
                return;
        else
                counter = 0;
        // this loop executes 8 times with n value half in every call
        for i = 1 to 8 do
                function (n/2);
        // this loop executes n^3 times with constant time loop
        for I =1 to n^3 do
                counter = counter + 1;
}
```

$T(n)$ can be defined as follows:

$$T(n) = 1 \; if \; n < 2,$$
$$= 8T(\frac{n}{2}) + n^3 + 1 \; otherwise.$$

Using the master theorem gives, $T(n) = \Theta(n^{log_2^8} \log n) = \Theta(n^3 \, log \, n)$.

Problem-45 Find the complexity of the below function.

```
temp = 1
repeat
        for i = 1 to n
                temp = temp + 1;
        n = n/2;
until n <= 1
```

Solution: Consider the comments in below function:

```
//const time
temp = 1
repeat
```

```
            // this loops executes n times
            for i = 1 to n
                  temp =  temp + 1;
            //recursive call with n/2 value
            n  = n/2;
      until n <= 1
```

The recurrence for this function is $T(n) = T(\frac{n}{2}) + n$.
Using master theorem, we get, $T(n) = O(n)$.

Problem-46 Running time of following program?

```
function(int n)
{
      for( i = 1 ; i <= n ; i + + )
            for( j = 1 ; j <= n ; j * = 2 )
                  print( "*" ) ;
}
```

Solution: Consider the comments in below function:

```
function(int n)
{
      // this loops executes n times
      for( i = 1 ; i <= n ; i + + )
            // this loops executes logn times from our logarithms
            //guideline
            for( j = 1 ; j <= n ; j * = 2 )
                  print( "*" ) ;
}
```

Complexity of above program is : $O(nlogn)$.

Problem-47 Running time of following program?

```
function(int n)
{
      for( i = 1 ; i <= n/3 ; i + + )
            for( j = 1 ; j <= n ; j += 4 )
                  print( "*" ) ;
}
```

Solution: Consider the comments in below function:

```
function(int n)
{
```

```
        // this loops executes n/3 times
        for( i = 1 ; i <= n/3 ; i + + )
                // this loops executes n/4 times
                for( j = 1 ; j <= n ; j += 4)
                        print( "*" ) ;
}
```

The time complexity of this program is : $O(n^2)$.

Problem-48 Find the complexity of the below function.

```
  void function(int n)
  {
        if(n <=  1)
                return;
        if (n > 1)
          {
                print ("*");
                function( n/2 );
                function( n/2 );
          }
  }
```

Solution: Consider the comments in below function:

```
void function(int n)
{
        //constant time
        if(n <= 1)
                return;
        if (n > 1)
        {
                //constant time
                print ("*");
                //recursion with n/2 value
                function( n/2 );
                //recursion with n/2 value
                function( n/2 );
        }
}
```

The recurrence for this function is: $T(n) = 2T\left(\frac{n}{2}\right) + 1$
Using master theorem, we get $T(n) = O(n)$.

Problem-49 Find the complexity of the below function.

```
function()
{
        int i=1;
        while (i < n)
        {
                int j=n;
                while(j > 0)
                        j = j/2;
                i=2*i;
        } // i
}
```

Solution:

```
function()
{
        int i=1;
        while (i < n)
        {
                int j=n;
                while(j > 0)
                        j = j/2;   logn code ..

                i=2*i; //logn times
        } // i
}
```

Time Complexity: $O(logn * logn) = O(log^2 n)$.

Chapter 12 SEARCHING

What is Searching?

In computer science, searching is the process of finding an item with specified properties among a collection of items. The items may be stored as records in a database, simple data elements in arrays, text in files, nodes in trees, vertices and edges in graphs or may be elements of other search space.

Why Searching?

Searching is one of core computer science algorithms. We know that today's computers store lot of information. To retrieve this information efficiently we need very efficient searching algorithms.

There are certain ways of organizing the data which improves the searching process. That means, if we keep the data in some proper order then it is easy to search the required element. Sorting is one of the techniques for making the elements ordered.

In this chapter we will see different searching algorithms.

Types of Searching

The following are the types of searches which we will be discussing in this book.

- Unordered Linear Search
- Sorted/Ordered Linear Search
- Binary Search
- Symbol Tables and Hashing
- String Searching Algorithms: Tries, Ternary Search and Suffix Trees

Unordered Linear Search

Let us assume that given an array whose elements order is not known. That means the elements of the array are not sorted. In this case if we want to search for an element then we have to scan the complete array and see if the element is there in the given list or not.

int UnsorteddLinearSearch (int A[], int n, int data)

```
{
        for (int i  = 0; i < n; i++)
        {
                if (A[i] == data)
                        return i;
        }
        return -1;
}
```

Time complexity of this algorithm is $O(n)$. This is because in the worst case we need to scan the complete array.
Space complexity: $O(1)$.

Sorted/Ordered Linear Search

If the elements of the array are already sorted then in many cases we don't have to scan the complete array to see if the element is there in the given array or not. In the below algorithm, it can be seen that, at any point if the value at $A[i]$ is greater than the $data$ to be searched then we just return -1 without searching the remaining array.

```
int SortedLinearSearch(int A[], int n, int data)
{
        for (int i  = 0; i < n; i++)
        {
                if (A[i] == data)
                        return i;
                else if(A[i] > data)
                        return -1;
        }
        return -1;
}
```

Time complexity of this algorithm is $O(n)$. This is because in the worst case we need to scan the complete array. But in the average case it reduces the complexity even though the growth rate is same.
Space complexity: $O(1)$.

Note: For the above algorithm we can make further improvement by incrementing the index at faster rate (say, 2). This will reduce the number of comparisons for searching in the sorted list.

Binary Search

If we consider searching of a word in a dictionary, in general we directly go some approximate page [generally middle page] start searching from that point. If the *name* that we are searching is same then we are done with the search. If the page is before the selected pages then apply the same process for the first half otherwise apply the same process to the second half. Binary search also works in the same way. The algorithm applying such a strategy is referred to as *binary search* algorithm.

