Dedicated
to
My Grandmother
Late Smt Gauri Devi Agarwal

Contents

CHAPTER 1 ■ INTRODUCTION TO ALGORITHMS 1

1.1 Introduction 1
1.2 Why study algorithms ? 1
1.3 Definitions 2
1.4 Algorithms vs. Programs 3
1.5 Algorithm Design Techniques 4
1.6 Algorithm Classification 5
1.7 Algorithm Analysis 7
1.8 Formal and Informal Algorithm Analysis 10
1.9 How to Calculate Running Time of an Algorithm ? 11
1.10 Loop Invariants 11

CHAPTER 2 ■ GROWTH OF FUNCTIONS 13

2.1 Complexity of Algorithms 14
2.2 RAM Machine 14
2.3 Best, Worst and Average - Case Complexity 15
2.4 Growth Rate of Functions 17
2.5 Asymptotic Analysis 17
2.6 Analysing Algorithm Control Structures 23
2.7 Logarithms and Exponents 32

CHAPTER 3 ■ RECURRENCES 35

3.1 Introduction 35
3.2 The Substitution Method 36
3.3 The Iteration Method 38
3.4 Master Method 42

CHAPTER 4 ■ ANALYSIS OF SIMPLE SORTING ALGORITHMS 55

4.1 Bubble Sort 56
4.2 Selection Sort 59
4.3 Insertion Sort 62

CHAPTER 5 ▪ MERGE SORT 65

5.1 Introduction 65
5.2 Analysis of Merge Sort 68
5.3 Insertion Sort and Merge Sort 69

CHAPTER 6 ▪ HEAP SORT 73

6.1 Binary Heap 73
6.2 Heap Property 74
6.3 Height of a Heap 75
6.4 Heapify 77
6.5 Building a Heap 80
6.6 Heap Sort Algorithm 80
6.7 Heap-Insert 87
6.8 Heap-Delete 88
6.9 Heap-Extract-Max 88

CHAPTER 7 ▪ QUICK SORT 91

7.1 Introduction 91
7.2 Partitioning the Array 92
7.3 Performance of Quick Sort 94
7.4 Versions of Quick Sort 98

CHAPTER 8 ▪ SORTING IN LINEAR TIME 103

8.1 Stability 104
8.2 Counting Sort 104
8.3 Radix Sort 106
8.4 Bucket Sort 107

CHAPTER 9 ▪ MEDIANS AND ORDER STATISTICS 111

9.1 Selection Problem 111
9.2 Finding Minimum (or Maximum) 111
9.3 Finding Minimum & Maximum Simultaneously 112
9.4 Selection of *i*th-order Statistic in Linear Time 112
9.5 Worst-Case Linear-Time Order Statistics 114
9.6 Choosing the Pivot 115

CHAPTER 10 ▪ DICTIONARIES AND HASH TABLES 117

10.1 Dictionaries 117
10.2 Log Files 118
10.3 Hash Tables 118
10.4 Hashing 120
10.5 Hash Functions 123
10.6 Universal Hashing 125
10.7 Hashing With Open Addressing 125
10.8 Rehashing 132

CHAPTER 11 ▪ ELEMENTARY DATA STRUCTURES 135

11.1 Introduction 135
11.2 Abstract Data Type 136
11.3 Stack 138
11.4 Queue 141
11.5 Linked-List 144
11.6 Binary Tree 145

CHAPTER 12 ▪ BINARY SEARCH TREE 147

12.1 Binary-Search-Tree Property 147
12.2 Binary-Search-Tree Property vs Heap Property 148
12.3 Querying a Binary Search Tree 149

CHAPTER 13 ▪ AVL TREE 155

13.1 Introduction 155
13.2 Balance Factor 156
13.3 Insertion in an AVL Search Tree 156
13.4 Efficiency of AVL trees 164

CHAPTER 14 ▪ SPLAY TREES 165

14.1 Introduction 165
14.2 Splaying 166
14.3 Splay vs. Move-to-root 168

CHAPTER 15 ■ RED-BLACK TREES 171

15.1 Introduction 171
15.2 Operations on RB Trees 173
15.3 Elementary Properties of Red-Black Tree 181

CHAPTER 16 ■ AUGMENTING DATA STRUCTURES 187

16.1 Augmenting a red-black tree 187
16.2 Retrieving an element with a given rank 188
16.3 Determining the rank of an element 189
16.4 Data Structure Maintenance 189
16.5 An Augmentation Strategy 190
16.6 Interval trees 191

CHAPTER 17 ■ B-TREES 195

17.1 Introduction 195
17.2 Definition of *B*-tree 197
17.3 Searching for a Key *k* in a *B*-tree 198
17.4 Creating an Empty *B*-Tree 199
17.5 How do we Search for a Predecessor ? 199
17.6 Inserting a Key into a *B*-tree 200

CHAPTER 18 ■ BINOMIAL HEAPS 213

18.1 Mergeable Heaps 213
18.2 Binomial Trees and Binomial Heaps 214
18.3 Binomial Heaps 215

CHAPTER 19 ■ FIBONACCI HEAPS 229

19.1 Introduction 229
19.2 Structure of Fibonacci Heaps 230
19.3 Potential function 231
19.4 Operations 232
19.5 Bounding the Maximum Degree 242

CHAPTER 20 ■ DATA STRUCTURES FOR DISJOINT SETS 245

20.1 Introduction 245
20.2 Disjoint-Set Operations 245
20.3 UNION-FIND Algorithm 246
20.4 Applications of Disjoint-set Data Structures 246
20.5 Linked-list Representation of Disjoint Sets 247
20.6 Disjoint-Set Forests 248
20.7 Properties of Ranks 250

CHAPTER 21 ■ DYNAMIC PROGRAMMING 253

21.1 Introduction 253
21.2 Common Characteristics 255
21.3 Matrix Multiplication 255
21.4 Memoization 261
21.5 Longest common subsequence (LCS) 262

CHAPTER 22 ■ GREEDY ALGORITHMS 271

22.1 Introduction 271
22.2 An Activity-Selection Problem 271
22.3 Knapsack Problems 274
22.4 Huffman Codes 277
22.5 Prefix Codes 278
22.6 Greedy Algorithm for Constructing a Huffman Code 279
22.7 Activity or Task Scheduling Problem 281
22.7 Travelling Sales Person Problem 283
22.9 Matroids 286
22.10 Minimum Spanning Tree 287

CHAPTER 23 ■ BACKTRACKING 289

23.1 Introduction 289
23.2 Recursive Maze Algorithm 291
23.3 Hamiltonian Circuit Problem 292
23.4 Subset-sum Problem 296
23.5 *N*-Queens Problem 297

CHAPTER 24 ▪ BRANCH AND BOUND 301
24.1 Introduction 301
24.2 Live Node, Dead Node and Bounding Functions 303
24.3 FIFO Branch-and-Bound Algorithm 304
24.4 Least Cost (LC) Search 305
24.5 The 15-Puzzle : An Example 306
24.6 Branch-and-Bound Algorithm for TSP 308
24.7 Knapsack Problem 308A
24.8 Assignment Problem 308B

CHAPTER 25 ▪ AMORTIZED ANALYSIS 309
25.1 Introduction 309
25.2 Aggregate Analysis 310
25.3 Accounting Method or Taxation Method 311
25.4 Potential Method 312

CHAPTER 26 ▪ ELEMENTARY GRAPHS ALGORITHMS 315
26.1 Introduction 315
26.2 How is a Graph Represented ? 316
26.3 Breadth First Search 318
26.4 Depth First Search 323
26.6 Topological Sort 328
26.7 Strongly Connected Components 329

CHAPTER 27 ▪ MINIMUM SPANNING TREE 333
27.1 Spanning Tree 333
27.2 Kruskal's Algorithm 334
27.3 Prim's Algorithm 337

CHAPTER 28 ▪ SINGLE-SOURCE SHORTEST PATHS 343
28.1 Introduction 343
28.2 Shortest Path : Existence 344
28.3 Representing Shortest Paths 345
28.4 Shortest Path : Properties 345
28.5 Dijkstra's Algorithm 346
28.6 The Bellman-Ford Algorithm 350
28.7 Single-source shortest paths in directed acyclic graphs 353

CHAPTER 29 ▪ ALL-PAIRS SHORTEST PATHS 357
29.1 Introduction 357
29.2 Matrix Multiplication 358
29.3 The Floyd-Warshall Algorithm 358
29.4 Transitive Closure 364
29.5 Johnson's Algorithm 365

CHAPTER 30 ▪ MAXIMUM FLOW 367
30.1 Flow Networks and Flows 367
30.2 Network Flow Problems 370
30.3 Residual Networks, Augmenting Paths, and Cuts 371
30.4 Max-flow min-cut Theorem 374
30.5 Ford-Fulkerson Algorithm 375

CHAPTER 31 ▪ SORTING NETWORKS 381
31.1 Comparison Networks 381
31.2 Bitonic Sorting Network 384
31.3 Merging Network 385

CHAPTER 32 ▪ ALGORITHMS FOR PARALLEL COMPUTERS 387
32.1 Parallel Computing 387
32.2 Why Use Parallel Computing ? 389
32.3 von Neumann Architecture 389
32.4 Flynn's Classical Taxonomy 390
32.5 Parallel Algorithms 393
32.6 Concurrent Versus Exclusive Memory Accesses 393
32.7 List Ranking by Pointer Jumping 394
32.8 The Euler-Tour Technique 395
32.9 CRCW Algorithms Versus EREW Algorithms 397
32.10 Brent's Theorem 398

CHAPTER 33 ▪ MATRIX OPERATIONS 401
33.1 Introduction 401
33.2 Operations on Matrices 403

33.3 Strassen's Algorithm for Matrix Multiplication 408
33.4 Solving Systems of Linear Equations 410
33.5 Computing an LU decomposition 411
33.6 Computing an LUP decomposition 413

CHAPTER 34 ■ NUMBER-THEORETIC ALGORITHMS 417

34.1 Some Facts From Elementary Number Theory 417
34.2 Euclid's GCD Algorithm 419
34.3 The Chinese Remainder Theorem 421
34.4 The RSA Public-key Cryptosystem 422

CHAPTER 35 ■ POLYNOMIALS AND THE FFT 425

35.1 Polynomial 425
35.2 Representation of Polynomials 425
35.3 Evaluating Polynomial Functions 426
35.4 Complex Roots of Unity 427
35.5 Discrete Fourier Transform 429
35.6 Fast Fourier Transform (FFT) 429

CHAPTER 36 ■ STRING MATCHING 433

36.1 Introduction 433
36.2 The Naive String-matching Algorithm 434
36.3 The Rabin-Karp Algorithm 435
36.4 String Matching with Finite Automata 438
36.5 The Knuth-Morris-Pratt (KMP) Algorithm 439
36.6 The Boyer-Moore Algorithm 441

CHAPTER 37 ■ COMPUTATIONAL GEOMETRY 445

37.1 Introduction 445
37.2 Strengths and Limitations of Computational Geometry 446
37.3 Polygons 446
37.4 Convex Polygon 447
37.5 Convex Hulls 448
37.6 Graham's Scan Algorithm 449
37.7 Jarvis's March Algorithm 456

CHAPTER 38 ■ NP-COMPLETENESS 459

38.1 Classes of Problems 459
38.2 The Class NP (Non Deterministic Polynomial) 461
38.3 *NP*-Completeness 462
38.4 The Class co-NP 464
38.5 Satisfiability (SAT) 465
38.6 Circuit Satisfiability 465
38.7 Proving NP-Completeness 467
38.8 Techniques for NP-complete Problem 468
38.9 (Formula) Satisfiability 469
38.10 3-CNF Satisfiability 471
38.11 The Clique Problem 472
38.12 The Vertex-cover Problem 474
38.13 The Subset-Sum Problem 474
38.14 The Hamiltonian-Cycle Problem 474
38.15 The traveling-Salesman Problem 475

CHAPTER 39 ■ APPROXIMATE ALGORITHMS 477

39.1 Introduction 477
39.2 Performance Ratios 478
39.3 Examples of Approximate Algorithms 479

CHAPTER 40 ■ RANDOMIZED ALGORITHM 483

40.1 Introduction 483
40.2 Applications 484
40.3 Advantages and Disadvantages 487

BIBLIOGRAPHY 489

CHAPTER

1

Introduction to Algorithms

1.1 Introduction

A common man thinks that a computer can do anything and everything and it is very difficult to make people understand that it is not really the computer but the man behind computer who does the whole thing.

In the modern world man feels just by entering what he wants to search into the computers he can get information as desired by him. He believes that, the computer does this. A common man rarely understands that a man made procedure called **search** has done the entire job and the only support provided by the computer is the execution speed and organized storage of information.

In the above instance, a designer of the information system should know what one frequently searches for. He should make a structured organization of all those details to store in memory of the computer. Based on the requirement, the right information is brought out. This is accomplished through a set of instructions created by the designer of the information system to search the right information matching the requirement of the user. This set of instructions is termed as **program**.

1.2 Why Study Algorithms ?

As the speed of processors increases, performance is frequently said to be less central than the other software quality characteristics (*e.g.*, security, extensibility, re-usability, etc.) However, large problem sizes are common place in the area of computational science; which makes performance a very important factor. This is because longer computation time, to name a few,

mean slower results, less through research, and higher cost of computation (if buying CPU hours from an external party). The study of algorithms, therefore, gives us a language to express performance as a function of problem size.

1.3 Definitions

The name 'Algorithm' is given after the name of **Abu Ja'far Muhammad ibn Musa Al-Khwarizmi**, Ninth Century, is defined as follows :

- An algorithm is a set of rules for carrying out calculation either by hand or on a machine.
- An algorithm is a sequence of computational steps that transform the input into the output.
- An algorithm is a sequence of operations performed on data that have to be organized in data structures.
- A finite set of instruction that specify a sequence of operations to be carried out in order to solve a specific problem or class of problems is called an algorithm.
- An algorithm is an abstraction of a program to be executed on a physical machine (model of computation).

[illegible] means "restoring" referring to the process of moving a subtracted quantity to the other side of an equation, or multiplying two terms to subtracting equal quantities from both sides of an equation.

An algorithm must have the following properties :

- **Finiteness.** Algorithm must complete after a finite number of instructions have been executed.
- **Absence of ambiguity.** Each step must be clearly defined, having only one interpretation.
- **Definition of sequence.** Each step must have a unique defined preceding and succeeding step. The first step (start step) and last step (halt step) must be clearly noted.
- **Input/output.** Number and types of required inputs and results must be specified.
- **Feasibility.** It must be possible to perform each instruction.

Let us try to present the scenario of a man brushing his own teeth as an algorithm as follows :

Step 1. Take the brush
Step 2. Put the paste on it
Step 3. Start brushing
Step 4. Clean
Step 5. Wash
Step 6. Stop

If one goes through these 6 steps without being aware of the statement of the problem, he could possibly feel that this is the algorithm for cleaning a toilet. This is because of several ambiguities while comprehending every step. The step 1 may imply toothbrush, paintbrush, toilet brush etc. Such an ambiguity doesn't an instruction an algorithmic step. Thus every step should be made unambiguous. An unambiguous step is called '**definite instruction**'. Even if the step 2 is rewritten as apply the toothpaste, to eliminate ambiguities yet the conflicts such as, where to apply the toothpaste and where is the source of the toothpaste, need to be resolved. Hence, the act of

applying the toothpaste is not mentioned. Although unambiguous, such unrealizable steps can't be included as algorithmic instruction as they are not effective.

The definiteness and effectiveness of an instruction implies the successful termination of that instruction. However the above two may not be sufficient to guarantee the termination of the algorithm. Therefore, while designing an algorithm care should be taken to provide a proper termination for algorithm.

Characteristics of an Algorithm

Every algorithm should have the following *five* characteristic features :

1. Input 2. Output 3. Definiteness 4. Effectiveness 5. Termination

Characteristics (1) and (2) require that an algorithm produces one or more outputs and have zero or more inputs that are externally supplied.

According to characteristic (3) each operation must be definite meaning that it must be perfectly clear what should be done. Instructions such as "compute x/o" or "subtract 7 or 6 to x" are not permitted because it is not clear which of the two possibilities should be done or what the result is. That is definiteness means each instruction is clear and unambiguous. Characteristic (4) requires that each operation be *effective* that is each step must be such that it can be done by a person using pencil and paper in a finite amount of time. The (5) characteristic for algorithm is that it terminates after a finite number of operations. A related consideration is that the time for termination should be reasonably short.

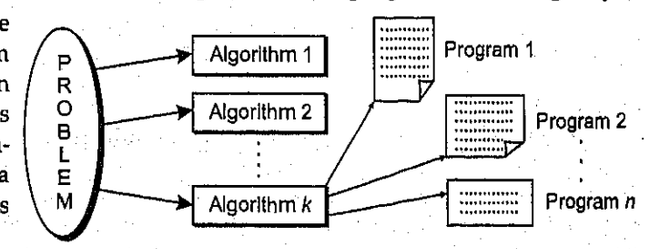
Therefore, an **algorithm can be defined as a sequence of definite and effective instructions, while terminates with the production of correct output from the given input.**

In other words, viewed little more formally, an algorithm is a step-by-step formalization of a mapping function to map input set onto an output set.

1.4 Algorithms vs. Programs

In computational theory, we distinguish between an algorithm and a program. Program does not have to satisfy the finiteness condition. For example, we can think of an operating system that continues in a "wait" loop until more jobs are entered. Such a program does not terminate unless the system crashes. Since our programs always terminate, we use "algorithm" and "program" interchangeably.

Given a problem to solve, the **design phase** produces an algorithm and the **implementation phase** then produces a program that expresses the designed algorithm. So, the concrete expression of an algorithm in a particular programming language is called a program.



Example :

Problem *Finding the largest value number among n numbers.*

Input The value of n and n numbers.

Output The largest value number.

Steps

1. Let the value of the first be the largest value denoted by BIG
2. Let R denote the number of remaining number. $R = n - 1$
3. If $R! = 0$ then it is implied that the list is still not exhausted. Therefore look the next number called NEW.
4. Now R becomes $R - 1$
5. If NEW is greater than BIG then replace BIG by the value of NEW
6. Repeat steps 3 to 5 until R becomes zero.
7. Print BIG
8. Stop

End of algorithm

1.5 Algorithm Design Techniques

For a given problem, there are many ways to design algorithms for it, (*e.g.*, Insertion sort is an incremental approach). The following is a list of several popular design approaches.

- Divide-and-Conquer (*D* and *C*)
- Greedy Approach
- Dynamic Programming
- Branch-and-Bound
- Randomized Algorithms
- Backtracking Algorithms

Divide and Conquer

- Divide the original problem into a set of sub problems
- Solve every sub problem individually, recursively
- Combine the solutions of the sub problems (top level) into a solution of the whole original problem.

Greedy approach

Greedy algorithms seek to optimize a function by making choices (greedy criterion) which are the best locally but do not look at the global problem. The result is a good solution but not necessarily the best one. The greedy algorithm does not always guarantee the optimal solution however it generally produces solutions that are very close in value to the optimal.

Dynamic Programming

Dynamic programming is a technique for efficiently computing recurrences by storing partial results. It is a method of solving problems exhibiting the properties of overlapping subproblems and optimal substructure that takes much less time than naive methods.

Branch-and-Bound

In a branch and bound algorithm a given sub problem, which cannot be bounded, has to be divided into at least two new restricted sub problems. Branch and bound algorithms are methods for global optimization in nonconvex problems. Branch and bound algorithms can be (and often are) slow, however, in the worst case they require effort that grows exponentially with problem size, but in some cases we are lucky, and the methods converge with much less effort.

Randomized Algorithms

A randomized algorithm is defined as an algorithm that is allowed to access a source of independent, unbiased random bits, and it is then allowed to use these random bits to influence its computation.

Backtracking Algorithms

Backtracking algorithms try each possibility until they find the right one. It is a depth-first search of the set of possible solutions. During the search, if an alternative doesn't work, the search backtracks to the choice point, the place which presented different alternatives, and tries the next alterative. When the alternatives are exhausted, the search returns to the previous choice point and try the next alternative there. It there are no more choice points, the search fails.

1.6 Algorithm Classification

Algorithms that use a similar problem-solving approach can be grouped together. This classification is neither exhaustive nor disjoint. Algorithm types we will consider include :

- Recursive algorithms
- Backtracking algorithms
- Divide and conquer algorithms
- Dynamic programming algorithms
- Greedy algorithms
- Branch and bound algorithms
- Brute force algorithms
- Randomized algorithms

1.6.1 Recursive Algorithms

A recursive algorithm :

- Solves the base cases directly
- Recurs with a simpler sub problem
- Does some extra work to convert the solution to the simpler sub problem into a solution to the given problem.
- *Examples* :
  - To count the number of elements in a list.
  - To test if a value occurs in a list.

1.6.2 Backtracking Algorithms

Backtracking algorithms are based on a depth-first recursive search. A backtracking algorithm :

- Tests to see if a solution has been found, and if so, returns it ; otherwise
- For each choice that can be made at this point,
  1. Make that choice 2. Recur 3. If the recursion returns a solution, return it
- If no choices remain, return failure

Example, To color a map with no more than four colors :

Color (Country *n*) :

```
If all countries have been colored (n > number of countries) return success ; otherwise,
    For each color c  four colors,
        If country n is not adjacent to country that has been colored c
            Color country n with color c
            recursively color country n + 1
If successful, return success
If loop exists, return failure
```

1.6.3 Divide and Conquer

A divide and conquer algorithm consists of *two* parts :

1. Divide the problem into smaller sub problems of the same type, and solve these sub problems recursively.
2. Combine the solutions to the sub problems into a solution to the original problem.

Traditionally, an algorithm is only called "divide and conquer" if it contains at least two recursive calls. *Example,* Quick sort, Merge sort.

1.6.4 Dynamic Programming Algorithms

A dynamic programming algorithm remembers past results and uses them to find new results. Dynamic programming is generally used for optimization problems. In dynamic programming multiple solutions exist, need to find the "best" one. It "requires substructure" and "overlapping sub problems". Optimal substructure means optimal solution contains solutions to sub problems. Overlapping sub problems means solutions to sub problems can be stored and reused in a bottom-up fashion.

This differs from Divide and Conquer, where sub problems generally need not overlap.

1.6.5 Greedy Algorithm

An optimization problem is one in which we want to find, not just a solution, but the best solution. A "greedy algorithm" sometimes works well for optimization problems. A greedy algorithm works in phases :

At each phase :

- We take the best we can get right now, without regard for future consequences.
- We hope that by choosing a *local* optimum at each step, we will end up at a *global* optimum

1.6.6 Branch and Bound Algorithms

Branch and bound algorithms are generally used for optimization problem. As the algorithm progresses, a tree of subproblems is formed. The original problem is considered the "root problem".

A method is used to construct an upper and lower bound for a given problem.

- At each node, apply the bounding methods
- If the bounds match, it is deemed a feasible solution to that particular sub problem.
- If bounds do not match, partition the problem represented by that node, and make the two sub problems into children nodes.
- Continue, using the best known feasible solution to trim sections of the tree, until all nodes have been solved or trimmed.

1.6.7 Brute Force Algorithm

A brute force algorithm simply tries *all* possibilities until a satisfactory solution is found. Such an algorithm can be :

- **Optimizing.** Find the best solution. This may require finding all solution, or if a value for the best solution is known, it may stop when any best solution is found. Example : Finding the best path for a travelling salesman.
- **Satisfying.** Stop as soon as a solution is found that is *good enough*. Example, finding a travelling salesman path that is within 10% of optimal.

1.6.8 Randomized Algorithms

A randomized algorithm uses a random number at least once during the computation to make a decision

- *Example,* In Quick sort, using a random number to choose a pivot
- *Example,* Trying to factor a large number by choosing random numbers as possible divisors.

1.7 Algorithm Analysis

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

The purpose of analysis of algorithms is not to give a formula that will tell us exactly how many seconds or computer cycles a particular algorithm will take. This is not useful information because we would then need to talk about the type of computer, whether it has one or many users at a time, what processor it has, how fast its clock is, whether it has a complex or reduced instruction set processor chip, and how well the compiler optimizes the executable code.

All of those will have an impact on how fast a program for an algorithm will run. To talk about analysis in those terms would mean that by moving a program to a faster computer, the algorithm would become better because it now completes its job faster. That's not true, so, we do our analysis without regard to any specific computer.

In the case of a small or simple routine it might be possible to count the exact number of operations performed as a function of N. Most of the time, however, this will not be useful. In fact, we will see that the difference between an algorithm that does $N + 5$ operations and one that does $N + 250$ operations becomes meaningless as N gets very large.

Another reason we do not try to count every operation that is performed by an algorithm is that we could fine-tune an algorithm extensively but not really make much of a difference in its overall performance. For instance, let's say that we have an algorithm that counts the number of different characters in a file. An algorithm for that might look like the following :

```
for all 256 characters do
    assign zero to the counter
end for
while there are more characters in the file do
    get the next character
    increment the counter for this character by one
end while
```

When we look at this algorithm, we see that there are 256 passes for the initialization loop. If there are N characters in the input file, there are N passes for the second loop. So the question becomes what do we count ? In a for loop, we have the initialization of the loop variable and then

for each pass of the loop, a check that the loop variable is within the bounds, the execution of the loop, and the increment of the loop variable. This means that the initialization loop does a set of 257 assignments (1 for the loop variable and 256 for the counters), 256 increments of the loop variable, and 257 checks that this variable is within the loop bounds (the extra one is when the loop stops). For the second loop, we will need to do check of the condition $N+1$ times (the +1 is for the last check when the file is empty), and we will increment N counters. The total number of operations is

- Increment $N+256$
- Assignments 257
- Conditional checks $N+258$

So, if we have 500 characters in the file, the algorithm will do a total of 1771 operations, of which 770 are associated with the initialization (43%). Now consider what happens as the value of N gets large. If we have a file with 50,000 characters, the algorithm will do a total of 100,771 operations, of which there are still only 770 associated with the initialization (less than 1% of the total work). The number of initialization operations has not changed, but they become a much smaller percentage of the total as N increases. Let's look at this way. Computer organization information shows that copying large blocks of data is as quick as an assignment. We could initialize the first 16 counters to zero and then copy this block 15 times to fill in the rest of the counters. This would mean a reduction in the initialization pass down to 33 conditional checks, 33 assignments, and 31 increments. This reduces the initialization operation to 97 from 770, a saving of 87%. When we consider this relative to the work of processing the file of 50,000 characters, we have saved less than 0.7%(100,098 vs. 100,771). Notice we could save even more time if we did all of these initializations without loops, because only 31 pure assignments would be needed, but this would only save an additional 0.7%. It's not worth the effort. We see that the importance of the initialization is small relative to the overall execution of this algorithm. In analysis terms, the cost of the initialization becomes meaningless as the number of input values increases. The earliest work in analysis of algorithms determined the computability of an algorithm on a Turing machine. The analysis would count the number of times that the transition function needed to be applied to solve the problem.

An analysis of the space needs of an algorithm would count how many cells of a Turing machine tape would be needed to solve the problem. The sort of analysis is a valid determination of the relative speed of two algorithms, but it is also time-consuming and difficult. To do this sort of analysis, you would first need to determine the process used by the transition functions of the Turing machine that carries out the algorithm. Then you would need to determine how long it executes–a very tedious process. An equally valid way to analyze an algorithm, and the one we will use, is to consider the algorithm as it is written in a higher-level language. This language can be Pascal, C, C++, Java, or a general pseudocode. The specifics don't really matter as long as the language can express the major control structures common to algorithms. This means that any language that has a looping mechanism, like a for or while, and a selection mechanism, like an if, case, or switch, will serve our needs. Because we will be concerned with just one algorithm at a time, we will rarely write more than a single function or code fragment, and so the power of many of the languages mentioned will not even come into play. For this reason, a generic pseudocode will be used in this book. Some languages use short-circuit evaluation when determining the value of a Boolean expression. This means that in the expression A and B, the term B will only be evaluated if A is true, because if A is false, the result will be false no matter what B is. Likewise, for A or B, B will not be evaluated if A is true. As we will see, counting a compound expression as one or two comparisons will not be significant. So, once we are past the basics in this chapter, we will not worry about short-circuited evaluations.

Algorithm analysis requires a set of rules to determine how operations are to be counted. There is no generally accepted set of rules for algorithm analysis. In some cases, an exact count of operations is desired ; in other cases, a general approximation is sufficient.

The rules presented that follow are typical of those intended to produce an exact count of operations.

Exact Analysis Rules

1. We assume an arbitrary time unit.
2. Execution of one of the following operations takes time 1 (one) :
   (*a*) Assignment operations (*b*) Single I/O operations
   (*c*) Single boolean operations, numeric comparisons
   (*d*) Single arithmetic operations (*e*) Function return
   (*f*) Array index operations, pointer dereferences
3. Running time of a selection statement (if, switch) is the time for the condition evaluation + the maximum of the running times for the individual clauses in the selection.
4. Loop execution time is the sum, over the number the loop is executed, of the body time + time for the loop check and update operations, + time for the loop setup. (Always assume that the loop executes the maximum number of iterations possible.)
5. Running time of a function call is 1 for setup + the time for any parameter calculations + the time required for the execution of the function body.

In other case, the analysis of an algorithm is to predict the resources that the algorithm requires, and such analysis is based on individual computational models. In the following several popular computational models are listed.

- **RAM.** time and space (traditional serial computers)
- **PRAM.** parallel time, number of processors, and read-and-write restrictions (SIMD type of parallel computers)
- **Message Passing Model.** communication cost (number of message), and computational cost (usually the cost for local computation is ignored) (Distributed computing, peer-to-peer networking, MIMD type of Machine)
- **Turing Machine.** time and space (abstract theoretical machine)

The analysis of an algorithm is to evaluate the performance of the algorithm based on the given models and metrics.

- **Input size**
- **Running time (worst-case and average-case) :** The running time of an algorithm on a particular input is the number of primitive operations or steps executed. Unless otherwise specified, we shall concentrate on finding only the worst case running time.
- **Order of growth.** To simplify the analysis of algorithms, we are interested in the growth rate of the running time, *i.e.*, we only consider the leading terms of a time formula. *e.g.*, the leading term is n^2 in the expression $n^2+100n+50000$.

Most algorithms are designed to work with inputs of arbitrary length. Usually the efficiency or complexity of an algorithm is stated as a function relating the input length to the number of steps (time complexity) or storage locations.

Algorithm analysis is an important part of a broader computational complexity theory, which provides theoretical estimates for the resources needed by any algorithm which solves a given computational problem. These estimates provide an insight into reasonable directions of search of efficient algorithms.

In theoretical analysis of algorithms it is common to estimate their complexity in asymptotic sense, *i.e.*, to estimate the complexity function for reasonably large length of input. Big O notation, omega notation and theta notation are used to this end. For instance, binary search is said to run an amount of steps proportional to a logarithm, or in $O(\log(n))$, colloquially "in logarithmic time". **Usually asymptotic estimates are used because different implementations of the same algorithm may differ in efficiency.** However the efficiencies of any two "reasonable" implementations of a given algorithm are related by a constant multiplicative factor called hidden constant.

Exact (not asymptotic) measures of efficiency can sometimes be computed but they usually require certain assumptions concerning the particular implementation of the algorithm, called **model of computation.** A model of computation may be defined in terms of an abstract computer, *e.g.*, Turing machine, and/or by postulating that certain operations are executed in unit time. For example, if the sorted set to which we apply binary search N elements, and we can guarantee that a single binary lookup can be done in unit time, then at most $\log_2 N+1$ time units are needed to return an answer.

Exact measures of efficiency are useful to the people who actually implement and use algorithms, because they are more precise and thus enable them to know how much time they can expect to spend in execution. To some people (*e.g.*, game programmers), a hidden constant can make all the difference between success and failure.

Time efficiency estimates depend on what we define to be a step. For the analysis to make sense, the time required to perform as step must be guaranteed to be bounded above by a constant. One must be careful here ; for instance, some analysis count and addition of two numbers as a step. This assumption may not be warranted in certain contexts. For example, if the numbers involved in a computation may be arbitrarily large, addition no longer can be assumed to require constant time (compare the time we need to add two 2-digit integers and two 1000-digits integers using a pen and paper).

1.8 Formal and Informal Algorithm Analysis

There are a number of standard criteria which can be used to evaluate any algorithm, which can be divided into formal and informal techniques. A formal criterion is one which yields a result which can be expressed in a logical or arithmetic statement, an informal criterion is one which yields a less precisely stated result.

The criteria, which are about to be introduced, can be used to evaluate two or more different possible algorithms in order to choose which one should actually be implemented for a particular requirement. The techniques can also be used when a proposed optimization of an existing implementation is considered. Some of the most useful informal criteria which can be used include the following.

Effectiveness is the most fundamental criterion and is concerned with an algorithm fulfilling its specification. As it is sometimes difficult to determine if an algorithm is always effective for all possible sets of input data, there are formal techniques which can be used in an attempt to determine this.

Termination is strictly an attribute of effectiveness but is sufficiently for it to be considered as a distinct criterion. For an algorithm to be useful it must at some stage terminate and it is again not always possible to be certain of this for all possible sets of input data.

Generality. The definition of any algorithm should specify the input data which it will require and the output data which it will produce. An algorithm which will accept a greater range of input data and/or produce a greater range of output data is more general and likely to be a more useful algorithm, as it may be usable in a wider range of situations without requiring amendments.

Efficiency is sometimes taken as the most important criterion and is not always adequately specified. Possible sub-criteria include speed of execution, amount of main or secondary storage required or amount of complexity of maintenance. The basic rule for considering efficiency is that it is not possible to optimize an algorithm against any efficiency sub-criterion until it has first been shown to be effective.

Elegance. This is the most subjective aspect of informal algorithm analysis. An elegant algorithm is one which is both simple and ingenious ; of these two factors simplicity is far more important than ingenuity.

1.9 How to Calculate Running Time of an Algorithm ?

We can calculate the running time of an algorithm reliably by running the implementation of the algorithm on a computer. Alternatively we can calculate the running time by using a technique called algorithm analysis. We can estimate an algorithm's performance by counting the number of basic operations required by the algorithm to process an input of a certain size.

Basic Operation. The time to complete a basic operation does not depend on the particular values of its operands. So it takes a constant amount of time.

Examples, Arithmetic operation (addition, subtraction, multiplication, division), Boolean operation (AND, OR, NOT), Comparison operation, Module operation, Branch operation etc.

Input Size. It is the number of input processed by the algorithm.

Example, For sorting algorithm the input size is measured by the number of records to be sorted.

Growth Rate. The growth rate of an algorithm is the rate at which the running time (cost) of the algorithm grows as the size of the input grows. The growth rate has a tremendous effect on the resources consumed by the algorithm.

1.10 Loop Invariants

This is a justification technique. We use loop Invariants to help us understand why an algorithm is correct. To prove some statement S about a loop is correct, define S in terms of a series of smaller statements $S_0, S_1, \ldots S_k$ where,

(*i*) The initial claim, so is true before the loop begins.

(*ii*) If S_{i-1} is true before iteration i begins, then one can show that S_i will be true after iteration i is over.

(*iii*) The final statement S_k, implies the statement S that we wish to justify as being true.

Example. *Consider the following search problem :*

Input : A sequence $A = \langle a_1, a_2 \ldots, a_n \rangle$ and a value of v.

Output : An index i such that $v = A[i]$ or the special value Nil if v does not appear in A.

Write pseudocode for the linear search, which scans through the sequence, looking for v. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three conditions.

Solution. Below is a sample algorithm to solve this linear search problem :

```
1.  i ←1
2.  while (i <=n) do
3.      If (v == A[i])
4.          return i
5.      else i <i + 1
6.  return Nil
```

We now show that either the code returns with the index of v when $v \in A[1..n]$; or the following loop invariant holds at the beginning of the while loop :

$$v \notin A[1..i-1]$$

At the beginning of the very first loop, $i = 1$. There is no way for the code to return, but the invariant is true, since $A[1..0]$ is empty.

Assume that the code has yet to return, and the loop invariant holds at the beginning of the loop when $i = i_0$ *i.e.*, $v \notin A[1..i_0 - 1]$. Once in the loop, if $v = A[i] (= A[i_0])$, then line 4 returns with the value of i_0 ; otherwise line 5 increments i to $i_0 + 1$, which is the value of i in the next loop. Hence, either the code returns with the index of v ; or at the beginning of the next loop, we have $v \notin A[1..i_0]$, *i.e.*, $v \notin A[1..i-1]$. Thus, the maintenance case holds.

If the code does not return for any $i \le n$, the code continues till $i = n+1$ when the invariant still holds at the beginning of that loop. Thus, $v \notin A[1..n]$ since $i = n+1$.

Exercise

1. Why do we study algorithms ?
2. What is the main difference between algorithms and programs ?
3. Define the main characteristics of an algorithm.
4. Define different design approaches of an algorithm.
5. What do you mean by analysis of an algorithm ?
6. How would you calculate running time of an algorithm ?

CHAPTER 2

Growth of Functions

The key idea that supports most of the theory of algorithms is the method of quantifying the execution time of an algorithm. Execution time depends on many parameters which are independent of the particular algorithm : machine clock rate, quality of code produced by compiler, whether or not the computer is multi-programmed etc.

The execution time of an algorithm is a function of the values of its input parameters. If we know this function (at least, to within a scalar multiple determined by the machine parameters) then we can predict the time the algorithm will require without running it. (We might not wish to run it until we knew that it would complete in a reasonable time). Unfortunately this execution time function measure is generally very complicated, too complicated to know exactly. But now a number of observations come to the release.

The first is that is very often possible to get meaningful predictive results by concentrating on one parameter alone : the total size of the input (rather than the accurate values of each input parameter). The second is that execution time functions are often the sum of a large number of terms and many of these terms make an insignificant contribution to function values : in mathematical terms, it is asymptotic form of the function that matters rather than the precise form. The final observation is that, in practice, we wish only to know either an upper bound on the execution time (to be assured that it will perform acceptably) or a lower bound (possibly to encourage us to develop a better algorithm). To sum up these observations : we measure the execution time of an algorithm as a function $T(n)$ of a single quantity n (the size of the entire input).

2.1 Complexity of Algorithms

It is very convenient to classify algorithms based on the relative amount of time or relative amount of space they require and specify the growth of time/space requirements as a function of the input size.

1. **Time Complexity.** Running time of the program as a function of the size of input.

2. **Space Complexity.** Most of what we will be discussing is going to be how efficient various algorithms are in terms of time, but some forms of analysis could be done based on how much space an algorithm needs to complete its task. This space complexity analysis was critical in the early days of computing when storage space on a computer (both internal and external) was limited. When considering space complexity, algorithms are divided into those that need extra space to do their work and those that work in place. It was not unusual for programmers to choose an algorithm that was slower just because it worked in place, because there was not enough extra memory for a faster algorithm.

Computer memory was at a premium ; so another form of space analysis would examine all of the data being stored to see if there were more efficient ways to store it. For example, suppose we are storing a real number that has only one place of precision after the decimal point and ranges between – 10 and +10. If we store this as a real number, most computers will use between 4 and 8 bytes of memory, but if we first multiply the value by 10, we can then store this as an integer between – 100 and +100. This needs only 1 byte, a saving of 3 to 7 bytes. A program that stores 1000 of these values can save 3000 to 7000 bytes. When you consider that computers as recently as the early 1980s might have only had 65,536 bytes of memory, these savings are significant. It is this need to save space on these computers along with the longevity of working computer programs that lead to all of the Y2K bug problems. When you have a program that works with a lot of dates, you use half the space for the year by storing it as 99 instead of 1999. Also, people writing programs in the 1980s and earlier never really expected their programs to still be in use in 2000.

Looking at software that is on the market today, it is easy to see that space analysis is not being done. Programs, even simple ones, regularly quote space needs in a number of megabytes. Software companies seem to feel that making their software space efficient is not a consideration because customers who don't have enough computer memory can just go out and buy another 32 megabytes (or more) of memory to run the program or a bigger hard disk to store it. This attitude drives computers into obsolescence long before they really are obsolete.

A recent change to this is the popularity of personal digital assistants (PDAs). These small handheld devices typically have between 2 and 8 megabytes for both their data and software. In this case, developing small programs that store data compactly is not only important, it is critical.

2.2 RAM Machine

In computing time complexity, one good approach is to count primitive operations. This approach of simply counting primitive operations gives rise to a computational model called the Random Access Machine (RAM). Primitive operations are simply :

- Basic computations performed by an algorithm
- Identifiable in pseudo code

- Largely independent from the programming language
- Exact definition not important (we will see why later)
- Assumed to take a constant amount of time in the RAM model

Some examples of primitive operations are :

- Assigning a value to a variable
- Calling a method
- Performing an arithmetic operation
- Comparing two numbers
- Indexing into an array
- Following an object reference
- Returning from a method

RAM model views a computer simply as a CPU connected to a bank of memory cells. Each memory cell stores a word, which can be a number, a character string, or an address *i.e.*, the value of a base type. The term "random access" refers to the ability of the CPU to access an arbitary memory cell with one primitive operation. To keep the model simple, we do not place any specific limits on the size of numbers that can be stored in words of memory. We assume the CPU in the RAM model can perform any primitive operation in a constant number of steps, which do not depend on the size of the input. Thus, an accurate bound on the number of primitive operations an algorithm performs corresponds directly to the running time of that algorithm in the RAM model. By inspecting the pseudo code, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size.

2.3 Best, Worst and Average - Case Complexity

Using the random-access machine (RAM) model of computation, we can count how many steps our algorithm will take on any given input instance by simply executing it on the given input. In the RAM model, instructions are executed one after another, with no concurrent operations. However, to really understand how good or bad an algorithm is, we must know how it works over *all* instances.

To understand the notions of the best, worst and average-case complexity, one must think about running an algorithm on all possible instances of data that can be fed to it. The best, worst and average cases of a given algorithm express what the resource usage is *at least, at most* and *on average*, respectively. Usually the resource being considered is running time, but it could also be memory or other resource. The *worst-case complexity* of the algorithm is the function defined by the maximum number of steps taken on any instance of size *n*. The *best-case complexity* of the algorithm is the function defined by the minimum number of steps taken on any instance of size *n*. Finally, the *average-case complexity* of the algorithm is the function defined by the average number of steps taken on any instance of size *n*.

Worst-case Running Time. The behaviour of the algorithm with respect to the worst possible case of the input instance. The worst-case running time of an algorithm is an upper bound on the running time for any input. Knowing it gives us a guarantee that the algorithm will never take any longer time.

Average case Running Time. The expected behaviour when the input is randomly drawn from a given distribution. The average-case running time of an algorithm is an estimate of the running time for an "average" input. Computation of a average-case running time

entails knowing all possible input sequences, the probability distribution of occurrence of these sequences, and the running times for the individual sequences.
Often it is assumed that all inputs of a given size are equally likely.

Amortized Running Time. Here the time required to perform a sequence of (related) operations is averaged over all the operations performed. Amortized analysis can be used to show that the average cost of an operation is small, if one averages over a sequence of operations, even though a simple operation might be expensive. Amortized analysis guarantees the average performance of each operation in the worst case.

Worst-case analysis and average-case analysis

Worst case performance analysis and average case performance analysis have similarities, but usually require different tools and approaches in practice. Determining what average input means is difficult, and often that average input has properties which make it difficult to characterise mathematically (consider, for instance, algorithms that are designed to operate on strings of text). Similarly, even when a sensible description of a particular "average case" is possible, they tend to result in more difficult to analyse equations.

Worst case analysis has similar problems, typically it is impossible to determine the exact worst case scenario. Instead, a scenario is considered which is at least as bad as the worst case. For example, when analysing an algorithm, it may be possible to find the longest possible path through the algorithm (by considering maximum number of loops, for instance) even if it is not possible to determine the exact input that could generate this. Indeed, such an it is input may not exist. This leads to a safe analysis (the worst case is never underestimated), but which is *pessimistic*, since no input might require this path.

Alternatively, a scenario which is thought to be close to (but not necessarily worse than) the real worst case may be considered. This may lead to an *optimistic* result, meaning that the analysis may actually underestimate the true worst case.

In some situations it may be necessary to use a pessimistic analysis in order to guarantee safety. Often, however, a pessimistic analysis may be too pessimistic, so an analysis that gets closer to the real value but may be optimistic (perhaps with some known low probability of failure) can be a much more practical approach.

When analyzing algorithms which often take a small time to complete, but periodically require a much larger time, amortized analysis can be used to determine the worst case running time over a (possible infinite) series of operations. This amortized worst case cost can be much closer to the average case cost, while still providing a guaranteed upper limit on the running time.

The other thing that has to be decided when making these considerations is whether what is of interest is the *average* performance of a program, or whether it is important to guarantee that even in the worst case, performance obeys certain rules. In many every-day applications, the average case will be the more important one, and saving time there may be more important than guaranteeing good behaviour in the worst case. On the other hand, when considering time-critical problems (think of a program that keeps track of the planes in a certain sector), it may be totally unacceptable for the software to take very long if the worst comes to the worst.

Again, algorithms often trade efficiency of the average case versus efficiency of the worst case *i.e.*, the most efficient program on average might have a particularly bad worst case. We have seen concrete examples for these when we considered algorithms for searching.

For example,

- Consider the problem of finding the minimum in a list of elements.
  Worst case = $O(n)$; Average case = $O(n)$
- Quick sort
  Worst case = $O(n^2)$; Average case = $O(n \log n)$
- Merge sort, Heap sort
  Worst case = $O(n^2)$; Average case = $O(n^2)$
- Binary search tree : Search for an element
  Worst case = $O(n)$; Average case = $O(\log n)$

2.4 Growth Rate of Functions

Resources for an algorithm are usually expressed as a function of input. Often this function is messy and difficult to work. To study function growth easily, we reduce the function down to the important part.

Let $f(n) = an^2 + bn + c$.

In this function, the n^2 term dominates the function, that is when n gets sufficiently large, the other terms bare factor into the result.

Dominate terms are what we are interested in to reduce a function, in this we ignore all constants and coefficients and look at the highest order term in relation to n.

2.5 Asymptotic Analysis

Asymptotic means a line that tends to converge to a curve, which may or may not eventually touch the curve. It is a line that stays within bounds.

Asymptotic notation is a shorthand way to write down and talk about 'fastest possible' and 'slowest possible' running times for an algorithm, using high and low bounds on speed. These are also referred to as 'best case' and 'worst case' scenarios respectively.

2.5.1 Why are Asymptotic Notations Important ?

1. They give a simple characterization of an algorithm's efficiency.
2. They allow the comparison of the performances of various algorithms.

2.5.2 Asymptotic Notations

1. Big-oh notation. Big-oh is the formal method of expressing the upper bound of an algorithm's running time. It is the measure of the longest amount of time it could possibly take for the algorithm to complete. More formally, for non-negative functions, $f(n)$ and $g(n)$, if there exists an integer n_0 and a constant $c > 0$ such that for all integers $n > n_0$.

$$f(n) \le cg(n)$$

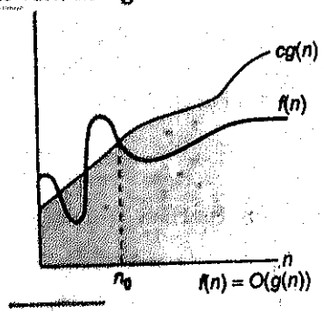


Figure 2.1

Then $f(n)$ is Big-oh of $g(n)$. This is denoted as

$$"f(n) \in O(g(n))"$$

i.e., the set of functions which, as n gets large, grow no faster than a constant times $f(n)$.

2. Big-omega notation. For non-negative functions, $f(n)$ and $g(n)$, if there exists an integer n_0 and a constant $c > 0$ such that for all integers $n > n_0$, $f(n) \geq cg(n)$ then $f(n)$ is big omega of $g(n)$. This is denoted as

$$"f(n) \in \Omega(g(n))".$$

This is almost the same definition as Big-oh, except that "$f(n) \geq (g(n))$", this makes $g(n)$ a lower bound function instead of an upper bound function. It describes the best that can happen for a given data size.

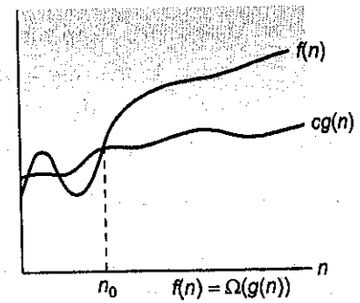


Figure 2.2

3. Theta notation. The lower and upper bound for the function 'f' is provided by the theta notation (θ). For non-negative functions $f(n)$ and $g(n)$, if there exists an integer n_0 and positive constants c_1 and c_2 *i.e.,* $c_1 > 0$ and $c_2 > 0$ such that for all integers $n > n_0$, $c_1 g(n) \leq f(n) \leq c_2 g(n)$ then $f(n)$ is theta of $g(n)$. This is denoted as "$f(n) \in \theta(g)$" we mean "f is order g".

Theorem

If $f(n) = a_m n^m + a_{m-1} n^{m-1} + \ldots a_1 n + a_0$ *and* $a_m > 0$, *then* $f(n) = O(n^m)$

Proof.

$$f(n) \leq \sum_{k=0}^{m} |a_k| n^k$$

$$\leq n^m \sum_{k=0}^{m} |a_k| n^{k-m} \leq n^m \sum_{k=0}^{m} |a_k| \text{ for } n \geq 1$$

So $\quad f(n) = O(n^m)$.

Properties. Let $f(n)$ and $g(n)$ be such that $\lim_{n \to \infty} \frac{f(n)}{g(n)}$ exists, then

(*i*) Function $f \in O(g)$ *i.e.,* $f(n) \in O(g(n))$ if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c < \infty, \text{ also including the case in which limit is } 0.$$

(*ii*) Function $f(n) \in \Omega(g(n))$ if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} > 0, \text{ including the case in which limit is } \infty.$$

(*iii*) Function $f(n) \in \theta(g(n))$ if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c \text{ for some constant } c \text{ such that } 0 < c < \infty.$$

4. Little-oh notation (o). Asymptotic upper bound provided by O-notation may not be asymptotically tight. So o-notation is used to denote an upper bound that is asymptotically tight.

$$o(g(n)) = \{ f(n) : \text{ for any +ve constant } c > 0, \text{ if a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \ \forall\, n \geq n_0 \}$$

The function $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity, *i.e.,*

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

For example, $\quad 2n = o(n^2)$

$$2n^2 \neq o(n^2)$$

5. Little-Omega Notation. As o-notation is to O-notation, we have ω-notation as Ω-notation. Little-omega (ω) is used to denote an lower bound that is asymptotically tight,

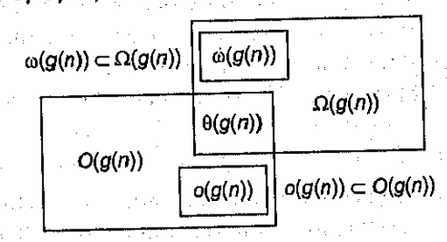
$$\omega(g(n)) = \{ f(n) : \text{ for any +ve constant } c > 0, \text{ if a constant } n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \}$$

Here $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$.

Thus asymptotic notation gives us a way to express their relationship.

- If $f(n)$ is $O(g(n))$ *i.e.,* $f(n)$ grows no faster than $g(n)$
- If $f(n)$ is $o(g(n))$ *i.e.,* $f(n)$ grows slower than $g(n)$
- If $f(n)$ is $\omega(g(n))$ *i.e.,* $f(n)$ grows faster than $g(n)$.
- If $f(n)$ is $\Omega(g(n))$ *i.e.,* $f(n)$ grows no slower than $g(n)$.
- If $f(n)$ is $\theta(g(n))$ *i.e.,* $f(n)$ and $g(n)$ grow at the same rate.

Relationships between O, o, θ, Ω, ω notations



θ is a subset of Ω and of O.

(*i*) $\omega(g)$ union $\theta(g)$ is a subset of $\Omega(g)$

i.e., $\omega(g) \cup \theta(g) \subset \Omega(g)$

(*ii*) $o(g)$ union $\theta(g)$ is a subset of $O(g)$.

i.e., $o(g) \cup \theta(g) \subset O(g)$

(*iii*) $\theta(g)$ is the intersection of $O(g)$ and $\Omega(g)$.

i.e., $\theta(g) = \Omega(g) \cap O(g)$

In other words

$$f(n)=\theta(g(n)) \text{ iff } f(n)=o(g(n)) \text{ and } f(n)=\Omega(g(n))$$

Figure 2.2A shows how the various measures of complexity compare with one another. The horizontal axis represents the size of the problem–for example, the number of records to process in a search algorithm. The vertical axis represents the computational effort required by algorithms of each class. This is not indicative of the running time or the CPU cycles consumed ; it merely gives an indication of how the computational resources will increase as the size of the problem to be solved increases.

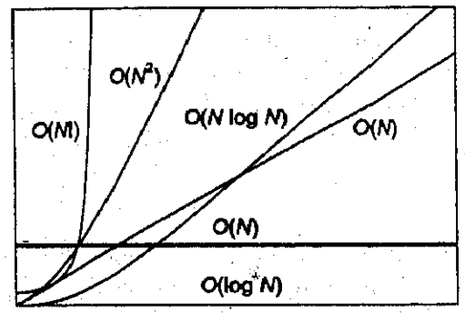


Figure 2.2A

Referring back at the list, you may have noticed that none of the orders contain constants. That is, if an algorithm's expected runtime performance is proportional to N, $2\times N$, $3\times N$, or even $100\times N$, in all cases the complexity is defined as being $O(N)$. This may seem a little strange at first–surely $2\times N$ is better than $100\times N$–but as mentioned earlier, the aim is not to determine the exact number of operations but rather to provide a means of comparing different algorithms for relative efficiency. In other words, an algorithm that runs in $O(N)$ time will generally outperform another algorithm that runs in $O(N^2)$. Moreover, when dealing with large values of N, constants make less of a difference : As a ratio of the overall size, the difference between 1,000,000,000 and 20,000,000,000 is almost insignificant even though one is actually 20 times bigger.

Of course, at some point you will want to compare the actual performance of different algorithms, especially if one takes 20 minutes to run and the other 3 hours, even if both are $O(N)$. The thing to remember, however, is that it's usually much easier to halve the time of an algorithm that is $O(N)$ than it is to change an algorithm that's inherently $O(N^2)$ to one that is $O(N)$.

Why not use θ notation all the time ?

At the first glance, it would appear that θ is all we really want or need-after all, it is the right answer. Where the order of $g(n)$ is exactly $f(n)$. That is, if we know the real order of $f(n)$, then we know $\theta(g(n))$.

Unfortunately, it is not so simple – there are functions that do not have a θ at all. For example, consider a function $f(n)$ that is $O(n)$ when n is even and $O(1)$ if n is odd. Therefore

$$f(n)=O(n) \quad \text{but} \quad f(n)=\Omega(1).$$

Helpful Hints

1. Not every pair of functions is comparable.
2. It may be easier to test for $o(g)$ and $\omega(g)$. Try these first and then try O, Ω, and θ.
3. Sometimes we can deduce several relationships from the knowledge of only one. For example, if a function is $o(g)$ it is also $O(g)$, but never $\theta(g)$, $\Omega(g)$ or $\omega(g)$.
4. When is doubt, graph the functions.

Example 1. *Prove* $n!=o(n^n)$

Solution. We know $n!=n(n-1)(n-2)\ldots 2\cdot 1$

$$=n^n\left[1-\frac{1}{n}\right]\ldots\frac{2}{n}\cdot\frac{1}{n}$$

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\lim_{n\to\infty}\frac{n^n\left[1-\frac{1}{n}\right]\ldots\frac{2}{n}\cdot\frac{1}{n}}{n^n}=0$$

$\therefore\ n!=o(n^n)$.

Example 2. *Prove* $n!=w(2^n)$

Solution. $$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\lim_{n\to\infty}\frac{n^n\left[1-\frac{1}{n}\right]\left[1-\frac{2}{n}\right]\ldots\frac{2}{n}\cdot\frac{1}{n}}{2^n}=\frac{\infty}{2^\infty}=\infty.$$

Hence $n!=w(2^n)$.

Example 3. *Prove that* $f(n)=\log_2 n$ *is* $O(n^\alpha)$ *for any* $\alpha>0$.

Solution. By definition $f(n)\in O(n)$ if

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=0$$

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\lim_{n\to\infty}\frac{\log_2 n}{n^\alpha}=\lim_{n\to\infty}\frac{\lg n}{\lg 2\cdot n^\alpha}$$

i.e.,

$$\frac{f(n)}{g(n)}=\frac{\lg n}{\lg 2\cdot n^\alpha}$$

$$\frac{f'(n)}{g'(n)}=\frac{\frac{1}{n}}{\lg 2\cdot\alpha n^{\alpha-1}}=\frac{n}{n\alpha\lg 2n^\alpha}=\frac{1}{\alpha\lg 2\cdot n^\alpha}.$$

By L's Hospital Rule,

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\lim_{n\to\infty}\frac{f'(n)}{g'(n)}=\lim_{n\to\infty}\frac{1}{\alpha\lg 2\cdot n^\alpha}=\frac{1}{\infty}=0.$$

$\therefore\ f(n)\in O(n^\alpha)$.

2.5.3 Asymptotic Notation Properties

1. Reflexivity
2. Symmetry
3. Transpose Symmetry
4. Transitivity

1. Reflexivity

$$f(n)=\theta(f(n))$$
$$f(n)=O(f(n))$$
$$f(n)=\Omega(f(n))$$

2. Symmetry

$$f(n)=\theta(f(n)) \text{ if and only if } g(n)=\theta(f(n))$$

3. Transpose Symmetry

$$f(n)=O(f(n)) \text{ if and only if } g(n)=\Omega(f(n))$$
$$f(n)=o(g(n)) \text{ if and only if } g(n)=\omega(f(n))$$

4. Transitivity

$$f(n)=\theta(g(n)) \text{ and } g(n)=\theta(h(n)) \text{ imply } f(n)=\theta(h(n))$$
$$f(n)=O(g(n)) \text{ and } g(n)=O(h(n)) \text{ imply } f(n)=O(h(n))$$
$$f(n)=\Omega(g(n)) \text{ and } g(n)=\Omega(h(n)) \text{ imply } f(n)=\Omega(h(n))$$
$$f(n)=o(g(n)) \text{ and } g(n)=o(h(n)) \text{ imply } f(n)=o(h(n))$$
$$f(n)=\omega(g(n)) \text{ and } g(n)=\omega(h(n)) \text{ imply } f(n)=\omega(h(n))$$

Manipulating asymptotic notations :

1. $cO(f(n))=O(f(n))$
2. $O(O(f(n)))=O(f(n))$
3. $O(f(n)g(n))=f(n)O(g(n))$
4. $O(f(n))O(g(n))=O(f(n)g(n))$
5. $O(f(n)+g(n))=O(\max(f(n), g(n)))$
6. $\sum_{i=1}^{n} O(f(i))=O\left(\sum_{i=1}^{n} f(i)\right)$

2.5.4 Asymptotic Notations in equations

(*i*) $n=O(n^2)$

It generally means $n \in$ in the set of $O(n^2)$

(*ii*) $2n^2+6n+5=2n^2+\theta(n)$

It means, we do not care about details of $\theta(n)$

It means $\theta(n)$ is some function $f(n) \in \theta(n)$

$f(n)$ is an anonymous function.

(*iii*) If asymptotic notation appears on the left

i.e., $2n^2+\theta(n)=\theta(n^2)$

We can always choose a right hand side to satisfy equality.

(*iv*) $2n^2+16n+2=2n^2+\theta(n)=\theta(n^2)$

2.6 Analysing Algorithm Control Structures

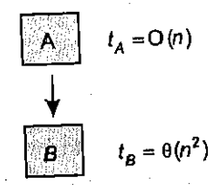
The analysis of an algorithm is calculated by considering its individual instructions. The individual instructions are calculated and then according to the control structures, we combine these times. Some algorithm control structures are :

1. Sequencing
2. If-then-else
3. "for" loop
4. "while" loop
5. Recursion

1. Sequencing

Suppose our algorithm consists of two parts A and B. A takes time t_A and B takes t_B time for computation. The total computaion "t_A+t_B" is according to the sequencing rule. According to maximum rule this computation time is (max (t_A, t_B)).

Example. Suppose $t_A=O(n)$ and $t_B=\theta(n^2)$. Calculate the total computation time



$$\text{Computation time } = t_A+t_B$$
$$=(\max(t_A, t_B))$$
$$=(\max(O(n), \theta(n^2))=\theta(n^2)$$

2. If-then-else

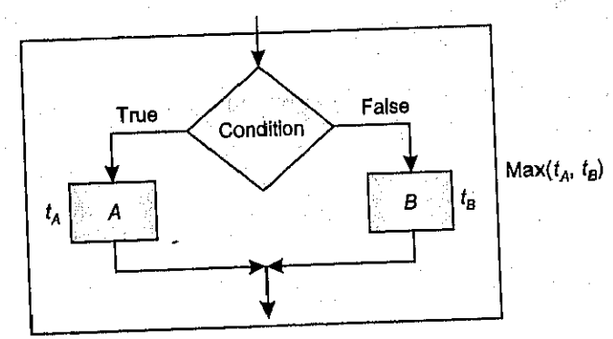


Figure 2.3

The total computation time is according to the conditional rule - "if-then-else". According to the maximum rule this computation time is max $(t_A\ t_B)$.

Example, Suppose $t_A = O(n^2)$ and $t_B = \theta(n^2)$. Calculate the total computation time for the following :

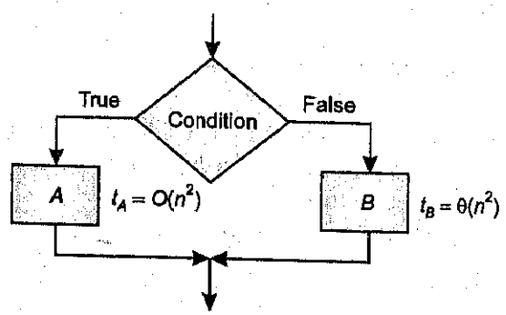


Figure 2.4

$$\text{Total computation} = \max(t_A, t_B)$$
$$= \max(O(n^2), \theta(n^2)) = \theta(n^2)$$

3. "For" loop

Consider the following loop.

```
for i ← 1 to m.
{
    P(i)
}
```

If the computation time t_i for $P(i)$ various as a function of "i", then total computation time for the loop is given not by a multiplication but by a sum. *i.e.,*

```
for i ← 1 to m.
{
    P(i)
}
```

takes $\sum_{i=1}^{m} t_i$ time. *i.e.,* $\sum_{i=1}^{m} \theta(1) = \theta \sum_{i=1}^{m} 1 = \theta(m)$.

If the algorithm consists of nested "for" loops then the total computation time is

```
for i ← 1 to m.
{
    for j ← 1 to m.
    {
        P(ij)
    }
}
```

$$\sum_{i=1}^{m} \sum_{j=1}^{m} t_{ij}$$

Example. *Consider following 'for' loops, calculate the total computation time for the following*

```
for i ← 2 to m − 1.
{
    for j ← 3 to i.
    {
        sum ← sum + A [i] [j]
    }
}
```

Solution. The total computation time is

$$\sum_{i=2}^{m-1} \sum_{j=3}^{i} t_{ij} = \sum_{i=2}^{m-1} \sum_{j=3}^{i} \theta(1)$$

$$= \sum_{i=2}^{m-1} \theta(i)$$

$$= \theta\left(\sum_{i=2}^{m-1} i\right) = \theta(m^2/2 + \theta(m))$$

$$= \theta(m^2).$$

4. "while" loop

In this, there is no obvious method which determines how many times we shall have to repeat the loop. The simple technique for analysing the loops is to firstly determine function of variables involved whose value decreases each time around. Secondly for terminating the loop, it is necessary that this value must be a positive integer. By keeping the track of how many times the value of function decreases, one can obtain the number of repetition of the loop. The other approach for analysing "while" loops is to treat them as recursive algorithms.

Example. *The running time of algorithm array Max for computing the maximum element in an array of n integers is O(n).*

Solution. array Max (A, n).

```
current max ← A[0].
for i ← 1 to n − 1
    do if current max < A[i]
        then current max ← A[i]
return current max.
```

The number of primitive operations $t(n)$ executed by this algorithm is atleast

$$2 + 1 + n + 4(n-1) + 1 = 5n$$

and at most $\quad 2 + 1 + n + 6(n-1) + 1 = 7n - 2.$

The best case $t(n)=5n$ occurs when $A[o]$ is the maximum element. The worst case $t(n)=7n-2$ occurs when the elements are sorted in increasing order.

We may therefore apply the big-oh definition with $c=7$ and $n_0=1$ and conclude that running time of this is $O(n)$.

Example. *Let $f(n)$ and $g(n)$ be asymptotically non-negative functions. Using the basic definition of θ-notation prove that*

$$max(f(n), g(n))=\theta(f(n)+g(n))$$

Solution. This is true if there exists $c_1, c_2, n_0>0$ such that

$$0 \le c_1(f(n)+g(n)) \le \max(f(n), g(n)) \le c_2(f(n)+g(n)) \quad \text{for all } n \ge n_0.$$

◄ Choose for instance $c_1=\frac{1}{2}$ $\Rightarrow$ two situations.

1. $f(n)>g(n) \Rightarrow \frac{1}{2}(f(n)+g(n))<\frac{1}{2}(f(n)+f(n))=f(n)=\max(f(n),g(n))$

2. $f(n)<g(n) \Rightarrow \frac{1}{2}(f(n)+g(n))<\frac{1}{2}(g(n)+g(n))=g(n)=\max(f(n),g(n))$

◄ Choose for instance $c_2=1$.

$$\max(f(n), g(n)) \le 1(f(n)+g(n))=f(n)+g(n)$$

$$\max(f(n), g(n)) \le f(n)+g(n)$$

So, choose $c_1=\frac{1}{2}$, $c_2=1$ and the statement holds.

Actually, any $c_1 \le \frac{1}{2}$ and $c_2 \ge \frac{1}{2}$ works.

Example. *"Explain why the statement "The running time of algorithm A is at least $O(n^2)$", is meaningless".*

Solution. "The running time of algorithm. A is atleast $O(n^2)$" is like saying that $T(n)$ is asymptotically greater than or equal to all those functions that are $O(n^2)$.

O-notation is meant to indicate the upper bound.

Therefore, $O(n^2)$ by itself means "at most $c \cdot n^2$. So the statement "The running time of algorithm A is at least at most $c \cdot n^2$ does not make sense.

Example. *Prove that $(n+a)^b=\theta(n^b)$, $b>0$*

Solution. It means, we have to show that

$$c_1 n^b \le (n+a)^b \le c_2 n^b.$$

$$(n+a)^b=(n+a)(n+a)(n+a)\ldots(n+a) \text{ } b \text{ times}$$

$$=n^b+k_1 n^{b-1}+k_2 n^{b-2}+\ldots+k_{b-1}n^1+k_b$$

$$c_1 n^b \le n^b+k_1 n^{b-1}+k_2 n^{b-2}+\ldots+k_{b-1}n^1+k_b \le c_2 n^b.$$

Now, $$c_1 n^b \le n^b+k_1 n^{b-1}+k_2 n^{b-2}+\ldots+k_{b-1}n^1+k_b$$

$$c_1 \le 1+\frac{k_1}{n}+\frac{k_2}{n^2}+\ldots+\frac{k_b}{n^b}.$$

It is possible to find c_1 small enough such that for particular values of $k_1, k_2, \ldots k_{b-1}, k_b$ even if values are negative for $n \ge n_0$.

i.e., $$(n+a)^b=\Omega(n^b) \quad \ldots(1)$$

Now, $$n^b+k_1 n^{b-1}+k_2 n^{b-2}+\ldots k_{b-1}n^1+k_b \le n^b+k_1 n^b+k_2 n^b+\ldots k_{b-1}n^b+k_b n^b=kn^b$$

i.e., $$n^b+k_1 n^{b-1}+k_2 n^{b-2}+\ldots+k_b \le c_2 n^b \quad \forall\ n \ge n_0.$$

$$(n+a)^b=O(n^b) \quad \ldots(2)$$

From (1) and (2), we get

$$(n+a)^b=\theta(n^b)$$

Example. *"Is $2^{n+1}=O(2^n)$? Is $2^{2n}=O(2^n)$?"*

Solution. Is $2^{n+1}=O(2^n)$ *i.e.,* $2^{n+1} \le c \cdot 2^n$ for any value of c and all $n \ge n_0$.

Yes, choose $c \ge 2$ and the statement is true.

Hence $2^{n+1}=O(2^n)$ is true.

Now is $2^{2n}=O(2^n)$ *i.e.,* $2^{2n} \le c \cdot 2^n$ for any c and all $n \ge n_0$.

We now if $\lim_{n\to\infty} \frac{f(n)}{g(n)}=\text{constant}$, then $f(n)=O(g(n))$

In this case the ratio

$$\frac{2^{2n}}{c \cdot 2^n}=\frac{(2^n)^2}{c \cdot 2^n}=\frac{2^n}{c}$$

which grows unbounded with n. This $c \cdot 2^n$ can never be an upper bound to 2^{2n}, regardless how big we choose c.

Example. *Let $p(n)=\sum_{i=0}^{d} a_i n^i$ where $a_d>0$ be a degree-d polynomial in n and let k be a constant. Use the definitions of the asymptotic notations to prove the following properties :*

(a) If $k \ge d$, then $p(n)=O(n^k)$

(b) If $k=d$, then $p(n)=\theta(n^k)$.

Solution. $p(n)=\sum_{i=0}^{d} a_i n^i$ where $a_d>0$.

(*a*) Show that if $k \geq d$, then $p(n) = O(n^k)$.

This is true if and only if

$$0 \leq \sum_{i-0}^{d} a_i n^i \leq c \cdot n^k \text{ for all } n > n_0 \text{ and some constant } c.$$

Divide with n^k we get $0 \leq \sum_{i-0}^{d} a_i n^{i-k} \leq c$.

Since $k \geq d$

all $i - k \leq 0$ *i.e.,* all n^{i-k} will be less than 1.

So, choose $c = \sum_{i=0}^{d} a_i$ the statement is true.

(*b*) Show that if $k = d$, then $p(n) = \theta(n^k)$.

This is true if and only if

$$0 \leq c_1 \cdot n^k \leq \sum_{i=0}^{d} a_i n^i \leq c_2 \cdot n^k$$

Divide by n^k, we get $0 \leq c_1 \leq \sum_{i=0}^{d} a_i n^{i-k} \leq c_2$

Since $k = d$,

Thus $0 \leq c_1 \leq a_0 n^{0-k} + a_1 n^{1-k} + \ldots + a_k n^{k-k} \leq c_2$

i.e., $0 \leq c_1 \leq a_0 n^{-k} + a_1 n^{1-k} + \ldots + a_k n^0 \leq c_2$

Here, we choose $c_1 = a_d$ and $c_2 = \sum_{i=0}^{d} a_i$, the statement is true.

Example. *Prove that for any two functions $f(n)$ and $g(n)$, we have*

$f(n) = \theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Solution. We need to prove $f(n) = O(g(n))$ and $f(n) = \Omega(g(n)) \leftrightarrow f(n) = \theta(g(n))$

To prove one implication,

$f(n) = O(g(n))$ *i.e.,* $f(n) \leq c_0 g(n)$, $n \geq n_0$ R.H.S

$f(n) = \Omega(g(n))$ *i.e.,* $f(n) \geq c_1 g(n)$, $n \geq n_1$

which means $c_1 g(n) \leq f(n) \leq c_0\, g(n)$ $n \geq \max(n_0, n_1)$

Now, to prove the other side of the implication.

$f(n) = \theta(g(n))$ means $c_0\, g(n) \leq f(n) \leq c_1\, g(n)$, $n \geq n_0$ L.H.S.

Hence, $f(n) \leq c_0\, g(n)$, $n \geq n_0$ means $f(n) = O(g(n))$

$f(n) \geq c_1\, g(n)$, $n \geq n_0$ means $f(n) = \Omega(g(n))$

Both sides of the implication were proved true. Hence

$f(n) = O(g(n))$ and $f(n) = \Omega(g(n)) \leftrightarrow f(n) = \theta(g(n))$

Example. *Prove that $o(g(n)) \cap \omega(g(n))$ is the empty set.*

Solution. Proof by contradiction assume $f(n) = o(g(n))$ and $f(n) = \omega(g(n))$

According to definition of little-*o*

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \quad \ldots(1)$$

and definition of little-omega says $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$ ...(2)

which is clearly impossible for any function $f(n)$.

Hence $f(n)$ cannot be both *o* and ω.

Hence $o(g(n)) \cap \omega(g(n))$ is the empty set.

Example. *Prove the equation $\lg(n!) = \theta(n \lg n)$. Also prove that $n! = \omega(2^n)$ and $n! = o(n^n)$.*

Solution. (*a*) $\lg(n!) = \lg[n(n-1)(n-2)\ldots 1] \leq \lg[n \times n \times n \ldots n]$

$= \lg(n^n) = n \lg n$.

Thus $\lg(n!) \leq c_1 n \lg n$ for $c_1 \geq 1$ and $n \geq 0$.,

This proves $\lg(n!) = O(n \lg n)$

To prove that $\lg(n!) = \Omega(n \lg n)$

We use Striling's approximation :

$$n! = \sqrt{2\pi n}\left[\frac{n}{e}\right]^n \left[1 + \theta\left(\frac{1}{n}\right)\right]$$

where e is the base of the natural logarithm gives us a tighter upper bound and a lower bound as well.

Now, prove $n! \geq cn^n$.

$$n! \geq \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \frac{c}{n}\right)$$

$$= \sqrt{2\pi}\left(\frac{n^{n+0.5}}{e^n}\right)\left(1 + \frac{c}{n}\right) = \sqrt{2\pi}\left(\frac{n^{n+0.5}}{e^n}\right) + \sqrt{2\pi}\left(\frac{cn^{n-0.5}}{e^n}\right)$$

$\geq 2\pi n^n$ for $c > 0$, $n \geq 1$

Hence $n! \geq n^n$ for $c > 0$, $n \geq 1$.

If we take log of both sides, the inequality will still be true because log is monotonic function.

$\log(n!) \geq \log(n^n) = n \log n$ for $c > 0$, $n \geq 1$

Hence $\lg(n!) = \Omega(n \lg n)$

Since, we proved $\lg(n!) = \Omega(n \lg n)$ and $\lg(n!) = O(n \lg n)$

$\lg(n!) = \theta(n \lg n)$.

Now prove $n! = \omega(2^n)$

$$n*(n-1)*(n-2)...1 > c*2*2*2...2$$

Divide both sides by 2^n.

$$\frac{n}{2}*\frac{(n-1)}{2}*\frac{(n-2)}{2}...\frac{1}{2} > c$$

Each element on the left side (except ½) is greater than 1.

Hence if $c > \frac{1}{2}$, the inequality holds for $n > 0$, therefore

$$n! = \omega(2^n)$$

Now prove $n! = o(n^n)$

$$n*(n-1)*...2*1 < c*n*n...n$$

Inequality holds for $n > 1$ and $c > 1$.

Example. *Find two non-negative functions $f(n)$ and $g(n)$ such that neither $f(n) = O(g(n))$ nor $g(n) = O(f(n))$.*

Solution. Consider $f(n) = 1 + \cos(n)$ and $g(n) = 1 + \sin(n)$.

Therefore, both $f(n)$ and $g(n)$ are periodic and take values in [0...2]

But when $f(n) = 0$, $g(n) = 2$ and when $g(n) = 0$, $f(n) = 2$.

Therefore, we cannot find a positive constant c, such that $f(n) \le c \cdot g(n)$ for large n, because $g(n)$ always comes back to 0 when $f(n)$ is 2.

Therefore, $f(n)$ cannot be $O(g(n))$. The same argument works for the other case.

Example. *"Let $f(n)$ and $g(n)$ be asymptotically positive functions. Prove or disprove each of the following conjectures".*

(a) $f(n) = O(g(n))$ implies $g(n) = O(f(n))$

(b) $f(n) + g(n) = \theta(\min(f(n), g(n)))$

(c) $f(n) = O(g(n))$ implies $\lg(f(n)) = O(\lg(g(n)))$ where $\lg(g(n)) \ge 1$ and $f(n) \ge 1$ for all sufficiently large n.

(d) $f(n) = O(g(n))$ implies $2^{f(n)} = O(2^{g(n)})$

(e) $f(n) = O((f(n))^2)$

(f) $f(n) = O(g(n))$ implies $g(n) = \Omega(f(n))$

(g) $f(n) = \theta\left(f\left(\frac{n}{2}\right)\right)$

(h) $f(n) + o(f(n)) = \theta(f(n))$

Solution. (a) No, $f(n) = O(g(n))$ does not imply $g(n) = O(f(n))$

Clearly, $n = O(n^2)$ but $n^2 \ne O(n)$

(b) No, $f(n) + g(n)$ is not $\theta(\min(f(n), g(n)))$

e.g., if $f(n) = n$

$g(n) = 1$ then $f(n) + g(n) = n + 1 \ne \theta(\min(n, 1))$

i.e., $n + 1 \ne \theta(1)$

other example is if $f(n) = n^{17}$ $g(n) = n^2$

then $n^{17} + n^2 \ne \theta(n^2)$.

(c) $f(n) = O(g(n)) \Leftrightarrow 0 \le f(n) \le c \cdot g(n)$ for some $c > 0$ and all $n > n_0$.

$\Rightarrow \lg(f(n)) \le \lg(c \cdot g(n))$

$\Rightarrow \lg(f(n)) \le \lg c + \lg(g(n))$

$\le c_2 \lg(g(n))$ for some $c_2 > 1$

i.e., $\lg(f(n)) = O(\lg(g(n)))$

This only holds if $\lg(g(n))$ is not approaching 0 as n grows.

(d) No, $f(n) = O(g(n))$ does not imply $2^{f(n)} = O(2^{g(n)})$

If $f(n) = 2n$ and $g(n) = n$, then

$2^{2n} = 4^n \ne O(2^n)$

(e) If $f(n) < 1$ for large n, then

$(f(n))^2 < f(n)$

and the upper band will not hold otherwise $f(n) > 1$ and the statement is trivally true.

e.g., suppose $f(n) = \frac{1}{n} \ne O\left(\frac{1}{n^2}\right)$

(f) $f(n) = O(g(n)) \Leftrightarrow 0 \le f(n) \le c_1 g(n)$ for some $c_1 > 0$ and all $n > n_0$

$$\Leftrightarrow 0 \le \left(\frac{1}{c_1}\right) f(n) \le g(n)$$

$$\Leftrightarrow g(n) = \Omega(f(n))$$

(g) No, let $f(n) = 4^n$

$4^n \ne \theta(4^{n/2} = 2^n)$

(h) Show that $f(n) + o(f(n)) = \theta(f(n))$

$$o(f(n)) = \{g(n) : 0 \le g(n) < c \cdot f(n), \ \forall c > 0, \ \forall n \ge n_0\}$$

At most $f(n) + o(f(n))$ can be $f(n) + c f(n) = (1 + c) f(n)$.

At least $f(n) + o(f(n))$ can be $f(n) + 0 = f(n)$.

Thus $0 < f(n) < f(n) + o(f(n)) < (1 + c) f(n)$ which is the same as saying

$$f(n) + o(f(n)) = \theta(f(n))$$

2.7 Logarithms and Exponents

One of the interesting and sometimes even surprising aspects of the analysis of data structures and algorithms is the ubiquitous presence of logarithms and exponents, where we say

$$\log_b a = c \text{ if } a = b^c$$

i.e., An logarithm is simply an inverse exponential function. Exponential functions are functions that grow at a distressingly fast rate, as anyone who however tried to pay off a mortgage or bank loan understands.

Thus, inverse exponential functions, *i.e.,* logarithms grow refreshingly slowly. If we have an algorithm that runs in $O(\log n)$ time, take it and run. This will be blindingly fast even on very large problem instances. Binary search is an example of an algorithm that takes $O(\log n)$ time. The power of binary search and logarithms is one of the most fundamental ideas in the analysis of algorithms. Two mathematical properties of logarithms are important to understand :

1. The base of the logarithm has relatively little impact on the growth rate. Compare the following *three* values :

- $\log_2 (1{,}000{,}000) = 19.9316$
- $\log_3 (1{,}000{,}000) = 12.5754$
- $\log_{100} (1{,}000{,}000) = 3$

A big change in the base of the logarithm produces relatively little difference in the value of the log.

This is a consequence of the formula for changing the base of the logarithm :

$$\log_b a = \frac{\log_c a}{\log_c b}$$

Changing the base of the log from a to c involves multiplying or dividing by $\log_c a$. This will be lost to the big oh-notation whenever a and c are constants as is typical. Thus we are usually justified in ignoring the base of the logarithm, when analyzing algorithms.

2. Logarithms cut any function down to size. The growth rate of the logarithm of any polynomial function is $O(\lg n)$.

This follows because

$$\log_a n^b = b \log_a n.$$

The power of binary search on a wide range of problems is a consequence of this observation.

For example, note that doing a binary search on a sorted array of n^2 things requires only twice as many comparisons as a binary search on n things. Thus, logarithms efficiently cut any function down to size.

Example. *For each function $f(n)$ and time t in the following table, determine the largest size n of a problem that can be solved in time t, assuming that the algorithm to solve the problem takes $f(n)$ microseconds.*

	1 second	*1 minute*	*1 hour*	*1 day*	*1 month*	*1 year*	*1 century*
$\lg n$							
$\sqrt{n}$							
n							
$n \lg n$							
n^2							
n^3							
2^n							
$n!$							

Solution.

	1 second	*1 minute*	*1 hour*	*1 day*	*1 month*	*1 year*	*1 century*
$\lg n$	2^{10^6}	$2^{6 \cdot 10^7}$	$2^{36 \cdot 10^8}$	$2^{864 \cdot 10^8}$	$2^{2592 \cdot 10^9}$	$2^{94608 \cdot 10^{10}}$	$2^{94608 \cdot 10^{12}}$
$\sqrt{n}$	10^{12}	$36 \cdot 10^{14}$	$1296 \cdot 10^{16}$	$746496 \cdot 10^{16}$	$6718464 \cdot 10^{18}$	$8950673664 \cdot 10^{20}$	$8950673664 \cdot 10^{24}$
n	10^6	$6 \cdot 10^7$	$36 \cdot 10^8$	$864 \cdot 10^8$	$2592 \cdot 10^9$	$94608 \cdot 10^{10}$	$94608 \cdot 10^{12}$
$n \lg n$	62746	2801417	??	??	??	??	??
n^2	10^3	24494897	$6 \cdot 10^4$	293938	1609968	30758413	307584134
n^3	10^2	391	1532	4420	13736	98169	455661
2^n	19	25	31	36	41	49	56
$n!$	9	11	12	13	15	17	18

We assume that all months are 30 days and all years are 365.

Exercise

1. Find the Big-oh(O) notation for the following functions.
   (a) $f(n) = 6993$
   (b) $f(n) = 6n^2 + 135$
   (c) $f(n) = 7*n^2 + 8*n + 39$
   (d) $f(n) = n^4 + 35n^2 + 84$
2. Find the Big theta (θ) and Big omega (Ω) notation for the following :
   (a) $f(n) = 14*7 + 83$
   (b) $f(n) = 83n^3 + 84n$
   (c) $f(n) = n^2 + n$
   (d) $f(n) = 13n + 8$
3. Prove that the following are the loose bounds.
   (a) $17n^3 + 8n^2 = O(n^4)$
   (b) $8n + 16 = O(n^2)$
   (c) $8n^3 + 6n^2 = \Omega(n^2)$
   (d) $8n + 3 = \Omega(1)$
4. Arrange the following growth rates in the increasing order.
   $O(n^4)$, $O(1)$, $O(n^3)$, $O(n)$, $O(n\log n)$, $O(n^2 \log n)$, $\Omega(n^{0.5})$, $\Omega(n^2 \lg n)$, $\theta(n^2)$, $\theta(n^{1.5})$, $\theta(n\lg n)$.
5. Obtain the running time for the following "for" loops.

(a)
```
for (i ← 1 to k)
{
    for (i ← 1 to k²)
    {   for (i ← 1 to k³)
        {
            p ← p*p ;
        }
    }
}
```

(b)
```
for (i ← 2 to k − 1)
{
    for (j ← 3 to k − 2)
        {
            A ← A + 2 ;
        }
}
```

CHAPTER 3

Recurrences

3.1 Introduction

A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs. To solve a recurrence relation means to obtain a function defined on the natural numbers that satisfies the recurrence. For example the worst-case running time $T(n)$ of the MERGE-SORT procedure is described by the recurrence.

$$T(n) = \begin{cases} \theta(1) & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + \theta(n) & \text{if } n > 1 \end{cases}$$

There are three methods for solving this : **Substitution method**, we guess a bound and then use mathematical induction to prove our guess correct. The **iteration method** converts the recurrence into a summation and then relies on techniques for bounding summations to solve the recurrence and the **master method** provides bounds for recurrences of the form.

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \quad \text{where } a \geq 1, b > 1 \text{ and } f(n) \text{ is a given function.}$$

When we state and solve recurrences, we often omit floors, ceilings and boundary conditions.

3.2 The Substitution Method

It involves guessing the form of the solution and then using mathematical induction to find the constants and show that the solution works. This method is powerful, but it can be applied only in cases when it is easy to guess the form of the answer. The substitution method can be used to establish either upper or lower bounds on a recurrence.

Example. *Consider the recurrence* $T(n)=T\left(\left\lfloor \frac{n}{2} \right\rfloor\right)+1$

we have to show that it is asymptotically bound by $O(\log n)$.

Solution. For $T(n)=O(\log n)$

we have to show that for some constant c.

$$T(n)\le c\log n.$$

Put this in the given recurrence equation.

$$T(n)\le c\log\left(\left\lfloor \frac{n}{2} \right\rfloor\right)+1$$

$$\le c\log\left(\frac{n}{2}\right)+1=c\log n-c\log_2 2+1$$

$$\le c\log n \text{ for } c\ge 1$$

Thus $T(n)=O(\log n)$

Example. *Consider the recurrence* $T(n)=2T\left(\left\lfloor \frac{n}{2} \right\rfloor+16\right)+n$.

We have to show that it is asymptotically bound by $O(n\log n)$.

Solution. For $T(n)=O(n\log n)$ we have to show that for some constant c.

$$T(n)\le cn\log n.$$

Put this in the given recurrence equation.

$$T(n)\le 2\left[c\left(\left\lfloor \frac{n}{2} \right\rfloor+16\right)\log\left(\left\lfloor \frac{n}{2} \right\rfloor+16\right)\right]+n$$

$$=cn\log\left(\frac{n}{2}\right)+32+n=cn\log n-cn\log 2+32+n$$

$$=cn\log n-cn+32+n=cn\log n-(c-1)n+32$$

$$\le cn\log n \quad (\text{for } c\ge 1)$$

Thus $T(n)=O(n\log n)$

Example. *Consider the recurrence*

$$T(n)=\begin{cases} 1 & n=1 \\ 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right)+n & n>1 \end{cases}$$

Find an asymptotic bound on T.

Solution. We guess the solution is $O(n\lg n)$. Thus for a constant 'c'

$$T(n)\le cn\log n.$$

Put this in the given recurrence equation.

Now $T(n)\le 2c\left\lfloor \frac{n}{2} \right\rfloor\log\left\lfloor \frac{n}{2} \right\rfloor+n$

$$\le cn\log n-cn\lg 2+n$$

$$=cn\log n-n(c\lg 2-1)$$

$$\le cn\log n \ \forall\ c\ge 1.$$

By mathematical induction, we require to show that our solution holds for the boundary conditions.

$$T(1)\le c\cdot 1\log 1=0.$$

Thus for any value of c, this will not hold, so by asymptotic definition we need to prove

$$T(n)\le cn\lg n \quad \text{for all } n\ge n_0.$$

Now $T(2)\le c\cdot 2\log 2$ $\quad T(3)\le c\cdot 3\log 3$

Thus the above relation holds for $c\ge 2$.

Hence our guess $T(n)=O(n\lg n)$ is true.

For most of the recurrences we shall examine, it is straightforward to extend boundary conditions to make the inductive assumption work for small n.

3.2.1 Making a good guess

1. If a recurrence is similar to one we have seen before, then guessing a similar solution is reasonable. For example,
   $T(n)=2T\left(\left\lfloor \frac{n}{2} \right\rfloor+17\right)+n$ which looks difficult because of added "17" in the argument to T, but this additional term cannot substantially affect the solution to the recurrence. Because, when n is large, the difference between $T\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$ and $T\left(\left\lfloor \frac{n}{2} \right\rfloor+17\right)$ is not large and both cut n nearly evenly in half. Hence, we make the guess that $T(n)=O(n\lg n)$.
2. Another way to make a good guess is to prove loose upper and lower bounds on the recurrence and then reduce the range of uncertainly.
3. There are times when we can correctly guess at an asymptotic bound on the solution of a recurrence, but somehow the math doesn't seem to work out in the induction. When we hit such a situation revising the guess by subtracting a lower order term often permits the math to go through.
4. **Changing Variables.** Sometimes, a little algebraic manipulation can make an unknown recurrence similar to one we have seem before.

Example. *Consider the following recurrence*

$T(n)=2T(\sqrt{n})+\log n$. *Solve it by changing variable.*

Solution. $T(n)=2T(\sqrt{n})+\log n$

Suppose $m=\log_2 n \Rightarrow n=2^m$

$\therefore \quad n^{1/2}=2^{m/2} \Rightarrow \sqrt{n}=2^{m/2}$

Put the values, we get

$$T(2^m)=2T(2^{m/2})+m$$

Again consider,

$$S(m)=T(2^m)$$

We have $\quad S(m)=2S\left(\frac{m}{2}\right)+m$

We know this recurrence has the solution.

$$S(m)=O(m\log m)$$

Substitute the values of m, we get

$$T(n)=S(m)=O(m\log m)=O(\lg n\lg\lg n)$$

Example. *Solve the recurrence :* $T(n)=2T(\sqrt{n})+1$. *By making a change of variables.*

Solution. $T(n)=2T(\sqrt{n})+1$

Suppose $m=\log n \Rightarrow n=2^m \Rightarrow \sqrt{n}=2^{m/2}$

Thus $T(2^m)=2T(2^{m/2})+1$

$$S(m)=T(2^m)$$

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

We know the solution to above recurrence is

$$S(m)=O(m)$$

Substituting for m, we get

$$T(n)=S(m)=O(m)=O(\lg n)$$

3.3 The Iteration Method

In iteration method the basic idea is to expend the recurrence and express it as a summation of terms dependent only on 'n' and the initial conditions.