```
//Iterative Binary Search Algorithm
int BinarySearchIterative[int A[], int n, int data)
{
        int low =  0;
        int high = n-1;
        while (low <= high)
        {
                mid  = low + (high-low)/2; //To avoid overflow

                if (A[mid] == data)
                        return mid;
                else if (A[mid] < data)
                        low =  mid + 1;
                else
                        high =  mid - 1;
        }
        return -1;
}

//Recursive Binary Search Algorithm
int BinarySearchRecursive[int A[], int low, int high, int data)
{
        int mid  = low + (high-low)/2; //To avoid overflow

        if (A[mid] == data)
                return mid;
        else if (A[mid] < data)
                return BinarySearchRecursive (A, mid + 1, high, data);
        else
                return BinarySearchRecursive (A, low, mid - 1 , data);

        return -1;
}
```

Recurrence for binary search is $T(n) = T(\frac{n}{2}) + \Theta(1)$. This is because we are always considering only half of the input list and throwing out the other half. Using *Divide and Conquer* master theorem, we get, $T(n) = O(logn)$.

Time Complexity: $O(logn)$.
Space Complexity: $O(1)$ [for iterative algorithm].

Comparing Basic Searching Algorithms

Implementation	Search-Worst Case	Search-Avg. Case
Unordered Array	n	$\frac{n}{2}$
Ordered Array	$logn$	$logn$
Unordered List	n	$\frac{n}{2}$
Ordered List	n	$\frac{n}{2}$
Binary Search (arrays)	$logn$	$logn$
Binary Search Trees (for skew trees)	n	$logn$

Note: For discussion on binary search trees refer *Trees* chapter.

Symbol Tables and Hashing

Refer *Symbol Tables* and *Hashing* chapters.

String Searching Algorithms

Refer *String Algorithms* chapter.

Problems on Searching

Problem-1 Given an array of n numbers. Give an algorithm for checking whether there are any duplicated elements in the array or not?

Solution: This is one of the simplest problems. One obvious answer to this is, exhaustively searching for duplicated in the array. That means, for each input element check whether there is any element with same value. This we can solve just by using two simple *for* loops. The code for this solution can be given as:

```
void CheckDuplicatesBruteForce(int A[], int n)
{
        int i =  0, j=0;
```

```
        for(i = 0; i < n; i++)
        {
                for(j = i+1; j < n; j++)
                {
                        if(A[i] == A[j])
                        {
                                printf("Duplicates exist: %d", A[i]);
                                return;
                        }
                }

        }
        printf("No duplicates in given array.");
}
```

Time Complexity: $O(n^2)$. This is because of two nested for loops.
Space Complexity: $O(1)$.

Problem-2 Can we improve the complexity of Problem-1's solution?

Solution: Yes. Sort the given array. After sorting all the elements with equal values come adjacent. Now, just do another scan on this sorted array and see if there are elements with same value and adjacent.

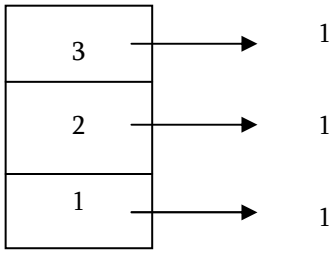
```
void CheckDuplicatesBruteForce(int A[], int n)
{
        //sort the array
        Sort(A, n);

        for(int i = 0; i < n-1; i++)
        {
                if(A[i] == A[i+1])
                {
                        printf("Duplicates exist: %d", A[i]);
                        return;
                }
        }
        printf("No duplicates in given array.");
}
```

Time Complexity: $O(nlogn)$. This is because of sorting.
Space Complexity: $O(1)$.

Problem-3 Is there any other way of solving the Problem-1?

Solution: Yes, using hash table. Hash tables are a simple and effective method to implement dictionaries. *Average* time to search for an element is $O(1)$, while worst-case time is $O(n)$. Refer *Hashing* chapter for full details on hashing algorithms.

For example, consider the array, $A = \{3, 2, 1, 2, 2, 3\}$. Scan the input array and insert the elements into the hash. For inserted element, keep the *counter* as 1. This indicates that the corresponding element has occurred already. For the given array, the hash table will look like (after inserting first three elements 3, 2 and 1):



If we try inserting 2, since the counter value of 2 is already 1, then we can say the element is appearing twice.

Time Complexity: $O(n)$.
Space Complexity: $O(n)$.

Problem-4 Can we further improve the complexity of Problem-1's solution?

Solution: Let us assume that the array elements are positive numbers and also all the elements are in the range 0 to $n - 1$. For each element $A[i]$, we go to the array element whose index is $A[i]$. That means we select $A[A[i]]$ and mark - $A[A[i]]$ (that means we negate the value at $A[A[i]]$). We continue this process until we encounter the element whose value is already negated. If one such element exists then we say duplicate elements exist in the given array. As an example, consider the array, $A = \{3, 2, 1, 2, 2, 3\}$.

Initially,

3	2	1	2	2	3
0	1	2	3	4	5

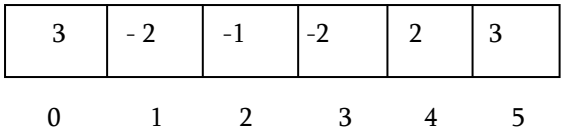
At step-1, negate A[abs(A[0])],

3	2	1	-2	2	3
0	1	2	3	4	5

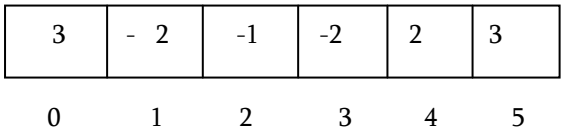
At step-2, negate A[abs(A[1])],

3	2	-1	-2	2	3
0	1	2	3	4	5

At step-3, negate A[abs(A[2])],



At step-4, negate A[abs(A[3])],



At step-4, we can observe that $A[abs(A[3])]$ is already negative. That means we have encountered the same value twice.

The code for this algorithm can be given as:

```
void CheckDuplicates(int A[], int n)
{
        int i =  0;
        for(i = 0; i < n; i++)
        {
                if(A[abs(A[i])] < 0)
                {
                        printf("Duplicates exist:%d", A[i]);
                        return;
                }
                else
                {
                        A[A[i]] = - A[A[i]];
                }
        }
        printf("No duplicates in given array.");
}
```

Time Complexity: $O(n)$. Since, only one scan is required.
Space Complexity: $O(1)$.

Note:

- This solution does not work if the given array is read only.
- This solution will work only if all the array elements are positive.
- If the elements range is not in 0 to $n - 1$ then it may give exceptions.

Problem-5 Given an array of n numbers. Give an algorithm for finding the first element in the array which is repeated?

For example, consider the array, $A = \{3, 2, 1, 2, 2, 3\}$. In this array the first repeated number is 3 (not 2). That means, we need to return the first element among the repeated elements.

Solution: We can use the brute force solution of Problem-1. Because it for each element it checks whether there is a duplicate for that element or not. So, whichever element duplicates first then that element is returned.

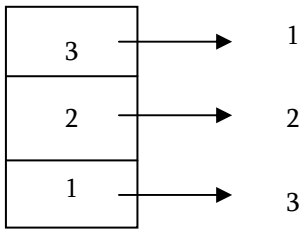
Problem-6 For Problem-5, can we use sorting technique?

Solution: No. For proving the failed case, let us consider the following array. For example, $A = \{3, 2, 1, 2, 2, 3\}$. Then after sorting we get $A = \{1, 2, 2, 2, 3, 3\}$. In this sorted array the first repeated element is 2 but the actual answer is 3.

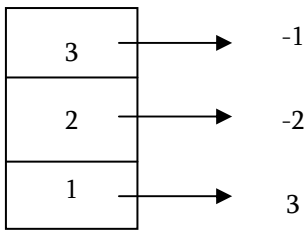
Problem-7 For Problem-5, can we use hashing technique?

Solution: Yes. But the simple technique which we used for Problem-3 will not work. For example, if we consider the input array as $A = \{3, 2, 1, 2, 3\}$, in this case the first repeated element is 3 but using our simple hashing technique we the answer as 2. This is because of the fact that 2 is coming twice before 3. Now let us change the hashing table behavior so that we get the first repeated element.

Let us say, instead of storing 1 value, initially we store the position of the element in the array. As a result the hash table will look like (after inserting 3, 2 and 1):



Now, if we see 2 again, we just negate the current value of 2 in the hash table. That means, we make its counter value as -2. The negative value in the hash table indicates that we have seen the same element two times. Similarly, for 3 (next element in input) also, we negate the current value of hash table and finally the hash table will look like:



After scanning the complete array, we scan the hash table and return the highest negative indexed value from it (i.e., -1 in our case). The highest negative value indicates that we have seen that element first (among repeated elements) and also repeating.

What if the element is repeated more than two times?

In this case, what we can do is, just skip the element if the corresponding value i already negative.

Problem-8 For Problem-5, can we use Problem-3's technique (negation technique)?

Solution: No. As a contradiction example, for the array $A = \{3, 2, 1, 2, 2, 3\}$ the first repeated element is 3. But with negation technique the result is 2.

Problem-9 Given an array of n elements. Find two elements in the array such that their sum is equal to given element K?

Solution: Brute Force Approach
One simple solution to this is, for each input element check whether there is any element whose sum is K. This we can solve just by using two simple for loops. The code for this solution can be given as:

```
void BruteForceSearch[int A[], int n, int K)
{
        int i =  0, j = 0;
        for (i = 0; i < n; i++)
        {
                for(j = i; j < n; j++)
                {
                        if(A[i]+A[j] == K)
                        {
                                printf("Items Found:%d %d", i, j);
                                return;
                        }
                }
        }
        printf("Items not found: No such elements");
}
```

Time Complexity: $O(n^2)$. This is because of two nested for loops.
Space Complexity: $O(1)$.

Problem-10 Does the solution of Problem-9 works even if the array is not sorted?

Solution: Yes. Since we are checking all possibilities, the algorithm ensures that we get the pair of numbers if they exist.

Problem-11 For the Problem-9, can we improve the time complexity?

Solution: Yes. Let us assume that we have sorted the given array. This operation takes $O(n\ logn)$. On the sorted array, maintain indices $loIndex = 0$ and hiIndex = $n - 1$ and compute $A[loIndex] + A[hiIndex]$. If the sum equals K, then we are done with the solution. If the sum is less than K, decrement $hiIndex$, if the sum is greater than K, increment $loIndex$.

```
void Search[int A[], int n, int K)
{
        int i, j, temp;
        Sort(A, n);
        for(i = 0, j = n-1; i < j; )
        {
                temp = A[i] + A[j];
                if (temp == K)
                {
                        printf("Elements Found: %d %d", i, j);
                        return;
                }
                else if (temp < K)
                        i = i + 1;
                else
                        j = j - 1;
        }
        return;
}
```

Time Complexity: $O(nlogn)$. If the given array is already sorted then the complexity is $O(n)$.
Space Complexity: $O(1)$.

Problem-12 Is there any other way of solving the Problem-9?

Solution: Yes, using hash table.

Since our objective is to find two indexes of the array whose sum is K. Let us say those indexes are X and Y. That means, $A[X] + A[Y] = K$.

What we need is, for each element of the input array $A[X]$, check whether $K - A[X]$ also exists in input array. Now, let us simplify that searching with hash table.

Algorithm

- For each element of the input array, insert into the hash table. Let us say the current element is $A[X]$.
- Before proceeding to the next element we check whether $K - A[X]$ also exists in hash table or not.

- Existence of such number indicates that we are able to find the indexes.
- Otherwise proceed to the next input element.

Time Complexity: $O(n)$.
Space Complexity: $O(n)$.

Problem-13 Given an array A of n elements. Find three elements, i, j and k in the array such that $A[i]^2 + A[j]^2 = A[k]^2$?

Solution:

Algorithm

- For each array index i compute $A[i]^2$ and store in array.
- Now, the problem reduces to finding three indexes , i, j and k such that $A[i] + A[j] = A[k]$. This is same as that of Problem-9.

Problem-14 Two elements whose sum is closest to zero

Given an array with both positive and negative numbers. We need to find the two elements such that their sum is closest to zero. For the below array, algorithm should give -80 and 85.

Example: $1\ 60\ -10\ 70\ -80\ 85$

Solution: Brute Force Solution.

For each element, find the sum of it with every other element in the array and compare sums. Finally, return the minimum sum.

```
void TwoElementsWithMinSum(int A[], int n)
{
        int inv_count = 0;
        int i, j, min_sum, sum, min_i, min_j;

        if(n < 2)
        {
                printf("Invalid Input");
                return;
        }

        /* Initialization of values */
        min_i = 0;
        min_j = 1;
        min_sum = A[0] + A[1];
```

```
        for(i= 0; i < n - 1; i ++)
        {
                for(j = i + 1; j < n; j++)
                {
                        sum = A[i] + A[j];
                        if(abs(min_sum) > abs(sum))
                        {
                                min_sum = sum;
                                min_i = i;
                                min_j = j;
                        }
                }
        }
        printf(" The two elements are %d and %d", arr[min_i], arr[min_j]);
}
```

Time complexity: $O(n^2)$.
Space Complexity: $O(1)$.

Problem-15 Can we improve the time complexity of Problem-14?

Solution: Use Sorting.

Algorithm

1) Sort all the elements of the given input array.
2) Find the two elements on either side of zero (if they are all positive or all negative then we are done with the solution)
3) If the one is positive and other is negative then add the two values at those positions. If the total is positive then increment the negative index, if it is negative then increment the positive index. If it is zero then stop.
4) loop step (3) until we hit a zero total or reached the end of array. Store the best total as you go.

Time Complexity: $O(nlogn)$, for sorting.

Problem-16 Given an array of n elements. Find three elements in the array such that their sum is equal to given element K?

Solution: Brute Force Approach.

The default solution to this is, for each pair of input elements check whether there is any element whose sum is K. This we can solve just by using three simple for loops. The code for this solution can be given as:

```
void BruteForceSearch[int A[], int n, int data)
{
        int i =  0, j = 0, k = 0;
        for (i = 0; i < n; i++)
        {
                for(j = i+1; j < n; j++)
                {
                        for(k = j+1; k < n; k++)
                        {
                                if(A[i] + A[j] + A[k]== data)
                                {
                                        printf("Items Found:%d %d %d", i, j, k);
                                        return;
                                }
                        }
                }
        }
        printf("Items not found: No such elements");
}
```

Time Complexity: $O(n^3)$. This is because of three nested for loops.
Space Complexity: $O(1)$.

Problem-17 Does the solution of Problem-16 works even if the array is not sorted?

Solution: Yes. Since we are checking all possibilities, the algorithm ensures that we can find three numbers whose sum is K if they exist.

Problem-18 Can we use sorting technique for solving Problem-16*?*

Solution: Yes.