Example. *Consider the recurrence :* $T(n)=3T\left(\left\lfloor\frac{n}{4}\right\rfloor\right)+n$

Solution. We iterate it as follows :

$$T(n)=n+3T\left(\left\lfloor\frac{n}{4}\right\rfloor\right)$$

$$=n+3\left(\left\lfloor\frac{n}{4}\right\rfloor+3T\left(\left\lfloor\frac{n}{16}\right\rfloor\right)\right)$$

$$=n+3\left(\left\lfloor\frac{n}{4}\right\rfloor+3\left(\left\lfloor\frac{n}{16}\right\rfloor+3T\left(\left\lfloor\frac{n}{64}\right\rfloor\right)\right)\right)$$

$$=n+3\left\lfloor\frac{n}{4}\right\rfloor+9\left\lfloor\frac{n}{16}\right\rfloor+27T\left(\left\lfloor\frac{n}{64}\right\rfloor\right)$$

$$\le n+\frac{3n}{4}+\frac{9n}{16}+\ldots 3^iT\left(\frac{n}{4^i}\right)$$

The series terminates when $\frac{n}{4^i}=1 \Rightarrow n=4^i$ or $i=\log_4 n$.

$$T(n)\le n+\frac{3n}{4}+\frac{9n}{16}+\frac{27n}{64}+\ldots+3^{\log_4 n}T(1)$$

$$\le n+\frac{3n}{4}+\frac{9n}{16}+\frac{27n}{64}+\ldots+3^{\log_4 n}\,\theta(1)$$

$$\le n\sum_{i=0}^{\infty}\left(\frac{3}{4}\right)^i+\theta(n^{\log_4 3}) \text{ as } 3^{\log_4 n}=n^{\log_4 3}$$

$$\le n\cdot\frac{1}{1-\frac{3}{4}}+o(n) \text{ as } \log_4 3<1 \text{ i.e., } \theta(n^{\log_4 3})=o(n)$$

$$=4n+o(n)=O(n)$$

Example. *Consider the recurrence*

$$T(n)=T(n-1)+1 \text{ and } T(1)=\theta(1). \text{ Solve it}$$

Solution. $T(n)=T(n-1)+1$

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

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

$$=T(n-4)+4=T(n-5)+1+4$$

$$=T(n-5)+5$$

$$=T(n-k)+k$$

where $k=n-1$

i.e., $\quad T(n-k)=T(1)=\theta(1)$

i.e., $\quad T(n)=\theta(1)+(n-1)=1+n-1=n=\theta(n)$.

Example. $T(n)=T\left(\frac{n}{3}\right)+n^{4/3}$. *Solve this recurrence by iteration method.*

Solution. $T(n)=T\left(\frac{n}{3}\right)+n^{4/3}=n^{4/3}+T\left(\frac{n}{3}\right)$

$$=n^{4/3}+\left(\frac{n}{3}\right)^{4/3}+T\left(\frac{n}{3^2}\right)$$

$$\because \quad T(n) = n^{4/3} + \left(\frac{n}{3}\right)^{4/3} + \left(\frac{n}{3^2}\right)^{4/3} + \ldots \left(\frac{n}{3^k}\right)^{4/3}$$

Let $k = \log_3 n$ when $\frac{n}{3^k} = 1$ *i.e.,* $3^k = n$.

$$\therefore \quad T(n) = n^{4/3} + n^{4/3}\left(\frac{1}{3^{4/3}}\right) + n^{4/3}\left(\frac{1}{3^{4/3}}\right)^2 + \ldots n^{\frac{4}{3}}\left(\frac{1}{3^{4/3}}\right)^k$$

$$= n^{4/3} \sum_{i=0}^{k} \left(\frac{1}{3^{4/3}}\right)^i$$

$$\le n^{4/3} \sum_{i=0}^{\infty} \left(\frac{1}{3^{4/3}}\right)^i \le n^{4/3} \frac{1}{1 - \frac{1}{3^{4/3}}} = O(n^{4/3}).$$

Example. Solve $T(n) = T(n-1) + \frac{1}{n}$.

Solution. By iteration method.

$$T(n) = \frac{1}{n} + T(n-1)$$

$$= \frac{1}{n} + \frac{1}{n-1} + T(n-2)$$

$$= \frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + T(n-3)$$

$$= \frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + \ldots + T(1)$$

$$= \sum_{i=0}^{n-2} \frac{1}{n-i} + T(1) \le \sum_{i=0}^{\infty} \frac{1}{n-i} + \theta(1)$$

Let $\quad n - i = x$

$\quad -di = dx$

Thus, it can be transformed into an integral.

$$\sum_{i=0}^{n} \frac{1}{n-i} = -\int_{n}^{0} \frac{dx}{x} = \log n.$$

Thus $\quad T(n) = \theta(\log n) + \theta(1) = \theta(\log n)$

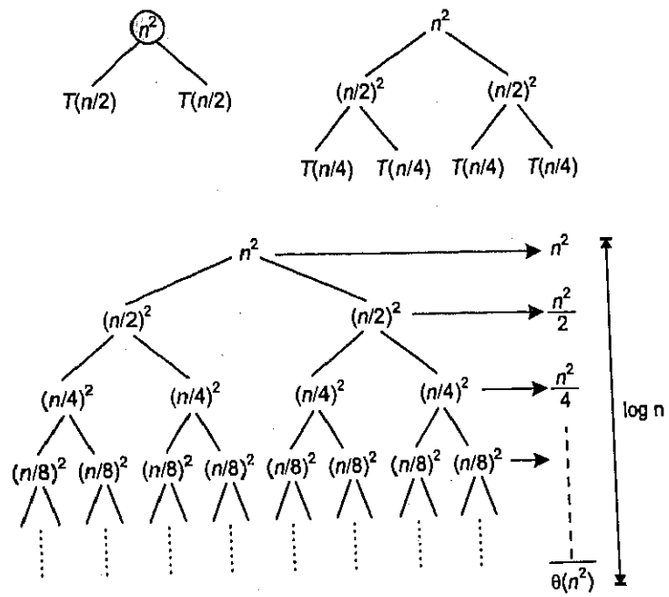
3.3.1 Recursion Tree

Recursion tree method is a pictorial representation of an iteration method, which is in the form of a tree, where at each level nodes are expanded. In general, we consider second term in recurrence as root. It is useful when divide and conquer algorithm is used.

Example. *Consider* $T(n) = 2T\left(\frac{n}{2}\right) + n^2$.

We have to obtain the asymptotic bound using recursion tree method.

Solution. The recursion tree for the above recurrence is



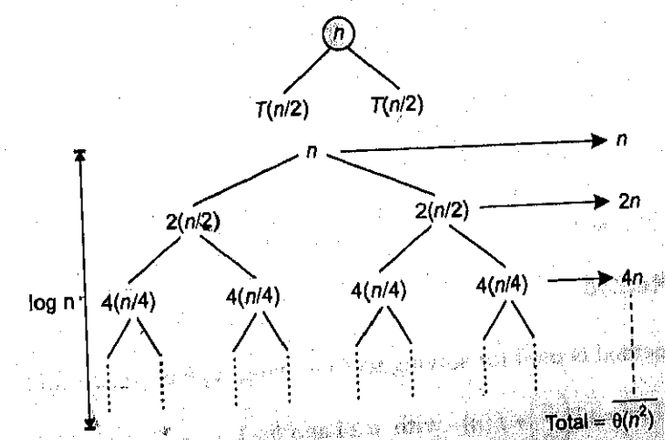
Hence solution is $\theta(n^2)$, because the values decrease geometrically, the total is at most a constant factor more than largest (first) term, and hence the solution is $\theta(n^2)$.

Example. *Consider the following recurrence*

$$T(n) = 4T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n$$

Obtain the asymptotic bound using recursion tree method.

Solution. The recursion tree for the above recurrence



We have $n+2n+4n+\ldots \log_2 n$ times

$$=n(1+2+4+\ldots \log_2 n \text{ times})$$

$$=n\frac{(2^{\log_2 n}-1)}{(2-1)}=\frac{n(n-1)}{1}=n^2-n=\theta(n^2)$$

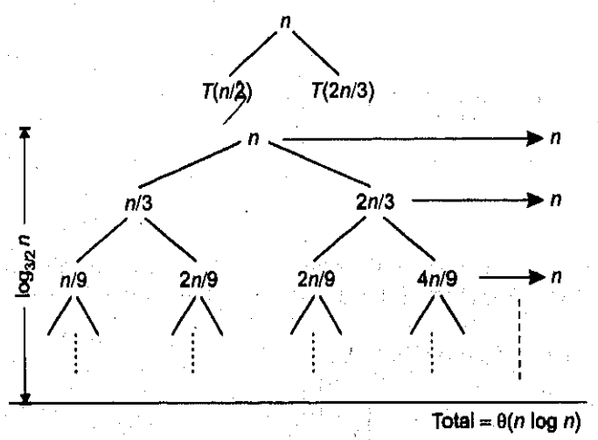
$$\therefore \quad T(n)=\theta(n^2)$$

Example. *Consider the following recurrence*

$$T(n)=T\left(\frac{n}{3}\right)+T\left(\frac{2n}{3}\right)+n$$

Obtain the asymptotic bound using recursion tree method.

Solution. The given recurrence has the following recursion tree.



When we add the values across the levels of the recursion tree, we get a value of n for every level. The longest path from the root to a leaf is

$$n \to \frac{2}{3}n \to \left(\frac{2}{3}\right)^2 n \to \ldots 1$$

Since $\left(\frac{2}{3}\right)^i n=1$ when $i=\log_{3/2} n$.

Thus the height of the tree is $\log_{3/2} n$.

$$\therefore \quad T(n)=n+n+n+\ldots+\log_{3/2} n \text{ times.}$$
$$=\theta(n \log n)$$

3.4 Master Method

The master method is used for solving the following type of recurrence.

$$T(n)=aT\left(\frac{n}{b}\right)+f(n) \quad \text{with } a\geq 1 \text{ and } b>1.$$

In this, problem is divided into 'a' subproblems, each of size $\frac{n}{b}$ where a and b are positive constants. The cost of dividing the problem and combining the results of the subproblems is described by the function $f(n)$.

The master method depends on the following theorem.

Master Theorem

Let $T(n)$ be defined on the non-negative integers by the recurrence.

$T(n)=aT\left(\frac{n}{b}\right)+f(n)$ where $a\geq 1$ and $b>1$ be constants and $f(n)$ be a function and $\frac{n}{b}$ can be interpreted as $\left\lfloor\frac{n}{b}\right\rfloor$ or $\left\lceil\frac{n}{b}\right\rceil$.

Then $T(n)$ can be bound asymptotically as follows :

1. If $f(n)=O(n^{\log_b a-\epsilon})$ for some constant $\epsilon>0$, then $T(n)=\theta(n^{\log_b a})$
2. If $f(n)=\theta(n^{\log_b a})$, then $T(n)=\theta(n^{\log_b a}\lg n)$
3. If $f(n)=\Omega(n^{\log_b a+\epsilon})$ for some constant $\epsilon>0$ and if a $f\left(\frac{n}{b}\right)\leq cf(n)$ for some $c<1$ and all sufficiently large n, then $T(n)=\theta(f(n))$, a $f\left(\frac{n}{b}\right)\leq cf(n)$ is called the "regularity" condition.

Note *It is important to note that the three cases do not cover all the possibilities for $f(n)$. There is a gap between cases 1 and 2. When $f(n)$ is smaller than $n^{\log_b a}$ but not polynomially smaller. Similarly there is a gap between cases 2 and 3 when $f(n)$ is larger than $n^{\log_b a}$ but not polynomially larger. If the function $f(n)$ falls into one of these gaps, or if the regularity condition in case 3 fails to hold, the master method cannot be used to solve the recurrence.*

Example. *Solve the recurrence using master method.*

$$T(n)=9T\left(\frac{n}{3}\right)+n$$

Solution. Compare, the given recurrence with the $T(n)=aT\left(\frac{n}{b}\right)+f(n)$

We get $a=9, \quad b=3, \quad f(n)=n$.

Now, $\quad n^{\log_b a}=n^{\log_b a}=n^2=\theta(n^2)$

$$f(n)=n=n^{\log_3 9-1}=n^{2-1}$$

Hence we can apply case 1 and the solution is $\theta(n^{\log_3 9})=\theta(n^2)$.

Example. *Solve $T(n)=T\left(\frac{2n}{3}\right)+1$ by master method.*

Solution. Here $a=1, \; b=\frac{3}{2}, \; f(n)=1$.

$$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$$

$f(n) = n^{\log_b a}$, hence case 2 applies. Thus solution is

$$T(n) = \theta(n^{\log_b a} \lg n) = \theta(\lg n).$$

Example. *Solve the recurrence* $T(n) = 3T\left(\frac{n}{4}\right) + n \lg n$ *by master method.*

Solution. Here $a = 3$, $b = 4$ $f(n) = n \lg n$.

$$n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$$

$$f(n) = \Omega(n^{\log_4 3 + \epsilon}) \quad \text{where } \epsilon = 0.2$$

Case 3 applies, now for regularity condition *i.e.*,

$$af\left(\frac{n}{b}\right) \le c f(n)$$

$$3\left(\frac{n}{4}\right)\log\left(\frac{n}{4}\right) \le \left(\frac{3}{4}\right) n \lg n = c f(n) \text{ for } c = \frac{3}{4}.$$

Therefore, the solution is

$$T(n) = \theta(n \lg n)$$

Example. *Solve* $T(n) = 16T\left(\frac{n}{4}\right) + n^3$

Solution. By master theorem

$$a = 16 \quad b = 4 \quad f(n) = n^3$$

$$n^{\log_b a} = n^{\log_4 16} = n^2$$

$$f(n) = n^3 = n^{\log_b a + \epsilon} = n^{2+1} \quad \therefore \epsilon = 1$$

Now regularity condition *i.e.*,

$$16 f\left(\frac{n^3}{4^3}\right) \le c f(n^3)$$

$$16\frac{n^3}{64} \le cn^3 \quad \text{for } c = \frac{1}{4}. \text{ So regularity condition holds.}$$

Hence by case 3, the solution is

$$T(n) = \theta(f(n)) = \theta(n^3)$$

3.4.1 Proof of the Master theorem

The proof is in two parts. The first part analyzes the recurrence $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ under the simplifying assumption that $T(n)$ is defined only on the exact power of $b > 1$ *i.e.*, for $n = 1, b, b^2, b^3 \ldots$. The second part shows how the analysis can be extended to all positive integers n.

Proof for exact powers

The first part of the proof of the master theorem analysis the recurrence $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ under the assumption that n is an exact power of $b > 1$, where b need not be an integer. The analysis is broken into three lemmas.

The first lemma reduces the problem into the problem of evaluating an expression that contains a summation. The second determines bounds on this summation and the third lemma puts the first two together to prove a version of the master theorem in which n is an exact power of b.

Lemma 1

Let $a \ge 1$ and $b > 1$ be constants and let $f(n)$ be a non-negative function defined on exact powers of b. Then define $T(n)$ on exact powers of b by the recurrence.

$$T(n) = \begin{cases} \theta(1) & \text{if } n = 1 \\ aT\left(\frac{n}{b}\right) + f(n) & \text{if } n = b^i \end{cases}$$

where i is a positive integer, then $T(n) = \theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^j}\right)$

Proof. $T(n) = f(n) + aT\left(\frac{n}{b}\right)$

$$= f(n) + af\left(\frac{a}{b}\right) + a^2 T\left(\frac{n}{b^2}\right)$$

$$= f(n) + af\left(\frac{n}{b}\right) + a^2 f\left(\frac{n}{b^2}\right) + \ldots a^i T\left(\frac{n}{b^i}\right)$$

$$\because \left(\frac{n}{b^i} = 1 \Rightarrow n = b^i \Rightarrow i = \log_b n\right)$$

$$\therefore \quad T(n) = f(n) + af\left(\frac{n}{b}\right) + a^2 f\left(\frac{n}{b^2}\right) + \ldots + a^{\log_b n} T(1)$$

$$= f(n) + af\left(\frac{n}{b}\right) + a^2 f\left(\frac{n}{b^2}\right) + \ldots + a^{\log_b n - 1} T\left(\frac{n}{b^{\log_b n - 1}}\right) + \theta(n^{\log_b a})$$

$$(\because a^{\log_b n} = n^{\log_b a})$$

$$T(n) = \sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^j}\right) + \theta(n^{\log_b a})$$

which completes the proof.

Lemma 2

Let $a \ge 1$ and $b > 1$ be constants and let $f(n)$ be a non-negative function defined on exact powers of b. A function $g(n)$ defined over exact powers of b by

$$g(n) = \sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^j}\right) \quad \ldots(A)$$

it can be bounded asymptotically for exact powers of b, as follows :

1. If $f(n)=O(n^{\log_b a-\epsilon})$ for some constant $\epsilon>0$ then
   $g(n)=O(n^{\log_b a})$
2. If $f(n)=\theta(n^{\log_b a})$ then $g(n)=\theta(n^{\log_b a}\lg n)$
3. If $af\left(\frac{n}{b}\right)\le c\,f(n)$ for some constant $c<1$ and all $n\ge b$ then $g(n)=\theta(f(n))$

Proof.

Case 1

We have $f(n)=O(n^{\log_b a-\epsilon})$ implies that

$$f\left(\frac{n}{b^j}\right)=O\left(\left(\frac{n}{b^j}\right)^{\log_b a-\epsilon}\right)$$

Put this in equation (A), we get

$$g(n)=O\left(\sum_{j=0}^{\log_b n-1} a^j\left(\frac{n}{b^j}\right)^{\log_b a-\epsilon}\right)$$

Now
$$\sum_{j=0}^{\log_b n-1} a^j\left(\frac{n}{b^j}\right)^{\log_b a-\epsilon}=n^{\log_b a-\epsilon}\sum_{j=0}^{\log_b n-1}\left(\frac{ab^\epsilon}{b^{\log_b a}}\right)^j$$

$$=n^{\log_b a-\epsilon}\sum_{j=0}^{\log_b n-1}(b^\epsilon)^j$$

$$=n^{\log_b a-\epsilon}\left(\frac{b^{\epsilon\log_b n}-1}{b^\epsilon-1}\right)$$

$$=n^{\log_b a-\epsilon}\left(\frac{n^\epsilon\log_b b-1}{b^\epsilon-1}\right)$$

$$=n^{\log_b a-\epsilon}\left(\frac{n^\epsilon-1}{b^\epsilon-1}\right)$$ where b and ϵ are constants.

Thus, expression reduces to

$$n^{\log_b a-\epsilon}O(n^\epsilon)=O(n^{\log_b a})$$

$$\therefore\quad g(n)=O(n^{\log_b a})$$

For case 2

We have $f(n)=\theta(n^{\log_b a})$

So $f\left(\frac{n}{b^j}\right)=\theta\left(\left(\frac{n}{b^j}\right)^{\log_b a}\right)$

Put this value in equation (A), we get

$$g(n)=\theta\left(\sum_{j=0}^{\log_b n-1} a^j\left(\frac{n}{b^j}\right)^{\log_b a}\right)$$

We bound the summation with in the θ as in case 1, but this time we do not obtain a geometric series. Instead, we discover that every term of the smmation is the same.

$$\sum_{j=0}^{\log_b n-1} a^j\left(\frac{n}{b^j}\right)^{\log_b a}=n^{\log_b a}\sum_{j=0}^{\log_b n-1}\left(\frac{a}{b^{\log_b a}}\right)^j$$

$$=n^{\log_b a}\sum_{j=0}^{\log_b n-1}(1)^j=n^{\log_b a}\log_b n$$

Put this, we get $g(n)=\theta(n^{\log_b a}\log_b n)$

$$g(n)=\theta(n^{\log_b a}\lg n)$$

Case 3

Since $f(n)$ appears in the definition of $g(n)$ and all terms of $g(n)$ are non-negative, we can conclude that $g(n)=\Omega(f(n))$ for exact powers of b. Under the assumption that $af\left(\frac{n}{b}\right)\le c\,f(n)$ for some constant $c<1$ and $n\ge b$, we have

$$a^j f\left(\frac{n}{b^j}\right)\le c^j f(n)$$

Put this in equation (A) and simplifying we get a geometric series but this series has decreasing terms.

$$g(n)\le\sum_{j=0}^{\log_b n-1} a^j f\left(\frac{n}{b^j}\right)$$

$$\le\sum_{j=0}^{\log_b n-1} c^j f(n)\le f(n)\sum_{j=0}^{\infty}c^j$$

$$=f(n)\left(\frac{1}{1-c}\right)=O(f(n))$$

Since c is a constant. Thus we can conclude that $g(n)=\theta(f(n))$ for exact powers of b which completes the proof.

We can now prove the case in which n is an exact power of b.

Lemma 3

Let $a\ge1$ and $b>1$ be constants and let $f(n)$ be a non-negative function defined on exact powers of b. Define $T(n)$ on exact powers of b by the recurrence.

$$T(n)=\begin{cases}\theta(1) & \text{if } n=1\\ aT\left(\frac{n}{b}\right)+f(n) & \text{if } n=b^i\end{cases}$$

where i is a positive integer. Then $T(n)$ can be bounded asymptotically for exact powers of b as follows :

1. If $f(n)=O(n^{\log_b a-\epsilon})$ for some constant $\epsilon>0$ then $T(n)=\theta(n^{\log_b a})$

2. If $f(n)=\theta(n^{\log_b a})$ then $T(n)=\theta(n^{\log_b a}\lg n)$
3. If $f(n)=\Omega(n^{\log_b a+\epsilon})$ for some constant $\epsilon>0$ and if $af\left(\frac{n}{b}\right)\le cf(n)$, for some constant $c<1$ and all sufficiently large 'n' then $T(n)=\theta(f(n))$

Proof. We use the bounds in lemma 2 to evaluate the summation

For case 1, we have

$$T(n)=\theta(n^{\log_b a})+O(n^{\log_b a})$$
$$=\theta(n^{\log_b a})$$

and for case 2

$$T(n)=\theta(n^{\log_b a})+\theta(n^{\log_b a}\lg n)$$
$$=\theta(n^{\log_b a}\lg n)$$

For case 3, the condition $af\left(\frac{n}{b}\right)\le cf(n)$ implies

$$f(n)=\Omega(n^{\log_b a+\epsilon})\text{ consequently}$$
$$T(n)=\theta(n^{\log_b a})+\theta(f(n))$$
$$=\theta(f(n))$$

Floor and Ceilings

To complete the proof of the master theorem, we must now extend our analysis to the situation in which floors and ceilings are used in the master recurrence. So that recurrence is defined for all integers.

Obtaining a lower bound on

$$T(n)=aT\left(\left\lceil\frac{n}{b}\right\rceil\right)+f(n) \qquad ...(1)$$

and an upper bound on

$$T(n)=aT\left(\left\lfloor\frac{n}{b}\right\rfloor\right)+f(n) \qquad ...(2)$$

and $\left\lfloor\frac{n}{b}\right\rfloor\le\frac{n}{b}$ and $\left\lceil\frac{n}{b}\right\rceil\ge\frac{n}{b}$.

To iterate the recurrence (1), we obtain a sequences of recursive invocations on the arguments

n

$\lceil n/b\rceil$

$\lceil\lceil n/b\rceil/b\rceil$

$\lceil\lceil\lceil n/b\rceil/b\rceil/b\rceil$

the ith element in the sequence by n_i is defined as

$$n_i=\begin{cases} n & \text{if } i=0\\ \lceil n_{i-1}/b\rceil & \text{if } i>0\end{cases}$$

The first aim is to determine the number of iterations k such that n_k is a constant. Using the inequality $\lceil x\rceil\le x+1$, we get

$$n_0\le n;$$
$$n_1\le\frac{n}{b}+1$$
$$n_2\le\frac{n}{b^2}+\frac{1}{b}+1;$$
$$n_3\le\frac{n}{b^3}+\frac{1}{b^2}+\frac{1}{b}+1$$

In general $n_i\le\frac{n}{b^i}+\sum_{j=0}^{i-1}\frac{1}{b^j}$

$$\le\frac{n}{b^i}+\frac{b}{b-1}$$

and thus, when $i=\lfloor\log_b n\rfloor$, we obtain $n_i\le b+\frac{b}{b-1}=O(1)$

We can now iterate recurrence, obtaining

$$T(n)=f(n_0)+aT(n_1)$$
$$=f(n_0)+af(n_1)+a^2T(n_2)$$
$$\le f(n_0)+a\,f(n_1)+a^2f(n_2)+\ldots+a^{\lfloor\log_b n\rfloor-1}f\left(n_{\lfloor\log_b n\rfloor-1}\right)+a^{\lfloor\log_b n\rfloor}T\left(n_{\lfloor\log_b n\rfloor}\right)$$
$$=\theta(n^{\log_b a})+\sum_{j=0}^{\lfloor\log_b n\rfloor-1}a^jf(n_j)$$

Now $g(n)=\sum_{j=0}^{\lfloor\log_b n\rfloor-1}a^jf(n_j)$ ...(A)

We can evalute this summation like lemma 2. Starting with case 3, if $af\left(\left\lceil\frac{n}{b}\right\rceil\right)\le c\,f(n)$ for $n>b+\frac{b}{b-1}$ where $c<1$ is a constant then it follows that $a^jf(n_j)\le c^jf(n)$. Therefore, the sum in equation (A) can be evaluated just as in lemma 2.

For case 2, we have $f(n)=\theta(n^{\log_b a})$. If we can show that

$$f(n_j)=O\left(\frac{n^{\log_b a}}{a^j}\right)=O\left(\left(\frac{n}{b^j}\right)^{\log_b a}\right),\text{ then the proof is same.}$$

Observe that $j\le\lfloor\log_b n\rfloor$ implies $\frac{b^j}{n}\le 1$. Then bound $f(n)=O(n^{\log_b a})$ implies that there exists a constant $c>0$ such that for sufficiently large n_j.

$$f(n_j) \le c\left(\frac{n}{b^j}+\frac{b}{b-1}\right)^{\log_b a}$$

$$= c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1+\left(\frac{b^j}{n}\cdot\frac{b}{b-1}\right)\right)^{\log_b a}$$

$$\le c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1+\frac{b}{b-1}\right)^{\log_b a}$$

$$\le O\left(\frac{n^{\log_b a}}{a^j}\right)$$

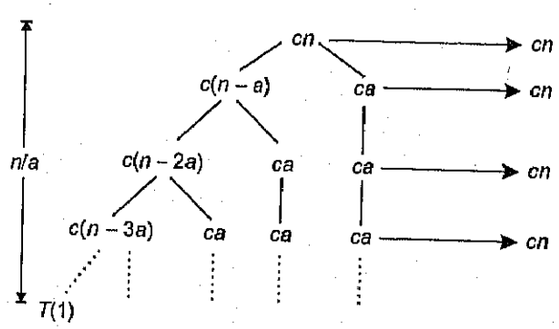
Since $c\left(1+\frac{b}{b-1}\right)^{\log_b a}$ is constant.

Thus case 2 is proved. The proof of case 1 is almost identical. The key is to prove the bound $f(n_j)=O(n^{\log_b a-\epsilon})$ which is similar to the corresponding proof of case 2.

Example. *Use a recursion tree to give an asymptotically tight solution to the recurrence*

$$T(n)=T(n-a)+T(a)+cn \text{ where } a\ge 1 \text{ and } c>0 \text{ are constants.}$$

Solution.



Suppose the tree ends when $n-ka=1$

$$k=(n-1)/a \approx \frac{n}{a} \text{ levels.}$$

Total sum is $T(n)=\left(\frac{n}{a}\right)\cdot cn=O(n^2)$

Now verify that $T(n)=O(n^2)$ *i.e.*, $T(n)<dn^2$

Assume that $T(k)<dk^2 \quad \forall\, k<n$.

Now

$$T(n)=T(n-a)+T(a)+cn$$
$$<d(n-a)^2+d(a^2)+cn$$
$$=dn^2-2adn+2da^2+cn$$
$$=dn^2-(2adn-2da^2-cn)$$

For $T(n)<dn^2$

$$2adn-2da^2-cn>0$$

e.g., $$d=\frac{(c+1)}{2a}$$

we get, $$T(n)<dn^2-(n-ac-a)$$

$\therefore$ $T(n)<dn^2$ for large value of n.

i.e., $T(n)=O(n^2)$.

Example. *"The recurrence $T(n)=7T\left(\frac{n}{2}\right)+n^2$ describes the running time of an algorithm A. A competing algorithm A′ has a running time of $T'(n)=aT'\left(\frac{n}{4}\right)+n^2$. What is the largest integer value for 'a' such that A′ is asymptotically faster than A?"*

Solution. $T(n)=7T\left(\frac{n}{2}\right)+n^2$

The master method gives us $a=7,\ b=2\ \ f(n)=n^2$

$$n^{\log_b a}=n^{\log_2 7}\approx n^{2.8}$$

It is the first case because $f(n)=n^2=O(n^{\log 7-\epsilon})$

where $\epsilon=0.8$ which gives $T(n)=\theta(n^{\log 7})$

The other recurrence $T'(n)=aT'\left(\frac{n}{4}\right)+n^2$ is a bit more difficult to analyse because when 'a' is unknown it is not so easy to say which of the three cases applies in the master method.

But $f(n)$ is same in both algorithms which leads us to try the first case. $\log_4 a$ must be log7, which happens when $a=48$.

In other words, A' is asymptotically faster than A as long as $a<48$. The other cases in the master method do not apply for $a>48$.

Hence A' is asymptotically faster than A up to $a=48$.

Example. *Give asymptotic upper and lower bounds for $T(n)$ in each of the following recurrences. Assume that $T(n)$ is constant for $n\le 2$. Make your bounds as tight as possible, and justify your answer.*

(a) $T(n)=2T\left(\frac{n}{2}\right)+n^3$

(b) $T(n)=T\left(\frac{9n}{10}\right)+n$

(c) $T(n)=16T\left(\frac{n}{4}\right)+n^2$

(d) $T(n)=7T\left(\frac{n}{3}\right)+n^2$

(e) $T(n)=7T\left(\frac{n}{2}\right)+n^2$

(f) $T(n)=2T\left(\frac{n}{4}\right)+\sqrt{n}$

(g) $T(n)=T(n-1)+n$

(h) $T(n)=T(\sqrt{n})+1$

Solution. (a) $T(n)=2T\left(\frac{n}{2}\right)+n^3$

Here $a=2 \quad b=2 \quad f(n)=n^3$

$$n^{\log_b a}=n$$

$$f(n)=\Omega(n^{1+\epsilon}) \text{ for } \epsilon=2.$$

$$2f\left(\frac{n}{2}\right)=cf(n) \text{ if } c=\frac{1}{4}$$

Thus from case 3 $T(n)=\theta(n^3)$

(b) $T(n)=T\left(\frac{9n}{10}\right)+n$

$a=1 \quad b=\frac{10}{9} \quad f(n)=n$

$$n^{\log_b a}=n^0 \quad \therefore \quad f(n)=\Omega(n^{0+\epsilon}) \text{ with } \epsilon=1$$

$$f\left(\frac{9n}{10}\right)\le cf(n) \text{ for large values of } n \text{ if } \frac{9}{10}<c<1$$

Thus $T(n)=\theta(n)$

(c) $T(n)=16T\left(\frac{n}{4}\right)+n^2$

$a=16 \quad b=4 \quad f(n)=n^2$

$$n^{\log_b a}=n^2 \quad \text{and} \quad f(n)=n^{\log_b a}$$

$\therefore$ By case 2. $\quad T(n)=\theta(n^2\log n)$

(d) $T(n)=7T\left(\frac{n}{3}\right)+n^2$

$a=7 \quad b=3 \quad f(n)=n^2$

$$n^{\log_b a}=n^{\log_3 7} \quad \text{we know } 1<\log_3 7<2$$

$$f(n)=\Omega(n^{\log_3 7+\epsilon}) \text{ where } \epsilon=2-\log_3 7$$

$\therefore \quad T(n)=\theta(n^2)$

(e) $T(n)=7T\left(\frac{n}{2}\right)+n^2$

$a=7 \quad b=2 \quad f(n)=n^2$

$$n^{\log_b a}=n^{\log 7}\approx n^{2.8} \quad f(n)=O(n^{\log 7-\epsilon}) \text{ where } \epsilon\approx 0.8$$

$\therefore \quad T(n)=\theta(n^{\log 7})$

(f) $T(n)=2T\left(\frac{n}{4}\right)+\sqrt{n}$

$a=2 \quad b=4 \quad f(n)=\sqrt{n}$

$$n^{\log_b a}=\sqrt{n} \quad f(n)=n^{\log_b a}$$

$\therefore \quad T(n)=\theta(\sqrt{n}\lg n)$

(g) $T(n)=T(n-1)+n$

In this case the master method does not apply, but by recussion tree, we obtain $T(n)=\theta(n^2)$

(h) $T(n)=T(\sqrt{n})+1$

In this case the master method does not apply but by recursion tree, we obtain $T(n)=\theta(\lg\lg n)$.

Example. *Solve* $T(n)=2T\left(\frac{n}{2}\right)+\frac{n}{\lg n}$.

Solution. $T(n)=2T\left(\frac{n}{2}\right)+\frac{n}{n\lg n}$

$$\le 2T\left(\frac{n}{2}\right)-n\lg n.$$

$$T(n)=2T\left(\frac{n}{2}\right)-\theta(n\lg n)$$

This recurrence cannot be solved by master method, so using the iteration method, we get

$$2T\left(\frac{n}{2}\right)=\sum_{i=0}^{k-1}2^i 2^{k-i}\log_2(2^{k-i})$$

$$=2^k\sum_{i=0}^{k-1}(k-i)=2^{k-1}k(k+1)$$

As $\quad k=\log n.$

$$=O(n\log^2 n)$$

So $T(n)=O(n\log^2 n)-\theta(n\lg n)$

$$=O(n\log^2 n)$$

Because the term $O(n\log^2 n)$ is greater than $\theta(n\lg n)$.

Example. *Solve the recurrence*

$$T(1)=1$$

$$T(n)=3T\left(\frac{n}{2}\right)+2n^{1.5}$$

Solution. We have $T(n)=3T\left(\frac{n}{2}\right)+2n^{1.5}$

Divide by 2, we get

$$\frac{1}{2}T(n)=\frac{3}{2}T\left(\frac{n}{2}\right)+n^{1.5}$$

Let $\frac{T(n)}{2} = S(n)$

$$S(n) = 3S\left(\frac{n}{2}\right) + n^{1.5} \quad ...(A)$$

and $S(1) = \frac{1}{2} = 0.5$

Solve the recurrence equation (3) with the help of master theorem

Here $a = 3$, $b = 2$ $f(n) = n^{1.5}$

$$n^{\log_b a} = n^{\log_2 3} \approx n^{1.59}$$

$$f(n) = n^{\log_b a + \epsilon} = (n^{1.59-0.9}) \text{ i.e., } \epsilon = .09$$

By case 1 : $S(n) = \theta(n^{1.59}) = O(n^{\log_2 3})$

Hence $T(n) = 2\,S(n) = O(n^{\log_2 3})$

$\therefore$ $T(n) = O(n^{\log_2 3})$

Exercise

1. Consider the recurrence

$$T_n = 3T_{n-1} + 15$$
$$T_0 = 0$$

Guess the solution, and prove it by induction.

2. Consider the recurrence

$$T(n) = 14T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n^2$$

Find the asymptotic bound.

3. Using master theorem find the asymptotic bound for the following recurrences :

(a) $T(n) = 2T\left(\frac{n}{2}\right) + n^3$ (b) $T(n) = 5T\left(\frac{n}{4}\right) + n^2$

(c) $T(n) = 7T\left(\frac{n}{2}\right) + O(n^2)$ (d) $T(n) = 2T\left(\frac{n}{2}\right) + O(n^2)$

(e) $T(n) = 2T\left(\frac{n}{2}\right) + O(\sqrt{n})$ (f) $T(n) = 4T\left(\frac{n}{4}\right) + O(n^2)$

4. Can the master method be applied to the recurrence. $T(n) = 4T\left(\frac{n}{3}\right) + n^2 \lg n$? Why or why not ? **Give** an asymptotic upper bound for this recurrence.

5. Use a recursion tree to give an asymptotically tight solution to the recurrence.

$$T(n) = T(\alpha n) + T((1-\alpha)n) + cn,$$

where α is a constant in the range $0 < \alpha < 1$ and $c < 0$ is also a constant.

CHAPTER 4

Analysis of Simple Sorting Algorithms

One of the fundamental problems of computer science is ordering a list of items. There's a surplus of solutions to this problem, known as **sorting algorithms**. Some sorting algorithms are simple, such as the bubble sort. Others, such as the quick sort, are extremely complicated, but produce lightning-fast results.

The common sorting algorithms can be divided into two classes by the complexity of their algorithms. Algorithmic complexity is generally written in a form known as Big-O notation, where the 'O represents the complexity of the algorithm and a value n represents the size of the set the algorithm is run against.

For example, $O(n)$ means that an algorithm has a linear complexity. In other words, it takes ten times longer to operate on a set of 100 items than it does on a set of 10 items ($10*10 = 100$). If the complexity was $O(n^2)$ (quadratic complexity), then it would take 100 times longer to operate on a set of 100 items than it does on a set of 10 items.

The two classes of sorting algorithms are $O(n^2)$, which includes the bubble, insertion, selection, and shell sorts ; and $O(n \log n)$ which includes the heap, merge, and quick sorts.

In addition to algorithmic complexity, the speed of the various sorts can be compared with empirical data. Since the speed of a sort can vary greatly depending on what data set it sorts, accurate empirical results require several runs of the sort be made and the results averaged together.

Here, we are going to look at three simple sorting techniques : *Bubble Sort, Selection Sort,* and *Insertion Sort.*

4.1 Bubble Sort

Bubble sort, also known as exchange sort, is a simple sorting algorithm. It works by repeatedly stepping through the list to be sorted, comparing two items at a time and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which means the list is sorted. The algorithm gets its name from the way smaller elements "bubble" to the top (*i.e.*, the beginning) of the list via the swaps. Because it only uses comparisons to operate on elements, it is a **comparison sort**. This is the easiest comparison sort to implement.

Compare each element (except the last *one*) with its neighbour to the right

- If they are out of order, swap them
- This puts the largest element at the very end
- The last element is now in the correct and final place

Compare each element (except the last *two*) with its neighbour to the right

- If they are out of order, swap them
- This puts the second largest element next to last
- The last two elements are now in their correct and final places

Compare each element (except the last *three*) with its neighbour to the right. Continue as above until you have no unsorted elements on the left.

The bubble sort is generally considered to the most inefficient sorting algorithm in common usage. Under best-case conditions (the list is already sorted), the bubble sort can approach a constant $O(n)$ level of complexity. General-case is an extremely bad $O(n^2)$.

```
Bubble Sort (A)
for i ← 1 to length [A]
    for j ← length [A] downto i + 1
        if A[j] < A[j − 1]
            exchange (A[j], A[j − 1])
```

The outer loop is executed $n-1$ times. Each time the outer loop is executed, the inner loop is executed. Inner loop executes $n-1$ times at first, linearly dropping to just once. On average, inner loop executes about $n/2$ times for each execution of the outer loop. In the inner loop, the comparison is always done (constant time), the swap might be done (also constant time). Thus result is $n * n/2 * k$, that is $O(n^2)$.

Example. *Illustrate the operation of Bubble sort on the array* $A = \langle 5,2,1,4,3,7,6 \rangle$.

Solution. $A = \langle 5,2,1,4,3,7,6 \rangle$

Here length $[A] = 7$

$i = 1$ to 7 and $j = 7$ to 2

$i = 1, \quad j = 7$

$A[7] = 6$ and $A[6] = 7$. So $A[7] < A[6]$

Now exchange $(A[7], A[6])$

i.e., $A[\ ] =$

5	2	1	4	3	6	7

Now, $i = 1, j = 6$ then $A[6] = 6$

$A[5] = 3$ and $A[5] < A[6]$

Now, $i = 1, j = 5$ then $A[5] = 3$

$A[4] = 4$ and $A[5] < A[4]$

So, exchange $(A[5], A[4])$

and $A[\ ] =$

5	2	1	3	4	6	7

Now, $i = 1, j = 4$ then $A[4] = 3$

$A[3] = 1$ and $A[4] > A[3]$

Now, $i = 1, j = 3$ then $A[3] = 1$

$A[2] = 2$ and $A[3] < A[2]$

So, exchange $(A[3], A[2])$

then $A[\ ] =$

5	1	2	3	4	6	7

Now, $i = 1, j = 2$ then $A[2] = 1$

$A[1] = 5$ and $A[2] < A[1]$

So, exchange $(A[2], A[1])$

then $A[\ ] =$

1	5	2	3	4	6	7

Now, $i = 1, j = 7$ then $A[7] = 7$

$A[6] = 6$ and $A[7] > A[6]$. No exchange

Similarly, $i = 2, j = 6, 5, 4$. No change

then $i = 2, \ j = 3$

$A[3] = 2$

$A[2] = 5$ and $A[3] < A[2]$

So, exchange $(A[3], A[2])$

nd $A[\ ] =$

1	2	5	3	4	6	7

Now, $i = 3, j = 7, 6, 5$ No change

en $i = 3, j = 4$

$A[4] = 3$

$A[3] = 5$ and $A[4] < A[3]$

So, exchange $(A[4], A[3])$

en $A[\ ] =$

1	2	3	5	4	6	7

Now $i=4$, $j=7,6$ No change

Now $i=4$, $j=5$ then $A[5]=4$

$A[4]=5$ and $A[5]<A[4]$

So exchange $(A[5], A[4])$

and $A[\,]=$

1	2	3	4	5	6	7

is the sorted array.

Example. (*a*) *Let A' denote the output of the procedure. To prove that it is correct, we need to prove that it terminates and* $A'[1] \le A'[2] \le \ldots \le A'[n]$.

What else must be proved to show that it actually sorts ?

(*b*) *State explicitly a loop invariant for the for loop in lines 2-4, and prove that it holds for the loop.*

(*c*) *Using the termination condition proved in the last part, state a loop invariant for the for loop in lines 1-4 that will allow you to prove the property as stated in the above part a.*

(*d*) *What is the worst-case running time for bubble sort ? How does it compare to that of the insertion sort ?*

Solution. (*a*) We need to show that the elements of A' form a permutation of the elements of A, *i.e.*, they only come from A.

(*b*) The loop invariant may be stated as follows :

At the start of each iteration of the for loop of lines 2 – 4,

$$A[j]=\min\{A[k] \mid j \le k \le n\}$$

and the sub array $A[j..n]$ is a permutation of the values that were in $A[j..n]$ at the time that the loop started.

Now the proof :

- Initially, $j=n$, and the sub array $A[j..n]$ consists of single elements $A[n]$. Thus, the loop invariant trivially holds.
- For the maintenance part, consider an iteration for a given value of $j=j_0$, by the loop invariant, $A[j_0]$ is the smallest value in $A[j_0..n]$. Lines 3-4 exchange $A[j_0]$ and $A[j_0-1]$ if $A[j_0]$ is less than $A[j_0-1]$, and so $A[j_0-1]$ will be the smallest value in $A[j_0-1..n]$ afterward.

  Since the only change to the subarray $A[j_0-1..n]$ is the possible exchange, and the subarray $A[j_0..n]$ is a permutation of the values that were in $A[j_0..n]$ at the time that the loop started, we see that $A[j_0-1..n]$ is a permutation of the values that were in $A[j_0-1..n]$ at the time that the loop started. Decrementing j to j_0-1 for the next iteration maintains the invariant.
- Finally, the loop terminates when j reaches i. By the statement of the loop invariant,

  $$A[i]=\min\{A[k] \mid i \le k \le n\}$$

  and $A[i..n]$ is a permutation of the values that were in $A[i..n]$ at the time that the loop started.

(*c*) We can have the following loop invariant for the loop in lines 1 – 4.

At the start of each iteration of the this for loop, the subarray $A[1..i-1]$ consists of the $i-1$ smallest values originally in $A[1..n]$, in sorted order, and $A[i..n]$ consists of the $n-i+1$ remaining values originally in $A[1..n]$.

Now the proof :

- Before the first iteration of the loop, $i=1$. The subarray $A[1..i-1]$ is empty, and so the loop invariant trivially holds.
- Consider an iteration for a given value of $i=i_0$. By the loop invariant, $A[1..i_0-1]$ consists to the i_0 smallest values in $A[1..n]$, in sorted order.

  The above part *b* showed that after executing the for loop of lines 2 – 4, $A[i_0]$ is the smallest value in $A[i_0..n]$, and so $A[1..i_0]$ is now the i_0 smallest values originally in $A[1..n]$, in sorted order. Morever, since the for loop of lines 2 – 4 permutes $A[i_0..n]$, the subarray $A[i_0+1..n]$ consists of the $n-i_0$ remaining values originally in $A[1..n]$. Increment of i to i_0+1 makes the loop invariant holds at the beginning of the next loop.
- Finally, the for loop of lines 1 – 4 terminates when $i=n+1$, so that $i-1=n$. By the statement of the loop invariant, $A[1..i-1]$ is the entire array $A[1..n]$, and it consists of the original array $A[1..n]$, in sorted order.

(*d*) The running time depends on the number of iteration of the for loop of lines 2-4. For a given value of i, this loop makes $n-i$ iterations, and $i \in [1, n]$

Thus the total number of iterations is

$$B(n)=\sum i=1^{n}(n-i)=\frac{1}{2}n(n-1).$$

4.2 Selection Sort

The idea of selection sort is rather simple : we repeatedly find the next largest (or smallest) element in the array and move it to its final position in the sorted array. Assume that we wish to sort the array in increasing order, *i.e.*, the smallest element at the beginning of the array and the largest element at the end. We begin by selecting the largest element and moving it to the highest index position. We can do this by swapping the element at the highest index and the largest element. We then reduce the effective size of the array by the one element and repeat the process on the smaller sub array. The process stops when the effective size of the array becomes 1 (an array of 1 element is already sorted). Thus, the selection sort works by selecting the smallest unsorted item remaining in the list, and then swapping it with the item in the next position to be filled. The selection sort has a complexity of $O(n^2)$.

```
Selection Sort (A)
n ← length [A]
    for j ← 1 to n − 1
        smallest ← j
        for i ← j + 1 to n
            if A[i] < A [smallest]
                then smallest ← i
        exchange (A[j], A [smallest])
```

Selection sort is very easy to analyze since none of the loops depends on the data in the array. Selecting the lowest elements requires scanning all n elements (this takes $n-1$ comparisons) and then swapping it into the first position. Finding the next lowest element requires scanning the remaining $n-1$ elements and so on, for a total of $(n-1)+(n-2)+\ldots+2+1=\Theta(n^2)$ comparisons. Each of these scans requires one swap for a total of $n-1$ swaps (the final element is already in place). Thus, the comparisons dominate the running time, which is $\Theta(n^2)$.

Example. *Sort the following array using selection sort : $A[\,]=\langle 5,2,1,4,3\rangle$.*

Solution.

A[ ] =

1	2	3	4	5
5	2	1	4	3

Here $n=5$

For $j=1$ to 4.

$j=1$, smallest = 1

For $i=2$ to 5

$i=2$, smallest = 1

$A[2]=2$ $A[1]=5$ $A[2]<A[1]$

then smallest = 2

Now $i=3$, smallest = 2

$A[3]=1$ $A[2]=2$ $A[3]<A[2]$

then smallest = 3

Now $i=4$ smallest = 3

$A[4]=4$

$A[3]=1$ $A[4]>A[3]$ No change

Now $i=5$, smallest = 3

$A[5]=3$

$A[3]=1$ $A[5]>A[3]$ So, No change

then

exchange ($A[1]$, A[smallest])

i.e., exchange (5, 1)

Now

A[ ] =

1	2	5	4	3

Now $j=2$, smallest = 2

For $i=3$ to 5

Now $i=3$, smallest = 2

$A[3]=5$

$A[2]=2$ $A[3]>A[2]$ No change

Now $i=4$, smallest = 2. No change

$i=5$, smallest = 2. No change

Now $j=3$, smallest = 3

For $i=4$ to 5

$i=4$ smallest = 3

$A[4]=4$

$A[3]=5$ $A[4]<A[3]$ then smallest = 4

Now $i=5$ smallest = 4

$A[5]=3$

$A[4]=4$ $A[5]<A[4]$ then smallest = 5

Now exchange [$A[3]$, $A[5]$]

then

A[ ] =

1	2	3	4	5

Now $j=4$, smallest = 4, $i=5$

$A[5]=5$

$A[4]=4$ $A[5]>A[4]$. No change

Hence sorted array is

1	2	3	4	5

Example. *Write a pseudocode for the Selection sort algorithm. What loop invariant does it maintain ? Why does it need to run for only the first $n-1$ elements, rather than for all n elements ? Give the best and worst-case analysis of this sort.*

Solution. The pseudocode for selection sort is

```
Selection sort (A)
1.  n ← length [A]
2.      for j ← 1 to n − 1
3.          smallest ← j
4.              for i ← j + 1 to n
5.                  if A[i] < A [smallest]
6.                      then smallest ← i
7.              exchange (A[j], A [smallest])
```

The algorithm maintains the loop invariant that at the start of each iteration of the for loop between line 2 and line 7, the sub array $A[1..j-1]$ consists of the $j-1$ smallest elements in the array $A[1..n]$, and this sub array is in sorted order.

Morever, after the first $n-1$ elements, according to the previous invariant, the sub array $A[1..n-1]$ contains the smallest $n-1$ elements, sorted. Hence, element $A[n]$ must be the largest element.

Regarding the running time of the algorithm, it is easy to see that, for all cases, we have to do the following amount of work :

$$S(n)=\sum_{j=1}^{n-1}\sum_{i=j+1}^{n}=\sum j=1^{n-1}(n-j)=n(n-1)-\sum_{j=1}(n-1)$$

$$=n(n-1)-\frac{1}{2}n(n-1)=\frac{1}{2}n(n-1)=\Theta(n^2).$$

It yields a 60% performance improvement over the bubble sort, but the insertion sort is over twice as fast as the bubble sort and is just as easy to implement as the selection sort. In short, there really isn't any reason to use the selection sort – use the insertion sort instead.

4.3 Insertion Sort

The insertion sort works just like its name suggests – it inserts each item into its proper place in the final list. The simplest implementation of this requires two list structures – the source list and the list into which sorted items are inserted. To save memory, most implementations use an in-place sort that works by moving the current item pass the already sorted items and repeatedly swapping it with the preceding item until it is in place.

Like the bubble sort, the insertion sort has a complexity of $O(n^2)$. Although it has the same complexity, the insertion sort is a little over twice as efficient as the bubble sort. It has various advantages :

- It is simple to implement
- It is efficient on small data sets

- It is efficient on data sets which are already substantially sorted : it runs in $O(n+d)$ time, where d is the number of inversions
- It is more efficient in practice than most other simple $O(n^2)$ algorithms such as selection sort or bubble sort : the average time is $\frac{n^2}{4}$ and it is linear in the best case.
- It is stable (does not change the relative order of elements with equal keys)
- It is In-place (only requires a constant amount $O(1)$ of extra memory space)
- It is an online algorithm, in that it can sort a list as it receives it.

Every iteration of an insertion sort removes an element from the input data, inserting it at the correct position in the already sorted list, until no elements are left in the input. The choice of which element to remove from the input is arbitrary and can be made using almost any choice algorithm. Sorting is typically done in-place. The resulting array after k iterations contains the first k entries of the input array and is sorted. In each step, the first remaining entry of the input is removed, inserted into the result at the right position, thus extending the result.

```
Insertion-Sort (A)
for j ← 2 to length [A]
    do key ← A[j]
        ▷Insert A[j] into the sorted sequence A[1..j − 1]
        i ← j − 1
        while i > 0 and A[i] > key
            do A[i + 1] ← A[i]
                i ← i − 1
        A[i + 1] ← key
```

Example. *Illustrate the operation of INSERTION SORT on the array* $A=\langle 2,13,5,18,14\rangle$.

Solution.

	1	2	3	4	5
$A[\,]=$	2	13	5	18	14

For $j=2$ to 5

$j=2$, key $=A[2]$

key $=13$

$i=2-1=1$, $i=1$

while $i>0$ and $A[1]>13$

condition false, so no change.

Now $j=3$, key $= A[3]=5$

$i=3-1=2$

$i=2$, key $=5$

while $i>0$ and $A[2]>$ key

condition is true

So $A[2+1] \leftarrow A[2]$

$A[3] \leftarrow A[2]$

i.e.,

2		13	18	14

and $i=2-1=1$, $i=1$

while $1>0$ and $A[1]>$ key

condition **false**. So no change

then $A[1+1] \leftarrow$ key

$A[2] \leftarrow 5$

That is

2	5	13	18	14

For $j=4$

key $= A[4]$

key $= 18$, $i=3$

Now, while $3>0$ and $A[3]>18$

condition is **false**. No change

Similarly, $j=5$

key $= A[5]$

So key $=14$, $i=4$

Now, while $4>0$ and $A[4]>14$

condition is **true**

So $A[5]=18$ and $i=4-1=3$

Now, while $3>0$ and $A[3]>14$

condition is **false**.

So $A[3+1]=A[4]=14$

and the sorted array is

$A[\,]=$

2	5	13	14	18

Analysis of Insertion Sort

The time taken by INSERTION - SORT procedure depends on the input. We start by presenting the INSERTION - SORT procedure with the time "cost" of each statement and the number of times each statement is executed.

	Insertion-Sort (A)	*Cost*	*times*
1.	for $j \leftarrow 2$ to length [A]	c_1	n
2.	do key $\leftarrow A[j]$	c_2	$n-1$
3.	▷ insert $A[j]$ into the sorted sequence $A[i..j-1]$	0	$n-1$
4.	$i \leftarrow j \leftarrow 1$	c_4	$n-1$
5.	while $i>0$ and $A[i]>$ key	c_5	$\sum_{j=2}^{n} t_j$
6.	do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7.	$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8.	$A[i+1] \leftarrow$ key	c_8	$n-1$

The running time of the algorithm is the sum of running times for each statements executed.

To compute $T(n)$, the running time, we sum the product of the cost and times columns.

$$T(n)=c_1 n+c_2(n-1)+c_4(n-1)+c_5\sum_{j=2}^{n} t_j+c_6\sum_{j=2}^{n}(t_j-1)+c_7\sum_{j=2}^{n}(t_j-1)+c_8(n-1)$$

Best case occurs if the array is already sorted. For each $j=2,3,\ldots n$, we then find that $A[i] \le$ key in line 5 when $i=j-1$.

Thus $t_j=1$ for $j=2,3,\ldots n$

$$T(n)=c_1 n+c_2(n-1)+c_4(n-1)+c_5(n-1)+c_8(n-1)$$

$$=(c_1+c_2+c_4+c_5+c_8)n-(c_2+c_4+c_5+c_8)$$

$=an+b$, where a and b are constants.

= linear function of $n=O(n)$.

In worst case, the array is in reverse order. We must compare each element $A[j]$ with each element in the entire sorted, subarray $A[1\ldots j-1]$ and $t_j=j$ for $j=2,3,\ldots n$.

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} j + c_6 \sum_{j=2}^{n} (j-1) + c_7 \sum_{j=2}^{n} (j-1) + c_8(n-1).$$

$$= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$

$$\left(\because \sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \text{ and } \sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}\right)$$

$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8)$$

$= an^2 + bn + c$ for a, b and c are constant.

= quadratic function of n.

$= O(n^2)$.

We use loop invariant to help us understand why an algorithm is correct. We must show *three* things about a loop invariant :

1. Initialization
2. Maintenance
3. Termination

- **Initialization.** It is true prior to the first iteration of the loop.
- **Maintenance.** If it is true before an iteration of the loop, it remains true before the next iteration.
- **Termination.** When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

The third property is the most important one, since we are using the loop invariant to show correctness.

Example. *Insertion sort can be expressed as a recursive procedure : recursively sort $A[1..n-1]$, then insert $A[n]$ into the sorted array. Write a recurrence for the running time of this recursive version.*

Solution. Since it takes $\theta(n)$ time in the worst case to insert $A[n]$ into the sorted array of $A[1..n-1]$, let $T(n)$ be the total time it takes to insertion sort a list with n elements, we get the recurrence

$$T(n) = \theta(1) \text{ if } n = 1 \text{ and}$$
$$T(n) = T(n-1) + \theta \text{ otherwise.}$$

The solution is this recurrence is that $T(n) = \theta(n^2)$.

Example. *Can we use a binary search to replace the linear search the insertion sort algorithm to improve the worst-case running to $\theta(n \log n)$?*

Solution. The modified code, which uses a procedure similar to the binary search to find out the appropriate position, k in the line 4 ; then move all the items k and $j-1$ one position to the right in lines 5–6.

```
Insertion-Sort-with-Binary Search (A)
1.  for j ← 2 to length [A]
2.      do key ← A[j]
3.  // Insert A[j] into the sorted A[1..j−1]
4.  k ← Binary Search for Position (key)
5.  for m ← j down to k + 1
6.      A[m] ← A[m − 1]
7.  A[k] ← key
```

In the above code, we use the binary search to look for the insertion position of key, which takes $\theta(\log j)$ to complete. Since we work with the worst case, when the input is a reversibly sorted list, it always comes back with the position of 1. Then, lines 5–6 always takes $j-1$ movements, plus the two made in line 2 and 7, for each j, $j+1$ movements have to be made. Hence, the total time is the following :

$$T(n) = \sum_{j=2}^{n} [\log j + (j+1)]$$
$$= \theta(n \log n) + \theta(n^2)$$
$$= \theta(n^2)$$

Example. *Write the pseudocode of Insertion Sort as a recursive procedure.*

Solution.

```
Insertion - Sort (A, p, r)
1.  if p < r
2.      then INSERTION - SORT (A, p, r − 1)
3.          INSERT (A[p, ... r − 1], A[r])
```

where INSERT ($A[p..r]$, key) inserts key into the sorted array $A[p..r]$.

The pseucode for INSERT is shown below and it consists of one iteration of the non-recursive INSERTION-SORT.

```
INSERT (A [p..r], key )
1.  i ← r − 1
2.      while i > p − 1 and A[i] > key
3.          do A[i + 1] ← A[i]
4.              i ← 1 − 1
5.  A[i + 1] ← key
```

Exercise

1. Rewrite the Bubble sort, Selection sort and Insertion sort procedure which sorts the given list of elements in a non-increasing order.
2. Illustrate the operation of INSERTION SORT on the array $A = \langle 41,31,69,16,38,62 \rangle$
3. Illustrate the operation of Selection sort and Bubble sort on the following array :
   (a) $\langle 10,2,13,15,19,2,18 \rangle$
   (b) $\langle 31,41,59,26,41,48,101,99,78 \rangle$
   (c) $\langle 70,80,40,50,60,35,85,2 \rangle$
   (d) $\langle 1,3,5,8,9,10 \rangle$
   (e) $\langle 1,1,1,1,1,1 \rangle$
4. Design an algorithm to find the sum of smallest $\log_2 n$ elements in an unsorted array of n distinct elements.
5. Analyse the Insertion sort in worst case.

CHAPTER 5

Merge Sort

5.1 Introduction

This algorithm was invented by John von Neumann in 1945. It closely follows the divide-and-conquer paradigm.

Conceptually, it works as follows :

1. **Divide.** Divide the unsorted list into two sub lists of about half the size
2. **Conquer.** Sort each of the two sub lists recursively until we have list sizes of length 1, in which case the list itself is returned
3. **Combine.** Merge the two-sorted sub lists back into one sorted list.

We note that the recursion "bottoms out". When the sequence to be sorted has length 1, in which case there is no work to be done, and since every sequence of length 1 is already in sorted order. The key operation of the merge sort algorithm is the merging of two sorted sequences in the "combine" step. To perform the merging, we use an auxiliary procedure MERGE (A, p, q, r), where A is an array and p, q, and r are indices numbering elements of the array such that $p \le q < r$. The procedure assumes that the sub arrays $A[p..q]$ and $A[q+1..r]$ are in sorted order. It *merges* them to form a single sorted sub array that replaces the current sub array $A[p..r]$.

```
MERGE (A, p, q, r)

n1 ← q − p + 1
n2 ← r − q
create arrays L[1..n1 + 1] and R[1..n2 + 1]
for i ← 1 to n1
    do L[i] ← A[p + i − 1]
for j ← 1 to n2
    do R[j] ← A[q + j]
L[n1 + 1] ← ∞
R[n2 + 1] ← ∞
i ← 1
j ← 1
for k ← p to r
    do if L[i] ≤ R[j]
        then A[k] ← L[i]
            i ← i + 1
        else A[k] ← R[j]
            j ← j + 1
```

It is easy to imagine a MERGE procedure that takes time $\theta(n)$, where $n = r - p + 1$ is the number of elements being merged.

We can now use the MERGE procedure as a subroutine in the merge sort algorithm. The procedure MERGE-SORT (A, p, r) sorts the elements in the sub array $A[p..r]$. If $p \le r$, the sub array has at most one element and is therefore already sorted. Otherwise, the divide step simply computes an index q that partitions $A[p..r]$ into two sub arrays: $A[p..q]$, containing $\lceil n/2 \rceil$ elements, and $A[q+1..r]$, containing $\lfloor n/2 \rfloor$ elements.

```
MERGE-SORT (A, p, r)

if p < r
    then q ← ⌊(p + r) / 2⌋
        MERGE-SORT (A, p, q)
        MERGE-SORT (A, q + 1, r)
        MERGE (A, p, q, r)
```

floor function

To sort the entire sequence $A = (A[1], A[2], \ldots A[n])$, we call MERGE-SORT $(A, 1, length[A])$, where once again $length[A] = n$. If we look at the operation of the procedure bottom-up when n is a power of two, the algorithm consists of merging pairs of 1-item sequences to form sorted sequences of length 2, merging pairs of sequences of length 2 to form sorted sequences of length 4, and so on, until two sequences of length $n/2$ are merged to form the final sorted sequence of length n. Figure shows this process in the array $\langle 5, 2, 4, 6, 1, 3, 2, 6 \rangle$.

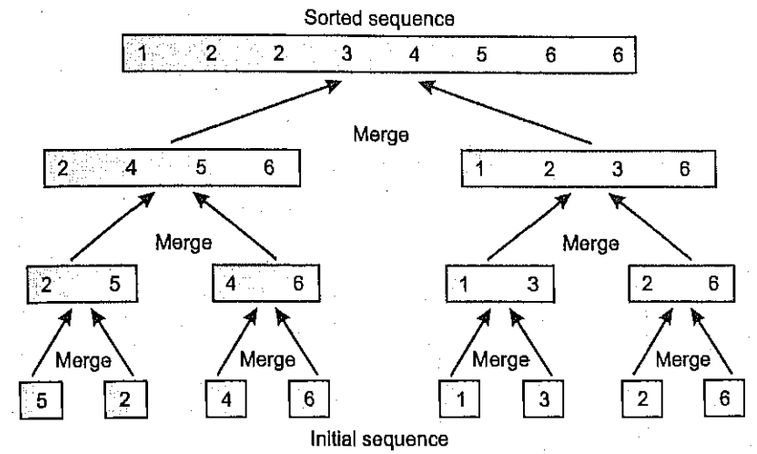


Figure 5.1

Example of Merge Sort

0 MS	85	24	63	45	17	31	96	50
1 Div	85	24	63	45	17	31	96	50
1 MS	85	24	63	45	[illegible]	[illegible]	[illegible]	[illegible]
2 Div	85	24	63	45	[illegible]	[illegible]	[illegible]	[illegible]
2 MS	85	24	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
3 Div	85	24	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
3 MS	85	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
3 MS	[illegible]	24	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
3 Merge	24	85	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
2 MS	[illegible]	[illegible]	63	45	[illegible]	[illegible]	[illegible]	[illegible]
3 Div	[illegible]	[illegible]	63	45	[illegible]	[illegible]	[illegible]	[illegible]
3 MS	[illegible]	[illegible]	63	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
3 MS	[illegible]	[illegible]	[illegible]	45	[illegible]	[illegible]	[illegible]	[illegible]
3 Merge	[illegible]	[illegible]	45	63	[illegible]	[illegible]	[illegible]	[illegible]
2 Merge	24	45	63	85	[illegible]	[illegible]	[illegible]	[illegible]

0 MS	85	24	63	45	17	31	96	50
1 Div	85	24	63	45	17	31	96	50
1 MS	85	24	63	45	17	31	96	50
...	...	...	...	...	...	...	...	...
2 Merge	24	45	63	85	17	31	96	50
1 MS	24	45	63	85	17	31	96	50
2 Div	24	45	63	85	17	31	96	50
2 MS	24	45	63	85	17	31	96	50
3 Div	24	45	63	85	17	31	96	50
3 MS	24	45	63	85	17	31	96	50
3 MS	24	45	63	85	17	31	96	50
3 Merge	24	45	63	85	17	31	96	50
2 MS	24	45	63	85	17	31	96	50
3 Div	24	45	63	85	17	31	96	50
3 MS	24	45	63	85	17	31	96	50
3 MS	24	45	63	85	17	31	96	50
3 Merge	24	45	63	85	17	31	50	96
2 Merge	24	45	63	85	17	31	50	96
1 Merge	17	24	31	45	50	63	85	96

5.2 Analysis of Merge Sort

Although the pseudocode for MERGE-SORT works correctly when the number of elements is not even, our recurrence-based analysis is simplified if we assume that the original problem size is a power of two. Each divide step then yields two subsequence of size exactly $n/2$. This assumption does not affect the order of growth of the solution to the recurrence.

Merge sort on just one element takes constant time. When we have $n > 1$ elements, we break down the running time as follows.

- **Divide.** The divide step just computes the middle of the sub array, which takes constant time. Thus, $D(n) = \theta(1)$.
- **Conquer.** We recursively solve two sub problems, each of size $n/2$, which contributes $2T(n/2)$ to the running time.
- **Combine.** We have already noted that the MERGE procedure on an n-element sub array takes time $\theta(n)$, so $C(n) = \theta(n)$.

When we add the functions $D(n)$ and $C(n)$ for the merge sort analysis, we are adding a function that is $\theta(n)$ and a function that is $\theta(1)$. This sum is a linear function of n, that is, $\theta(n)$. Adding it to the $2T(n/2)$ term from the "conquer" step gives the recurrence for the worst-case running time $T(n)$ of merge sort :

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

By Master Theorem , we shall show that $T(n)$ is $\theta(n \lg n)$, where $\lg n$ stands for $\log_2 n$. For large enough inputs, merge sort, with its $\theta(n \lg n)$ running time, outperforms insertion sort, whose running time is $\theta(n^2)$, in the worst case

Merge sort's merge operation is useful in online sorting, where the list to be sorted is received a piece at a time, instead of all at the beginning. In this application, we sort each new piece that is received using any sorting algorithm, and then merge it into our sorted list so far using the merge operation. However, this approach can be expensive in time and space if the received pieces are small compared to the sorted list – a better approach in this case is to store the list in a self-balancing binary search tree and add elements to it as they are received.

Although heap sort has the same time bounds as merge sort, it requires only $\theta(1)$ auxiliary space instead of merge sort's $\theta(n)$, and is consequently often faster in practical implementations. Quick sort, however, is considered by many to be the fastest general-purpose sort algorithm although there are proofs to show that merge sort is indeed the fastest sorting algorithm out of the three. Its average-case complexity is $O(n \log n)$ even with perfect input and given optimal pivots quick sort is not as fast as merge sort and it is quadratic in the worst case. On the plus side, merge sort is a stable sort, parallelizes better, and is more efficient at handling slow-to-access sequential media. Merge sort is often the best choice for sorting a linked list: in this situation it is relatively easy to implement a merge sort in such a way that it requires only $\theta(1)$ extra space, and the slow random-access performance of a linked list makes some other algorithms (such as Quick sort) perform poorly, and others (such as heap sort) completely impossible.

As of Perl 5.8, merge sort is its default-sorting algorithm (it was Quick sort in previous versions of Perl). In Java, the Arrays sort( ) methods use merge sort or a tuned Quick sort depending on the data types and for implementation efficiency switch to insertion sort when fewer than seven array elements are being sorted.

Merge sort incorporates *two* main ideas to get better its runtime:

1. A small list will take fewer steps to sort than a large list.
2. Fewer steps are required to construct a sorted list from two sorted lists than two unsorted lists. For example, you only have to traverse each list once if they're already sorted (see the merge function below for an example implementation).

5.3 Insertion Sort and Merge Sort

Suppose we are comparing implementation of insertion sort and merge sort on the same machine. For inputs of size n, insertion sort runs in $8n^2$ steps, while merge sort runs in $64\, n \log n$ steps.

For $n \le 43$, $8n^2 \le 64\, n \log n$ and insertion sort beats merge sort.

Thus for $n \le 43$ insertion sort beats merge sort.

Although merge sort runs in $\theta(n \lg n)$ worst case time and insertion sort runs in $\Theta(n^2)$ worst-case time, the constant factors in insertion sort make it faster for small n. Therefore, it makes sense to use insertion sort within merge sort when subproblems become sufficiently small. Consider a modification of merge sort in which subarrays of size k or less (for some k) are not divided further, but sorted explicitly with Insertion sort.

```
MERGE-SORT (A, p, r)
  if p < r − k + 1
     then q ← ⌊(p + r)/2⌋
        MERGE-SORT (A, p, q)
        MERGE-SORT (A, q + 1, r)
        MERGE (A, p, q, r)
     else INSERTION-SORT (A[p..r])
```

(i) *The total time spent on all calls to Insertion sort is in the worst-case* $\theta(nk)$.

Proof. If $n \le k$, then this modified MERGE-SORT will just call. INSERTION-SORT on the whole array.

Therefore, the running time will be $\theta(n^2) = \theta(n \cdot n) = O(nk)$

So let us assume that $n > k$.

First note that the length of subarray $A[p..r]$ is $l = r - p + 1$. So the condition $p < r - k + 1$ is the same as $l > k$. Therefore, any subarray of length l on which we call INSERTION-SORT has to satisfy $l \le k$ (this is actually in the given of the algorithm). Moreover, any subarray $A[p..r]$ of length l on which we call INSERTION-SORT has to satisfy $\frac{k}{2} \le l$. The reason for this is the following : if INSERTION-SORT is called on $A[p..r]$, then $A[p..r]$ must be the first or second half of another subarray $A[p'..r']$ that was divided (we assumed that $n > k$ so $A[p..r]$ cannot be $A[1..n]$). Thus, $A[p'..r']$ has length $l' > k$. Therefore, since $l \ge \left\lfloor \frac{l'}{2} \right\rfloor$ and $l' > k$, then $l \ge \frac{k}{2}$.

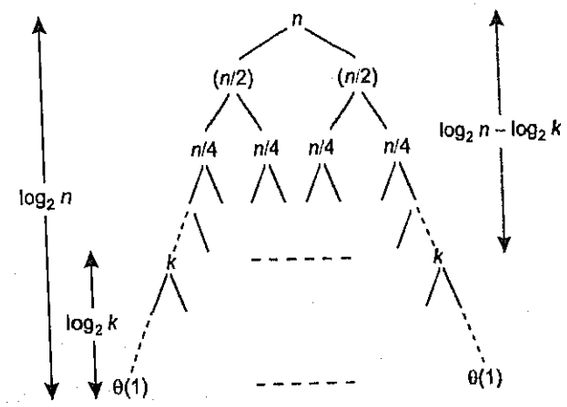
The running time of INSERTION-SORT on $A[p..r]$ is $\Theta(l^2)$, where $l = r - p + 1$ is the length of $A[p..r]$. But since $\frac{k}{2} \le l \le k$, then $l = \Theta(k)$ and the running time of INSERTION-SORT on $A[p..r]$ is $\Theta(k^2)$.

Now let us see how many such sub arrays (on which we call INSERTION-SORT) we have. Since all the sub arrays are disjoint (*i.e.*, their indices do not overlap) and for every one of them $\frac{k}{2} \le l \le k$, then the number of these sub arrays, call it m, satisfies $\frac{n}{k} \le m \le \frac{2n}{k}$. Therefore $m = \Theta(n/k)$.

The total time spent on INSERTION-SORT is therefore $\Theta(k^2) \cdot \Theta(n/k) = \Theta(nk)$.

(ii) *Show that the total time spent on merging is in the worst-case* $\theta(n \log(n/k))$.

Proof. The merging proceeds in the same way as with the basic version of MERGE-SORT except that it stops whenever the subarray reaches a length $l \le k$ (instead of 1). Therefore, we need to determine the number of levels in the recursive tree. With the basic version of MERGE-SORT, we argued that the number of levels is $\Theta(\log n)$ because we divide n by 2 with every level until we reach 1. With this modification, we divide n by 2 with every level until we reach k (or less). Therefore the number of levels is $\theta(\log n) - \theta(\log k)$, which is the number of levels originally minus the number of levels that we would have had if we continued beyond k. But $\theta(\log n) - \theta(\log k) = \theta(\log(n/k))$. So the total time that we spend on merging is $\Theta(n \log(n/k))$ since we spend $\Theta(n)$ time in every level as before.



Therefore, this modified merge sort runs in $\theta(nk) + \theta(n \log(n/k))$ (sorting + merging).

Thus, the total running time is $\theta(nk + n \log(n/k))$.

(iii) *What is the largest asymptotic (θ-notation) value of k as a function of n for which the modified algorithm has the same asymptotic running time as standard merge sort ? How should k be chosen in practice ?*

Proof. Obviously, if k grows faster than $\log n$ asymptotically, then we would get something worse than $\Theta(n \log n)$ because of the $\Theta(nk)$ term. Therefore, we know that $k = O(\log n)$. So lets pick $k = \Theta(\log n)$ and see if this works. If $k = \Theta(\log n)$ then the running time of the modified MERGE-SORT will be

$$\Theta\left(n \log n + n \log \frac{n}{\log n}\right) = \Theta(n \log n + n \log n - n \log n(\log n)) = \Theta(n \log n).$$

So if $k = \Theta(\log n)$, the running time of the modified algorithm has the same asymptotic running time as the standard one. In practice, k should be the maximum input size on which Insertion sort is faster then Merge sort.

Example. *Rewrite the MERGE procedure so that it does not use sentinels, instead stopping once either array has had all its items copied back to A.*

Solution. The code could be the following :

```
MERGE (A, p, q, r)
1.    n1 ← q − p + 1
```

```
n2 ← r − q
create arrays L[1..n] and R[1..n2]
for i ←1 to n1
    do L[i] ←A[p + i − 1]
for j ←1 to n2
    do R[j] ←A[q + i]
i ←1
j ←1
for k ←p to r
    do if (i <= n1)
        if (j <= n2)
        then if (L[i] <= R[j])
            then A[k] ←L[i]
                i ←i + 1
            else A[k] ← R[j])
                j ← j + 1
        else A[k] ←L[i]
            i <i + 1
    else if (j <= n2)
        then A[k] ←R[j]
            j ← j + 1
```

The idea is that once a list, either L or R is completed, everything in the order list is copied back.

Exercise

1. Consider following list of elements as 50, 40, 20, 70, 15, 35, 20, 60
   Sort the above list using Merge sort.
2. Among, Merge sort, Insertion sort and Bubble sort which sorting techniques is the best in the worst case. Support your argument with an example and analysis.
3. Analyse the Merge sort algorithm. Argue on its best case, average case and worst case time complexity.
4. Sort the list 70, 80, 40, 50, 60, 12, 35, 95, 10. Using Merge sort.
5. Show that the merge sort associated with an execution of merge-sort on a sequence of size n has height $\lceil \log n \rceil$.
6. Show that merging two sorted sequences S_1 and S_2 takes $O(n_1 + n_2)$ time, where n_1 is the size of S_1 and n_2 is the size of S_2.
7. Show that the running time of the merge-sort algorithm on the n-element sequence is $O(n \log n)$, even when n is not a power of 2.
8. Let S be a sequence of n elements on which a total order relation is defined. An inversion in S is a pair of elements x and y such that x appears before y in S but $x > y$. Describe an algorithm running in $O(n \log n)$ time for determining the number of inversions in S.

CHAPTER 6

Heap Sort

6.1 Binary Heap

The binary heap data structure is an array that can be viewed as a complete binary tree. Each node of the binary tree corresponds to an element of the array. The array is completely filled on all levels except possibly lowest.

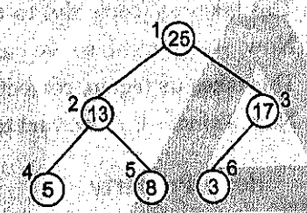


Figure 6.1

We represent heaps in level order, going from left to right. The array corresponding to the heap in Fig. 6.1 above is

	1	2	3	4	5	6
A	25	13	17	5	8	3

If an array A contains key values of nodes in a heap, length $[A]$ is the total number of elements.

heap-size $[A]$ = length $[A]$ = number of elements.

The root of the tree $A[1]$ and given index i of a node the indices of its parent, left child and right child can be computed.

```
PARENT (i)
    return floor (i / 2)
LEFT (i)
    return 2i
RIGHT (i)
    return 2i + 1
```

Let us try these out on a heap to make sure we believe they are correct, take this heap

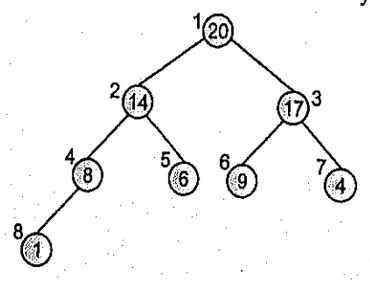


Figure 6.2

which is represented by the array

1	2	3	4	5	6	7	8	9
20	14	17	8	6	9	4	1	

The index of the 20 is 1

To find the index of the left child, we calculate $1*2=2$

This takes us (correctly) to the 14.

Now, we go right, so we calculate $2*2+1=5$.

This takes us (again, correctly) to the 6.

Now, 4's index is 7, we want to go to the parent, so we calculate $7/2=3$ which takes us to the 17.

6.2 Heap Property

In a Max-heap, for every node i other than the root, the value of a node is greater than or equal (at most) to the value of its parent

$$A[\text{PARENT}(i)] \geq A[i]$$

Thus, the largest element in a heap is stored at the root. Following is an example of MAX-HEAP.

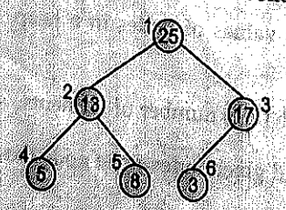


Figure 6.3

and a Min-Heap is organized in the opposite way *i.e.*, the min-heap property is that for every node i other than the root is

$$A[\text{PARENT}(i)] \leq A[i]$$

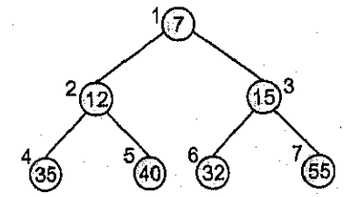


Figure 6.4

By the definition of a heap, all the tree levels are completely filled except possibly for the lowest level which is filled from the left up to a point.

Following is not a heap, because it only has the heap property – it is not a complete binary tree. Recall that to be complete, a binary tree has to fill up all of its levels with the possible exception of the last one, which must be filled in from the left side.

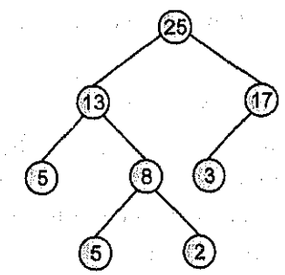


Figure 6.5

6.3 Height of a Heap

Height of a node

We define the height of a node in a tree to be a number of edges on the longest simple downward path from a node to a leaf.

Height of a tree

The number of edges on a simple downward path from a root to a leaf.

Note that the height of a tree with n node is $\lfloor \lg n \rfloor$ which is $\Theta(\lg n)$. This implies that an n-element heap has height $\lfloor \lg n \rfloor$.

In order to show this let the height of n-element heap be h. From the bounds obtained on maximum and minimum number of elements in a heap, we get

$$2^h \leq n \leq 2^{h+1} - 1$$

Where n is the number of elements in a heap.

$$2^h \leq n \leq 2^{h+1}$$

Taking logarithms to the base 2

$$h \le \lg n \le h+1$$

It follows that $h = \lfloor \lg n \rfloor$

Clearly, a heap of height h has the **minimum number** of elements when it has just one node at the lowest level. The levels above the lowest level form a complete binary tree of $h-1$ and $2^h - 1$ nodes.

Hence, the minimum number of nodes possible in a heap of height h is 2^h.

Clearly, a heap of height h, has the **maximum number** of elements when its lowest level is completely filled. In this case the heap is a complete binary tree of height h and hence has $2^{h+1} - 1$ nodes.

We know from above that largest element resides in root, $A[1]$. The natural question to ask is **where in a heap might the smallest element resides** ? Consider any path from root of the tree to a leaf. Because of the heap property, as we follow that path, the elements are either decreasing or staying the same. If it happens to be the case all elements in the heap are distinct, then the above implies that the smallest is in a leaf of the tree. It could also be that an entire sub tree of the heap is the smallest element or indeed that there is only one element in the heap, which is the smallest element, so the smallest element is everywhere. Note that anything below the smallest element must equal the smallest element.

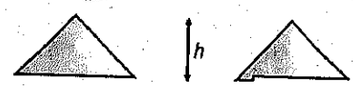
Example. *What are the minimum and maximum numbers of elements in a heap of height h ?*

Solution. Let n is the number of elements in a heap where the height is fixed to h.

The largest value of n occurs when the heap is shaped as the left Fig. below. We cannot pack any more elements into such a heap. The maximum number of elements is thus

$$n = 2^{h+1} - 1$$

height starts from 0. CNP.



$\frac{r^{n+1}-1}{r-1}$

In the other situation when there is only one element on the last row, we can't have any fewer elements in such a heap without reducing the height h. The minimum number of elements is thus $n = 2^{h+1-1} - 1 + 1 = 2^h$.

In conclusion, $2^h \le n \le 2^{h+1} - 1$

Example. *Show that an n-element heap has height* $\lfloor \lg n \rfloor$

Solution. From above, we can derive the following constraints by taking the logarithm of both sides :

$$h \le \lg n \le \lg(2^{h+1} - 1)$$

Since $\lg(2^{h+1} - 1) < \lg(2^{h+1}) = h+1$, we can write

$$h \le \lg n \le h+1$$

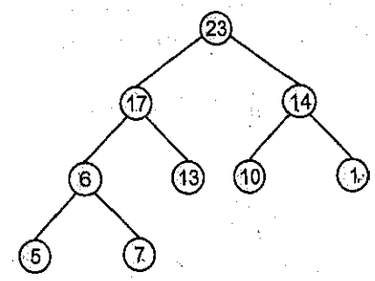
which is the same as saying $h = \lfloor \lg n \rfloor$.

Example. *Where in a max-heap might the smallest element reside, assuming that all elements are distinct ?*

Solution. The smallest element must reside in one of the leaf nodes.

Example. *Is the sequence* $\langle 23,17,14,6,13,10,1,5,7,12 \rangle$ *a max heap ?*

Solution. No, the sequence doesn't fulfill the heap-property since the element 7 is below element 6.



6.4 Heapify

Maintaining the heap property

Heapify is a procedure for manipulating heap data structures. It is given an array A and index i into the array. The subtree rooted at the children of $A[i]$ are heap but node $A[i]$ itself may possibly violate the heap property *i.e.*, $A[i] < A[2i]$ or $A[i] < A[2i+1]$. The procedure 'Heapify' manipulates the tree rooted at $A[i]$ so it becomes a heap. In other words, 'Heapify' is let the value at $A[i]$ "float down" in a heap so that subtree rooted at index i becomes a heap.

Outline of procedure heapify

Heapify picks the largest child key and compares it to the parent key. If parent key is larger then heapify quits, otherwise it swaps the parent key with the largest child key. So that the parent now becomes larger than its children. It is important to note that swap may destroy the heap property of the subtree rooted at the largest child node. If this is the case, heapify calls itself again using largest child node as the new root.

```
MAX-HEAPIFY (A, i)
    l ← left [i]
    r ← right [i]
    if l ≤ heap-size [A] and A[l] > A[i]
        then largest ← l
        else largest ← i
    if r ≤ heap-size [A] and A[r] > A [largest]
        then largest ← r
    if largest ≠ i
    { then exchange A[i] ↔ A [largest]
            MAX-HEAPIFY (A, largest) }
```

Analysis

If we put a value at root that is less than every value in the left and right subtree, then 'Heapify' will be called recursively until leaf is reached. To make recursive calls traverse the longest path to a leaf, choose value that makes 'Heapify' always recurs on the left child. It follows the left branch when left child is greater than or equal to the right child, so putting 0 at the root and 1 at all other nodes, for example, will accomplish this task. With such values 'Heapify' will be called h times, where h is the heap height so its running time will be $\theta(h)$ (since each call does $\theta(1)$ work), which is $\theta(\lg n)$. Since we have a case in which Heapify's running time $\theta(\lg n)$, its worst-case running time is $\Omega(\lg n)$.

Example : Suppose we have a complete binary tree.

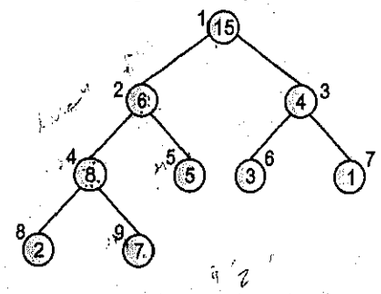


Figure 6.6

In this complete binary tree, the subtrees of 6 are not max-heap. So we call MAX-HEAPIFY $(A, 2)$

$l = 4,$

$r = 5$: and $i = 2$

Here heap-size $(A) = 9$

$l \le$ heap-size and $A[l] = 8$ and $A[i] = 6$

So, $A[l] > A[i]$

then largest $= 4$

$r \le$ heap-size $[A]$ and $A[r] = 5$

$A[r] < A$[largest]

Here, largest $\ne i$

then $A[2] = 8$ and $A[4] = 6$ after exchanging $A[i]$ and A [largest]

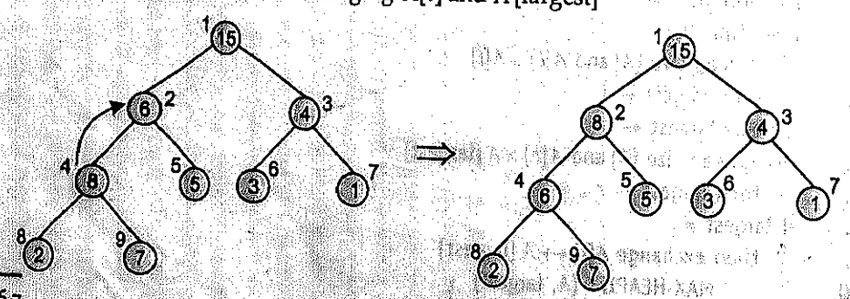


Figure 6.7

Now, call MAX-HEAPIFY $(A, 4)$

$l = 8$

$r = 9$ and $A[l] = 2$

$A[r] = 7$

$A[i] = 6$

$l \le$ heap-size $[A]$ and $A[l] < A[i]$ so largest $= i$

$r \le$ heap-size $[A]$ and $A[r] > A$ [largest] so largest $= r$

largest $\ne i$ then exchange A [largest] $\leftrightarrow A[i]$

Thus the final tree is

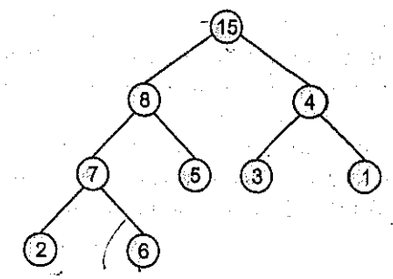


Figure 6.8

Example. *What is the effect of calling MAX-HEAPIFY (A, i) when the element $A[i]$ is larger than its children ?*

Solution. Nothing happens.

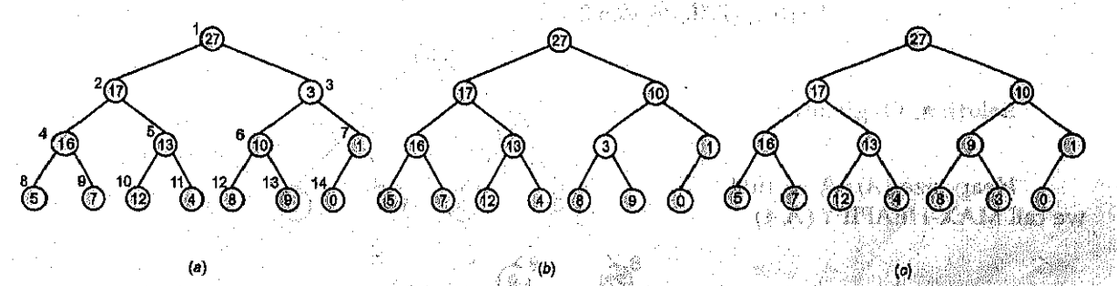
Example. *What is the effect of calling MAX-HEAPIFY (A, i) for $I >$heap-size $[A]/2$?*

Solution. Nothing happens (l and $r >$ heap-size $[A]$).

Example. *Using Fig. illustrate the operation of MAX-HEAPIFY $(A,3)$ on the array*

$A = \langle 27,13,3,16,13,10,1,5,7,12,4,8,9,0 \rangle$

Solution. The following happens when MAX-HEAPIFY $(A,3)$ is called.