```
void Search[int A[], int n, int data)
{
        int i, j;
        Sort(A, n);
        for(k = 0; k < n; k++)
        {
                for(i =  k + 1, j = n-1; i < j;  )
                {
                        if (A[k] + A[i] + A[j]  == data)
                        {
                                printf("Items Found:%d %d %d", i, j, k);
```

```
                        return;
                    }
                    else if (A[k] + A[i] + A[j]  < data)
                        i =  i + 1;
                    else
                        j =  j - 1;
                }
            }
            return;
}
```

Time Complexity: Time for sorting + Time for searching in sorted list $= O(nlogn) + O(n^2) \approx O(n^2)$. This is because of two nested *for* loops.
Space Complexity: $O(1)$.

Problem-19 Can we use hashing technique for solving Problem-16*?*

Solution: Yes. Since our objective is to find three indexes of the array whose sum is K. Let us say those indexes are X, Y and Z. That means, $A[X] + A[Y] + A[Z] = K$.

Let us assume that we have kept all possible sums along with their pairs in hash table. That means the key to hash table is $K - A[X]$ and values for $K - A[X]$ are all possible pairs of input whose sum is $K - A[X]$.

Algorithm

- Before starting the searching, insert all possible sums with pairs of elements into the hash table.
- For each element of the input array, insert into the hash table. Let us say the current element is $A[X]$.
- Check whether there exists a hash entry in the table with key: $K - A[X]$.
- If such element exists then scan the element pairs of $K - A[X]$ and return all possible pairs by including $A[X]$ also.
- If no such element exists (with $K - A[X]$ as key) then go to next element.

Time Complexity: Time for storing all possible pairs in Hash table + searching = $O(n^2) + O(n^2) \approx O(n^2)$.
Space Complexity: $O(n)$.

Problem-20 Given an array of n integers, the $3 - sum\ problem$ is to determine find three integers whose sum is closest to $zero$.

Solution: This is same as that of Problem-16. In this case, K value is zero.

Problem-21 Given an array of n numbers. Give an algorithm for finding the element which appears maximum number of times in the array?

Solution: Brute Force.

One simple solution to this is, for each input element check whether there is any element with same value and for each such occurrence, increment the counter. Each time, check the current counter with the max counter and update it if this its value is greater than max counter. This we can solve just by using two simple for loops. The code for this solution can be given as:

```
int CheckDuplicatesBruteForce(int A[], int n)
{
        int i =  0, j=0;
        int counter =0, max=0;
        for(i = 0; i < n; i++)
        {
                counter=0;
                for(j = 0; j < n; j++)
                {
                        if(A[i] == A[j])
                                counter++;
                }
                if (counter > max)
                        max = counter;
        }
        return max;
}
```

Time Complexity: $O(n^2)$. This is because of two nested for loops.
Space Complexity: $O(1)$.

Problem-22 Can we improve the complexity of Problem-21 solution?

Solution: Yes. Sort the given array. After sorting all the elements with equal values come adjacent. Now, just do another scan on this sorted array and see which element is appearing maximum number of times.

Time Complexity: $O(nlogn)$. (for sorting).
Space Complexity: $O(1)$.

Problem-23 Is there any other way of solving Problem-21?

Solution: Yes, using hash table. For each element of the input keep track of how many times that element appeared in the input. That means the counter value represents the number of occurrences for that element.

Time Complexity: $O(n)$.
Space Complexity: $O(n)$.

Problem-24 For Problem-21, can we improve the time complexity? Assume that the elements range is o to $n-1$. That means all the elements are within this range only.

Solution: Yes. We solve this problem in two scans. We *cannot* use the negation technique of Problem-3 for this problem because of number of repetitions.

In the first scan, instead of negating we add the value n. That means for each of occurrence of an element we add the array size to that element.

In the second scan we check the element value by dividing it with n and we return the element whichever gives the maximum value. The code based on this method is given below.

```
void MaxRepititions(int A[], int n)
{
        int i =  0;
        int max = 0;
        for(i = 0; i < n; i++)
        {
                A[A[i]%n] +=n;
        }
        for(i = 0; i < n; i++)
        {
                if(A[i]/n > max)
                {
                        max = A[i]/n;
                        max =i;
                }
        }
        return max;
}
```

Note:

- This solution does not work if the given array is read only.
- This solution will work only if the array elements are positive.
- If the elements range is not in 0 to $n-1$ then it may give exceptions.

Time Complexity: $O(n)$. Since no nested for loops are required.

Space Complexity: $O(1)$.

Problem-25 Let A be an array of n distinct integers. Suppose A has the following property: there exists an index $1 \leq k \leq n$ such that $A[1], \ldots, A[k]$ is an increasing sequence and $A[k+1], \ldots, A[n]$ is a decreasing sequence. Design and analyze an efficient algorithm for finding k.

Similar question:

Lets us assume that the given array is sorted but starts with negative numbers and ends with positive numbers [such functions are called monotonically increasing function]. In this array find the starting index of the positive numbers. Let us assume that we know the length of the input array. Design a $O(logn)$ algorithm.

Solution: We use a variant of the binary search.

```
int Search (int A[], int first, int last)
{
        int first = 0;
        int last = n-1;
        int mid;

        while(first <= last)
        {
                // if the current array has size 1
                if(first == last)
                        return A[first];
                // if the current array has size 2
                else if(first == last-1)
                        return max(A[first], A[last]);
                // if the current array has size 3 or more
                else
                {
                        mid = first + (last-first)/2;