6.5 Building a Heap

We can use the procedure 'Heapify' in a bottom-up fashion to convert an array $A[1..n]$ into a heap.

```
BUILD-MAX-HEAP (A)
    heap-size (A) ← length [A]
    For i ← floor(length [A]/2) down to 1 do
        MAX-HEAPIFY (A, i)
```

We can build a heap from an unordered array in linear time.

6.6 Heap Sort Algorithm

The heap sort combines the best of both merge sort and insertion sort. Like merge sort, the worst case time of heap sort is $O(n \log n)$ and like insertion sort, heap sort sorts in-place. The heap sort algorithm starts by using procedure MAX-BUILD-HEAP to build a heap on the input array $A[1..n]$. Since the maximum element of the array stored at the root $A[1]$, it can be put into its correct final position by exchanging it with $A[n]$ (the last element in A). If we now discard node n from the heap then the remaining elements can be made into heap. Note that the new element at the root may violate the heap property. All that is needed to restore the heap property.

```
HEAP-SORT(A)
    BUILD-MAX-HEAP (A)
    for i ← length [A] down to 2
        do exchange A[1] ↔ A[i]
            heap-size [A] ← heap-size [A] −1
            MAX-HEAPIFY (A, 1)
```

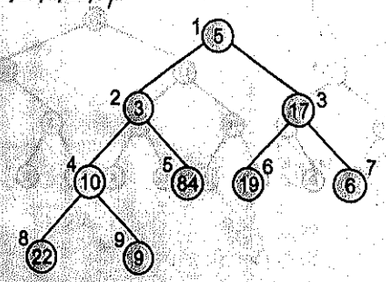
Analysis

We have seen that the running time of 'Build-heap' is $O(n)$. The heap-sort algorithm makes a call to 'Build-heap' for creating a (max) heap, which will take $O(n)$ time and each of the $(n-1)$-calls to Max-heapify to fix up the new heap (which is created after exchanging the root and by decreasing the heap size). We know 'MAX-HEAPIFY' takes time $O(\lg n)$.

Thus the total running time for the heap-sort is $O(n \lg n)$.

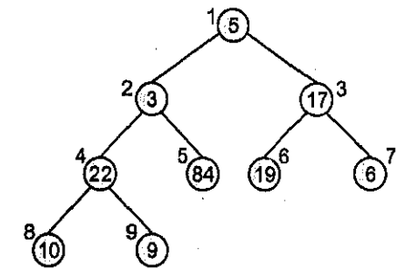
Example. *Using Fig. illustrate the operation of BUILD-MAX-HEAP on the array* $A = \langle 5,3,17,10,84,19,6,22,9 \rangle$.

Solution. Originally :

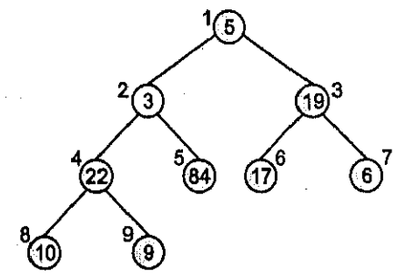


Heap-size $(A) = 9$, so first we call MAX-HEAPIFY $(A, 4)$

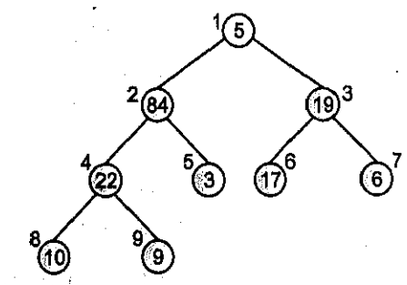
After MAX-HEAPIFY $(A, 4)$



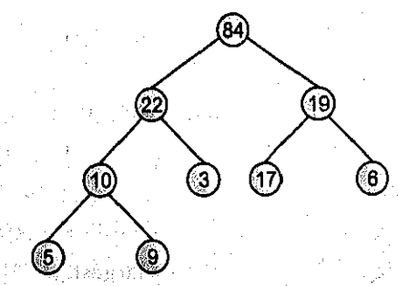
After MAX-HEAPIFY $(A, 3)$



After MAX-HEAPIFY $(A, 2)$

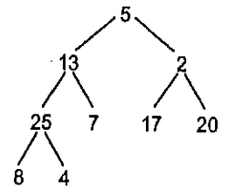


After MAX-HEAPIFY $(A, 1)$



Example. *Illustrate the operation of HEAP-SORT on the array* $A = \langle 5,13,2,25,7,17,20,8,4 \rangle$

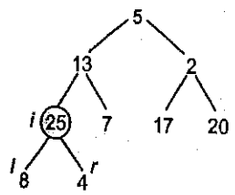
Solution. Originally,



First we call BUILD-MAX-HEAP

heap-size $(A) = 9$

So, $i = 4$ to 1. Call Max-Heapify (A, i).



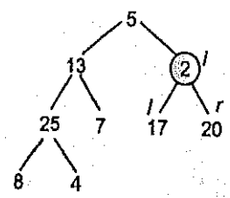
i.e., first we call MAX-HEAPIFY $(A, 4)$

$A[l] = 8 \quad A[i] = 25 \quad A[r] = 4$

$A[i] > A[l]$

$A[i] > A[r]$

Now we call MAX-Heapify $(A, 3)$.



$A[i] = 2 \quad A[l] = 17 \quad A[r] = 20$

$A[l] > A[i]$

$\therefore$ largest $= 6$

$A[r] > A$ [largest]

$20 > 17$

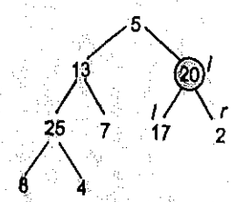
$\therefore$ largest $= 7$

largest $\neq i$

$\therefore$ $A[i] \leftrightarrow A$ [largest] *i.e.*, now

$A[i] > A[l]$

$A[i] > A[r]$



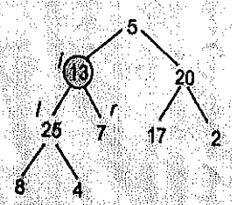
Now, we call MAX-HEAPIFY $(A, 2)$, *i.e.*,

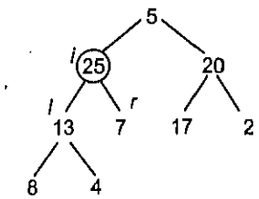
$A[i] < A[l]$

So, largest $= 4$

A [largest] $> A[r]$

$\therefore$ $i \neq$ largest, so $A[i] \leftrightarrow A$ [largest]

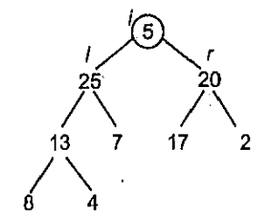




Now,

$A[i] > A[l]$

$A[i] > A[r]$



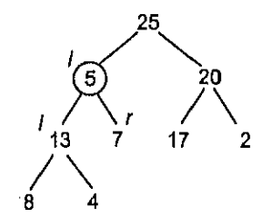
We call MAX-HEAPIFY $(A, 1)$.

$A[i] < A[l]$

largest $= 2$

A [largest] $> A[r]$ and largest $\neq i$

$A[i] \leftrightarrow A$ [largest]

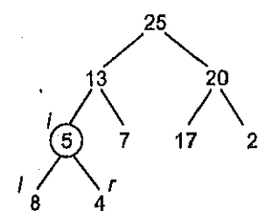


largest $= 2$, so $i = 2$

$A[i] < A[l]$ then largest $= 4$

and A [largest] $> A[r]$, largest $\neq i$

$A[i] \leftrightarrow A$ [largest]



Now,

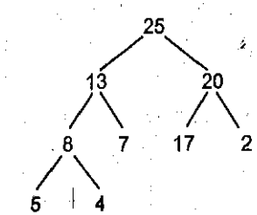
$A[i] < A[l]$

largest $= 8 \quad A$ [largest] $> A[r]$

largest $\neq i$

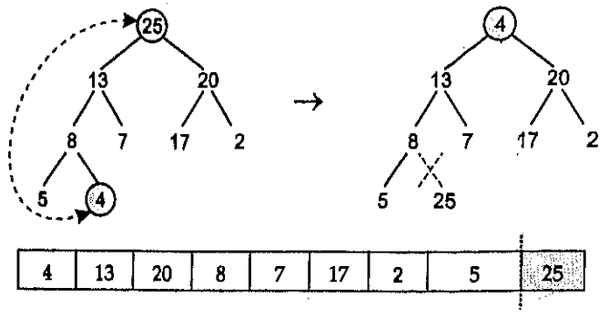
A [largest] $\leftrightarrow A[i]$

So, this is the final tree after BUILD-MAX-HEAP.

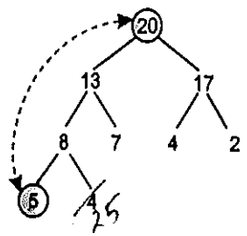


Now, $i = 9$ down to 2 exchange $A[i] \leftrightarrow A[i]$ and size = size -1 and call MAX-HEAPIFY $(A, 1)$ each time.

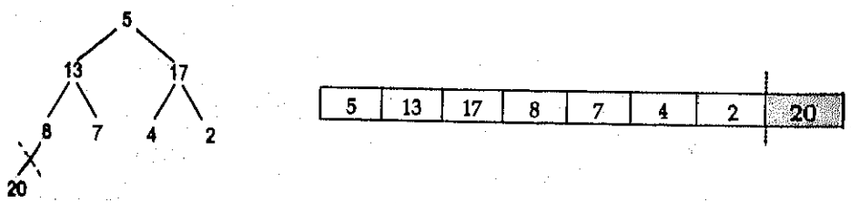
Exchanging $A[1] \leftrightarrow A[9]$, we get



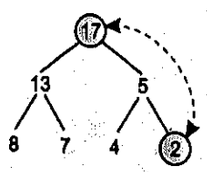
Now call MAX-HEAPIFY $(A,1)$, we get



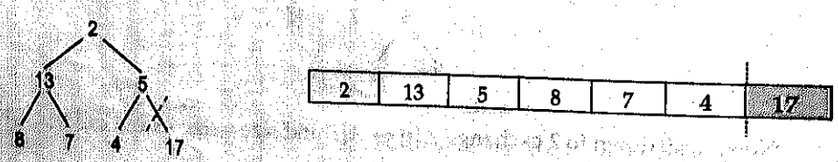
Now, exchange $A[1]$ and $A[8]$ and size = size $-1 = 8 - 1 = 7$



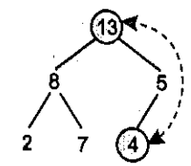
Again, call MAX-HEAPIFY $(A, 1)$, we get



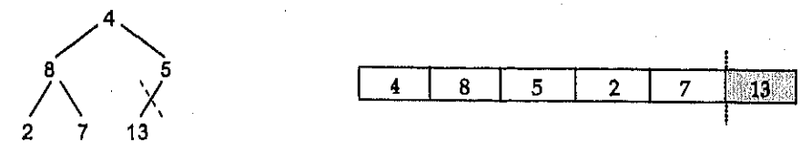
Exchange $A[1]$ and $A[7]$ and size = size $-1 = 7 - 1 = 6$.



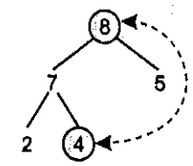
Again, call MAX-HEAPIFY $(A, 1)$, we get



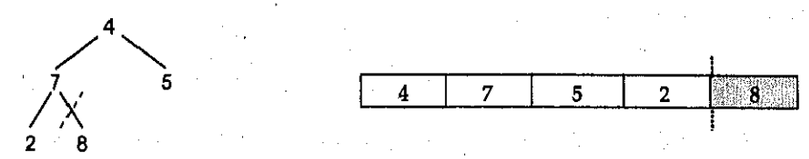
Exchange $A[1]$ and $A[6]$ and now size $= 6 - 1 = 5$.



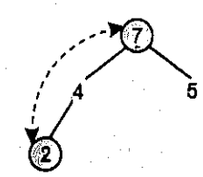
Again call MAX-HEAPIFY $(A,1)$, we get



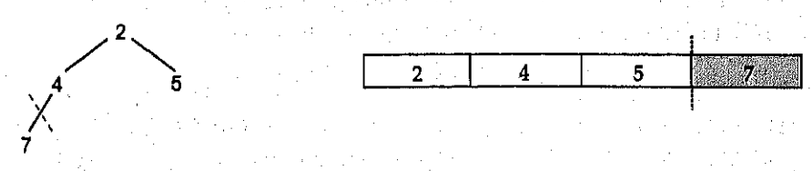
Exchange $A[1]$ and $A[5]$ and now size $= 5 - 1 = 4$.



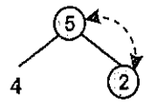
Again, call MAX-HEAPIFY $(A, 1)$, we get



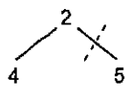
Exchange $A[1]$ and $A[4]$ and size $= 4 - 1 = 3$.



Call MAX-HEAPIFY $(A,1)$, we get

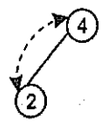


Exchange $A[1]$ and $A[3]$ and size $=3-1=2$.

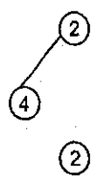


2	4	5

Call, MAX-HEAPIFY $(A,1)$, we get



Exchange $A[1]$ and $A[2]$ and size $=2-1=1$.



2	4

Thus the sorted array.

2	4	5	7	8	13	17	20	25

Example. *What is the running time of heap sort on an array A of length n that is already sorted in increasing order ? What about decreasing order ?*

Solution.

```
HEAP-SORT (A)
1.    BUILD-MAX-HEAP (A)
2.    for i ← length [A] down to 2
3.        do exchange A[1] ↔ A[i]
4.            heap-size [A] ← heap-size [A] − 1
5.            MAX-HEAPIFY (A, 1)
```

- **Increasing order.** Line 1 takes $O(n)$ time (worst case) while line 5 takes $O(\lg n)$ time since MAX-HEAPIFY must do $\lg n$ exchange. All together,

$$T(n)=O(n)+nO(\lg n)=O(n\lg n).$$

- **Decreasing order.** Line 1 results in $O(n/2)$ calls to MAX-HEAPIFY without any work *i.e.*, $O(n/2)$ in total. Line 5 costs just as much as in increasing order so in total we get

$$T(n)=O(n/2)+nO(\lg n)=O(n\lg n)$$

6.7 Heap-Insert

The procedure MAX-HEAP-INSERT implements the INSERT operation. It takes the key of new element as the input into max-heap A. The procedure first expands the max-heap by adding to the tree a new leaf whose key is $-\infty$. Then it calls HEAP-INCREASE-KEY to set the key of this new node to its correct value and maintain the max-heap property.

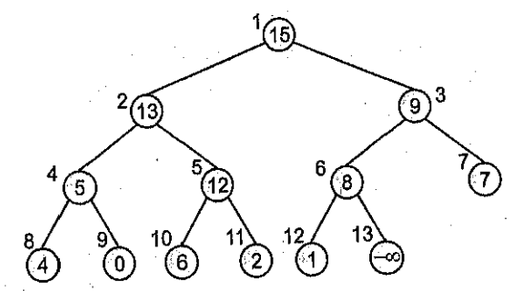
```
MAX-HEAP-INSERT (A, key)
    heap-size [A] ← heap-size [A] + 1
    A [heap-size [A] ] ← −∞
    HEAP-INCREASE-KEY (A, heap-size [A], key)

HEAP-INCREASE-KEY (A, i,. key)
    if key < A[i]
        then error "new key is smaller than current key".
    A[i] ← key
    while i > 1 and A [PARENT (i) < A[i] ]
        do exchange A[i] ↔ A [PARENT (i)]
            i ← PARENT (i)
```

The running time of MAX-HEAP-INSERT on an n-element heap is $O(\lg n)$.

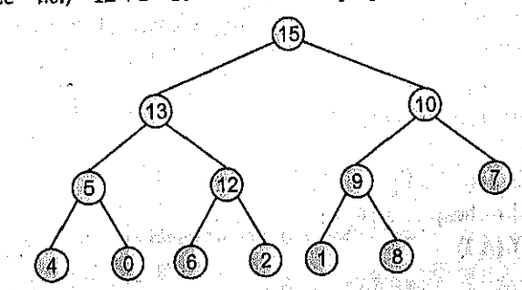
Example. *"Illustrate the operation of MAX-HEAP-INSERT $(A,10)$ on the heap*

$$A=\langle 15,13,9,5,12,8,7,4,0,6,2,1\rangle.$$

Solution. The Figures show the heap before and after the call HEAP-INCREASE-KEY $(A$, heap-size $[A]$, 10) which is called MAX-HEAP-INSERT $(A,10)$.



Increase heap size *i.e.*, $12+1=13$ and $A[13]=-\infty$



Final Heap

Example. *"The procedure BUILD-MAX-HEAP in section 6.5 can be implemented by repeatedly using MAX-HEAP-INSERT to insert the elements into the heap. Consider the following :*

```
BUILD-MAX-HEAP' (A)
1. heap-size [A] ← 1
2. for i ← 2 to length [A]
3.        do MAX-HEAP-INSERT (A, A[i])
```

(*a*) *Do the procedures BUILD-MAX-HEAP and BUILD-MAX-HEAP' always create the same heap when run on the same input array ? Prove that they do, or provide a counterexample.*

(*b*) *Show that in the worst case, BUILD-MAX-HEAP' requires $\Theta(n \lg n)$ time to build an n-element heap."*

Solution. The original algorithm to build a max-heap is

```
BUILD-MAX-HEAP (A)
1.  heap-size [A] ← length [A]
2.  for i ← ⌊length [A] / 2⌋ down to 1
3.      do MAX-HEAPIFY (A,i).
```

(*a*) No, they do not always create the same heap.

e.g., $\langle 0,1,2,3,4,5 \rangle$.

(*b*) The worst case occurs when MAX-HEAP-INSERT must let the element float all the way up to the root. This scenario can be provoked every time by presenting an array of decreasing values $\langle n, n-1, n-2, \ldots 1 \rangle$

Then, $T(n) = \theta(n \lg n)$.

6.8 Heap-Delete

HEAP-DELETE (A, i) is the procedure, which deletes the item in node i from heap A. HEAP-DELETE runs in $O(\lg n)$ time for an n-element max-heap.

```
HEAP-DELETE (A,i)
1.  A[i] ← A [heap-size [A]]
2.  heap-size [A] ← heap-size [A] – 1
3.  MAX-HEAPIFY (A, i)
```

6.9 Heap-Extract-Max

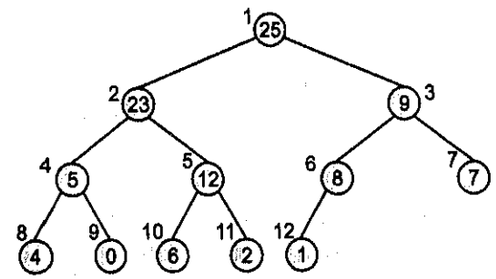
This operation removes and returns the element having largest key value from the set.

```
HEAP-EXTRACT-MAX (A)
1.  IF heap-size [A] < 1
2.      then error "heap underflow"
3.  max ← A[1]
4.  A[1] ← A [heap-size [A] ]
5.  heap-size [A] ← heap-size [A] –1
6.  MAX-HEAPIFY (A, 1)
7.  return max.
```


Example. *Illustrate HEAP-EXTRACT-MAX on the heap.*

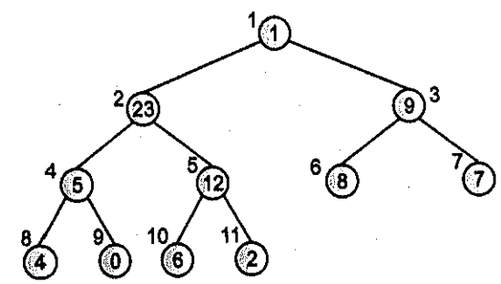
$$A = \langle 25,23,9,5,12,8,7,4,0,6,2,1 \rangle$$

Solution.

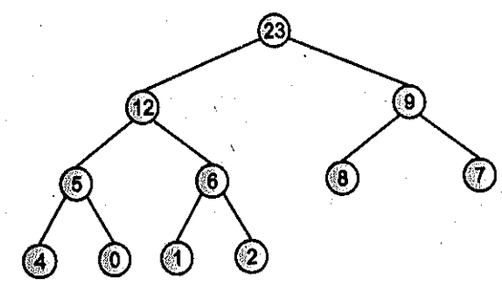


max ← $A[1]$

Thus max = 25 and $A[1] \leftarrow A[12]$, Now heap size $[A] = 12 - 1 = 11$



Now, call MAX-HEAPIFY $(A, 1)$ and readjust the heap and the final heap is as follows :



Exercise

1. Create a Min-Heap and Max-Heap for the following list

   (*a*) $L = \langle G, F, D, C, B, A, H, I, J, K \rangle$

(b) $L = \langle 20, 10, 1, 5, 4, 80, 60, 30 \rangle$

(c) $L = \langle 5, 15, 8, 3, 9, 65 \rangle$

(d) $L = \langle A, B, G, H, D, X \rangle$

2. Suppose the elements in an array are (starting at index 1) $\langle 25, 19, 15, 5, 12, 4, 13, 3, 7, 10 \rangle$
   Does this array represent a heap ? Justify your answer.
3. Sort the following array using Heap-Sort techniques.
   $$\langle 5, 8, 3, 9, 2, 10, 1, 45, 32 \rangle$$
4. Suppose the array to be sorted (into alphabetical order) by Heap-sort initially contains the following sequence of letters :
   ALGORITHMS
   show how they would be arranged in the array after the heap construction phase. (Build-Heap).
   How many key comparisons are done to construct the heap with these keys ?
5. Illustrate the performance of the heap-sort algorithm on the following input sequence ;
   $$\langle 2, 5, 16, 4, 10, 23, 39, 18, 26, 15 \rangle$$

CHAPTER 7

Quick Sort

7.1 Introduction

The basic version of quick sort algorithm was invented by C. A. R. Hoare in 1960 and formally introduced quick sort in 1962. It is used on the principle of divide-and-conquer. Quick sort is an algorithm of choice in many situations because it is not difficult to implement. It is a good "general purpose" sort and it consumes relatively fewer resources during execution.

Advantages

- It is in-place since it uses only a small auxiliary stack.
- It requires only $n \log(n)$ time to sort n items.
- It has an extremely short inner loop.
- This algorithm has been subjected to a thorough mathematical analysis ; a very precise statement can be made about performance issues.

Disadvantages

- It is recursive. Especially if recursion is not available, the implementation is extremely complicated.
- It requires quadratic (*i.e.*, n^2) time in the worst-case.
- It is fragile *i.e.*, a simple mistake in the implementation can go unnoticed and cause it to perform badly.

Quick sort works by partitioning a given array $A[p..r]$ into two non-empty sub arrays $A[p..q]$ and $A[q+1..r]$ such that every key in $A[p..q]$ is less than or equal to every key in $A[q+1..r]$. Then the two sub arrays are sorted by recursive calls to Quick sort. The exact position of the partition depends on the given array and index q is computed as a part of the partitioning procedure.

```
QUICK-SORT (A, p, r)
1.  If p < r then
2.      q ←PARTITION (A, p, r)
3.      QUICK SORT (A, p, q − 1)
4.      QUICK SORT (A, q + 1, r)
```

Note that to sort entire array, the initial call is Quick Sort (A, 1, length [A])

As a first step, Quick Sort chooses as pivot one of the items in the array to be sorted. Then array is partitioned on either side of the pivot. Elements that are less than or equal to pivot will move toward the left and elements that are greater than or equal to pivot will move toward the right.

7.2 Partitioning the Array

Partitioning procedure rearranges the sub arrays in-place.

```
PARTITION (A, p, r)
1.  x ← A[r]
2.  i ← p − 1
3.  for j ← p to r − 1
4.      do if A[j] ≤ x
5.          then i ← i + 1
6.              exchange A[i] ↔ A[j]
7.  exchange A[i + 1] ↔ A[r]
8.  return i + 1
```

Partition selects the first key, $A[p]$ as a pivot key about which the array will be partitioned :

- Keys $\le A[p]$ will be moved towards the left .
- Keys $\ge A[p]$ will be moved towards the right.

The running time of the partition procedure is $\theta(n)$ where $n = r - p + 1$ which is the number of keys in the array.

Another argument that running time of PARTITION on a sub array of size $\theta(n)$ is as follows: Pointer i and pointer j start at each end and move towards each other, conveying somewhere in the middle. The total number of times that i can be incremented and j can be decremented is therefore $O(n)$. Associated with each increment or decrement there are $O(1)$ comparisons and swaps. Hence, the total time is $O(n)$.

Example. "Using Fig. as a model, illustrate the operation of PARTITION on the array

$A = \langle 13,19,9,5,12,8,7,4,11,2,6,21 \rangle$".

13	19	9	5	12	8	7	4	11	2	6	21
13	19	9	5	12	8	7	4	11	2	6	21
13	19	9	5	12	8	7	4	11	2	6	21
13	19	9	5	12	8	7	4	11	2	6	21
13	19	9	5	12	8	7	4	11	2	6	21
13	19	9	5	12	8	7	4	11	2	6	21
13	19	9	5	12	8	7	4	11	2	6	21
13	19	9	5	12	8	7	4	11	2	6	21
13	19	9	5	12	8	7	4	11	2	6	21
13	19	9	5	12	8	7	4	11	2	6	21
13	19	9	5	12	8	7	4	11	2	6	21
13	19	9	5	12	8	7	4	11	2	6	21

Example. *What value of q does PARTITION return when all elements in the array $A[p..r]$ have the same value ?*

Solution. PARTITION will return r, identical elements are the worst-case for QUICK SORT since PARTITION will make the worst split : i and j both move to the right while elements are swapped with themselves.

Example. *Give a brief argument that the running time of PARTITION on a sub array of size n is $\theta(n)$.*

Solution. By examining the code we can see that

- Each element $A[I]$ is composed with x exactly once ;
- Each element $A[I]$ is swapped at most once.

Taken together, the running time must therefore be $\theta(n)$.

7.3 Performance of Quick Sort

The running time of quick sort depends on whether partition is balanced or unbalanced, which in turn depends on which elements of an array to be sorted are used for partitioning.

A very good partition splits an array up into two equal sized arrays. A bad partition, on other hand, splits an array up into two arrays of very different sizes. The worst partition puts only one element in one array and all other elements in the other array. If the partitioning is balanced, the Quick sort runs asymptotically as fast as merge sort. On the other hand, if partitioning is unbalanced, the Quick sort runs asymptotically as slow as insertion sort.

Best Case

The best thing that could happen in Quick sort would be that each partitioning stage divides the array exactly in half. In other words, the best to be a median of the keys in $A[p..r]$ every time procedure 'Partition' is called. The procedure 'Partition' always splits the array to be sorted into two equal sized arrays. If the procedure 'Partition' produces two regions of size $n/2$, the recurrence relation is then

$$T(n)=T(n/2)+T(n/2)+\theta(n)$$
$$=2T(n/2)+\theta(n)$$

And from case 2 of Master theorem

$$T(n)=\theta(n\lg n)$$

Worst case Partitioning

The worst-case occurs if given array $A[1..n]$ is already sorted. The PARTITION (A, p, r) call always returns p so successive calls to partition will split arrays of length n, $n-1$, $n-2,\ldots,2$ and running time proportional to $n+(n-1)+(n-2)+\ldots+2=[(n+2)(n-1)/2]=\theta(n^2)$. The worst-case also occurs if $A[1..n]$ starts out in reverse order.

Example. *Show that the running time of QUICK-SORT is $\theta(n^2)$ when the array A contains distinct elements and is sorted in decreasing order.*

Solution. We have thus an array whose elements are sorted in decreasing order, for instance $\langle 8,6,4,1\rangle$.

```
QUICK-SORT (A, p, r)
1.  if p < r then
2.      q ← PARTITION (A, p, r)
3.      QUICK-SORT (A, p, q − 1)
4.      QUICK-SORT (A, q + 1, r)
```

Since the pivot element is the leftmost element. PARTITION will make a lousy partitioning. Line 2 takes $\theta(n)$ time and sets $q \leftarrow p$. Line 3 takes $T(1)$ time. Line 4 takes $T(n-1)$ time.

In total, we get the recurrence

$$T(n)=T(n-1)+\theta(n)$$

Thus, $T(n)=\theta(n^2)$, by solving with recursion tree.

Example. *"Banks often record transaction on an account in order of the times of the transactions, but many people like to receive their bank statements with checks listed in order by check number. People usually write check in order by check number, and merchants usually cash them with reasonable dispatch. The problem of converting time-of-transaction ordering to check-number ordering is therefore the problem of sorting almost-sorted input. Argue that the procedure INSERTION-SORT would tend to beat the procedure QUICK-SORT on this problem."*

Solution. "Almost sorted" is almost the worst case for QUICK-SORT that is $\theta(n^2)$ and "Almost sorted" is almost the best case for INSERTION-SORT *i.e.*, $\theta(n)$.

Thus, in this problem, insertion-sort beats the Quick-sort.

Example. *"Show that quick sort's best-case running time is $\Omega(n\lg n)$.*

Solution. Best case : $T(n)=\min\limits_{1\le q\le n-1}(T(q)+T(n-q))+\theta(n)$

Guess that $T(n)\ge cn\lg n$ for some constant c

$$T(n)\ge \min_{1\le q\le n-1}(cq\lg q+c(n-q)\lg(n-q))+\theta(n)$$

$$=c\min_{1\le q\le n-1}(q\lg q+(n-q)\lg(n-q))+\theta(n)$$

Now, what is the minimum value of the function

$$f(q)=q\lg q+(n-q)\lg(n-q)\ ?$$

For $q=1$

$$f(q)=(n-1)\lg(n-1)$$

For $q=n-1$

$$f(q)=(n-1)\lg(n-1)$$

For $q=\frac{n}{2}$

$$f(q)=n\lg\left(\frac{n}{2}\right)<n\lg n.$$

Thus, the minimum appears somewhere between.

$$q=1 \text{ and } q=n-1.$$

Now, $$f'(q)=\frac{1}{q}q+\lg q+(n-q)\frac{1}{(n-q)}(-1)+(-1)\lg(n-q)$$

$$=1+\lg(q-1)-\lg(n-q)$$

$$=\lg q-\lg(n-q)$$

For minimum

$$f'(q)=0$$

i.e., $$\lg q-\lg(n-q)=0$$

which gives $q=\frac{n}{2}$

Put $q=\frac{n}{2}$ into the original equation.

$$T(n)\geq c\left(n\lg\left(\frac{n}{2}\right)\right)+\theta(n)$$
$$=cn\lg n-cn\lg 2+\theta(n)$$
$$\geq cn\lg n.$$

if we choose the constant c so that the constant in $\theta(n)>c\lg 2$.

Thus, quick-sort's best-case execution time is $\Omega(n\lg n)$.

Example. *"The running time of QUICK SORT can be improved in practice by taking advantage of the fast running time of insertion sort when its input is "nearly" sorted. When quick sort is called on a sub array with fewer than k elements, let it simply return without sorting the sub array.*

After the top-level call to quick sort return, run insertion sort on the entire array to finish the sorting process. Argue that this sorting algorithm runs in $O(nk+n\lg(n/k))$ expected time. How should k be picked, both in theory and in practice ?"

Solution. QUICK SORT without sorting sub arrays with less than k elements

$$T(n)=1 \text{ if } n<k$$
$$T(n)=2T(n/2)+n \text{ otherwise}$$

Expand the recurrence a few times.

$$T(n)=2T(n/2)+n$$
$$=n+2T(n/2)$$
$$=n+2(2T(n/2)+n/2)$$
$$=4T(n/4)+2n$$
$$=8T(n/8)+3n$$
$$\vdots$$

This goes on x times until $\frac{n}{2}x=k$ *i.e.,* $x=\lg\left(\frac{n}{k}\right)$

For the QUICK-SORT-part we get $T(n)=O(n\lg(n/k))$. The sorting is finished with an insertion sort where the number of elements is n elements, though divided up into blocks of k. It is important to notice that the elements do not move between blocks. Thus, we have n/k blocks that each can be sorted in $O(k^2)$ (worst-case), which gives $O(nk)$ time for the whole array.

The total becomes $O(nk)+O(n\lg(n/k))$

$$=O(nk+n\lg(n/k))$$

One should, of course pick k as to minimize the running time. For a certain n one can study the minimum of the function $f(n)=nk+n\lg(n/k)$.

In practice, one would run a series of bench marks to conclude the best k for various inputs. It suggests $k=9$.

Example. *"Professors Howard, Fine and Howard have proposed the following "elegant" sorting algorithm.*

```
STOOGE-SORT (A, i, j)
if A[i] > A[j]
    then exchange A[i] ↔ A[j]
if i + 1 ≥ j
    then return
k ← ⌊(j − i + 1) / 3⌋                ▷ Round down
STOOGE-SORT (A, i, j − k)            ▷ First two-thirds
STOOGE-SORT (A, i + k, j)            ▷ Last two-thirds.
STOOGE-SORT (A, i, j − k)            ▷ First two-thirds again.
```

(*a*) *Give a recurrence for the worst-case running time of STOOGE-SORT and a tight asymptotic (θ-notation) bound on the worst-case running time.*

(*b*) *Compare the worst-case running time of STOOGE-SORT with that of insertion sort, merge sort, heap sort, and quick sort. Do the professors deserve tenure ?"*

Solution. (*a*) The recurrence for STOOGE-SORT :

$$T(1)=\theta(1)$$
$$T(n)=3T(2n/3)+\theta(1) \text{ for } n>1$$

Apply master method.

$$f(n)=1 \quad a=3 \quad b=3/2 \quad \Rightarrow n^{\log_b a}=n^{\log_{3/2}3}$$
$$f(n)=1=O(n^{\log_{3/2}3-\epsilon}) \quad \text{for} \quad \epsilon=\log_{3/2}3.$$

which gives $$T(n)=\theta(n^{\log_{3/2}3})$$
$$\approx\theta(n^{2.7})$$

(*b*) Worst-case for a few sorting algorithms :

Insertion sort	:	$\theta(n^2)$
Merge sort	:	$\theta(n\lg n)$
Heap sort	:	$\theta(n\lg n)$
Quick sort	:	$\theta(n^2)$
Stooge sort	:	$\theta(n^{2.7})$

So, no tenure.

7.4 Versions of Quick Sort

Version 1. *Hoare's Algorithm*

```
QUICK SORT
    if p < r
    then q ← PARTITION (A, p, r)
        QUICK SORT (A, p, q)
        QUICK SORT (A, q + 1, r)
```

```
PARTITION (A, p, r)
1.  x ← A[p]
2.  i ← p − 1
3.  j ← r + 1
4.  while TRUE
5.      do repeat j ← j − 1
6.          until A[j] ≤ x
7.      repeat i ← i + 1
8.          until A[i] ≥ x
9.  if i < j
10.     then exchange A[i] ↔ A[j]
11. else return j
```

Technicalities :

- Pivot must be $A[p]$, not $A[r]$, if pivot is $A[r]$ and $A[r]$ contains the largest element in the array, then quick sort will loop forever.
- **Problem.** This version of partition can lead to one of the partitions containing 1 element.
  - This can happen when the smallest element is in $A[p]$.
  - When array is originally in sorted order, this leads to the worst case.

Version 2. *Lomuto's Algorithm*

```
QUICK SORT (A, p, r)
1.  if p < r
2.  then q ← PARTITION (A, p, r)
3.      QUICK SORT (A, p, q − 1)
4.      QUICK SORT (A, q + 1, r)
```



```
PARTITION (A, p, r)
1.  x ← A[r]
2.  i ← p − 1
3.  for j ← p to r − 1
4.      do if A[j] ≤ x
5.          then i ← i + 1
6.              exchange A[i] ↔ A[j]
7.  exchange A[i + 1] ↔ A[r]
8.  return i + 1
```

Version 3. Randomized Quick Sort

In the randomized version of Quick sort we impose a distribution on input. This does not improve the worst-case running time independent of the input ordering.

In this version we choose a random key for the pivot. Assume that procedure Random (a, b) returns a random integer in the range $[a, b)$; there are $b-a+1$ integers in the range and procedure is equally likely to return one of them. The new partition procedure, simply implemented the swap before actually partitioning.

```
RANDOMIZED-PARTITION (A, p, r)
1.  i ← RANDOM (p, r)
2.  exchange A[p] ↔ A[i]
3.  return PARTITION (A, p, r)
```

Now randomized quick sort call the above procedure in place of PARTITION

```
RANDOMIZED-QUICK SORT (A, p, r)
1.  if p < r
2.      then q ← RANDOMIZED-PARTITION (A, p ,r)
3.          RANDOMIZED-QUICK SORT (A, p, q)
4.          RANDOMIZED-QUICK SORT (A, q + 1, r)
```

Like other randomized algorithms, RANDOMIZED-QUICK SORT has the property that no particular input elicits its worst-case behavior; the behavior of algorithm only depends on the random-number generator. Even intentionally, we cannot produce a bad input for RANDOMIZED - QUICK SORT unless we can predict generator will produce next.

Version 4. MEDIAN-OF-3 PARTITION

```
Median-of-3-Partition (A, p, r)
1.  i ← RANDOM (p, r)
2.  j ← RANDOM (p, r)
3.  k ← RANDOM (p, r)
```

```
if (A[i] ≤ A[j] and A[j] ≤ A[k]) or (A[k] ≤ A[j] and A[j] ≤ A[i])
    then l ← j
else if (A[j] ≤ A[i] and A[i] ≤ A[k]) or (A[k] ≤ A[i] and A[i] ≤ A[j])
    then l ← i
else if (A[i] ≤ A[k] and A[k] ≤ A[j]) or (A[j] ≤ A[k] and A[k] ≤ A[i])
    then l ← k
exchange A[r] ↔ A[l]
return partition (A, p, r)
```

One common approach in median-of-3 method is choose the pivot as the median (middle element) of a set of 3 elements randomly selected. The median-of-3 partition algorithm makes it very improbable that the worst case partitioning will occur in every recursive call of quick sort. Thus this algorithm gives very good results in all but a few isolated cases.

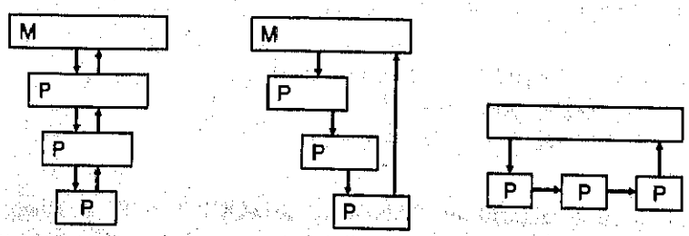
Stability

Quick Sort is not stable.

Tail Recursion

In the simple stack implementation of recursion, the local variables of the function will be pushed onto the stack as the recursive call is initiated. When the recursive call terminates, these local variables will be poped from the stack and thereby restored to their formal values.

But doing so is pointless if the very last action of a function is to make recursive call to itself as function now terminates and just restored local variables are immediately discarded.



(a) Recursion (b) Tail Recursion (c) Iteration

Thus, the special case when a recursive call is the last executed statement of the function, is called **tail recursion.**

A method used with Quick sort, which avoids the second recursive call, by using an iterative control structure is called tail recursion. A function is tail recursive if it either returns a value without making a recursive call, or returns directly the result of a recursive call. All possible branches of the function must satisfy one of these conditions.

If a function calls itself as the last thing it does then this call could be replaced by a "goto" to the beginning of the function after setting up the arguments correctly. For Dijkstra-aware programmers, this goto can be replaced by a while loop

This technique also works for calls to other functions although it is harder to use with some programming languages. Some compilers do this optimization automatically at higher levels of optimization.

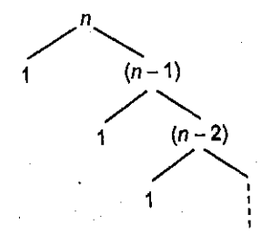
Example. *"Why do we analyze the average-case performance of a randomized algorithm and not its worst-case performance ?"*

Solution. Because the worst-case for a randomized algorithm is the same as the worst-case for the (non-randomized) original algorithm, which is rather a meaningless case to study.

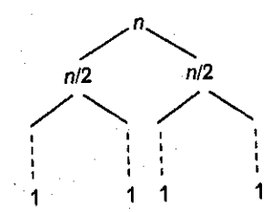
Example. *"During the running of the procedure RANDOMIZED-QUICK-SORT, how many calls are made in the random-number generator RANDOM in the worst case ? How about in the best case ? Give your answer in terms of the θ-notation".*

Solution. RANDOM is called the same number of times as RANDOMIZED-PARTITION.

◂ In the worst case this happens $\theta(n)$ times, the height of the following recursion tree.



◂ In the best case it happens $\theta(\lg n)$ times.



Exercise

1. Show how quick sort the following sequence of keys.

   (a) 2, 3, 18, 17, 5, 1

(*b*) 1, 1, 1, 1, 1, 1, 1

(*c*) 5, 5, 8, 9, 3, 4, 4, 3, 2

(*d*) 1, 3, 5, 7, 9, 12, 15

2. What is the running time of QUICK SORT when all elements of array A have the same value ?
3. Suppose that the splits at every level of quick sort are in the proportion $1-\alpha$ to α, where $0 < \alpha \le \frac{1}{2}$ is a constant.

   Show that the minimum depth of a leaf in the recursion tree is approximately $-\lg n / \lg \alpha$ and the maximum depth is approximately $-\lg n / \lg(1-\alpha)$.
4. Show that $q^2 + (n-q-1)^2$ achieves a maximum over $q = 0, 1, 2, \ldots n-1$ when $q = 0$ or $q = n-1$.
5. Show that RANDOMIZED-QUICK SORT's expected running time is $\Omega(n \lg n)$.
6. Show that randomized quick sort runs in $O(n \lg n)$ time with probability $1 - \frac{1}{n^2}$.

CHAPTER

8

Sorting in Linear Time

Merge sort, quick sort and heap sort algorithms share an interesting property: the sorted order they determine is based only on comparisons between the input elements. We call such sorting algorithms **comparison sorts**. Any comparison sort must make $\Omega(n \lg n)$ comparisons in the worst case to sort a sequence of n elements.

Heap Sort	$n \log n$
Insertion Sort	n^2
Quick Sort	n^2
Merge Sort	$n \log n$
Randomized Quick Sort	$n \log n$ expected

Figure 8.1

There are sorting algorithms that run faster than $O(n \lg n)$ time but they require special assumptions about the input sequence to be sort. Examples of sorting algorithms that run in linear time are counting sort, radix sort and bucket sort. Counting sort and radix sort assume that the

input consists of integers in a small range. Whereas, bucket sort assumes that a random process that distributes elements uniformly over the interval generates the input. Needless to say, these algorithms use operations other than comparisons to determine the sorted order.

8.1 Stability

We say that a sorting algorithm is *stable* if, when two records have the same key, they stay in their original order. This property will be important for extending bucket sort to an algorithm that works well when k is large. But first, which of the algorithms we've seen is stable ?

- Bucket sort ? Yes. We add items to the lists A[i] in order, and concatenating them preserves that order.
- Heap sort ? No. The act of placing objects into a heap (and heapifying them) destroys any initial ordering they might have.
- Merge sort ? Maybe. It depends on how we divide lists into two, and on how we merge them. For instance if we divide by choosing every other element to go into each list, it is unlikely to be stable. If we divide by splitting a list at its midpoint, and break ties when merging in favor of the first list, then the algorithm can be stable.
- Quick sort ? Again, maybe. It depends on how we do the partition step.

Any comparison-sorting algorithm can be made stable by modifying the comparisons to break ties according to the original positions of the objects, but only some algorithms are automatically stable.

8.2 Counting Sort

Counting sort assumes that each of the n input elements is an integer in the range 1 to k, for some integer k. When $k = O(n)$, the sort runs in $O(n)$ time. The basic idea of counting sort is to determine, for each input element x, the number of elements less than x. This information can be used to place element x directly into its position in the output array. For example, if there are 15 elements less than x, then x belongs in output position 16. This scheme must be modified slightly to handle the situation in which several elements have the same value, since we don't want to put them all in the same position. In the code for counting sort, we assume that the input is an array $A[1..n]$, and thus $length[A] = n$. We require two other arrays: the array $B[1..n]$ holds the sorted output, and the array $C[1..k]$ provides temporary working storage, where k is the highest number in array A.

```
COUNTING-SORT (A, B, k)
1.  for i ← 0 to k
2.      do C[i] ← 0
3.  for j ← 1 to length[A]
4.      do C[A[j]] ← C[A[j]] + 1
5.  /* C[i] now contains the number of elements equal to i. */
```



```
6.  for i ← 1 to k
7.      do C[i] ← C[i] + C[i - 1]
8.  /* C[i] now contains the number of elements less than or equal to i. */
9.  for j ← length[A] down to 1
10.     do B[C[A[j]]] ← A[j]
11.         C[A[j]] ← C[A[j]] - 1
```

An important property of counting sort is that it is ***stable:*** numbers with the same value appear in the output array in the same order as they do in the input array. That is, number appears first in the input array appears first in the output array. Of course, the property of stability is important only when satellite data are carried around with the element being sorted.

Example. *Suppose that the* **for** *loop in line 9 of the COUNTING-SORT procedure is rewritten :*

9 **for** $j \leftarrow 1$ **to** *length* $[A]$

Show that the algorithm still works properly. Is the modified algorithm stable ?

Solution. COUNTING-SORT will work correctly no matter what order A is processed in, however it is not stable. The modification to the for loop actually causes numbers with the same value to appear in reverse order in the output. Running a few examples we can see this.

Example. *Illustrate the operation of COUNTING-SORT on the array*

$$A = \langle 6,0,2,0,1,3,4,6,1,3,2 \rangle.$$

Solution.

	1	2	3	4	5	6	7	8	9	10	11
A	6	0	2	0	1	3	4	6	1	3	2

Here $k = 6$ (highest number in A)

For $i = 0$ to 6

$c[i] = 0$

i.e.,

	0	1	2	3	4	5	6
c	0	0	0	0	0	0	0

For $j \leftarrow 1$ to 11

	0	1	2	3	4	5	6
c	2	2	2	2	1	0	2

For $i = 1$ to 6.

	0	1	2	3	4	5	6
c	2	4	6	8	9	9	11

j	$A[j]$	$C[A[j]]$	$B[C[A[j]]] \leftarrow A[j]$	$C[A[j]] \leftarrow C[A[j]]-1$
11	2	6	$B[6] \leftarrow 2$	$C[2] \leftarrow 5$
10	3	8	$B[8] \leftarrow 3$	$C[3] \leftarrow 7$
9	1	4	$B[4] \leftarrow 1$	$C[1] \leftarrow 3$
8	6	11	$B[11] \leftarrow 6$	$C[6] \leftarrow 10$
7	4	9	$B[9] \leftarrow 4$	$C[4] \leftarrow 8$
6	3	7	$B[7] \leftarrow 3$	$C[3] \leftarrow 6$
5	1	3	$B[3] \leftarrow 1$	$C[1] \leftarrow 2$
4	0	2	$B[2] \leftarrow 0$	$C[0] \leftarrow 1$
3	2	5	$B[5] \leftarrow 2$	$C[2] \leftarrow 4$
2	0	1	$B[1] \leftarrow 0$	$C[0] \leftarrow 0$
1	6	10	$B[10] \leftarrow 6$	$C[6] \leftarrow 9$

	1	2	3	4	5	6	7	8	9	10	11
B	0	0	1	1	2	2	3	3	4	6	6

Thus, the final sorted array is B

8.3 Radix Sort

Radix-Sort is a sorting algorithm that is useful when there is a constant 'd' such that all the keys are d digit numbers. To execute Radix-Sort, for $p=1$ toward 'd' sort the numbers with respect to the pth digit from the right using any linear-time stable sort.

In a typical computer, which is a sequential random-access machine, radix sort is sometimes used to sort records of information that are keyed by multiple fields. For example, we might wish to sort dates by *three* keys: year, month, and day. We could run a sorting algorithm with a comparison function that, given two dates, compares years, and if there is a tie, compares months, and if another tie occurs, compares days. Alternatively, we could sort the information three times with a stable sort: first on day, next on month, and finally on year.

The code for radix sort is straightforward. The following procedure assumes that each element in the n-element array A has d digits, where digit 1 is the lowest-order digit and digit d is the highest-order digit.

```
RADIX-SORT (A, d)
for i ← 1 to d
    do use a stable sort to sort array A on digit i
```

Since a linear-time sorting algorithm is used 'd' times and d is a constant, the running time of Radix-Sort is linear. When each digit is in the range 1 to k, and k is not too large, counting sort is the obvious choice. Each pass over n d-digit numbers then takes time $\theta(n+k)$. There are d passes, so the total time for radix sort is $\theta(dn+kd)$. When d is constant and $k=O(n)$, radix sort runs in linear time.

Example : The first column is the input. The remaining columns show the list after successive sorts on increasingly significant digit positions. The vertical arrows indicate the digit position sorted on to produce each list from the previous one

329		720		720		329
457		355		329		355
657		436		436		436
839	⇒	457	⇒	839	⇒	457
436		657		355		657
720		329		457		720
355		839		657		839
		↑		↑		↑

8.4 Bucket Sort

Bucket sort runs in linear time on the average. Like counting sort, bucket sort is fast because it assumes something about the input. Whereas counting sort assumes that the input consists of integers in a small range, bucket sort assumes that the input is generated by a random process that distributes elements uniformly over the interval $U=[0,1)$.

To sort n input numbers, Bucket-Sort

1. partitions U into n non-overlapping intervals, called buckets,
2. puts each input number into its bucket,
3. sorts each bucket using a simple algorithm, *e.g.* Insertion-Sort, and then,
4. concatenates the sorted lists.

The idea of bucket sort is to divide the interval [0, 1) into n equal-sized subintervals, or **buckets**, and then distribute the n input numbers into the buckets. Since the inputs are uniformly distributed over [0, 1), we don't expect many numbers to fall into each bucket. To produce the output, we simply sort the numbers in each bucket and then go through the buckets in order, listing the elements in each.

Bucket sort assumes that the input is an n-element array A and that each element $A[i]$ in the array satisfies $0 \le A[i] < 1$. The code requires an auxiliary array $B[0..n-1]$ of linked lists (buckets) and assumes that there is a mechanism for maintaining such lists.

```
BUCKET-SORT (A)
n ← length [A]
for i ← 1 to n
    do insert A[i] into list B[⌊nA[i]⌋]
for i ← 0 to n − 1
    do sort list B[i] with insertion sort
concatenate the lists B[0], B[1],...B[n − 1] together in order
```

To see that this algorithm works, consider two elements $A[i]$ and $A[j]$. If these elements fall in the same bucket, they appear in the proper relative order in the output sequence because

insertion sort sorts their bucket. Suppose they fall into different buckets. Let these buckets be $B[i']$ and $B[j']$, respectively, and assume without loss of generality that $i' < j'$. When the lists of B are concatenated in line 6, elements of bucket $B[i']$ come before elements of $B[j']$, and thus $A[i]$ precedes $A[j]$ in the output sequence. Hence, we must show that $A[i] \le A[j]$. Assuming the contrary, we have

$$\begin{aligned} i' &= \lfloor nA[i] \rfloor \\ &\ge \lfloor nA[j] \rfloor \\ &= j', \end{aligned}$$

which is a contradiction, since $i' < j'$. Thus, bucket sort works.

To analyze the running time, observe that all lines except line 5 take $O(n)$ time in the worst case. The total time to examine all buckets in line 5 is $O(n)$, and so the only interesting part of the analysis is the time taken by the insertion sorts in line 5.

Since insertion sort runs in quadratic time, the expected time to sort the elements in bucket is

$$T(n) = \theta(n) + \sum_{i=0}^{n-1} O(n^2)$$

The total expected time to sort all the elements in all the buckets is therefore linear.

Despite of linear time usually these algorithms are not very desirable from practical point of view. Firstly, the efficiency of linear-time algorithms depend on the keys randomly ordered. If this condition is not satisfied, the result is the degrading in performance. Secondly, these algorithms require extra space proportional to the size of the array being sorted, so if we are dealing with large file, extra array becomes a real liability. Thirdly, the "inner-loop" of these algorithms contain quite a few instructions, so even though they are linear, they would not be as faster than quick sort.

Example. *Show how to improve the worst-case running time of bucket sort to $O(n \lg n)$.*

Solution. Simply replace the insertion sort used to sort the linked lists with some worst case $O(n \lg n)$ sorting algorithm, *e.g.* merge sort. The sorting then takes time:

$$\sum_{i=0}^{n-1} O(n_i \lg n_i) \le \sum_{i=0}^{n-1} O(n_i \lg n) = O(\lg n) \sum_{i=0}^{n-1} O(n_i) = O(n \lg n)$$

The total time of bucket sort is thus $O(n \lg n)$.

Example. Show how to sort n integers in the range 1 to n^2 in $O(n)$ time.

Solution. The number of digits used to represent an n^2 different numbers in a k-ary number system is $d = \log_k(n^2)$. Thus considering the n^2 numbers as radix n numbers gives us that :

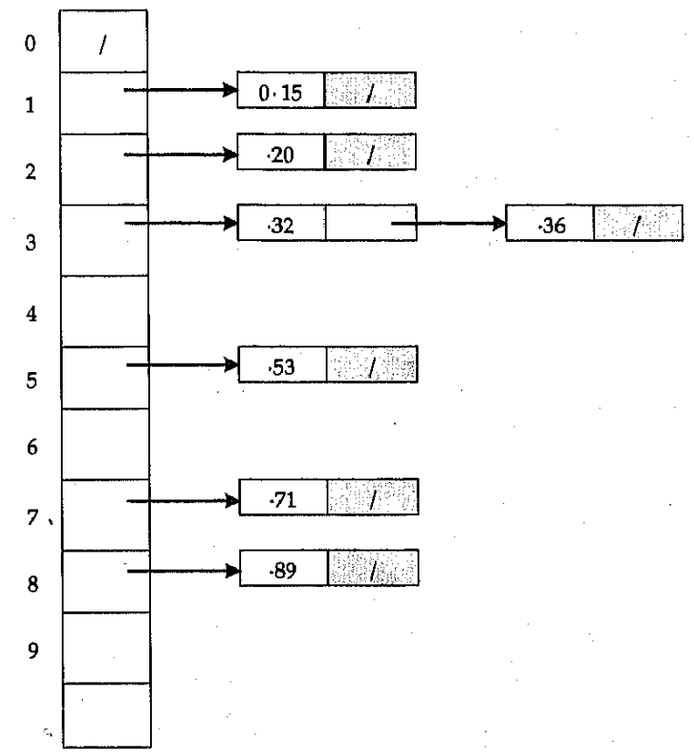
$$d = \log_n(n^2) = 2\log_n(n) = 2$$

Radix sort will then have a running time of $\theta(d(n+k)) = \theta(2(n+n)) = \theta(n)$.

Example . *Illustrate the operation of BUCKET-SORT on the array*

$$A = \langle 0.36, 0.15, 0.20, 0.89, 0.53, 0.71, 0.32 \rangle$$

Solution.



Now, Sorted array is

0	1	2	3	4	5	6
0.15	0.20	0.32	0.36	0.53	0.71	0.89

Exercise

1. Prove that counting sort is stable ?
2. What is a stable sorting algorithm ? Which of the sorting algorithm we have seen are stable and which are unstable ? Give the names.
3. Illustrate the operation of COUNTING SORT on the array

$$A = \langle 6, 14, 3, 25, 2, 10, 20, 3, 7, 6 \rangle$$

4. Suppose we radix sort n words of b bits each. Each could be viewed as having $\left(\frac{b}{r}\right)$ digits of r bits each, for some integer r.
   Now fill in the boxes :
   (a) Each pass of radix sort takes ☐ time.
   (b) At min r = ☐
   (c) $T(n,b)$ = ☐
5. What is time complexity of COUNTING-SORT ?
   Sort 1 9 3 3 4 5 6 7 7 8 by counting sort.
6. Modify the bucket sort algorithm such that the key of each record has value 0 or 1.

CHAPTER

9

Medians and Order Statistics

Let A be a set containing n distinct orderable elements : The ith order statistic is the number in A that is larger than exactly $i-1$ elements of A. We will consider some special cases of the order statistics problem :

- minimum = 1st order statistic
- maximum = nth order statistic
- median(s) = $\lceil (n+1)/2 \rceil$ and $\lfloor (n+1)/2 \rfloor$

9.1 Selection Problem

The *selection problem* can be specified formally as follows :

Input: a set A of n (distinct) numbers and a number i, with $1 \le i \le n$

Output: The element $x \in A$ that is larger than exactly $i-1$ other elements of A.

The selection problem can be solved in $O(n \lg n)$ time, since we can sort the numbers using heap sort or merge sort and then simply index the ith element in the output array.

9.2 Finding Minimum (or Maximum)

```
MINIMUM(A)
lowest ← A[1]
for i ← 2 to n do
    lowest ← min( lowest, A[i] )
```

Running Time :

- just scan input array
- exactly $n-1$ comparisons

9.3 Finding Minimum & Maximum Simultaneously

In some applications, we must find both the minimum and the maximum of a set of n elements. For example, a graphics program may need to scale a set of (x, y) data to fit onto a rectangular display screen or other graphical output device. To do so, the program must first determine the minimum and maximum of each coordinate.

It is not too difficult to devise an algorithm that can find both the minimum and the maximum of n elements using the asymptotically optimal $\Omega(n)$ number of comparisons.

Plan A : Find the minimum and maximum separately using $n-1$ comparisons for each $= 2n-2$ comparisons.

Plan B : Process elements in pairs. Compare pairs of elements from the input, first with each other, and then compare the smaller to the current min and the larger to the current max, changing current values of max and/or min if necessary. Simultaneous computation of max and min can be done in $3(n-3)/2$ steps.

```
MAX-AND-MIN (A, n)
max ← A[n]; min ← A[n]
for i ← 1 to ⌊n / 2⌋ do
    if A[2i − 1] ≥ max then
    {   if A[2i − 1] > max then
            max ← A[2i − 1]
        if A[2i] < min then
            min ← A[2i] }
    else {  if A[2i] > max then
            max ← A[2i]
            if A[2i − 1] < min then
            min ← A[2i − 1] }
return max and min
```

9.4 Selection of *i*th-order Statistic in Linear Time

Randomized-Select returns the ith smallest element of A. RANDOMIZED-SELECT uses the procedure RANDOMIZED-PARTITION .Thus, like RANDOMIZED-QUICKSORT, it is a randomized algorithm, since its behavior is determined in part by the output of a random-number generator. The following code for RANDOMIZED-SELECT returns the ith smallest element of the array $A[p..r]$.

```
RANDOMIZED-SELECT (A, p, r, i)
1.  if p = r
2.      then return A[p]
3.  q ← RANDOMIZED-PARTITION(A, p, r)
```



```
4.  k ← q − p + 1
5.  if i < k
6.      then return RANDOMIZED-SELECT (A, p, q, i)
7.      else return RANDOMIZED-SELECT (A, q + 1, r, i − k)
```

Randomized-Partition first swaps $A[r]$ with a random element of A and then proceeds as in the Partition subroutine of Quick sort.

```
RANDOMIZED-PARTITION (A, p, r)
1.  j ← Random(p, r)
2.  A[r] ↔ A[j]
3.  return Partition (A, p, r)
```

After RANDOMIZED-PARTITION is executed in line 3 of the RANDOMIZED-SELECT algorithm, the array $A[p..r]$ is partitioned into two nonempty subarrays $A[p..q]$ and $A[q+1..r]$ such that each element of $A[p..q]$ is less than each element of $A[q+1..r]$. Line 4 of the algorithm computes the number k of elements in the subarray $A[p..q]$. The algorithm now determines in which of the two subarrays $A[p..q]$ and $A[q+1..r]$ the ith smallest element lies. If $i < k$, then the desired element lies on the low side of the partition, and it is recursively selected from the subarray in line 6. If $i > k$, however, then the desired element lies on the high side of the partition. Since we already know k values that are smaller than the ith smallest element of $A[p..r]$ – namely, the elements of $A[p..q]$ – the desired element is the $(i-k)$th smallest element of $A[q+1..r]$, which is found recursively in line 7.

The worst-case running time for RANDOMIZED-SELECT is $\theta(n^2)$, even to find the minimum, because we could be extremely unlucky and always partition around the largest remaining element. The algorithm works well in the average case, though, and because it is randomized, no particular input elicits the worst-case behavior.

Example. *"Suppose we use RANDOMIZED-SELECT to select the minimum element of the array $A = \langle 3,2,9,0,7,5,4,8,6,1 \rangle$. Describe a sequence of partitions that results in a worst-case performance of RANDOMIZED-SELECT."*

Solution. Here is a sequence of partitions that results in a worst–case performance of RANDOMIZED-SELECT :

1	3	2	9	0	7	5	4	8	6	1	Choose 9 as a pivot element.
2	3	2	1	0	7	5	4	8	6	9	Choose 8 as a pivot element.
3	3	2	1	0	7	5	4	6	8		Choose 7 as a pivot element.
4	3	2	1	0	6	5	4	7			Choose 6 as a pivot element.
5	3	2	1	0	4	5	6				Choose 5 as a pivot element.
6	3	2	1	0	4	5					Choose 4 as a pivot element.
7	3	2	1	0	4						Choose 3 as a pivot element.
8	0	2	1	3							Choose 2 as a pivot element.
9	0	1	2								Choose 1 as a pivot element.
10	0	1									0 was the minimum element.

9.5 Worst-Case Linear-Time Order Statistics

We now examine a selection algorithm whose running time is $O(n)$ in the worst case. Like RANDOMIZED-SELECT, the algorithm SELECT finds the desired element by recursively partitioning the input array. The idea behind the algorithm, however, is to *guarantee* a good split when the array is partitioned. SELECT uses the deterministic partitioning algorithm PARTITION from quick sort modified to take the element to partition around as an input parameter.

SELECT (i, n)

1. Divide the n elements into groups of 5. Find the median of each 5-element group by rote.
2. Recursively SELECT the median x of the $\lfloor n/5 \rfloor$ group medians to be the pivot.
3. Partition around the pivot x. Let k = rank (x)
4. if $i = k$
   then return x
   elseif $i < k$
   then recursively SELECT the ith smallest element in the lower part
   else
   recursively SELECT the $(i-k)$th smallest element in the upper part

Example. *"In the algorithm SELECT, the input elements are divided into groups of 5. Will the algorithm work in linear time if they are divided into groups of 7 ? Argue that SELECT does not run in linear time if groups of 3 are used."*

Solution.

◄ **Groups of 7.**

The number of elements greater than $x \geq 4\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{7} \right\rceil \right\rceil\right) \geq \frac{4n}{14} - 8$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq d \\ T\left(\left\lceil \frac{n}{7} \right\rceil\right) + T\left(\frac{10n}{14} + 8\right) + O(n) & \text{if } n > d \end{cases}$$

Suppose $T(n) \leq cn$ for some c and all $n \leq d$

$$\begin{aligned} T(n) &\leq c\left\lceil \frac{n}{7} \right\rceil + c\left(\frac{10n}{14} + 8\right) + O(n) \\ &\leq c\frac{n}{7} + c + \frac{10cn}{14} + 8c + O(n) \\ &\leq \frac{12cn}{14} + 9c + O(n) \\ &\leq cn \end{aligned}$$

if $cn - \frac{12cn}{14} - 9c = c\left(n - \frac{12n}{14} - 9\right) = c\left(\frac{n}{7} - 9\right)$ dominates the constant in $O(n)$ then $T(n) \leq cn$, which happens when $n > 63$, since we can choose c freely.

◄ **Groups of 3.**

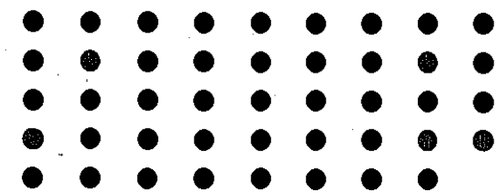
The number of elements greater than $x \geq 2\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{3} \right\rceil \right\rceil - 2\right) \geq \frac{2n}{6} - 4 = \frac{n}{3} - 4$.

Suppose $T(n) \leq cn$ for some c and all $n \leq d$.

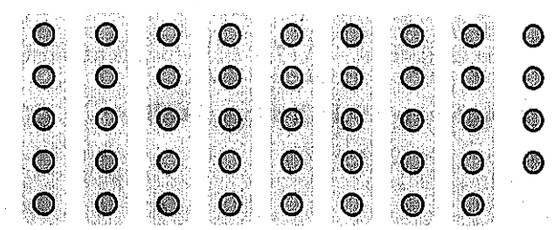
$$\begin{aligned} T(n) &\leq c\left\lceil \frac{n}{3} \right\rceil + c\left(\frac{2n}{3} + 4\right) + O(n) \\ &\leq \frac{cn}{3} + c + \frac{2cn}{3} + 4c + O(n) \\ &\leq cn + 5c + O(n) \not\leq cn \end{aligned}$$

In other words, 3 is too small.

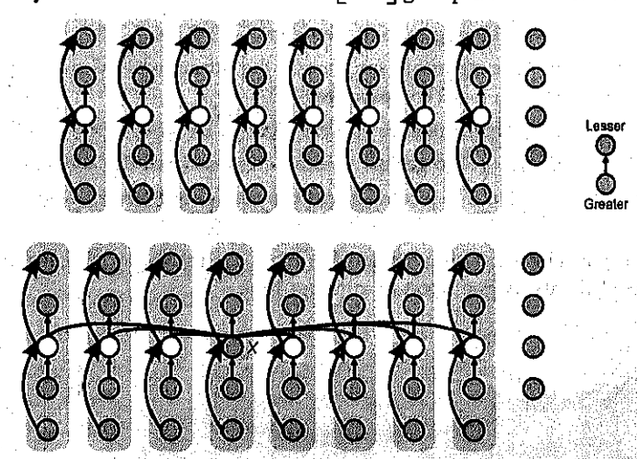
9.6 Choosing the Pivot



1. Divide the n elements into groups of 5. Find the median of each 5-element group by rote.



2. Recursively select the median x of the $\lfloor n/5 \rfloor$ group medians to be the pivot.



At least half the group medians are $\le x$, which is at least $\lfloor \lfloor n/5 \rfloor /2 \rfloor = \lfloor n/10 \rfloor$ group medians.

- Therefore, at least $3\lfloor n/10 \rfloor$ elements are $\le x$
- Similarly, at least $3\lfloor n/10 \rfloor$ elements are $\ge x$.

We can now develop a recurrence for the worst-case running time $T(n)$ of the algorithm SELECT. Steps 1, 2, and 4 take $O(n)$ time. (Step 2 consists of $O(n)$ calls of insertion sort on sets of size $O(1)$) Step 3 takes time $T(\lceil n/5 \rceil)$, and step 5 takes time at most $T(7n/10+6)$, assuming that T is monotonically increasing. Note that $7n/10+6<n$ for $n>20$ and that any input of 80 or fewer elements requires $O(1)$ time. We can therefore obtain the recurrence

$$T(n) \le T\left(\frac{n}{5}\right)+T\left(\frac{7n}{10}+6\right)+O(n)$$

We show that the running time is linear by substitution. Assume that $T(n) \le cn$ for some constant c and all $n>0$. Substituting this inductive hypothesis into the right-hand side of the recurrence yields

$$T(n) \le c\frac{n}{5}+c\left(\frac{7n}{10}+6\right)+O(n)$$

$$\le \frac{cn}{5}+c+\frac{7cn}{10}+6c+O(n)$$

$$=\frac{9cn}{10}+7c+O(n)=cn+\left(-\frac{cn}{10}+7c+an\right) \text{ which is at most } cn \text{ if } \left(-\frac{cn}{10}+7c+an\right) \le 0$$

The worst-case running time of SELECT is therefore linear *i.e.*, $O(n)$.

Exercise

1. Find the MEDIAN in the following set S of elements 1 8 3 3 5 4 7 7 6 8.
2. Design a divide-and-conquer algorithm for finding the minimum and the maximum element of n numbers using no more than $3n/2$ comparisons.
3. Show how to compute weighted median of n elements in $O(n \lg n)$ worst case time using sorting.
4. Write an algorithm MEDIAN (S) to get the median element from the sequence S of n elements.
5. Can you obtain less than $\lceil 3n/2 \rceil - 2$ comparison for obtaining both maximum and minimum of n number.
6. Devise an algorithm which can simultaneously obtain the minimum and maximum element from the given set of elements. Argue upon its running time.

CHAPTER

10

Dictionaries and Hash Tables

10.1 Dictionaries

A computer dictionary is similar to an ordinary dictionary because both are used to look things up. The main idea is that users can assign keys to elements and then use those keys later to look up or remove elements. So an abstract data type that support the operations insert, delete and search is called **dictionary**.

A dictionary stores key-element pairs (k, e) which we call items where k is the key and e is the element. We distinguish *two* types of dictionaries :

1. Unordered dictionary
2. Ordered dictionary

In either case, we use a key as an identifier that is assigned by an application or user to an associated element.

As an ADT, a dictionary D supports the following methods :

1. **Find element** (k) : If D contains an item with key equal to k then return the element of such item else return No-such-key.
2. **Insert item** (k, e) : Insert an item with element e and key k into dictionary D.
3. **Remove element** (k) : Remove from D an item with key equal to k and return its element. If D has no such item, then return the No-such-key.

10.2 Log Files

A simple way of realizing a dictionary D uses an unsorted sequence S which is implemented using a vector or list to store the key-element pairs. Such an implementation is called **log files**. The space required for a log file is $\theta(n)$ because both vector and linked list data structures can maintain their memory usage to be proportional to their size. The primary applications of a log file are situations where we wish to store small amounts of data or data that is unlikely to change much over time. We also refer to the log file implementation of D as an unordered sequence implementation.

In ordered dictionary, we wish to perform the usual dictionary operations and also maintain an order relation for the keys in our dictionary. We can use a comparator to provide the order relation among keys. Such an ordering helps us to efficiently implement the dictionary ADT. In addition, an ordered dictionary also supports the following methods :

Closest Key Before (k) : Return the key of the item with largest key less than or equal to k.

Closest Elem Before (k) : Return the element for the item with largest key less than or equal to k.

Closest Key After (k) : Return the key of the item with smallest key greater than or equal to k.

Closest Elem After (k) : Return the element for the item with smallest key greater than or equal to k.

Each of these methods returns the special No-such-key object if no item in the dictionary satisfies the query.

Thus, a dictionary is a data structure that supports the operations of Search, Insert, Delete.

There are *two* important variants of dictionary :

- **A static dictionary** is a restricted version that supports only the SEARCH operation. The goal here is to come up with a compact representation of the set while supporting very fast look-ups.
- When the elements of the set come from a totally ordered set, a dictionary on a total order supports MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR operations in addition to the three standard dictionary operations.

10.3 Hash Tables

Hash tables support one of the most efficient types of searching : ***hashing***. Fundamentally, a hash table consists of an array in which data is accessed via a special index called a *key*. The primary idea behind a hash table is to establish a mapping between the set of all possible keys and positions in the array using a *hash function*. A hash function accepts a key and returns its *hash coding*, or *hash value*. Keys vary in type, but hash codings are always integers. Since both computing a hash value and indexing into an array can be performed in constant time, the beauty of hashing is that we can use it to perform constant time searches. When a hash function can guarantee that no two keys will generate the same hash coding, the resulting hash table is said to be ***directly addressed***.

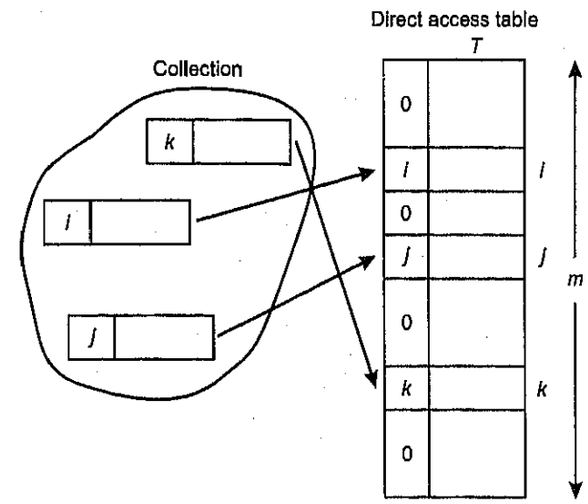


Figure 10.1

This is ideal, but direct addressing is rarely possible in practice.

Typically, the number of entries in a hash table is small relative to the universe of possible keys. Consequently, most hash functions map some keys to the same position in the table. When two keys map to the same position, they *collide*. A good hash function minimizes collisions, but we must still be prepared to deal with them.

10.3.1 Applications of Hash Tables

Some applications of hash tables are :

1. **Database systems.** Specifically, those that require efficient random access. Generally, database systems try to optimize between two types of access methods : sequential and random. Hash tables are an important part of efficient random access because they provide a way to locate data in a constant amount of time.

2. **Symbol tables.** The tables used by compilers to maintain information about symbols from a program. Compilers access information about symbols frequently. Therefore, it is important that symbol tables be implemented very efficiently.

3. **Tagged buffers.** A mechanism for storing and retrieving data in a machine-independent manner. Each data member resides at a fixed offset in the buffer. A hash table is stored in the buffer so that the location of each tagged member can be ascertained quickly. One use of a tagged buffer is sending structured data across a network to a machine whose byte ordering and structure alignment may not be the same as the original host's. The buffer handles these concerns as the data is stored and extracted member-by-member.

4. **Data dictionaries.** Data structures that support adding, deleting, and searching for data. Although the operations of a hash table and a data dictionary are similar, other data structures may be used to implement data dictionaries. Using a hash table is particularly efficient.

5. Associative arrays. Most commonly used in languages that do not support structured types. Associative arrays consist of data arranged so that the nth element of one array corresponds to the nth element of another. Associative arrays are useful for indexing a logical grouping of data by several key fields. A hash table helps to key into each array efficiently.

10.4 Hashing

Hashing is a widely used class of data structures that support the operations of insert, delete and search on a dynamic set of keys S. The keys are drawn from a universe $U(|U| >> |S|)$, which for our purposes will be a set of positive integers. These keys are mapped into a hash table using a hash function; the size of the hash table is usually within a constant factor of the number of elements in the current set S.

There are two main methods used to implement hashing : hashing with chaining and hashing with open addressing.

10.4.1 Hashing with Chaining

In hashing with chaining, the elements in S are stored in a hash table $T[0..m-1]$ of size m, where m is somewhat larger than n, the size of S. The hash table is said to have m slots. Associated with the hashing scheme is a hash function h which is a mapping from U to $\{0,..,m-1\}$. Each key $k \in S$ is stored in location $T[h(k)]$, and we say that key k is hashed into slot $h(k)$. If more than one key in S hashes into the same slot, then we have a **collision.**

In such a case, all keys that hash into the same slot are placed in a linked list associated with that slot; this linked list is called the **chain at that slot.** The **load factor** of a hash table is defined to be $\alpha = n/m$; it represents the average number of keys per slot. We typically operate in the range $m = \theta(n)$, so α is usually a constant (usually $\alpha < 1$). If a large number of insertions or deletions destroy this property, a more suitable value of m is chosen and the keys are re-hashed into a new table with a new hash function.

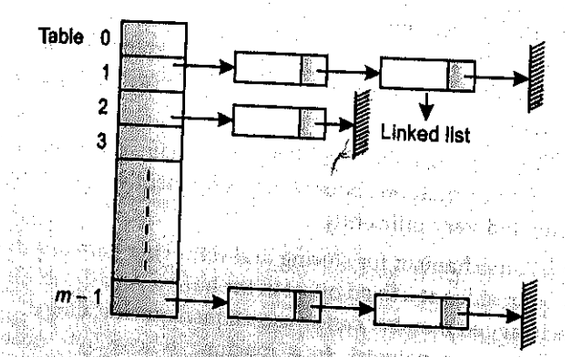


Figure 10.2

In simple uniform hashing we assume that we have a 'good' hash function h such that any element in U is equally likely to hash into any of the m slots in T, independent of the other keys in the table. We also assume that $h(k)$ can be computed in constant time.

In hashing with chaining, inserts and deletes can be performed in constant time with a suitable linked list representation for the chains. In the worst case, a search operation can take as long as n steps if all keys hash into the same slot. However, if we assume simple uniform hashing then the expected time for a search operation is easily shown to be $\theta(\alpha)$, which is a constant under the normal operation condition of $m = \theta(n)$. The main drawback with this method is the requirement that the scheme implements simple uniform hashing.

Analysis of hashing with chaining

Given a hash table T with m slots that stores n elements, we define the ***load factor*** α for T as n/m, that is, the average number of elements stored in a chain. The worst-case time for searching is thus $\theta(n)$ plus the time to compute the hash function—no better than if we used one linked list for all the elements. Clearly, hash tables are not used for their worst-case performance.

The average performance of hashing depends on how well the hash function h distributes the set of keys to be stored among the m slots, on the average.

Theorem

In a hash table in which collisions are resolved by chaining, an unsuccessful search takes time $\theta(1+\alpha)$, on the average, under the assumption of simple uniform hashing.

Proof : Under the assumption of simple uniform hashing, any key k is equally likely to hash to any of the m slots. The average time to search unsuccessfully for a key k is thus the average time to search to the end of one of the m lists. The average length of such a list is the load factor $\alpha = n/m$. Thus, the expected number of elements examined in an unsuccessful search is α, and the total time required (including the time for computing $h(k)$)) is $\theta(1+\alpha)$.

Example. *Let us consider the insertion of elements 5, 28, 19, 15, 20, 33, 12, 17, 10 into a chained-hash table. Let us suppose the hash table has 9 slots and the hash function be $h(k) = k \bmod 9$.*

Solution. The initial state of the chained-hash table

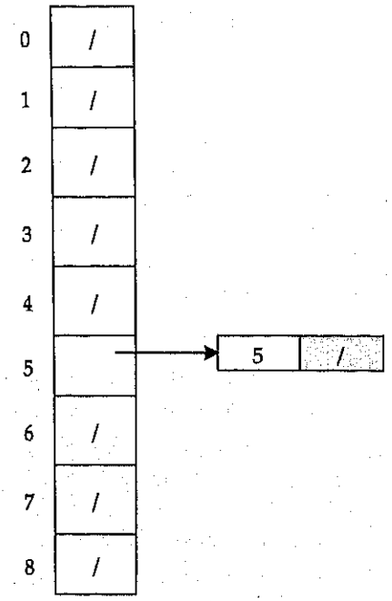
0	/
1	/
2	/
3	/
4	/
5	/
6	/
7	/
8	/

T

Insert 5 :

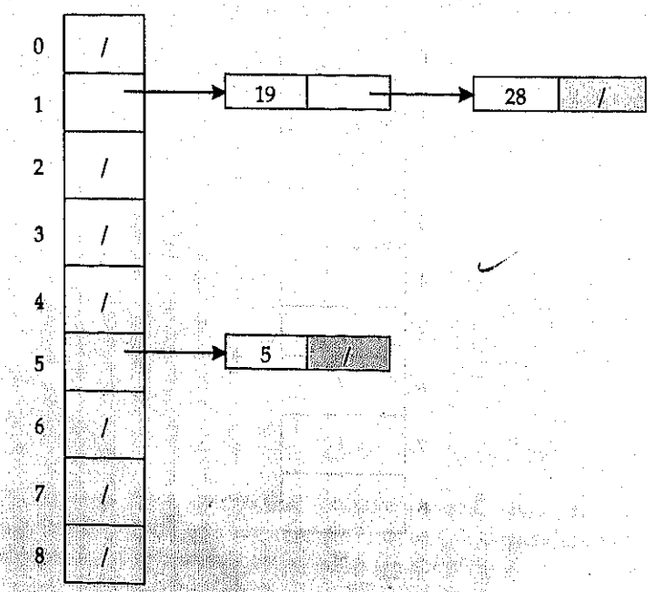
$h(5) = 5 \bmod 9 = 5$

Create a linked-list for T[5] and store value 5 in it.



Similarly insert 28. $h(28) = 28 \bmod 9 = 1$. Create a link-list for $T[1]$ and store value 28 in it.

Now insert 19 $h(19) = 19 \bmod 9 = 1$. Insert value 19 in the slot $T[1]$ in the beginning of the link-list.



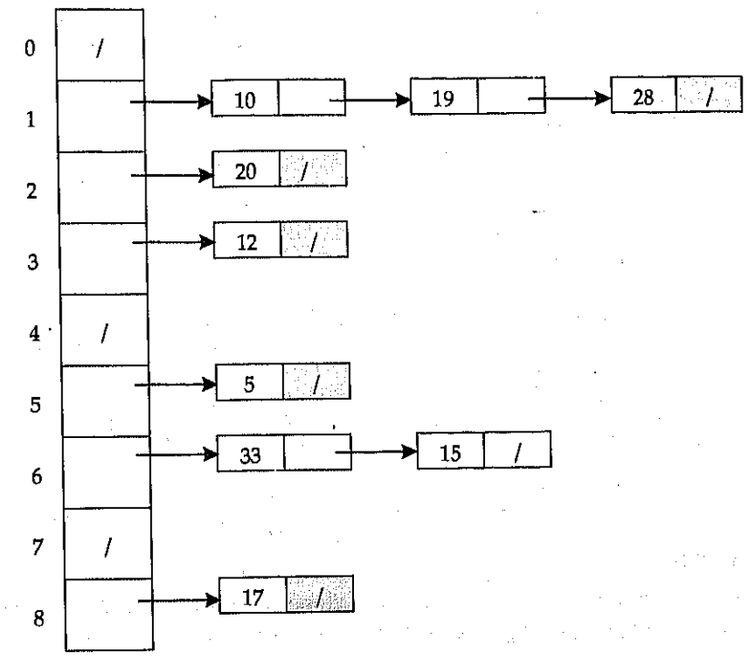
Now insert 15. $h(15) = 15 \bmod 9 = 6$. Create a link list for $T[6]$ and store value 15 in it. Similarly insert 20, $h(20) = 20 \bmod 9 = 2$ in $T[2]$. Insert 33, $h(33) = 33 \bmod 9 = 6$, in the beginning of the linked list for $T[6]$. Then,

Insert 12 *i.e.*, $h(12) = 12 \bmod 9 = 3$ in $T[3]$

Insert 17 *i.e.*, $h(17) = 17 \bmod 9 = 8$ in $T[8]$

Insert 10 *i.e.*, $h(10) = 10 \bmod 9 = 1$ in $T[1]$.

Thus the chained-hash-table after inserting key 10 is



10.5 Hash Functions

Hash Function is a function which, when applied to the key, produces an integer which can be used as an address in a hash table. The intent is that elements will be relatively randomly and uniformly distributed. **Perfect Hash Function** is a function which, when applied to all the members of the set of items to be stored in a hash table, produces a unique set of integers within some suitable range. Such function produces no collisions. **Good Hash Function** minimizes collisions by spreading the elements uniformly throughout the array.

There is no magic formula for the creation of the hash function. It can be any mathematical transformation that produces a relatively random and unique distribution of values within the address space of the storage.

Characteristics of a Good Hash Function

There are *four* main characteristics of a good hash function :

- The hash value is fully determined by the data being hashed.

- The hash function uses all the input data.
- The hash function "uniformly" distributes the data across the entire set of possible hash values.
- The hash function generates very different hash values for similar strings.

Let's examine why each of these is important:

Rule 1 If something else besides the input data is used to determine the hash, then the hash value is not as dependent upon the input data, thus allowing for a worse distribution of the hash values.

Rule 2 If the hash function doesn't use all the input data, then slight variations to the input data would cause an inappropriate number of similar hash values resulting in too many collisions.

Rule 3 If the hash function does not uniformly distribute the data across the entire set of possible hash values, a large number of collisions will result, cutting down on the efficiency of the hash table.

Rule 4 In real world applications, many data sets contain very similar data elements. We would like these data elements to still be distributable over a hash table.

However, finding a perfect hash function that works for a given data set can be extremely time consuming and very often it is just impossible. Therefore we must live with collisions and learn how to handle them.

Some common Hash functions are:

1. Division method

In the division method for creating hash functions, we map a key k into one of m slots by taking the remainder of k divided by m That is, the hash function is

$$h(k) = k \bmod m \quad \text{or} \quad h(k) = k \bmod m+1$$

For example, if the hash table has size $m=12$ and the key is $k=100$, then $h(k)=4$. Since it requires only a single division operation, hashing by division is quite fast.

When using the division method, we usually avoid certain values of m For example, m should not be a power of 2, since if $m=2^p$, then $h(k)$ is just the p lowest-order bits of k. Powers of 10 should be avoided if the application deals with decimal number of keys.

2. Multiplication method

The **multiplication method** for creating hash functions operates in two steps. First, we multiply the key k by a constant A in the range $0 < A < 1$ and extract the fractional part of kA. Then, we multiply this value by m and take the floor of the result. In short, the hash function is

$$h(k) = \lfloor m(k\,A \bmod 1) \rfloor,$$

where "$k\,A \bmod 1$" means the fractional part of kA, that is, $k\,A - \lfloor kA \rfloor$.

This method is not affected too much by the choice of m (which is the size of the hash table), but the value of A will impact the effectiveness of this method. 'Knuth' has suggested in his study that the following value of A is likely to work reasonably

$$A \approx \frac{(\sqrt{5}-1)}{2} = 0.6180339887 \ldots \text{ is a good choice.}$$

10.6 Universal Hashing

The main idea behind universal hashing is to select the hash function at random at run time from a carefully designed class of functions. This approach guarantees good average-case performance, no matter what keys are provided as input. Poor performance occurs only if the compiler chooses a random hash function that causes the set of identifiers to hash poorly, but the probability of this occurring is small and is the same for any set of identifiers of the same size.

Let H be a finite collection of hash functions that map a given universe U of keys into the range $\{0,1,\ldots m-1\}$. Such a collection is said to be **universal** if for each pair of distinct keys $x, y \in U$, the number of hash functions $h \in H$ for which $h(x) = h(y)$ is precisely $\frac{|H|}{m}$. In other words, with a hash function randomly chosen from H, the chance of a collision between x and y when $x \neq y$ is exactly $1/m$, which is exactly the chance of a collision if $h(x)$ and $h(y)$ are randomly chosen from the set $\{0,1,\ldots,m-1\}$.

10.7 Hashing With Open Addressing

In *open addressing*, all elements are stored in the hash table itself. That is, each table entry contains either an element of the dynamic set or NIL. When searching for an element, we systematically examine table slots until the desired element is found or it is clear that the element is not in the table. Thus, in open addressing, the load factor α can never exceed 1.

The **advantage** of open addressing is that it avoids pointers altogether. Instead of following pointers, we *compute* the sequence of slots to be examined. The extra memory freed by not storing pointers provides the hash table with a larger number of slots for the same amount of memory, potentially yielding fewer collisions and faster retrieval.

The process of examining the locations in the hash table is called **'Probing'**

To perform insertion using open addressing, we successively examine, or *probe*, the hash table until we find an empty slot in which to put the key.

```
HASH-INSERT (T, k)
i ← 0
repeat j ← h(k, i)
    if T [j] = NIL
        then T [j] ← k
        return j
    else i ← i + 1
until i = m
error "hash table overflow"
```

The algorithm for searching for key k probes the same sequence of slots that the insertion algorithm examined when key k was inserted. Therefore, the search can terminate (unsuccessfully) when it finds an empty slot, since k would have been inserted there and not later in its probe sequence. (Note that this argument assumes that keys are not deleted from the hash table.) The procedure HASH-SEARCH takes as input a hash table T and a key k, returning j if slot j is found to contain key k, or NIL if key k is not present in table T.

```
HASH-SEARCH(T, k)
i ← 0
repeat j ← h(k,i)
    if T[j] = j
        then return j
            i ← i + 1
until T[j] = NIL or i = m
return NIL
```

Deletion from an open-address hash table is difficult. When we delete a key from slot i, we cannot simply mark that slot as empty by storing NIL in it. Doing so might make it impossible to retrieve any key k during whose insertion we had probed slot i and found it occupied. One solution is to mark the slot by storing in it the special value DELETED instead of NIL. We would then modify the procedure HASH-SEARCH so that it keeps on looking when it sees the value DELETED, while HASH-INSERT would treat such a slot as if it were empty so that a new key can be inserted. When we do this, though, the search times are no longer dependent on the load factor α, and for this reason chaining is more commonly selected as a collision resolution technique when keys must be deleted.

Three techniques are commonly used to compute the probe sequences required for open addressing: **linear probing, quadratic probing, and double hashing.**

10.7.1 Linear Probing

Given an ordinary hash function $h' : U \{0,1,..,m-1\}$, the method of **linear probing** uses the hash function

$$h(k, i) = (h'(k) + i) \bmod m$$

where 'm' is the size of the hash table and $h'(k) = k \bmod m$ (basic hash function). For $i = 0,1,\ldots m-1$. Given key k, the first slot probed is $T[h'(k)]$. We next probe slot $T[h'(k)+1]$, and so on up to slot $T[m-1]$. Then we wrap around to slots $T[0]$, $T[1]$,..., until we finally probe slot $T[h'(k)-1]$. Since the initial probe position determines the entire probe sequence, only m distinct probe sequences are used with linear probing.

Example. *Consider inserting the keys 26, 37, 59, 76, 65, 86 into a hash-table of size $m = 11$ using linear probing, consider the primary hash function is $h'(k) = k \bmod m$.*

Solution. Initial state of the hash table

	0	1	2	3	4	5	6	7	8	9	10
T	/	/	/	/	/	/	/	/	/	/	/

1. Insert 26. We know $h(k,i) = [h'(k) + i] \bmod m$

Now $h(26,0) = [26 \bmod 11 + 0] \bmod 11$

$= (4+0) \bmod 11 = 4 \bmod 11 = 4$

Since $T[4]$ is free, insert key 26 at this place.

2. Insert 37. Now $h(37,0) = [37 \bmod 11 + 0] \bmod 11$

$= [4+0] \bmod 11 = 4$

Since $T[4]$ is not free, the next probe sequence is computed as

$h[37,1] = [37 \bmod 11 + 1] \bmod 11$

$= [4+1] \bmod 11 = 5 \bmod 11 = 5$

$T[5]$ is free, insert key 37 at this place.