                        if(A[mid-1] < A[mid] && A[mid] > A[mid+1])
                                return A[mid];
                        else if(A[mid-1] < A[mid] && A[mid] < A[mid+1])
                                first = mid+1;
                        else if(A[mid-1] > A[mid] && A[mid] > A[mid+1])
                                last = mid-1;
                        else
                                return INT_MIN ;
                } // end of else
        } // end of while
```

}

The recursion equation is $T(n) = 2T(n/2) + c$. Using master theorem, we get $O(logn)$.

Problem-26 If we don't know n, how do we solve the Problem-25?

Solution: Repeatedly compute $A[1], A[2], A[4], A[8], A[16]$, and so on until we find a value of n such that $A[n] > 0$.

Time Complexity: $O(logn)$, since we are moving at the rate of 2.
Refer *Introduction to Analysis of Algorithms* chapter for details on this.

Problem-27 Given an input array of size unknown with all $1's$ in the beginning and $0's$ in the end. Find the index in the array from where $0's$ start. Consider there are millions of $1's$ and $0's$ in the array. E.g. array contents 1111111…….1100000……..0000000.

Solution: This problem is almost similar to Problem-26. Check the bits at the rate of $2^K$where $k = 0, 1, 2$ ….

Since we are moving at the rate of 2, the complexity is $O(logn)$.

Problem-28 Given a sorted array of n integers that has been rotated an unknown number of times, give a $O(log\ n)$ algorithm that finds an element in the array.
Example: Find 5 in array (15 16 19 20 25 1 3 4 5 7 10 14)
Output: 8 (the index of 5 in the array)

Solution: Let us assume that the given array is $A[]$. Using solution of Problem-25, with extension. The below function $FindPivot$ returns the k value (let us assume that this function return the index instead of value). Find the pivot point, divide the array in two sub-arrays and call binary search.

The main idea for finding pivot is – for a sorted (in increasing order) and pivoted array, pivot element is the only element for which next element to it is smaller than it. Using above criteria and binary search methodology we can get pivot element in $O(logn)$ time

Algorithm

1) Find out pivot point and divide the array in two sub-arrays.
2) Now call binary search for one of the two sub-arrays.
    a. if element is greater than first element then search in left subarray
    b. else search in right subarray
3) If element is found in selected sub-array then return index *else* return -1.

```
int FindPivot(int A[], int start, int finish)
{
```

```
        if(finish - start == 0)
                return start;
        else if(start) == finish - 1)
        {
                if (A[start] >= A[finish])
                        return start;
                else
                        return finish;
        }
        else
        {
                mid = start + (finish-start)/2;
                if (A[mid] >= A[mid + 1])
                        return FindPivot(A, start, mid);
                else
                        return FindPivot(A, mid, finish);
        }
}

int Search(int A[], int n, int x)
{
        int pivot = FindPivot(A, 0, n-1);
        if(A[pivot] == x)
                return pivot;
        if(A[pivot] <= x)
                return BinarySearch(A, 0, pivot-1, x);
        else
                return BinarySearch(A, pivot+1, n-1, x);
}

int BinarySearch(int A[], int low, int high, int x)
{
        if(high >= low)
        {
                int mid = low + (high - low)/2;

                if(x == A[mid])
                        return mid;
                if(x > A[mid])
                        return BinarySearch(A, (mid + 1), high, x);
                else
                        return BinarySearch(A, low, (mid -1), x);
        }
        /*Return -1 if element is not found*/
```

```
        return -1;
}
```

Time complexity:$O(logn)$.

Problem-29 For Problem-28, can we solve in one scan?

Solution: Yes.

```
int BinarySearchRotated(int A[], int start, int finish, int data)
{
        int mid;
        if (start > finish)
                return -1;

        mid = start + (finish - start) / 2;

        if (data == A[mid])
                return mid;
        else if (A[start] <= A[mid])
        {       // start half is in sorted order.
                if (data >= A[start] && data < A[mid])
                        return BinarySearchRotated(A, start, mid - 1, data);
                else
                        return BinarySearchRotated(A, mid + 1, finish, data);
        }
        else
        {       // A[mid] <= A[finish], finish half is in sorted order.
                if (data > A[mid] && data <= A[finish])
                        return BinarySearchRotated(A, mid + 1, finish, data);
                else
                        return BinarySearchRotated(A, start, mid - 1, data);
        }
}
```

Time complexity:$O(logn)$.

Problem-30 Bitonic search.

An array is *bitonic* if it is comprised of an increasing sequence of integers followed immediately by a decreasing sequence of integers. Given a bitonic array A of n distinct integers, describe how to determine whether a given integer is in the array in $O(logn)$ steps.

Solution: This is same as Problem-25.

Problem-31 Yet, other way of asking Problem-25?
Let A[] be an array that starts out increasing, reaches a maximum, and then decreases. Design an $O(logn)$ algorithm to find the index of the maximum value.

Problem-32 Give an $O(nlogn)$ algorithm for computing the median of a sequence of n integers.

Solution: Sort and return element at $n/2$.

Problem-33 Given two sorted lists of size m and n, find the median of all elements in $O(log\ (m + n))$ time.

Solution: Refer *Divide and Conquer* chapter.

Problem-34 Give a sorted array A of n elements, possibly with duplicates, find the index of the first occurrence of a number in $O(logn)$ time.

Solution: To find the first occurrence of a number we need to check for the following condition. Return the position if any one of the following is true:

mid == low && A[mid] == data || A[mid] == data && A[mid-1] < data

```
int BinarySearchFirstOccurrence(int A[], int n, int low, int high, int data)
{
        int mid;
        if (high >= low)
        {
                mid = low + (high-low) / 2;

                if ((mid == low && A[mid] == data) || (A[mid] == data && A[mid - 1] < data))
                        return mid;

                // Give preference to left half of the array
                else if (A[mid] >= data)
                        return BinarySearchFirstOccurrence (A, n, low, mid - 1, data);
                else
                        return BinarySearchFirstOccurrence (A, n, mid + 1, high, data);
        }
        return -1;
}
```

Time Complexity: $O(logn)$.

Problem-35 Give a sorted array A of n elements, possibly with duplicates, find the index of the last occurrence of a number in $O(logn)$ time.

Solution: To find the last occurrence of a number we need to check for the following condition. Return the position if any one of the following is true:

mid == high && A[mid] == data || A[mid] == data && A[mid+1] > data

```
int BinarySearchLastOccurrence(int A[], int n, int low, int high, int data)
{
        int mid;
        if (high >= low)
        {
                mid = low + (high-low) / 2;

                if ((mid == high && A[mid] == data) || (A[mid] == data && A[mid + 1] > data))
                        return mid;

                // Give preference to right half of the array
                else if (A[mid] <= data)
                        return BinarySearchLastOccurrence (A, n, mid + 1, high, data);
                else
                        return BinarySearchLastOccurrence (A, n, low, mod - 1, data);
        }
        return -1;
}
```

Time Complexity: $O(logn)$.

Problem-36 Give a sorted array of n elements, possibly with duplicates, find the number of occurrences of a number.

Solution: Brute For Approach.

Do a linear search over the array and increment count as and when we find the element data in the array.