3. Insert 59. Now $h(59,0) = [59 \bmod 11 + 0] \bmod 11$

$= [4+0] \bmod 11 = 4$

$T[4]$ is not free, so the next probe sequence is computed as

$h[59,1] = [59 \bmod 11 + 1] \bmod 11$

$= 5$

$T[5]$ is also not free, so the next probe sequence is computed.

$h(59,2) = (59 \bmod 11 + 2) \bmod 11$

$= 6 \bmod 11 = 6$

$T[6]$ is free. Insert key 59 at this place.

4. Insert 76. $h(76,0) = (76 \bmod 11 + 0) \bmod 11$

$= (10+0) \bmod 11 = 10$

$T[10]$ is free, insert key at this place.

5. Insert 65. $h(65,0) = (65 \bmod 11 + 0) \bmod 11$

$= (10+0) \bmod 11 = 10$

$T[10]$ is occupied, the next probe sequence is computed as

$h(65,1) = (65 \bmod 11 + 1) \bmod 11$

$= (10+1) \bmod 11 = 11 \bmod 11 = 0$

$T[0]$ is free, insert key 65 at this place.

6. Insert 86. $h(86,0) = (86 \bmod 11 + 0) \bmod 11$

$= 9 \bmod 11 = 9$

$T[9]$ is free, insert key 86 at this place.

Thus,

	0	1	2	3	4	5	6	7	8	9	10
T	65	/	/	/	26	37	59	/	/	86	76

One main disadvantage of linear probing is that records tend to **'Cluster'** i.e. appear next to one another, when the load factor is greater than 50%. Such clustering substantially increases the average search time for a record. Two techniques to minimize clustering are as follows :

10.7.2 Quadratic Probing

Suppose a record R with key k has the hash address $H(k)=h$ then instead of searching the locations with addresses h, $h+1$, $h+2$,... We linearly search the locations with addresses

$$h,\ h+1,\ h+4,\ h+9,\ldots,h+i^2,\ldots$$

Quadratic probing uses a hash function of the form

$$h(k,i)=(h'(k)+c_1 i+c_2 i^2)\bmod m$$

where (as in linear probing) h' is an auxiliary hash function, c_1 and $c_2 \neq 0$ are auxiliary constants, and $i=0,1,\ldots,m-1$. The initial position probed is $T[h'(k)]$; later positions probed are offset by amounts that depend in a quadratic manner on the probe number i. This method works much better than linear probing, but to make full use of the hash table, the values of c_1, c_2 and m are constrained.

Example. *Consider inserting the keys 76, 26, 37, 59, 21, 65 into a hash table of size $m=11$ using quadratic probing with $c_1=1$ and $c_2=3$. Further consider that the primary hash function is $h'(k)=k \bmod m$*

Solution. For quadratic probing, we have

$$h(k,i)=[k \bmod m+c_1 i+c_2 i^2]\bmod m$$

0	1	2	3	4	5	6	7	8	9	10
/	/	/	/	/	/	/	/	/	/	/

This is the initial state of the hash table.

Here $c_1=1$ $c_2=3$

$$h(k,i)=[k \bmod m+i+3i^2]\bmod m$$

1. Insert 76.

$$h(76,0)=(76 \bmod 11+0+3\times 0)\bmod 11$$
$$=(10+0+0)\bmod 11=10$$

$T[10]$ is free, insert the key 76 at this place.

2. Insert 26.

$$h(26,0)=(26 \bmod 11+0+3\times 0)\bmod 11$$
$$=(4+0+0)\bmod 11=4.$$

$T[4]$ is free, insert the key 26 at this place.

3. Insert 37.

$$h(37,0)=(37 \bmod 11+0+3\times 0)\bmod 11$$
$$=(4+0+0)\bmod 11=4$$

$T[4]$ is not free, so next probe sequence is computed as

$$h(37,1)=(37 \bmod 11+1+3\times 1^2)\bmod 11$$
$$=(4+1+3)\bmod 11$$
$$=8 \bmod 11=8$$

$T[8]$ is free. Insert the key 37 at this place.

4. Insert 59.

$$h(59,0)=(59 \bmod 11+0+3\times 0)\bmod 11$$
$$=(4+0+0)\bmod 11=4 \bmod 11=4$$

$T[4]$ is not free, so next probe sequence is computed as

$$h(59,1)=(59 \bmod 11+1+3\times 1^2)\bmod 11$$
$$=(4+1+3)\bmod 11=8 \bmod 11=8$$

$T[8]$ is also not free, so the next probe sequence is computed as

$$h(59,2)=(59 \bmod 11+2+3\times 2^2)\bmod 11$$
$$=(4+2+12)\bmod 11=18 \bmod 11=7$$

$T[7]$ is free, insert key 59 at this place.

5. Insert 21.

$$h(21,0)=(21 \bmod 11+0+3\times 0)\bmod 11$$
$$=(10+0+0)\bmod 11=10 \bmod 11=10$$

$T[10]$ is not free, the next probe sequence is computed as

$$h(21,1)=(21 \bmod 11+1+3\times 1^2)\bmod 11$$
$$=(10+1+3)\bmod 11=14 \bmod 11=3$$

$T[3]$ is free, so insert key 21 at this place.

6. Insert 65.

$$h(65,0)=(65 \bmod 11+0+3\times 0)\bmod 11$$
$$=(10+0+0)\bmod 11=10 \bmod 11=10$$

Since $T[10]$ is not free, the next probe sequence is computed as

$$h(65,1)=(65 \bmod 11+1+3\times 1^2)\bmod 11$$
$$=(10+1+3)\bmod 11=14 \bmod 11=3$$

$T[3]$ is not free, so the next probe sequence is computed as

$$h(65,2)=(65 \bmod 11+2+3\times 2^2)\bmod 11$$
$$=(10+2+12)\bmod 11=24 \bmod 11=2$$

$T[2]$ is free, insert the key 65 at this place.

Thus, after inserting all keys, the hash table is

0	1	2	3	4	5	6	7	8	9	10
/	/	65	21	26	/	/	59	37	/	76

10.7.3 Double hashing

Double hashing is one of the best methods available for open addressing because the permutations produced have many of the characteristics of randomly chosen permutations.

Double hashing uses a hash function of the form

$$h(k,i) = (h_1(k) + ih_2(k)) \bmod m,$$

where h_1 and h_2 are auxiliary hash functions and m is the size of the hash table.

$h_1(k) = k \bmod m$ or $h_2(k) = k \bmod m'$. Here m' is slightly less than m (say $m-1$ or $m-2$).

The initial position probed is $T[h_1(k)]$; successive probe positions are offset from previous positions by the amount $h_2(k)$ modulo m. Thus, unlike the case of linear or quadratic probing, the probe sequence here depends in two ways upon the key k, since the initial probe position, the offset, or both, may vary.

Double hashing represents an improvement over linear or quadratic probing in that $\Theta(n^2)$ probe sequences are used, rather than $\Theta(m)$, since each possible $(h_1(k), h_2(k))$ pair yields a distinct probe sequence, and as we vary the key, the initial probe position $h_1(k)$ and the offset $h_2(k)$ may vary independently. As a result, the performance of double hashing appears to be very close to the performance of the "ideal" scheme of uniform hashing.

Example. *Consider inserting the keys 76, 26, 37, 59, 21, 65 into a hash table of size $m=11$ using double hashing. Consider that the auxiliary hash functions are $h_1(k) = k \bmod 11$ and $h_2(k) = k \bmod 9$.*

Solution. Initial state of hash table is

	0	1	2	3	4	5	6	7	8	9	10
T	/	/	/	/	/	/	/	/	/	/	/

1. Insert 76.

$h_1(76) = 76 \bmod 11 = 10$

$h_2(76) = 76 \bmod 9 = 4$

$$h(76,0) = (10 + 0 \times 4) \bmod 11$$
$$= 10 \bmod 11 = 10$$

$T[10]$ is free, so insert key 76 at this place.

2. Insert 26.

$h_1(26) = 26 \bmod 11 = 4$

$h_2(26) = 26 \bmod 9 = 8$

$$h(26,0) = (4 + 0 \times 8) \bmod 11$$
$$= 4 \bmod 11 = 4$$

$T[4]$ is free, so insert key 26 at this place.

3. Insert 37.

$h_1(37) = 37 \bmod 11 = 4$

$h_2(37) = 37 \bmod 9 = 1$

$$h(37,0) = (4 + 0 \times 1) \bmod 11 = 4 \bmod 11 = 4$$

$T[4]$ is not free, the next probe sequence is computed as

$$h(37,1) = (4 + 1 \times 1) \bmod 11 = 5 \bmod 11 = 5$$

$T[5]$ is free, so insert key 37 at this place.

4. Insert 59.

$h_1(59) = 59 \bmod 11 = 4$

$h_2(59) = 59 \bmod 9 = 5$

$$h(59,0) = (4 + 0 \times 5) \bmod 11 = 4 \bmod 11 = 4$$

Since, $T[4]$ is not free, the next probe sequence is computed as

$$h(59,1) = (4 + 1 \times 5) \bmod 11 = 9 \bmod 11 = 9$$

$T[9]$ is free, so insert key 59 at this place.

5. Insert 21.

$h_1(21) = 21 \bmod 11 = 10$

$h_2(21) = 21 \bmod 9 = 3$

$$h(21,0) = (10 + 0 \times 3) \bmod 11 = 10 \bmod 11 = 10$$

$T[10]$ is not free, the next probe sequence is computed as

$$h(21,1) = (10 + 1 \times 3) \bmod 11 = 13 \bmod 11 = 2$$

$T[2]$ is free, so insert key 21 at this place.

6. Insert 65.

$h_1(65) = 65 \bmod 11 = 10$

$h_2(65) = 65 \bmod 9 = 2$

$$h(65,0) = (10 + 0 \times 2) \bmod 11 = 10 \bmod 11 = 10$$

$T[10]$ is not free, the next probe sequence is computed as

$$h(65,1) = (10 + 1 \times 2) \bmod 11 = 12 \bmod 11 = 1$$

$T[1]$ is free, so insert key 65 at this place.

Thus after insertion of all keys the final hash table is

0	1	2	3	4	5	6	7	8	9	10
/	65	21	/	26	37	/	/	/	59	76

10.8 Rehashing

If at any stage the hash table becomes nearly full, the running time for the operations will start taking too much time, insert operation may fail than in such situation, the best possible solution is as follows :

1. Create a new hash table double in size.
2. Scan the original hash table, compute new hash value and insert into the new hash table.
3. Free the memory occupied by the original hash table.

Example. *Suppose we wish to search a linked list of length n, where each element contains a key k along with a hash value h(k). Each key is a long character string. How might we take advantage of the hash values when searching the list for an element with a given key?*

Solution. Each key is a long character thus to compare keys, at every node we need to perform a string comparison operation which is very time consuming. Instead we generate a hash value for the key (*i.e.*, generate a numeric value for each string) we are searching for and comparing hash values $h(k)$ along the length of the list, which turns out to be numeric values and the comparison is faster.

Example. *Consider inserting the keys 10, 22, 31, 4, 15, 28, 17, 88, and 59 into a hash table of length* $m=11$ *using open addressing with the primary hash function* $h'(k)=k \bmod m$ *Illustrate the result of inserting these keys using linear probing, using quadratic probing with* $c_1=1$ *and* $c_2=3$, *and using double hashing with* $h_2(k)=1+(k \bmod (m-1))$.

Solution. Using Linear probing the final state of the hash table would be :

0	22
1	88
2	/
3	/
4	4
5	15
6	28
7	17
8	59
9	31
10	10

Using Quadratic probing, with $c_1=1$, $c_2=3$, the final state of the hash table would be $h(k,i)=(h'(k)+c_1*i+c_2*i^2) \bmod m$ where $m=11$ and $h'(k)=k \bmod m$

0	22
1	88
2	/
3	17
4	4
5	/
6	28
7	59
8	15
9	31
10	10

Using double hashing the final state of the hash table would be:

0	22
1	/
2	59
3	17
4	4
5	15
6	28
7	88
8	/
9	31
10	10

Example. *Consider an open-address hash table with uniform hashing. Give upper bounds on the expected number of probes in an unsuccessful search and on the expected number of probes in a successful search when the load factor is ¾ and when it is 7/8.*

Solution. Given an open address hash table with load factor $\alpha=n/m<1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$, assuming uniform hashing.

$\alpha=\frac{3}{4}$, then the upper bound on the number of probes $=\frac{1}{\left(1-\frac{3}{4}\right)}=4$ **probes.**

$\alpha=\frac{7}{8}$, then the upper bound on the number of probes $=\frac{1}{\left(1-\frac{7}{8}\right)}=8$ **probes.**

Given an open address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in a successful search is at most $(1/\alpha)\ \ln\ (1/(1-\alpha))$, assuming uniform hashing and assuming that each key in the table is equally likely to be searched for.

$$\alpha = \frac{3}{4}\left(\frac{1}{3/4}\right)\ln\frac{1}{1-\frac{3}{4}} = 1.85 \text{ probes}$$

$$\alpha = \frac{7}{8}\left(\frac{1}{.875}\right)\ln\frac{1}{1-.875} = 2.38 \text{ probes}$$

Example. *Professor Marley hypothesizes that substantial performance gains can be obtained if we modify the chaining scheme so that each list is kept in sorted order. How does the professor's modification affect the running time for successful searches, unsuccessful searches, insertions, and deletions ?*

Solution. Consider keeping the chaining lists in sorted order. Searching will still take time proportional to the length of the list and therefore the running times are the same. The only difference is the insertions which now also take time proportional to the length of the list.

Exercise

1. Write an algorithm to search a record in chained method.
2. For data 46, 78, 6, 2, 43, 27, 62, 4 give the result of inserting the values into a hash table when using open addressing with quadratic collision resolution ($c_1 = c_2 = 1$) and

   Hash $(k) = k\%17 \quad (M = 17)$.
3. What is Hashing ? Explain why we require the table size to be a prime number in double hashing.
4. Let the table S and 12 memory locations and suppose records A, B, C, D, E, X, Y and Z are to be inserted with following hash address.

Record	A	B	C	D	E	X	Y	Z
	4	8	2	12	4	12	5	1

   Find successful and unsuccessful linear probes.
5. Consider inserting the keys 20, 12, 31, 4, 25, 28, 27, 38, 69 into a hash table of length $m = 11$ using open addressing with the primary hash function $h_1(k) = k \bmod m$.

   Illustrate the result of inserting keys using double hashing with $h_2(k) = 1 + (k \bmod (m-1))$

CHAPTER

11

Elementary Data Structures

11.1 Introduction

Abstraction refers to the treatment of problems whilst ignoring associated concrete details such as computer hardware or implementation details. One simple example of this abstraction is provided by the pre-defined type integer to be found in many programming languages. It is possible to discuss the properties of integers and how to use them without necessarily having to know how they are represented at machine level. Another example of abstraction is provided by the high-level program, which represents, as it were, the solution to a problem on an abstract machine. The high-level programming language provides a convenient notation for problem analysis and solution. The details of the physical machine on which the program may be executed have been abstracted away.

It is this process of abstraction that makes programming feasible.

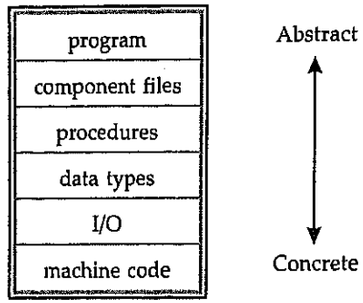


Figure 11.1

High-level programming languages make a selection of predefined atomic data types available. An atomic data type consists of a single component. These *primitive* types commonly include *integer, real or floating point, character,* and *boolean or logical*. In addition to these, programming languages offer a selection of predefined structured data types. A structured data type consists of a number of components associated together to form one logical whole. Simple examples are *arrays* and *array lists*. An array consists of a number of elements, which will be taken to be of the same basic type and in which the elements can be distinguished by means of an index. It should be appreciated that there is a considerable degree of abstraction here, which disguises the mapping between the indices and machine addresses. This abstraction is all the greater in the case of two-dimensional arrays since the basic machine memory is *linear*. An array list consists of a number of objects, which need not necessarily be of the same basic type.

Programming languages also provide a means of building new and more complex structures. The obvious example in Java is the *class*, which is a means of specifying objects, which comprise a set of variables, together with methods to do operations on those variables. An object displays *representational abstraction*, in as much as the implementation of the variables may be hidden from the outside, and *procedural abstraction*, in that the implementation of the methods is hidden. We will say lots more about this. In the mean time it is important to recognize that each data type has an associated set of operations by means of which the data values may be manipulated.

Data structures store a given set of data according to certain rules. These rules help in organizing data. Arrays and structures are built in data structures in most of the programming languages. Data can be stored in many other ways using other type of data structures. Thus, we can define data structure as a collection of data elements whose organization is characterized by accessing functions that are used to store and retrieve individual data elements. Data structures are represented at the logical level using a tool called **Abstract data type (ADT)** and actually implemented using algorithms.

11.2 Abstract Data Type

An ***abstract data type*** *(ADT)* consists of a data type together with a set of operations, which define how the type may be manipulated. This specification is stated in an implementation-independent manner.

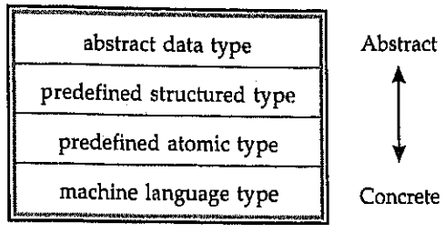


Figure 11.2

Abstract data types exist conceptually and concentrate on the (essential) mathematical properties of the data type ignoring implementation constraints and details. *Virtual* data types exist on a virtual processor such as a programming language. *Physical* data types exist on a physical processor such as the machine level of a computer. Abstract data types are implemented with virtual data types. Virtual data types are translated into physical data types. The advantages offered by abstract data types include :

1. modularity
2. precise specifications
3. information hiding
4. simplicity
5. integrity
6. implementation independence

While defining an ADT, we are not concerned with time and space efficiency or any other implementation details of the data structure. ADT is just a useful guideline to use and implement the data type.

An ADT has *two* parts :

1. Value definition
2. Operation definition

Value definition is again divided into *two* parts :

1. Definition clause
2. Condition clause

As the name suggests the **definition clause** states the contents of the data type and **condition clause** defines any condition that applies to the data type. **Definition clause** is mandatory while **condition clause** is optional.

In operational definition, there are *three* parts :

- Function
- Precondition
- Post condition

The function clause defines the role of the operation. If we consider the addition operation in integers, the function clause will state that two integers can be added using this function. In general, precondition specifies any restrictions that must be satisfied before the operation can be applied. This clause is optional. If we consider the division operation on integers then the

precondition will state that the divisor should not be zero. Precondition specifies any condition that may apply as a pre-requisite for the operation definition. Post condition specifies what the operation does *i.e.* it specifies the state after the operation is performed

The following figure 11.3 lists some typical abstract data types :

Vector	Direct access to each element and Fixed size.
List	*Unbounded.* Fast access to first element and thereafter sequential access.
Stack	*LIFO structure.* Insertions and deletions from one end only.
Queue	*FIFO structure.* Insertions and deletions from the ends.
Tree	Fast insertions and access. Hierarchical structures.
Hash table	Fast access.
Set	Fast unions and intersections. Elements not maintained in order.
Bag	As with sets but allows multiple occurrences of elements.
Dictionary	Values associated with key.
Graph	Collection of nodes and arcs. Unordered
File	External storage for persistent data. Slow access.

Figure 11.3

11.3 Stack

A stack is a temporary abstract data type and data structure based on the principle of **Last In First Out** (LIFO). Stacks are used extensively at every level of a modern computer system. Stack is a container of nodes and has two basic operations : **push** and **pop**. Push adds a given node to the top of the stack leaving previous nodes below. Pop removes and returns the current top node of the stack.

We can implement a stack of at most n elements with an array $S[1..n]$. The top $[S]$ indexes the most recently inserted element. The stack consists of elements S [1...top [S]], where $S[1]$ is the bottom element of the stack and S [top[S]] is the element at the top. When top[S]=0 the stack contains no elements and is empty. The stack can be tested for emptiness by the operation STACK-EMPTY. If an empty stack is popped, we say the stack **underflows**. If top[S] exceeds n, the stack **overflows**

The Stack operations can be implemented with following procedures

```
STACK-EMPTY (S)
1.   if top[S]=0
2.       then return TRUE
3.       else return FALSE
```



```
PUSH (S, x)
1.   top [S]← top[S]+1
2.   S [top [S]]←x
```

```
POP (S)
1.   if STACK-EMPTY(S)
2.       then error "underflow"
3.       else  top[S] ← top[S] − 1
4.            return S [top [S] + 1]
```

Each of the three stack operations takes $O(1)$ time.

11.3.1 Applications of Stacks

(a) Polish Notation

For most common arithmetic operations, the operator symbol is placed between its two operands. For example,

$$A+BC-DE*FG/H$$

This is called infix notation. With this notation, we must distinguish between $(A+B)*C$ and $A+(B*C)$ by using either parentheses or some operator-precedence convention such as the usual precedence levels. The order of the operators and operands in an arithmetic expression does not uniquely determine the order in which the operations are to be performed.

Polish notation, named after the Polish mathematician **Jan Lukasiewicz**, refers to the notation in which the operator symbol is placed before its two operands. For example,

$$+AB-CD*EF/GH$$

We translate, step by step, the following infix expressions into Polish notation using brackets [ ] to indicate a partial translation:

$$(A+B)*C=[+AB]*C=*+ABC$$

$$A+(B*C)=A+[*BC]=+A*BC$$

$$(A+B)/(C-D)=[+AB]/[-CD]=/+AB-CD$$

The fundamental property of Polish notation is that the order in which the operations are to be performed is completely determined by the positions of the operators and operands in the expression. Accordingly, one never needs parentheses when writing expressions in Polish notation.

Reverse Polish notation refers to the analogous notation in which the operator symbol is placed after its two operands :

$$AB+CDEF*GH/$$

Again, one never needs parentheses to determine the order of the operations in any arithmetic expression written in reverse Polish notation. This notation is frequently called postfix (or suffix) notation.

The computer usually evaluates an arithmetic expression written in infix notation in two steps. First, it converts the expression to postfix notation, and then it evaluates the postfix expression. In each step, the stack is the main tool that is used to accomplish the given task.

(b) Evaluation of a Postfix Expression

Suppose P is an arithmetic expression written in postfix notation. The following algorithm, which uses a STACK to hold operands, evaluates P.

Algorithm

This algorithm finds the VALUE of an arithmetic expression P written in postfix notation.

1. Add a right parenthesis ")" at the end of P. [This acts as a sentinel.]
2. Scan P from left to right and repeat Steps 3 and 4 for each element of P until the sentinel ")" is encountered.
3. If an operand is encountered, put it on STACK.
4. If an operator x is encountered, then :
   a. Remove the two top elements of STACK, where A is the top element and B is the next-top element.
   b. Evaluate B.A.
   c. Place the result of (*b*) back on STACK.
   [End of If structure.]

   [End of Step 2 loop.]
5. Set VALUE equal to the top element to STACK.
6. Exit.

We note that, when Step 5 is executed, there should be only one number on STACK.

(c) Transforming Infix Expressions into Postfix Expressions

Let Q be an arithmetic expression written in infix notation. Besides operands and operators, Q may also contain left and right parentheses. We assume that the operators in Q consist only of exponentiations ($\cdot$), multiplications ($*$), divisions ($/$), additions ($+$) and subtractions ($-$), and that they have the usual three levels of precedence as given above.

We also assume that operators on the same level, including exponentiation, are performed from left to right unless otherwise indicated by parentheses. (This is not standard, since expressions may contain unary operators and some languages perform the exponentiation from right to left. However, these assumptions simplify our algorithm.)

The following algorithm transforms the infix expression Q into its equivalent postfix expression P. The algorithm uses a stack to temporarily hold operators and left parentheses. The

postfix expression P will be constructed from left to right using the operands from Q and the operators, which are removed from STACK. We begin by pushing a left parenthesis onto STACK and adding a right parenthesis at the end of Q. The algorithm is completed when STACK is empty.

Algorithm

Suppose Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression P.

POLISH (Q, P)

1. Push "(" onto STACK, and add ")" to the end of Q.
2. Scan Q from left to right and repeat Steps 3 to 6 for each element of Q until the STACK is empty :
3. If an operand is encountered, add it to P.
4. If a left parenthesis is encountered, push it onto STACK.
5. If an operator . is encountered, then:
   (*a*) Repeatedly pop from STACK and add to P each operator (on the top of STACK) which has the same precedence as or higher precedence than.
   (*b*) Add. To STACK.
   [End of If structure.]
6. If a right parenthesis is encountered, then:
   (*a*) Repeatedly pop from STACK and add to P each operator (on the top of STACK) until a left parenthesis is encountered.
   (*b*) Remove the left parenthesis. [Do not add the left parenthesis to P.]
   [End of If structure.]
   [End of Step 2 loop.]
7. Exit

11.4 Queue

A queue is a linear list in which items may be added only at one end and items may be removed-only at the other end. The name "queue" likely comes from the everyday use of the term. Consider a queue of people waiting at a bus stop. Each new person who comes takes his or her place at the end of the line, and when the bus comes, the people at the front of the line board first. Clearly, the first person in the line is the first person to leave. Thus queues are also called **first-in first-out** (FIFO) lists. Another example of a queue is a batch of jobs waiting to be processed, assuming no job has higher priority than the others. A queue is a linear list of elements in which deletions can take place only at one end, called the head (front), and insertions can take place only at the other end, called the tail (rear). The terms "front" and "rear" are used in describing a linear list only when it is implemented as a queue. Queues are also called first-in first-out (FIFO) lists, since the first element in a queue will be the first element out of the queue. In other words, the order in which elements enter a queue is the order in which they leave. This contrasts with stacks, which are last-in first-out (LIFO) lists.

We call the INSERT operation on a queue ENQUEUE and we call the delete operation DEQUEUE. In ENQUEUE and DEQUEUE procedure the error checking for underflow and overflow has been omitted.

```
ENQUEUE (Q, x)
Q [tail[Q]] ← x
if tail [Q] = length [Q]
    then tail [Q] ← 1
    else tail[Q]←tail [Q] + 1
```

```
DEQUEUE (Q)
x ←Q [ head[Q] ]
if head[Q] = length[Q]
    then head[Q] ← 1
    else head[Q] ← tail[Q] + 1
return x
```

Both ENQUEUE and DEQUEUE operation takes O (1) time.

Example. *Explain how to implement two stacks in one array A[1..n] in such a way that neither stack overflows unless the total number of elements in both stacks together is n. The PUSH and POP operations should run in O(1) time.*

Solution. Two stacks can be implemented in a single array without overflows occurring if they grow from each end and towards the middle.

Example. *Show how to implement a queue using two stacks. Analyze the running time of the queue operations.*

Solution. Implement a queue using two stacks. Denote the two stacks *S1* and *S2*. The ENQUEUE operation is simply implemented as a push on *S1*. The dequeue operation is implemented as a pop on *S2*. If *S2* is empty, successively pop *S1* and push *S2*. This reverses the order of *S1* onto *S2*. The worst-case running time is $O(n)$.

```
Q-Insert (x)
if (top = max)
    then error "over flow"
else S1[top+1]← x
```

```
Q-delete ( )
if (top = 0)
    then error "under flow"
    flag ← 0
else t ← 0
while (top < 1)
{ x ← POP(S1)
```



```
    PUSH(S2, x)
    t ← t + 1
    top ← top - 1
}
y ← POP(S2)
top ← top-1
while (t ≠ 0)
{ x ← POP(S2)
    PUSH (S1, x)
    top ← top +1
}
return y
```

Example. *Show how to implement a stack using two queues. Analyze the running time of the stack operations.*

Solution. To implement the stack using two queues *Q1* and *Q2*, we can simply ENQUEUE elements into *Q1* whenever a push call is made. This takes $O(1)$ time. For POP calls we can DEQUEUE all elements of *Q1* and ENQUEUE them into *Q2* except for the last element which we set aside in a temp variable. We then return the elements to *Q1* by DEQUEUE from *Q2* and ENQUEUE into *Q1*. The last element that we set aside earlier is then return as the result of the POP. Thus POP takes $O(n)$ time.

11.4.1 Types of Queues

1. DEQUES

A deque (pronounced either "deck" or "dequeue") is a linear list in which elements can be added or removed at either end but not in the middle. The term deque is a contraction of the name **double-ended queue**.

There are various ways of representing a deque in a computer. Here, we will assume our deque is maintained by a circular array DEQUE with pointers LEFT and RIGHT, which point to the two ends of the deque. We assume that the elements extend from the left end to the right end in the array. The term "circular" comes from the fact that we assume that DEQUE [1] comes after DEQUE [*N*] in the array. The condition LEFT = NULL will be used to indicate that a deque is empty.

There are *two* kinds of a deque

- **Input-restricted deque** is a deque, which allows insertions at only one end of the list but allows deletions at both ends of the list;
- **Output-restricted deque** is a deque, which allows deletions at only one end of the list but allows insertions at both ends of the list.

2. PRIORITY QUEUES

A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules :

1. An element of higher priority is processed before any element of lower priority.
2. Two elements with the same priority are processed according to the order in which they were added to the queue.

A prototype of a priority queue is a timesharing system: programs of high priority are processed first, and programs with the same priority form a standard queue.

11.5 Linked-List

Linked lists and arrays are similar since they both store collections of data. The terminology is that arrays and linked lists store "elements" on behalf of "client" code. The specific type of element is not important since essentially the same structure works to store elements of any type. Linked lists have their own strengths and weaknesses, but they happen to be strong where arrays are weak. The array's features all follow from its strategy of allocating the memory for all its elements in one block of memory. Linked lists use an entirely different strategy. **Linked lists allocate memory for each element separately and only when necessary.**

A **linked list**, or **one-way list**, is a linear collection of data elements, called **nodes**, where the linear order is given by means of **pointers**. That is, each node is divided into two parts: the **first part** contains the information of the element, and the second part, called the link field or next pointer field, contains the address of the next node in the list.

11.5.1 Searching a node in a linked-list

The procedure LIST-SEARCH (L, k) finds the first element with key k in the list L by a simple linear search, returning a pointer to this element. If no object with key k appears in the list, then NIL is returned.

```
LIST-SEARCH (L, x)
1.  x ← head [L]
2.  while x ≠ NIL and key[x] ≠ k
3.      do x ← next[x]
4.  return x
```

To search a list of n objects, the LIST-SEARCH procedure takes $\theta(n)$ time in the worst case, since it may have to search the entire list.

11.5.2 Inserting into a linked-list

LIST-INSERT procedure inserts x onto the front of the linked list.

```
LIST-INSERT (L, x)
1.  next[x] ← head[L]
2.  if head[L] ≠ NIL
3.      then prev[head[L]] ← x
4.  head[L] ← x
5.  prev[x] ← NIL
```

The running time of LIST-INSERT procedure on a list of n elements is $O(1)$.

11.5.3 Deleting from a linked-list

The procedure LIST-DELETE removes an element x from a linked list L If we wish to delete an element with a given key, we must first call LIST- SEARCH to retrieve a pointer to the element. LIST-DELETE runs in $O(1)$ time , but if we wish to delete an element with a given key, $\theta(n)$ time is required in the worst case because we must first call LIST-SEARCH.

```
LIST-DELETE (L, x )
1.  if prev[x] ≠ NIL
2.      then next[prev[x] ] ← next[x]
3.      else head[L] ← next[x]
4.  if next[x] ≠ NIL
5.  then prev [next[x] ] ← prev [x]
```

11.6 Binary Tree

We represent each node of a tree by an object. As with linked lists, we assume that each node contains a **key** field. The remaining fields of interest are pointers to other nodes, and they vary according to the type of tree. We use the fields p, *left*, and *right* to store pointers to the parent, left child, and right child of each node in a binary tree T. If $p[x]$ = NIL, then x is the root. If node x has no left child, then *left* $[x]$ = NIL, and similarly for the right child. The root of the entire tree T is pointed to by the attribute *root*$[T]$. If *root* $[T]$ = NIL, then the tree is empty.

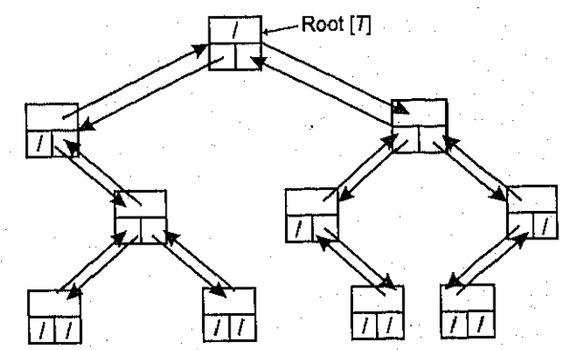


Figure 11.4 The representation of a binary tree T. Each node x has the fields $p[x]$ (top), left[x] (lower left), and right[x] (lower right). The key fields are not shown.

Exercise

1. Show how to implement a queue using two stacks. Analyze the running time of the queue operations.

2. A binary tree has 9 nodes. The Inorder and Preorder traversal of tree yield the following sequence of nodes :

   INORDER : E A C K F H D B G

   PREORDER : F A E K C D H G B

   Draw the tree.

3. Write an algorithm for insertion and deletion of elements for a queue. Use a boolean variable to distinguish between a queue being empty or full.
4. Write a C program that will split a circularly linked list into two circularly linked lists.
5. Convert the following infix expressions to the prefix expressions. Show all steps :

   (*i*) $A \wedge B * C - D + E / F / (G + H)$

   (*ii*) $(A + B) * (C \wedge (D - E) + F) - G$

6. Write a "C" program using stack to check whether a string is palindrome or not. Do not define empty, push, pop functions.

CHAPTER

12

Binary Search Tree

A binary search tree is organized in a binary tree. Such a tree can be represented by a linked data structure in which each node is an object. In addition to a *key* field, each node contains fields *left*, *right*, and *p* that point to the nodes corresponding to its left child, its right child, and its parent, respectively. If a child or the parent is missing, the appropriate field contains the value NIL. The root node is the only node in the tree whose parent field is NIL. The keys in a binary search tree are always stored in such a way as to satisfy the **binary-search-tree property**.

12.1 Binary-Search-Tree Property

Let x be a node in a binary search tree.

- If y is a node in the left sub tree of x, then key $[y] \le$ key $[x]$.
- If y is a node in the right sub tree of x, then key $[x] \le$ key $[y]$.

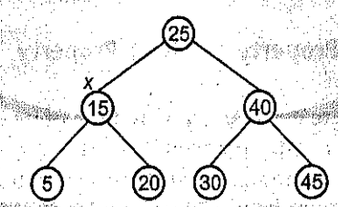


Figure 12.1

In this tree key $[x] = 15$

- If y is a node in the left subtree of x then key $[y] = 5$.

  i.e., key $[y] \le$ key $[x]$

and

- If y is a node in the right subtree of x then key $[y] = 20$.

  i.e., key $[x] \le$ key $[y]$

The binary-search-tree property allows us to print out all the keys in a binary search tree in sorted order by a simple recursive algorithm, called an ***inorder tree walk***. This algorithm derives its name from the fact that the key of the root of a sub tree is printed between the values in its left sub tree and those in its right sub tree. (Similarly, a ***preorder tree walk*** prints the root before the values in either sub tree, and a ***post order tree walk*** prints the root after the values in its sub trees.)

```
INORDER-TREE-WALK (x)
(During this type of walk, we visit the root of a sub tree between the left sub tree visit
and right sub tree visit.)
1.  if x ≠ NIL
2.      then INORDER-TREE-WALK (left[x])
3.           print key[x]
4.           INORDER-TREE-WALK (right[x])
```

It takes $\theta(n)$ time to walk a tree of n nodes. Note that the Binary Search Tree property allows us to print out all the elements in the Binary Search Tree in sorted order.

```
PREORDER-TREE-WALK (x)
(In which we visit the root node before the nodes in either sub tree)
1. if x ≠ NIL
2.     then print key[x]
3.          PREORDER-TREE-WALK (left[x])
4.          PREORDER-TREE-WALK (right[x])
```

```
POSTORDER-TREE-WALK (x)
(In which we visit the root node after the nodes in its sub trees)
1. if x ≠ NIL
2.     then POSTORDER-TREE-WALK (left[x])
3.          POSTORDER-TREE-WALK (right[x])
4.          print key[x]
```

It takes $O(n)$ time to walk (inorder, preorder and postorder) a tree of n nodes

12.2 Binary-Search-Tree Property vs Heap Property

In a heap, a nodes key is greater than equal to both of its children's keys. In binary search tree, a node's key is greater than or equal to its child's key but less than or equal to right child's key. Furthermore, this applies to entire sub tree in the binary search tree case. It is very important to note that the heap property does not help print the nodes in sorted order because this property does not tell us in which sub tree the next item is. If the heap property could be used to print the keys in sorted order in $O(n)$ time, this would contradict our known lower bound on comparison sorting.

The last statement implies that since sorting n elements takes $\Omega(n \lg n)$ time in the worst case in the comparison model, any comparison-based algorithm for constructing a Binary Search Tree from arbitrary list n elements takes $\Omega(n \lg n)$ time in the worst case.

We can show the validity of this argument (in case you are thinking of beating $\Omega(n \lg n)$ bound) as follows : let $c(n)$ be the worst-case running time for constructing a binary tree of a set of n elements. Given an n-node BST, the INORDER walk in the tree outputs the keys in sorted order (shown above). Since the worst-case running time of any computation based sorting algorithm is $\Omega(n \lg n)$, we have

$$c(n) + O(n) = \Omega(n \lg n)$$

Therefore, $$c(n) = \Omega(n \lg n).$$

12.3 Querying a Binary Search Tree

The most common operation performed on a binary search tree is searching for a key stored in the tree. Besides the SEARCH operation, binary search trees can support such queries as MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR. These operations run in $O(h)$ time where h is the height of the tree i.e., h is the number of links root node to the deepest node.

12.3.1 Searching

The TREE-SEARCH (x, k) algorithm searches the tree root at x for a node whose key value equals k. It returns a pointer to the node if it exists otherwise NIL

```
TREE-SEARCH (x, k)
1.  if x = NIL or k = key[x]
2.      then return x
3.  if k < key[x]
4.      then return TREE-SEARCH (left[x], k)
5.      else return TREE-SEARCH (right[x], k)
```

Clearly, this algorithm runs in $O(h)$ time where h is the height of the tree. The iterative version of above algorithm is very easy to implement

```
ITERATIVE-TREE-SEARCH (x, k)
1.  while x ≠ NIL and k ≠ key[x]
2.      do if k < key[x]
3.             then x ← left[x]
4.             else x ← right[x]
5.  return x
```

12.3.2 Minimum and Maximum

An element in a binary search tree whose key is a minimum can always be found by following *left* child pointers from the root until a NIL is encountered. The following procedure returns a pointer to the minimum element in the sub tree rooted at a given node x.

```
TREE-MINIMUM (x)
1.  while left[x] ≠ NIL
2.      do x ← left[x]
3.  return x
```

The pseudocode for TREE-MAXIMUM is symmetric.

```
TREE-MAXIMUM (x)
1.  while right[x] ≠ NIL
2.      do x ← right[x]
3.  return x
```

Both of these procedures run in $O(h)$ time on a tree of height h, since they trace paths downward in the tree.

12.3.3 Successor and predecessor

Given a node in a binary search tree, it is sometimes important to be able to find its successor in the sorted order determined by an inorder tree walk. If all keys are distinct, the successor of a node x is the node with the smallest key greater than $key[x]$. The structure of a binary search tree allows us to determine the successor of a node without ever comparing keys. The following procedure returns the successor of a node x in a binary search tree if it exists, and NIL if x has the largest key in the tree.

```
TREE SUCCESSOR (x)
1.  if right[x] ≠ NIL
2.      then return TREE-MINIMUM(right[x])
3.  y ← p[x]
4.  while y ≠ NIL and x = right[y]
5.      do x ← y
6.      y ← p[y]
7.  return y
```

The code for TREE-SUCCESSOR is broken into two cases. If the right sub tree of node x is nonempty, then the successor of x is just the left-most node in the right sub tree, which is found in line 2 by calling TREE-MINIMUM (*right* [x]). On the other hand, if the right sub tree of node x is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x. To find y, we simply go up the tree from x until we encounter a node that is the left child of its parent; lines 3-7 of TREE-SUCCESSOR accomplish this.

The running time of TREE-SUCCESSOR on a tree of height h is $O(h)$, since we either follow a path up the tree or follow a path down the tree. The procedure TREE-PREDECESSOR, which is symmetric to TREE-SUCCESSOR, also runs in time $O(h)$.

12.3.4 Insertion and deletion

Insertion

To insert a new value v into a binary search tree T, we use the procedure TREE-INSERT. The procedure is passed a node z for which $key[z] = v$, $left[z] = NIL$, and $right[z] = NIL$. It modifies T and some of the fields of z in such a way that z is inserted into an appropriate position in the tree.


```
TREE-INSERT (T, z)
1.  y ← NIL
2.  x ← root[T]
3.  while x ≠ NIL
4.      do y ← x
5.          if key[z] < key[x]
6.              then x ← left[x]
7.              else x ← right[x]
8.  p[z] ← y
9.  if y = NIL
10.     then root[T] ← z
11.     else if key[z] < key[y]
12.         then left[y] ← z
13.         else right[y] ← z
```

Like the procedures TREE-SEARCH and ITERATIVE-TREE-SEARCH, TREE-INSERT begins at the root of the tree and traces a path downward. The pointer x traces the path, and the pointer y is maintained as the parent of x. After initialization, the **while** loop in lines 3-7 causes these two pointers to move down the tree, going left or right depending on the comparison of $key[z]$ with $key[x]$, until x is set to NIL. This NIL occupies the position where we wish to place the input item z. Lines 8-13 set the pointers that cause z to be inserted.

Like the other primitive operations on search trees, the procedure TREE-INSERT runs in $O(h)$ time on a tree of height h.

Sorting

We can sort a given set of n numbers by first building a binary search tree containing these numbers by using TREE-INSERT (x) procedure repeatedly to insert the numbers one by one and then printing the numbers by an inorder tree walk. In best-case running time printing takes $O(n)$ time and n insertion cost $O(\lg n)$ each (tree is balanced, half the insertions are at depth $\lg(n)-1$). This gives the best-case running time $O(n \lg n)$. In worst-case running time printing still takes $O(n)$ time and n insertion costing $O(n)$ each (tree is a single chain of nodes) is $O(n^2)$. The n insertion cost 1, 2, 3, . . . n, which is arithmetic sequence so it is $n^2/2$.

Deletion

When deleting a node from a tree it is important that any relationships, implicit in the tree, be maintained. The deletion of nodes from a binary search tree will be considered. Three distinct cases can be identified:

1. ***Nodes with no children.*** This case is trivial. Simply set the parent's pointer to the node to be deleted to nil and delete the node.

2. ***Nodes with one child.*** This case is also easy to deal with. The single child of the node to be deleted becomes the child of its grandparent through the pointer that currently points to its parent, which is to be deleted. Examining the relationship between the grandparent node and its child, which is to be deleted, can see that this preserves the tree relationships. If the node to be deleted is a

left child then it is less than its parent. Whether or not the child of this node is left or right does not matter since by its place in the tree as the child of a left child it must be less than its grandparent and so can replace its parent in that position.

3. *Nodes with two children.* This case is a little more difficult since two sub trees need to be re-attached and both must maintain the same relation to the parent of the node being deleted and to each other. This can only be achieved if a new node is found to replace the deleted node as the root of these two sub trees. Whatever node is to become parent to these two hanging sub trees it must have the property that it is greater than any node in the left sub tree and less than any node in the right sub tree.

If a binary search tree is examined, it can be seen that there are just two possible nodes which can satisfy these conditions: the rightmost node in the left sub tree and the leftmost node in the right sub tree, hence one of these must replace the node to be deleted.

Inorder to find this node, move to the left and then keep searching right until there is a nil pointer. The node with the nil pointer is the node to be moved. The same process can be done by moving to the right of the node to be deleted and search left until a nil pointer is found.

Having come to this conclusion the best way to achieve this without wholesale reassignment of pointers is to copy the contents of the node doing the replacing into the node to be deleted. This effectively deletes the specified node and replaces it with an appropriate root for its two sub trees. All that remains is to delete the extra copy of the replacing node. This is going to be one of the other two trivial cases and so is easily handled.

The code for TREE-DELETE organizes these three cases a little differently.

```
TREE-DELETE (T, z)
if left[z] = NIL or right[z] = NIL
    then y ← z
    else y ← TREE-SUCCESSOR(z)
if left[y] ≠ NIL
    then x ← left[y]
    else x ← right[y]
if x ≠ NIL
    then p[x] ← p[y]
if p[y] = NIL
    then root[T] ← x
    else if y = left[p[y]]
        then left[p[y]] ← x
        else right[p[y]] ← x
if y ≠ z
    then key[z] ← key[y]
    if y has other fields, copy them, too.
return y
```

The procedure runs in $O(h)$ time on a tree of height h.