```
int LinearSearchCount(int A[], int n, int data)
{
        int count = 0;

        for (int i = 0; i < n; i++)
        {
                if (a[i] == k)
                        count++;
        }
        return count;
```

}

Time Complexity: $O(n)$.

Problem-37 Can we improve the time complexity of Problem-36?

Solution: Yes. We can solve this by using one binary search call followed by another small scan.

Algorithm

- Do a binary search for the $data$ in the array. Let us assume its position be K.
- Now traverse towards left from K and count the number of occurrences of $data$. Let this count be $leftCount$.
- Similarly, traverse towards right and count the number of occurrences of $data$. Let this count be $rightCount$.
- Total number of occurrences $= leftCount + 1 + rightCount$

Time Complexity – $O(logn + S)$ where S is the number of occurrences of $data$.

Problem-38 Is there any alternative way of solving the Problem-36*?*

Solution:

Algorithm

- Find the first occurrence of $data$ and call its index as $firstOccurrence$ (for algorithm refer Problem-34)
- Find the last occurrence of data and call its index as $lastOccurrence$ (for algorithm refer Problem-35)
- Return $lastOccurrence - firstOccurrence + 1$

Time Complexity $= O(logn + logn) = O(logn)$.

Problem-39 What is the next number in the sequence $1, 11, 21$ and why?

Solution: Read the given number loudly. This is just a fun problem.

One one
Two Ones
One two, one one→ 1211

So answer is, the next number is the representation of previous number by reading it loudly.

Problem-40 Finding second smallest number efficiently.

Solution: We can construct a heap of the given elements using up just less than n comparisons (Refer *Priority Queues* chapter for algorithm). Then we find the second smallest using $logn$ comparisons for the GetMax() operation. Overall, we get $n + logn + constant$.

Problem-41 Is there any other solution for Problem-40?

Solution: Alternatively, split the n numbers into groups of 2, perform $n/2$ comparisons successively to find the largest using a tournament-like method. The first round will yield the maximum in $n - 1$ comparisons. The second round will be performed on the winners of the first round and the ones the maximum popped. This will yield $logn - 1$ comparisons for a total of $n + lg\, n - 2$. The above solution is called as *tournament problem*.

Problem-42 An element is a majority if it appears more than $n/2$ times. Give and algorithm takes an array of n element as argument and identifies a majority (if it exists).

Solution: The basic solution is to have two loops and keep track of maximum count for all different elements. If maximum count becomes greater than $n/2$ then break the loops and return the element having maximum count. If maximum count doesn't become more than $n/2$ then majority element doesn't exist.

Time Complexity: $O(n^2)$.
Space Complexity: $O(1)$.

Problem-43 Can we improve the Problem-42 time complexity to $O(nlogn)$?

Solution: Using binary search we can achieve this.

Node of the Binary Search Tree (used in this approach) will be as follows.

```
struct TreeNode
{
        int element;
        int count;
        struct TreeNode *left;
        struct TreeNode *right;
}BST;
```

Insert elements in BST one by one and if an element is already present then increment the count of the node. At any stage, if count of a node becomes more than $n/2$ then return. The method works well for the cases where $n/2 + 1$ occurrences of the majority element is present in the starting of the array, for example $\{1, 1, 1, 1, 1, 2, 3, \text{and } 4\}$.

Time Complexity: If a binary search tree is used then worst time complexity will be $O(n^2)$. If a balanced-binary-search tree is used then $O(nlogn)$.

Space Complexity: $O(n)$.

Problem-44 Is there any other of achieving $O(nlogn)$ complexity for the Problem-42?

Solution: Sort the input array and scan the sorted array to find the majority element.

Time Complexity: $O(nlogn)$.
Space Complexity: $O(1)$.

Problem-45 Can we improve the complexity for the Problem-42?

Solution: If an element occurs more than $n/2$ times in A then it must be the median of A. But, the reverse is not true, so once the median is found, we must check to see how many times it occurs in A. We can use linear selection which takes $O(n)$ time (for algorithm refer *Selection Algorithms* chapter).

int CheckMajority(int A[], in n)
{
1) Use linear selection to find the median m of A.
2) Do one more pass through A and count the number of occurrences of m.
   a. If m occurs more than $n/2$ times then return true;
   b. Otherwise return false.
}

Problem-46 Is there any other way of solving the Problem-42?

Solution: Since only one element is repeating, we can use simple scan of the input array by keeping track of count for the elements. If the count is 0 then we can assume that the element is coming first time otherwise that the resultant element.

```
int MajorityNum(int[] A, int n)
{
    int majNum, count;
    element = -1; count = 0;
    for(int i = 0; i < n; i++)
    {
            // If the counter is 0 then set the current candidate to majority num and
            // we set the counter to 1.
            if(count == 0)
            {
                    element = A[i];
```

```
                count = 1;
            }
            else if(element == A[i])
            {
                // Increment counter If the counter is not 0 and
                // element is same as current candidate.
                count++;
            }
            else
            {
                // Decrement counter If the counter is not 0 and
                // element is different from current candidate.
                count--;
            }
        }
        return element;
}
```

Time Complexity: $O(n)$.
Space Complexity: $O(1)$.

Problem-47 Given an array of $2n$ elements of which n elements are same and the remaining n elements are all different. Find the majority element.

Solution: The repeated elements will occupy half the array. No matter what arrangement it is, only one of the below will be true,

- All duplicate elements will be at a relative distance of 2 from each other. Ex: n, 1, n, 100, n, 54, n ...
- At least two duplicate elements will be next to each other
  Ex: $n, n, 1, 100, n, 54, n, \ldots.$
  $n, 1, n, n, n, 54, 100 \ldots$
  $1, 100, 54, n, n, n, n \ldots.$

So, in worst case, we need will two passes over the array,

First Pass: compare $A[i]$ and $A[i+1]$
Second Pass: compare $A[i]$ and $A[i+2]$

Something will match and that's your element.

This will cost $O(n)$ in time and $O(1)$ in space.

Problem-48 Given an array with $2n + 1$ integer elements, n elements appear twice in arbitrary places in the array and a single integer appears only once somewhere inside. Find the lonely integer with $O(n)$ operations and $O(1)$ extra memory.

Solution: Since except one element all other elements are repeated. We know that $A\ XOR\ A\ = 0$. Based on this if we XOR all the input elements then we get the remaining element.

```
int solution(int* A)
{
        int i, res;
        for (i = res = 0; i < 2n+1; i++)
                res = res ^ A[i];
         return res;
}
```

Time Complexity: $O(n)$.
Space Complexity: $O(1)$.

Problem-49 Throwing eggs from an n-story building.
Suppose that we have an n story building and a set of eggs. Also assume that an egg breaks if it is thrown off floor F or higher, and will not break otherwise. Devise a strategy to determine the floor F, while breaking $O(logn)$ eggs.

Solution: Refer *Divide and Conquer* chapter.

Problem-50 Local minimum of an array.
Given an array A of n distinct integers, design an $O(logn)$ algorithm to find a *local minimum*: an index i such that $A[i-1] < A[i] < A[i+1]$.

Solution: Check the middle value $A[n/2]$, and two neighbors $A[n/2 - 1]$ and $A[n/2 + 1]$. If $A[n/2]$ is local minimum, stop; otherwise search in half with smaller neighbor.

Problem-51 Give an $n \times n$ array of elements such that each row is in ascending order and each column is in ascending order, devise an $O(n)$ algorithm to determine if a given element x in the array. You may assume all elements in the $n \times n$ array are distinct.

Solution: Let us assume that the given matrix is $A[n][n]$. Start with the last row, first column [or first row - last column]. If the element we are searching for is greater than the element at $A[1][n]$, then the column 1 can be eliminated. If the search element is less than the element at $A[1][n]$, then the last row can be completely eliminated. Now, once the first column or the last row is eliminated, now, start over the process again with left-bottom end of the remaining array. In this algorithm, there would be maximum n elements that the search element would be compared with.

Time Complexity: $O(n)$. This is because we will traverse at most 2 n points.

Space Complexity: $O(1)$.

Problem-52 Given an $n \times n$ array a of n^2 numbers, Give an $O(n)$ algorithm to find a pair of indices i and j such that $A[i][j] < A[i+1][j], A[i][j] < A[i][j+1], A[i][j] < A[i-1][j]$, and $A[i][j] < A[i][j-1]$.

Solution: This problem is same as Problem-51.

Problem-53 Given $n \times n$ matrix, and in each row all 1's are followed $0's$. Find row with maximum number of $0's$.

Solution: Start with first row, last column. If the element is 0 then move to the previous column in the same row and at the same time increase the counter to indicate the maximum number of $0's$. If the element is 1 then move to the next row in the same column. Repeat this process until we reach last row, first column.

Time Complexity: $O(2n) \approx O(n)$ (very much similar to Problem-51).

Problem-54 Given an input array of size unknown with all numbers in the beginning and special symbols in the end. Find the index in the array from where special symbols start.

Solution: Refer *Divide and Conquer* chapter.

Problem-55 Finding the Missing Number

We are given a list of $n-1$ integers and these integers are in the range of 1 to n. There are no duplicates in list. One of the integers is missing in the list. Given an algorithm to find the missing integer.

Example:

I/P $[1, 2, 4, 6, 3, 7, 8]$

O/P 5

Solution: Use sum formula

1) Get the sum of numbers, $sum = n * (n+1)/2$
2) Subtract all the numbers from sum and you will get the missing number.

Time Complexity: $O(n)$, this is because we need to scan the complete array.

Problem-56 In Problem-55, if the sum of the numbers goes beyond maximum allowed integer, then there can be integer overflow and we may not get correct answer. Can we solve this problem?

Solution:

1) XOR all the array elements, let the result of XOR be X.
2) XOR all numbers from 1 to n, let XOR be Y.
3) XOR of X and Y gives the missing number.

```
int FindMissingNumber(int A[], int n)
{
        int i, X, Y;
        for (i = 0; i < 9; i ++)
                X ^= A[i];
        for (i = 1; i <= 10; i ++)
                Y ^= i;
        //In fact, one variable is enough.
        return X ^ Y;
}
```

Time Complexity: $O(n)$, this is because we need to scan the complete array.

Problem-57 Find the Number Occurring Odd Number of Times

Given an array of positive integers, all numbers occurs even number of times except one number which occurs odd number of times. Find the number in $O(n)$ time & constant space.

Example:

I/P = $[1, 2, 3, 2, 3, 1, 3]$

O/P = 3

Solution: Do a bitwise XOR of all the elements. Finally we get the number which has odd occurrences. This is because of the fact that, $A\ XOR\ A = 0$.

Time Complexity: $O(n)$.

Problem-58 Find the two repeating elements in a given array

Given an array with $n + 2$ elements, all elements of the array are in range 1 to n and also all elements occur only once except two numbers which occur twice. Find those two repeating numbers.

Example: $6, 2, 6, 5, 2, 3, 1$ and $n = 5$

The above input has $n + 2 = 7$ elements with all elements occurring once except 2 and 6 which occur twice. So the output should be 6 2.

Solution: One simple way to scan the complete array for each element of the input elements. That means use two loops. In the outer loop, select elements one by one and count the number of occurrences of the selected element in the inner loop.

```
void PrintRepeatedElements(int A[], int n)
{
        int i, j;
        for(i = 0; i < n; i++)
                for(j = i+1; j <n; j++)
                        if(A[i] == A[j])
                                printf("%d", A[i]);
}
```

Time Complexity: $O(n^2)$.
Space Complexity: $O(1)$.

Problem-59 For the Problem-58, can we improve the time complexity?

Solution: Sort the array using any comparison sorting algorithm and see if there are any elements which contiguous with same value.

Time Complexity: $O(nlogn)$.
Space Complexity: $O(1)$.

Problem-60 For the Problem-58, can we improve the time complexity?

Solution: Use Count Array. This solution is like using a hash table. But for simplicity we can use array for storing the counts. Traverse the array once. While traversing, keep track of count of all elements in the array using a temp array $count[]$ of size n, when we see an element whose count is already set, print it as duplicate.

```
void PrintRepeatedElements(int A[], int n)
{
        int *count = (int *)calloc(sizeof(int), (n - 2));
        for(int i = 0; i < size; i++)
        {
                if(count[A[i]] == 1)
                        printf("%d", A[i]);
                else
                        count[A[i]]++;
        }
}
```

Time Complexity: $O(n)$.
Space Complexity: $O(n)$.

Problem-61 Consider the Problem-58. Let us assume that the numbers are in the range 1 to n. Is there any other way of solving the problem?

Solution: Using XOR Operation. Let the repeating numbers be X and Y, if we xor all the elements in the array and all integers from 1 to n, then the result is $X\ XOR\ Y$.

The 1's in binary representation of $X\ XOR\ Y$ is corresponding to the different bits between X and Y. Suppose that the k^{th} bit of $X\ XOR\ Y$ is 1, we can XOR all the elements in the array and all integers from 1 to n, whose k^{th} bits are 1. The result will be one of X and Y.

```
void PrintRepeatedElements (int A[], int size)
{
        int XOR = A[0];
        int right_most_set_bit_no;
        int n = size - 2;
        int X= 0, Y = 0;

        /* Compute XOR of all elements in A[]*/
        for(int i = 0; i < n; i++)
                XOR ^= A[i];

        /* Compute XOR of all elements {1, 2 ..n} */
        for(i = 1; i <= n; i++)
                XOR ^= i;

         /* Get the rightmost set bit in right_most_set_bit_no */
         right_most_set_bit_no = XOR & ~( XOR -1);

        /* Now divide elements in two sets by comparing rightmost set */
        for(i = 0; i < n; i++)
        {
                if(A[i] & right_most_set_bit_no)
                        X = X^ A[i]; /*XOR of first set in A[] */
                else
                        Y = Y ^ A[i]; /*XOR of second set inA[] */
        }
        for(i = 1; i <= n; i++)
        {
                if(i & right_most_set_bit_no)
                        X = X ^ i; /*XOR of first set in A[] and {1, 2, ...n }*/
                else
                        Y = Y ^ i; /*XOR of second set in A[] and {1, 2, ...n } */
        }

        printf("%d and %d",X, Y);
}
```

Time Complexity: $O(n)$.
Space Complexity: $O(1)$.

Problem-62 Consider the Problem-58. Let us assume that the numbers are in the range 1 to n. Is there yet other way of solving the problem?

Solution: We can solve this by creating two simple mathematical equations. Let us assume that two numbers which we are going to find are X and Y. We know the sum of n numbers is $n(n+1)/2$ and product is $n!$. Make two equations using these sum and product formulae, and get values of two unknowns using the two equations.

Let summation of all numbers in array be S and product be P and the numbers which are being repeated are X and Y.

$X + Y = S - n(n+1)/2$
$XY = P/n!$

Using above two equations, we can find out X and Y.
There can be addition and multiplication overflow problem with this approach.

Time Complexity: $O(n)$.
Space Complexity: $O(1)$.

Problem-63 Similar to Problem-58. Let us assume that the numbers are in the range 1 to n. Also, $n-2$ elements are repeating thrice and remaining two elements are repeating twice. Find the element which is repeating twice.

Solution: If we xor all the elements in the array and all integers from 1 to n, then the all the elements which are trice will become zero This is because, since the element is repeating trice and XOR with another time from range makes that element appearing four times. As a result, output of a $XOR\ a\ XOR\ a\ XOR\ a = 0$. Same is case with all elements which repeated thrice.

With the same logic, for the element which repeated twice, if we XOR the input elements and also the range, then the total number of appearances for that element is 3. As a result, output of $a\ XOR\ a\ XOR\ a = a$. Finally, we get the element which repeated twice.

Time Complexity: $O(n)$.
Space Complexity: $O(1)$.

Problem-64 Separate Even and Odd numbers

Given an array $A[\ ]$, write a function that segregates even and odd numbers. The functions should put all even numbers first, and then odd numbers.

Example:
Input = {12, 34, 45, 9, 8, 90, 3}
Output = {12, 34, 90, 8, 9, 45, 3}

In the output, order of numbers can be changed, i.e., in the above example 34 can come before 12 and 3 can come before 9.

Solution: The problem is very similar to *Separate 0's and 1's (*Problem-65*)* in an array, and both of these problems are variation of famous *Dutch national flag problem.*

Algorithm: Logic is little similar to Quick sort.

1) Initialize two index variables left and right: $left = 0$, $right = n - 1$
2) Keep incrementing left index until we see an odd number.
3) Keep decrementing right index until we see an even number.
4) If $left < right$ then swap $A[left]$ and $A[right]$

Implementation:

```
void DutchNationalFlag(int A[], int n)
{
        /* Initialize left and right indexes */
        int left = 0, right = n-1;
        while(left < right)
        {
                /* Increment left index while we see 0 at left */
                while(A[left]%2 == 0 && left < right)
                        left++;

                /* Decrement right index while we see 1 at right */
                while(A[right]%2 == 1 && left < right)
                        right--;

                if(left < right)
                {
                        /* Swap A[left] and A[right]*/
                        swap(&A[left], &A[right]);
                        left++;
                        right--;
                }
        }
}
```

Time Complexity: $O(n)$.

Problem-65 Other way of asking Problem-64 but with little difference.

Separate 0's and 1's in an array

We are given an array of 0's and 1's in random order. Separate 0's on left side and 1's on right side of the array. Traverse array only once.

Input array = [0, 1, 0, 1, 0, 0, 1, 1, 1, 0]
Output array = [0, 0, 0, 0, 0, 1, 1, 1, 1, 1]

Solution: Counting 0's or 1's

1. Count the number of 0's. Let count be C.
2. Once we have count, we can put C 0's at the beginning and 1's at the remaining $n - C$ positions in array.

Time Complexity: $O(n)$. This solution scans the array two times.

Problem-66 Can we solve the Problem-65 in once scan?

Solution: Yes. Use two indexes to traverse:

Maintain two indexes. Initialize first index left as 0 and second index right as $n - 1$.

Do following while $left < right$:

1) Keep incrementing index left while there are 0s at it
2) Keep decrementing index right while there are 1s at it
3) If left < right then exchange $A[left]$ and $A[right]$

```
/*Function to put all 0s on left and all 1s on right*/
void Separate0and1(int A[], int n)
{
        /* Initialize left and right indexes */
        int left = 0, right = n-1;
        while(left < right)
        {
                /* Increment left index while we see 0 at left */
                while(A[left] == 0 && left < right)
                        left++;

                /* Decrement right index while we see 1 at right */
                while(A[right] == 1 && left < right)
                        right--;

                /* If left is smaller than right then there is a 1 at left
                and a 0 at right.  Swap A[left] and A[right]*/
                if(left < right)
                {
```

```
                    A[left] = 0;
                    A[right] = 1;
                    left++;
                    right–;
                }
        }
}
```

Time Complexity: $O(n)$.
Space Complexity: $O(1)$.

Problem-67 Maximum difference between two elements

Given an array $A[]$ of integers, find out the difference between any two elements such that larger element appears after the smaller number in $A[]$.

Examples: If array is $[2, 3, 10, 6, 4, 8, 1]$ then returned value should be 8 (Diff between 10 and 2). If array is $[7, 9, 5, 6, 3, 2]$ then returned value should be 2 (Difference between 7 and 9)

Solution: Refer *Divide and Conquer* chapter.

Problem-68 Given an array of 101 elements. Out of them 25 elements are repeated twice, 12 elements are repeated 4 times and one element is repeated 3 times. Find the element which repeated 3 times in $O(1)$.

Solution: Before solving this problem let us consider the following XOR operation property.

$$a\ XOR\ a = 0$$

That means, if we apply the XOR on same elements then the result is 0. Let us apply this logic for this problem.

Algorithm:

- XOR all the elements of the given array and assume the result is A.
- After this operation, 2 occurrences of number which appeared 3 times becomes 0 and one occurrence will remain.
- The 12 elements which are appearing 4 times become 0.
- The 25 elements which are appearing 2 times become 0.

So just $XOR'ing$ all the elements give the result.

Time Complexity: $O(n)$, because we are doing only once scan.
Space Complexity: $O(1)$.

Problem-69 Given an array A of n numbers. Find all pairs of X and Y in the array such that $K = X * Y$. Give an efficient algorithm without sorting.

Solution: Create a hash table from the numbers that divide K. Divide K by a number and check for quotient in the table.

Problem-70 Given a number n, give an algorithm for finding the number of trailing zeros in $n!$.

Solution:

```
int NumberOfTrailingZerosInNumber(int n)
{
        int i, count = 0;
        if (n < 0)
                return -1;
        for (i = 5; n / i > 0; i *= 5)
                count += n / i;
        return count;
}
```

Time Complexity: $O(logn)$,