Example. *What is the difference between the binary-search-tree property and the heap property? Can the heap property be used to print out the keys of an n-node tree in sorted order in $O(n)$ time ? Explain how or why not ?*

Solution. The definitions clearly differ. If the heap property allowed the elements to be printed in sorted order in time $O(n)$. we would have an $O(n)$ time comparison sorting algorithm since BUILD-HEAP takes $O(n)$ time. This, however, is impossible since we know Ω $(n \lg n)$ is a lower bound for sorting.

Example. *Show that if a node in a binary search tree has two children then its successor has no left child and its predecessor has no right child.*

Solution. Let v is a node with two children. The nodes that immediately precede v must be in the left sub tree and the nodes that immediately follow v must be in the right sub tree. Thus the successor s must be in the right sub tree and s will be the next node from v in an inorder walk. Therefore s cannot have a left child since this child since this would come before s in the inorder walk. Similarly, the predecessor has no right child.

Example. *Show that the inorder walk of a n-node binary search tree implemented with a call to TREE-MINIMUN followed by $n-1$ calls to TREE-SUCCESSOR takes $O(n)$ time.*

Solution. Consider the algorithm at any given node during the algorithm. The algorithm will never go to the left sub tree and going up will therefore cause it to never return to this same node. Hence the algorithm only traverses each edge at most twice and therefore the running time is $O(n)$.

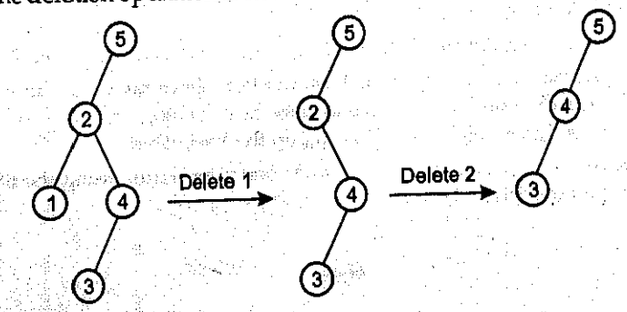
Example. *Give a recursive version of the TREE-INSERT procedure.*

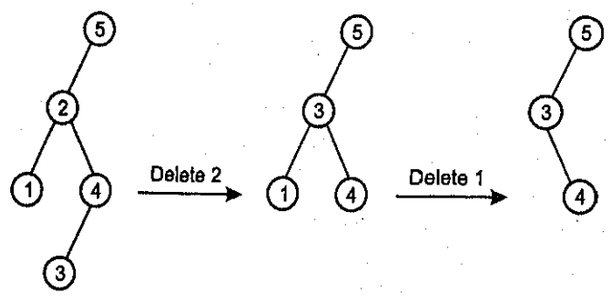
Solution.

```
TREE-INSERT(z, k)
if z = NIL
    then key[z] ← k
    left[z] ← NIL
    right[z] ← NIL
else if k < key[z]
    then TREE-INSERT(left[z], k)
    else TREE-INSERT(right[z], k)
```

Example. *Is the operation of deletion "commutative" in the sense that deleting x and then y from a binary search tree leaves the same tree as deleting y and then x ? Argue why it is or give a counter example.*

Solution. The deletion operation is not commutative. A counterexample is shown in the Fig.





Exercise

1. Suppose that we have numbers between 1 and 1000 in a binary search tree and want to search for the number 363. Which of the following sequences could not be the sequence of nodes examined ?
   (*a*) 2, 252, 401, 398, 330, 344, 397, 363. (*b*) 924, 220, 911, 244, 898, 258, 362, 363.
   (*c*) 925, 202, 911, 240, 912, 245, 363. (*d*) 2, 399, 387, 219, 266, 382, 381, 278, 363.
   (*e*) 935, 278, 347, 621, 299, 392, 358, 363.
2. A random binary search tree having n nodes, where $n = 2^k - 1$ for some positive integer k, is to be reorganized into a perfectly balanced binary search tree. Outline an efficient algorithm for this task. The algorithm should not use any memory locations other than those already used by the random binary search tree.
3. Consider three keys, k_1, k_2, k_3 such that $k_1 < k_2 < k_3$. A binary search tree is constructed with these three keys. Depending on the order in which the keys are inserted, five different binary search trees are possible. Write down the five binary search trees.
4. Given a set of 31 names, each containing 10 uppercase alphabets, we wish to set up a data structure that would lead to efficient average case performance of successful and unsuccessful searches on this set. It is known that not more than 5 names start with the same alphabet. Also, assume that successful and unsuccessful searches are equally likely and that in the event of a successful search, it is equally likely that any of the 31 names was searched for. The two likely choices for the data structure are :
   - A closed hash table with 130 locations where each location can accommodate one name.
   - A full binary search tree of height 4, where each of the 31 nodes contains a name and lexicographic ordering is used to set up the BST.
5. Answer the following questions :
   (*a*) What is the hashing function you would use for the closed hash table ?
   (*b*) Assume that the computation of the hash function above takes the same time as comparing two given names. Now, compute the average case running time of a search operation using the hash table. Ignore the time for setting up the hash table.
   (*c*) Compute the average case running time of a search operation using the BST. Ignore the time for setting up the BST.

CHAPTER 13

AVL Tree

Height Balanced Tree

13.1 Introduction

A height balanced tree is one in which the difference in the height of the two subtrees for any node is less than or equal to some specified amount. One type of height-balanced tree is the **AVL-tree** named after its originators : *Adel'son-Vel'ski* and *Landis*.

In this tree, the height difference may be no more than 1. In fact for an AVL tree it will never be greater than $1.44 \lg (n+2)$. Because of this, the AVL tree provides a method to ensure that tree searches always stay close to the theoretical minimum of $O(\lg n)$ and never degenerate to $O(n)$.

Definition

- *An empty binary tree B is an AVL tree.*
- If B is the non-empty binary tree with B_L and B_R are its left and right subtrees then B is an AVL tree if and only if
  (*i*) B_L and B_R are AVL trees, and
  (*ii*) $|h_L - h_R| \le 1$, where h_L and h_R are the heights of B_L and B_R subtrees, respectively.

Exercise

1. Insert items with the following (in the given order) into an initially empty splay tree and draw the tree after each operation :

   0, 2, 4, 6, 8, 11, 13, 15, 18, 20, 25

2. What does a splay tree look like if keys are accessed in increasing order ?

3. Describe how to implement operation JOIN (S_1, S_2) which returns a splay tree for the UNION of the elements in the splay trees. S_1 and S_2 under the condition that for all keys x in S_1 and all keys y in S_2 we have $x < y$.

   Splay trees S_1 and S_2 are destroyed by this operation.

CHAPTER

15

Red-Black Trees

15.1 Introduction

A **red-black tree** is a type of self-balancing binary search tree.It was invented in 1972 by Rudolf Bayer who called them "**symmetric binary B-trees**".

A red-black tree is a binary tree where each node has color as an extra attribute, either red or black. By constraining the coloring of the nodes it is ensured that the longest path from the root to a leaf is no longer than twice the length of the shortest path. This means that the tree is balanced. A red-black tree must satisfy these *properties* :

1. The root is always black.
2. A nil is considered to be black. This means that every non-NIL node has two children.
3. Black Children Rule : The children of each red node are black.
4. Black Height Rule : For each node v, there exists an integer bh (v) such that each downward path from v to a nil has exactly bh (v) black real (*i.e.* non-nil) nodes. Call this quantity the **black height** of v. We define the black height of an RB tree to be the black height of its root.

A tree T is an **almost red-black tree** (ARB tree) if the root is red, but other conditions above hold.

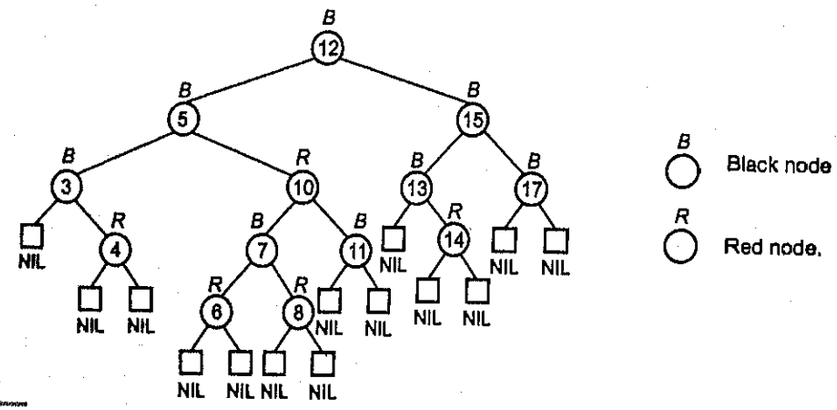


Figure 15.1

Lemma

Let T be an RB tree having some n internal nodes. Then the height of T is at most 2 lg (n + 1).

Proof. We first show that the sub tree rooted at any node x contains at least $2^{bh(x)}-1$ internal nodes. We prove this by induction.

If the height of x is 0 then x must be leaf (NIL), and the sub tree rooted at x contains at least $2^{bh(x)}-1=2^0-1=0$ internal nodes.

For inductive step, consider a node x that has positive height and is an internal node with two children. Each child has a black height of either $bh(x)$ or $bh(x)-1$, depending on whether its color is red or black, respectively. Since the height of a child of x is less than the height of x itself. So, we can apply the inductive hypothesis to conclude that each child has at least $2^{bh(x)-1}-1$ internal nodes.

Thus, the sub tree rooted at x contains at least $(2^{bh(x)-1}-1)+(2^{bh(x)-1}-1)+1=2^{bh(x)}-1$ internal nodes.

Let h be the height of T. Let $v_0; v_1,\ldots,v_{h+1}$ be an arbitrary length h downward path from the root to a nil, where v_0 is the root, v_h is the leaf, and v_{h+1} is the nil. v_0 is black and v_{h+1} is black. The number of red nodes among $v_1,\ldots,v_h$ is maximized when for all odd i v_i is red. So, the number of red nodes is at most $h/2$. This means that the number of black ones among $v_1,\ldots,v_h$ is at least $h-h/2$. Thus, $bh(T)\geq h/2$. If x is the root of the tree then $bh(x)$ must be at least $h/2$, thus

$$n\geq 2^{h/2}-1$$

Moving the 1 to the left hand side and taking logarithms on both sides, we get

$$\lg(n+1)\geq h/2$$

By solving this we have $h\leq 2\lg(n+1)$.

15.2 Operations on RB Trees

The search-tree operations TREE-INSERT and TREE-DELETE, when run on a red-black tree with n keys, take $O(\lg n)$ time. Because they modify the tree, the result may violate the red-black properties. To restore these properties, we must change the colours of some of the nodes in the tree and also change the pointer structure. We will study *two* operations, *insertion* and *deletion*. The two operations make use of two operations, Left-Rotate and Right-Rotate.

15.2.1 Rotations

Restructuring operations on red-black trees (and many other kinds of trees) can often be expressed more clearly in terms of the **rotation** operation, diagrammed below.

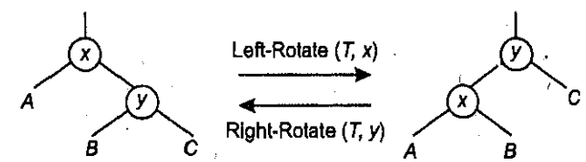
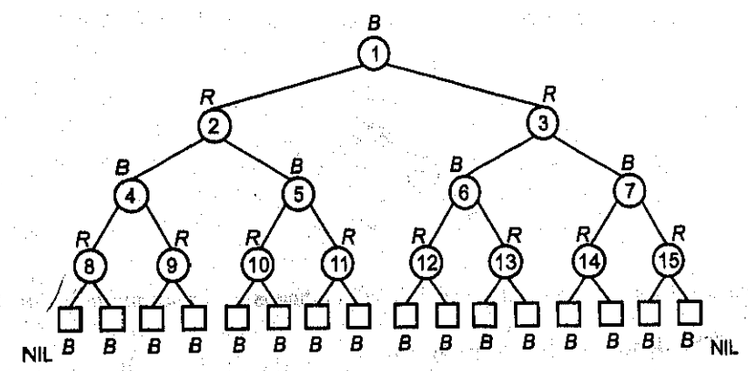


Figure 15.2

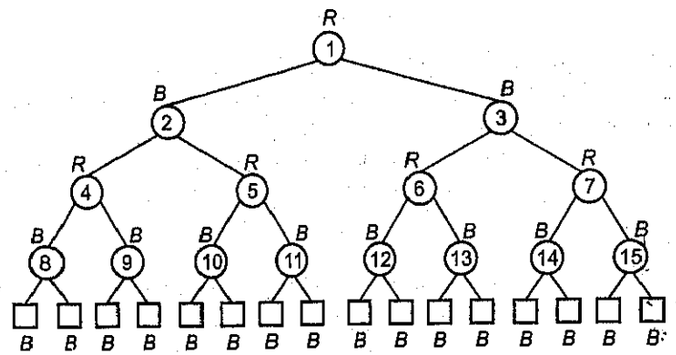
Clearly the order $\langle AxByC\rangle$ is preserved by the rotation operation. Therefore, if we start with a BST, and only restructure using rotations, then we will still have a BST i.e. rotations do not break the BST-property.

Example. *Draw the complete binary tree of height 3 on the keys {1, 2, 3, ..., 15}. Add the NIL leaves and colour the nodes in three different ways such that the black-heights of the resulting trees are : 2, 3 and 4.*

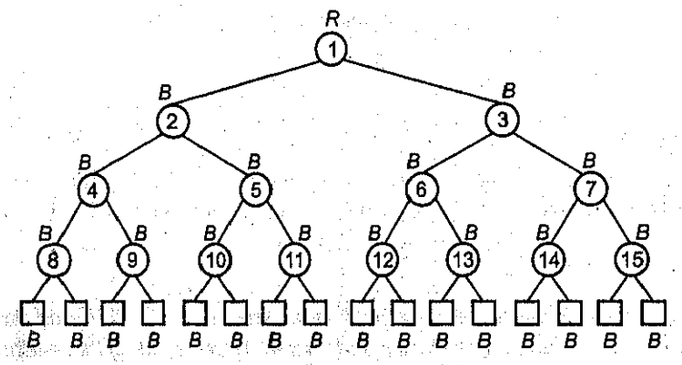
Solution.



Tree with black-height-2

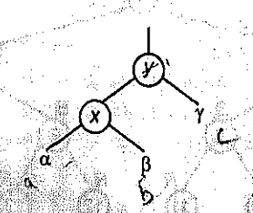


Tree with black-height-3



Tree with black-height-4

Example. *Let a, b, c be arbitrary nodes in subtrees α, β and γ respectively in the tree.*



How do the depths of a, b and c change when a left rotation is performed on node x in the figure?

Solution. The depth of *a* increases by +1.
The depth of *b* remains the same
The depth of *c* changes by −1.

15.2.2 Insertion

- Insert the new node the way it is done in binary search trees
- Color the node red
- If an inconsistency arises for the red-black tree, fix the tree according to the type of discrepancy.

A discrepancy can result from a parent and a child both having a red color. This type of discrepancy is determined by the location of the node with respect to its grand parent, and the color of the sibling of the parent.

Discrepancies, in which the sibling is red, are fixed by changes in color. Discrepancies, in which the siblings are black, are fixed through AVL-like rotations. Changes in color may propagate the problem up toward the root. On the other hand, at most one rotation is sufficient for fixing a discrepancy.

```
RB-INSERT (T, z)
y ← nil[T]
x ← root[T]
while x ≠ nil[T]
    do y ← x
        if key[z] < key[x]
        then x ← left[x]
        else x ← right[x]
p[z] ← y
if y = nil[T]
    then root[T] ← z
    else if key[z] < key[y]
        then left[y] ← z
        else right[y] ← z
left[z] ← nil[T]
right[z] ← nil[T]
color[z] ← RED
RB-INSERT-FIXUP (T, z)
```

After the insert new node, now the question arises which color do we give the new node. Coloring this new node into black may violate the black-height condition and coloring this new node into red may violate coloring condition *i.e.*, root is black and red node has no red children.

We know the black-height violations are hard. So we color the node red. After this if there is any color violation then we have to correct them by RB-INSERT-FIXUP procedure.

```
RB-INSERT-FIXUP(T, z)
while color[p[z]] = RED
    do if p[z] = left[p[p[z]]]
        then y ← right[p[p[z]]]
            if color[y] = RED
```

```
5.          then color[p[z]] ← BLACK                ▷ Case 1
6.               color[y] ← BLACK                   ▷ Case 1
7.               color[p[p[z]]] ← RED               ▷ Case 1
8.               z ← p[p[z]]                        ▷ Case 1
9.          else if z = right[p[z]]
10.              then z ← p[z]                      ▷ Case 2
11.                   LEFT-ROTATE (T, z)            ▷ Case 2
12.              color[p[z]] ← BLACK                ▷ Case 3
13.              color[p[p[z]]] ← RED               ▷ Case 3
14.              RIGHT-ROTATE (T, p[p[z]])          ▷ Case 3
15.     else (same as then clause
             with "right" and "left" exhanged)
16. color[root[T]] ← BLACK
```

There are actually six cases to consider in the **while** loop, but three of them are symmetric to the other three, depending on whether x's parent $p[x]$ is a left child or a right child of x's grandparent $p[p[x]]$, which is determined in line 4. We have given the code only for the situation in which $p[x]$ is a left child. We have made the important assumption that the root of the tree is black—a property we guarantee in line 18 each time we terminate—so that $p[x]$ is not the root and $p[p[x]]$ exists.

Case 1 is distinguished from cases 2 and 3 by the color of x's parent's sibling, or "uncle." Line 5 makes y point to x's uncle *right* $[p[p[x]]]$, and a test is made in line 6. If y is red, then case 1 is executed. Otherwise, control passes to cases 2 and 3. In all three cases, x's grandparent $p[p[x]]$ is black, since its parent $p[x]$ is red, and property 3 is violated only between x and $p[x]$.

Case 1 is executed when both $p[x]$ and y are red. Since $p[p[x]]$ is black, we can color both $p[x]$ and y black, thereby fixing the problem of x and $p[x]$ both being red, and color $p[p[x]]$ red, thereby maintaining property 4. The only problem that might arise is that $p[p[x]]$ might have a red parent; hence, we must repeat the **while** loop with $p[p[x]]$ as the new node x.

In cases 2 and 3, the color of x's uncle y is black. The two cases are distinguished by whether x is a right or left child of $p[x]$. In case 2, node x is a right child of its parent. We immediately use a left rotation to transform the situation into case 3, in which node x is a left child. Because both x and $p[x]$ are red, the rotation affects neither the black-height of nodes nor property 4. Whether we enter case 3 directly or through case 2, x's uncle y is black, since otherwise we would have executed case 1. We execute some color changes and a right rotation, which preserve property 4, and then, since we no longer have two red nodes in a row, we are done. The body of the **while** loop is not executed another time, since $p[x]$ is now black.

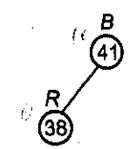
Since the height of a red-black tree on n nodes is $O(\lg n)$, the call to TREE-INSERT takes $O(\lg n)$ time. The **while** loop only repeats if case 1 is executed, and then the pointer x moves up the tree. The total number of times the **while** loop can be executed is therefore $O(\lg n)$. Thus, RB-INSERT takes a total of $O(\lg n)$ time. Interestingly, it never performs more than two rotations, since the **while** loop terminates if case 2 or case 3 is executed.

Example. *Show the red-black trees that result after successively inserting the keys 41, 38, 31, 12, 19, 8 into an initially empty red-black tree.*

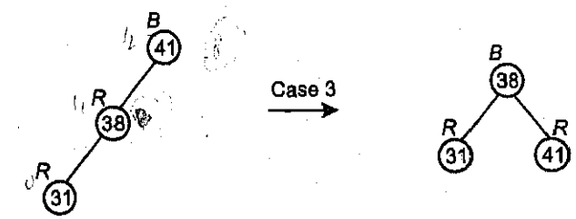
Solution.

◂ Insert 41

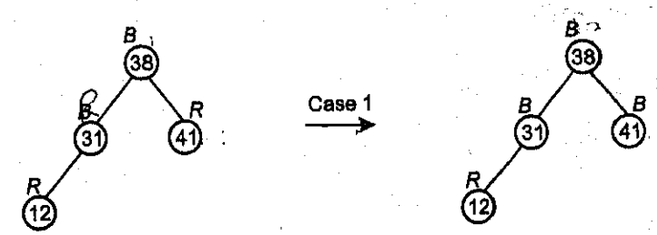
◂ Insert 38



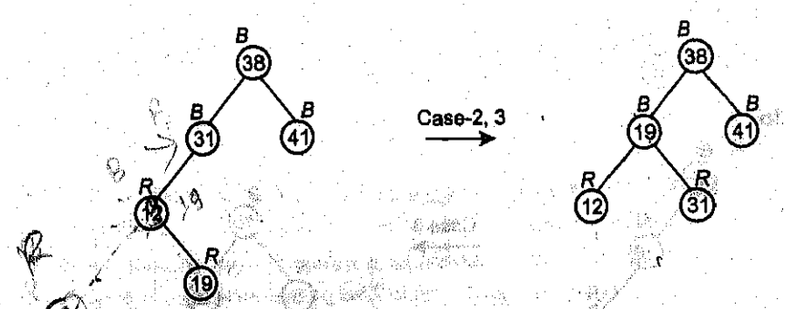
◂ Insert 31



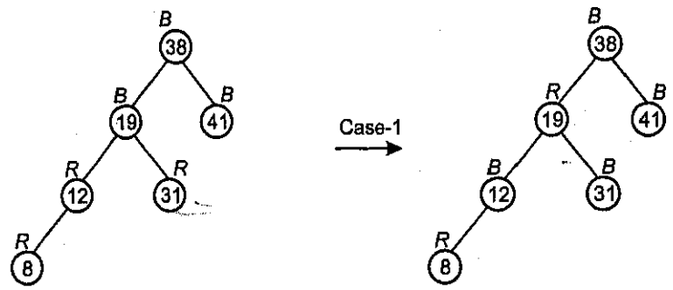
◂ Insert 12



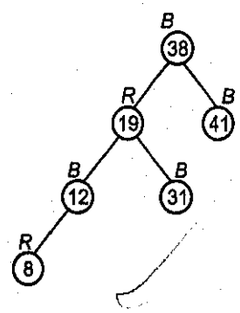
◂ Insert 19



◄ Insert 8



Thus final tree is

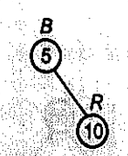


Example. *Show the red-black trees that result after successively inserting the keys 5, 10, 15, 25, 20 and 30 into an initially empty red-black tree.*

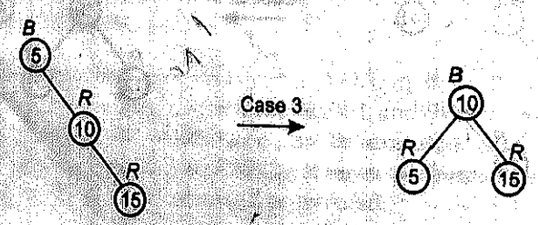
Solution.

◄ Insert 5

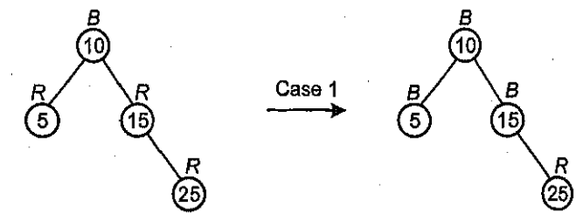
◄ Insert 10



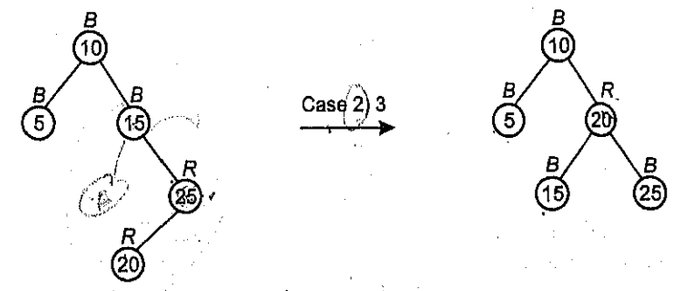
◄ Insert 15



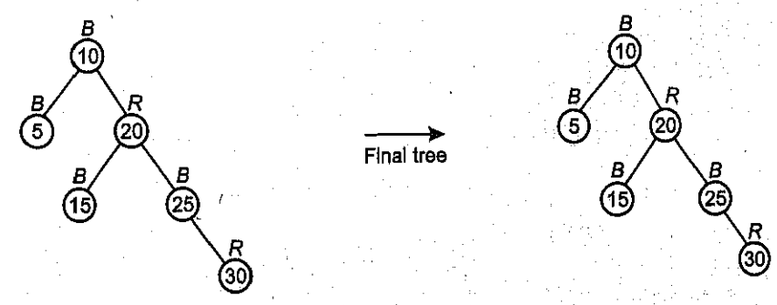
◄ Insert 25



◄ Insert 20



◄ Insert 30



15.2.3 Deletion

First, search for an element to be deleted :

- ◄ If the element to be deleted is in a node with only left child, swap this node with the one containing the largest element in the left sub tree. (This node has no right child).
- ◄ If the element to be deleted is in a node with only right child, swap this node with the one containing the smallest element in the right sub tree (This node has no left child).
- ◄ If the element to be deleted is in a node with both a left child and a right child, then swap in any of the above two ways. While swapping, swap only the keys but not the colors.

- The item to be deleted is now in a node having only a left child or only a right child. Replace this node with its sole child. This may violate red constraint or black constraint. Violation of red constraint can be easily fixed.
- If the deleted node is black, the black constraint is violated. The removal of a black node y causes any path that contained y to have one fewer black node.
- *Two* cases arise :
  1. The replacing node is red, in which case we merely color it black to make up for the loss of one black node.
  2. The replacing node is black.

The procedure RB-DELETE is a minor modification of the TREE-DELETE procedure. After splicing out a node, it calls an auxiliary procedure RB-DELETE-FIXUP that changes colors and performs rotations to restore the red-black properties.

RB-DELETE (T, z)

```
1.  if left[z] = nil[T] or right[z] = nil[T]
2.      then y ← z
3.      else y ← TREE-SUCCESSOR(z)
4.  if left[y] ≠ nil[T]
5.      then x ← left[y]
6.      else x ← right[y]
7.  p[x] ← p[y]
8.  if p[y] = nil[T]
9.      then root[T] ← x
10.     else if y = left[p[y]]
11.         then left[p[y]] ← x
12.         else right[p[y]] ← x
13. if y ≠ z
14.     then key[z] ← key[y]
15.         copy y's satellite data into z
16. if color[y] = BLACK
17.     then RB-DELETE-FIXUP (T,x)
18. return y
```

There are three differences between the procedures TREE-DELETE and RB-DELETE. First, all references to NIL in TREE-DELETE have been replaced by references to the sentinel $nil[T]$ in RB-DELETE. Second, the test for whether x is NIL in line 7 of TREE-DELETE has been removed, and the assignment $p[x] \leftarrow p[y]$ is performed unconditionally in line 7 of RB-DELETE. Thus, if x is the sentinel $nil[T]$, its parent pointer points to the parent of the spliced-out node y. Third, a call to RB-DELETE-FIXUP is made in lines 16-17 if y is black. If y is red, the red-black properties still hold when y is spliced out, since no black-heights in the tree have changed and no red nodes have been made adjacent. The node x passed to RB-DELETE-FIXUP is the node that was y's sole child before y was spliced out if y had a non-NIL child, or the sentinel $nil[T]$ if y had no children. In the latter case,

the unconditional assignment in line 7 guarantees that x's parent is now the node that was previously y's parent, whether x is a key-bearing internal node or the sentinel $nil[T]$.

We can now look at how the procedure RB-DELETE-FIXUP restores the red-black properties to the search tree.

RB-DELETE-FIXUP (T, x)

```
1.  while x ≠ root[T] and color[x] = BLACK
2.      do if x = left[p[x]]
3.          then w ← right[p[x]]
4.              if color[w] = RED
5.                  then color[w] ← BLACK                          ▷ Case 1
6.                       color[p[x]] ← RED                         ▷ Case 1
7.                       LEFT-ROTATE (T, p[x])                     ▷ Case 1
8.                       w ← right[p[x]]                           ▷ Case 1
9.              if color[left[w]] = BLACK and color[right[w]] = BLACK
10.                 then color[w] ← RED                            ▷ Case 2
11.                      x ← p[x]                                  ▷ Case 2
12.                 else if color[right[w]] = BLACK
13.                      then color[left[w]] ← BLACK               ▷ Case 3
14.                           color[w] ← RED                       ▷ Case 3
15.                           RIGHT-ROTATE (T, w)                  ▷ Case 3
16.                           w ← right[p[x]]                      ▷ Case 3
17.                      color[w] ← color[p[x]]                    ▷ Case 4
18.                      color[p[x]] ← BLACK                       ▷ Case 4
19.                      color[right[w]] ← BLACK                   ▷ Case 4
20.                      LEFT-ROTATE (T, P[x])                     ▷ Case 4
21.                      x ← root[T]                               ▷ Case 4
22.         else (same as then clause with "right" and "left" exchanged)
23. color[x] ← BLACK
```

Since the height of a red-black tree of n nodes is $O(\lg n)$, the total cost of the procedure without the call to RB-DELETE-FIXUP takes $O(\lg n)$ time. Within RB-DELETE-FIXUP, cases 1, 3, and 4 each terminate after performing a constant number of color changes and at most three rotations. Case 2 is the only case in which the while loop can be repeated, and then the pointer x moves up the tree at most $O(\lg n)$ times and no rotations are performed. Thus, the procedure RB-DELETE-FIXUP takes $O(\lg n)$ time and performs at most three rotations, and the overall time for RB-DELETE is therefore also $O(\lg n)$.

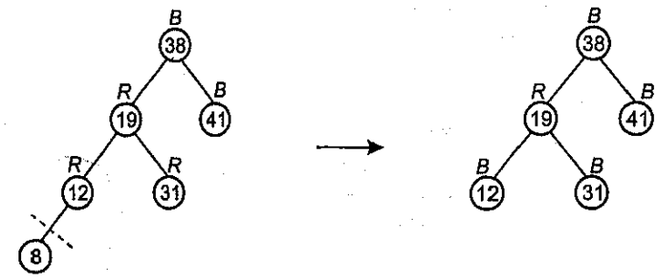
15.3 Elementary Properties of Red-Black Tree

1. Black-height of an red-black tree is the black height of its root. This equals the number of black edges to reach a leaf.
2. If Red-black tree has black-height 'bh' :
   (a) then it has atleast $2^{bh} - 1$ internal black nodes.
   (b) it has atleast $4^{bh} - 1$ internal nodes.

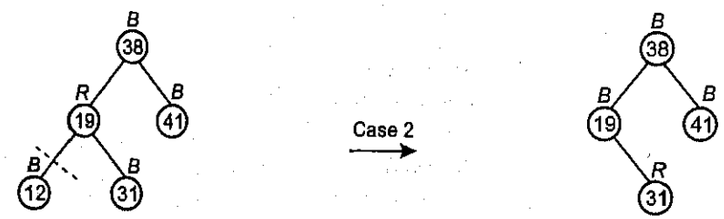
Example. *In previous exercise, we found the red-black tree that results from successively inserting the keys 41, 38, 31, 12, 19, 8 into an initially empty tree. Now show the red-black trees that result from the successive deletion of the keys in the order 8, 12, 19, 31, 38, 41.*

Solution.

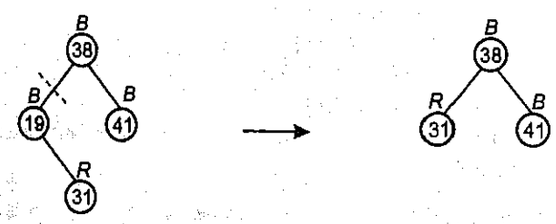
◂ Delete 8



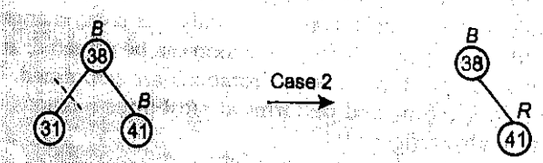
◂ Delete 12



◂ Delete 19



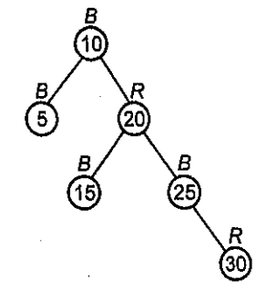
◂ Delete 31



◂ Delete 38

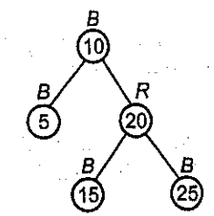
◂ Delete 41 No Tree.

Example. *Show the red-black tree that results from successively deleting the keys 30, 25, 20, 15, 10 and 5 from the final tree.*

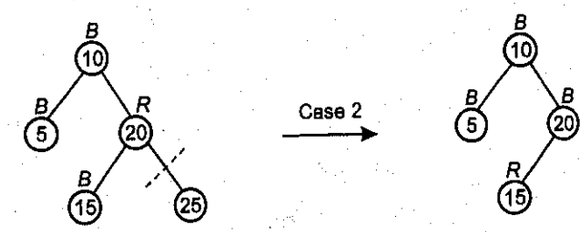


Solution.

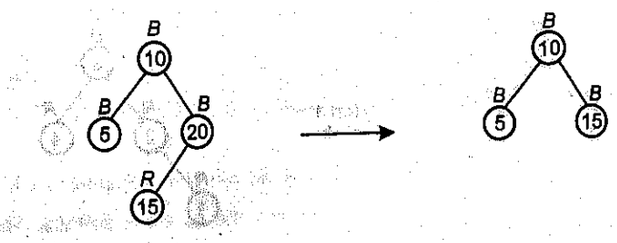
◂ Delete 30



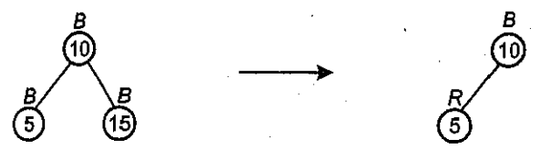
◂ Delete 25



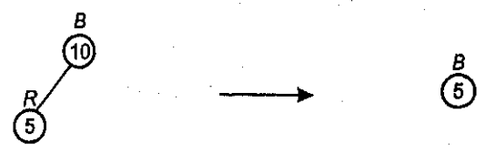
◂ Delete 20



◂| Delete 15



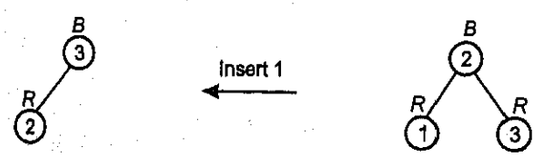
◂| Delete 10



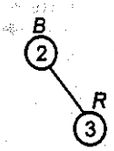
◂| Delete 5. No tree.

Example. *Suppose that a node x is inserted into a red-black tree with RB-INSERT and then immediately deleted with RB-DELETE. Is the resulting red-black tree the same as the initial red-black tree? Justify your anwer?*

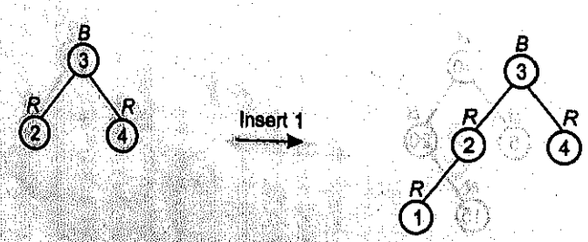
Solution. Inserting and immediately deleting need not yield the same tree. Here is an example that alters the structure and one that changes the color.



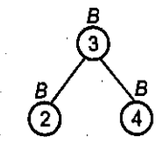
Now delete 1, we get



Again,



Now, delete 1, we get



Example. *Suppose that the root of a red-black tree is red. If we make it black, does the tree remain a red black tree?*

Solution. Coloring the root node black can obviously not violate any property.

Example. *Show that the longest simple path from a node x in a red black tree to a descendant leaf has length at most twice that of the shortest simple path from node x to a descendant leaf?*

Solution. By property 4 the longest and shortest path must contain the same number of black nodes. By property 3 every other nodes in the longest path must be black and therefore the length is at most twice that of the shortest path.

Example. *What is the largest possible number of internal nodes in a red black tree with black height k? What is the smallest possible number?*

Solution. Consider a red-black tree with black-height k. If every node is black, the total number of internal nodes is $2k-1$. If only every other node is black, we can construct a tree with $22k-1$ node.

Example. *In line 2 of RB-INSERT, we set the color of the newly inserted node x to red. Notice that if we had chosen to set x's color to black, then property 3 of a red-black tree would not be violated. Why didn't we choose to set x's color to black*

Solution. If we choose to set the color of a newly inserted node to black then property 3 is not violated but clearly property 4 is violated.

Exercise

1. Show how an AVL tree can be colored as a red-black tree.
2. Draw the red-black tree resulting from inserting the numbers 5, 16, 22, 45, 2, 10, 18, 30, 50, 12, 1 in this order.
3. A subtree of a red-black tree is red-black. Right or wrong, if wrong, provide counter example.
4. Draw a red-black that is not an AVL-tre.

5. Explain how to use a red-black tree to sort n comparable entries in $O(n \lg n)$ time in the worst case.
6. If the common black depth of a red-black tree is B what is the minimum number of black nodes.
7. Show the result of inserting 2, 1, 4, 5, 9, 3, 6, 7 in this order into an empty RB-tree. Do the same for a AVL tree.
8. You are adding an entry to a RB-tree. You are at a node whose children are both red, and the right child of the left child is also red. What steps do you take ? What information do you pass to the next level up ?
9. You are removing an entry from a RB-tree. All paths through the left child of the current node are short by one black node. The right child is red. What steps do you take ? Will you need to do anything at the next level up ?
10. After an addition in a red-black tree, what is the maximum number of rotations required to rebalance the tree ?
11. After a remove in RB tree, what is the maximum number of rotations required to rebalance the tree ?
12. What is the maximum height of a RB tree with 1,000,000 values ?

CHAPTER

16

Augmenting Data Structures

By augmenting already existing data structures one can build new data structures. Augmenting a data structure is not always straightforward, however, since the added information must be updated and maintained by the ordinary operations on the data structure.

16.1 Augmenting a red-black tree

For each node x, add a new field size(x), the number of non-NIL nodes in the sub tree rooted at x. Now with the size information, we can fast compute the dynamic order statistics and the rank, the position in the linear order.

An ***order-statistic tree*** T is simply a red-black tree with additional information stored in each node. Besides the usual red-black tree fields *key*[x], *color*[x], p[x], *left*[x], and *right*[x] in a node x, we have another field *size*[x]. This field contains the number of (internal) nodes in the subtree rooted at x (including x itself), that is, the size of the subtree.

If we define *size*[NIL] to be 0, then we have the identity

$$size[x] = size[left[x]] + size[right[x]] + 1.$$

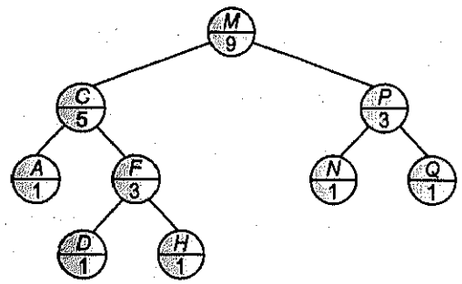


Figure 16.1

16.2 Retrieving an element with a given rank

The procedure OS-SELECT(x, i) returns a pointer to the node containing the ith smallest key in the sub tree rooted at x. To find the ith smallest key in an order-statistic tree T, we call OS-SELECT(*root*[T], i).

```
OS-SELECT(x,i)

1.  r ← size[left[x]] +1
2.  if i = r
3.      then return x
4.  elseif i < r
5.      then return OS-SELECT(left[x], i)
6.  else return OS-SELECT(right[x], i − r)
```

The value of *size*[*left*[x]] is the number of nodes that come before x in an inorder tree walk of the subtree rooted at x. Thus, *size*[*left*[x]]+1 is the rank of x within the subtree rooted at x.

In line 1 of OS-SELECT, we compute r, the rank of node x within the subtree rooted at x. If $i = r$, then node x is the ith smallest element, so we return x in line 3. If $i < r$, then the ith smallest element is in x's left sub tree, so we recurse on *left*[x] in line 5. If $i > r$, then the ith smallest element is in x's right subtree. Since there are r elements in the subtree rooted at x that come before x's right subtree in an inorder tree walk, the ith smallest element in the subtree rooted at x is the $(i-r)$th smallest element in the subtree rooted at *right*[x]. This element is determined recursively in line 6. Running time O(height)

To see how OS-SELECT operates, consider an example

OS-SELECT (root, 5)

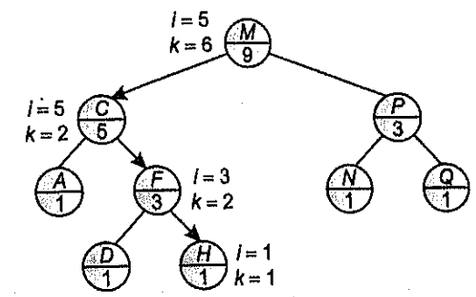


Figure 16.2

16.3 Determining the rank of an element

Given a pointer to a node x in an order-statistic tree T, the procedure OS-RANK returns the position of x in the linear order determined by an inorder tree walk of T.

```
OS-RANK (T, x)

1.  r ← size[left[x]] +1
2.  y ← x
3.  while y ≠ root[T]
4.      do if y = right[p[y]]
5.             then r ← r + size[left[p[y]]] +1
6.             y ← p[y]
7.  return r
```

Since each iteration of the **while** loop takes $O(1)$ time, and y goes up one level in the tree with each iteration, the running time of OS-RANK is at worst proportional to the height of the tree: $O(\lg n)$ on an n-node order-statistic tree.

16.4 Data Structure Maintenance

Example. *Why not keep the ranks themselves in the nodes instead of sub tree sizes ?*

Solution. They are hard to maintain when the tree is modified.

Given the *size* field in each node, OS-SELECT and OS-RANK can quickly compute order-statistic information. But unless these fields can be efficiently maintained by the basic modifying operations on red-black trees, We shall now show that sub tree sizes can be maintained for both **insertion** and **deletion** without affecting the asymptotic running times of either operation.

We know that insertion into a red-black tree consists of *two* phases. The first phase goes down the tree from the root, inserting the new node as a child of an existing node. The second phase goes up the tree, changing colors and ultimately performing rotations to maintain the red-black properties.

To maintain the subtree sizes in the first phase, we simply increment *size*[x] for each node x on the path traversed from the root down toward the leaves. The new node added gets a *size* of 1. Since there are $O(\lg n)$ nodes on the traversed path, the additional cost of maintaining the *size* fields is $O(\lg n)$.

In the second phase, the only structural changes to the underlying red-black tree are caused by rotations, of which there are at most two. Moreover, a rotation is a local operation: it invalidates only the two *size* fields in the nodes incident on the link around which the rotation is performed. Referring to the code for LEFT-ROTATE (T,x) in, we add the following lines :

```
13 size[y] ← size[x]
14 size[x] ← size[left[x]] + size[right[x]] + 1
```

Thus, the size has to be changed for only one node :

- the left-child of the rotated node in the case of right rotation and
- the right-child of the rotated node in the case of left rotation.

Since at most two rotations are performed during insertion into a red-black tree, only $O(1)$ additional time is spent updating *size* fields in the second phase. Thus, the total time for insertion into an n-node order-statistic tree is $O(\lg n)$-asymptotically the same as for an ordinary red-black tree.

Deletion from a red-black tree also consists of two phases: the first operates on the underlying search tree, and the second causes at most three rotations and otherwise performs no structural changes. The first phase splices out one node y. To update the sub tree sizes, we simply traverse a path from node y up to the root, decrementing the *size* field of each node on the path. Since this path has length $O(\lg n)$ in an n-node red-black tree, the additional time spent maintaining *size* fields in the first phase is $O(\lg n)$. The $O(1)$ rotations in the second phase of deletion can be handled in the same manner as for insertion.

Thus, both insertion and deletion, including the maintenance of the *size* fields, take $O(\lg n)$ time for an n-node order-statistic tree.

16.5 An Augmentation Strategy

16.5.1 How to augment a data structure

Augmenting a data structure can be broken into *four* steps :

1. choosing an **underlying data structure**,
2. determining **what kind of additional information** should be maintained in the underlying data structure,
3. **verify that the additional information can be maintained** during the execution of each basic modifying operation of the underlying data structure, and
4. developing **new operations**.

We followed these steps in to design our order-statistic trees. For step 1, we chose red-black trees as the underlying data structure. A clue to the suitability of red-black trees comes from their efficient support of other dynamic-set operations on a total order, such as MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR.

For step 2, we provided the *size* fields, which in each node x stores the size of the subtree rooted at x. Generally, the additional information makes operations more efficient. For example, we could have implemented OS-SELECT and OS-RANK using just the keys stored in the tree, but they would not have run in $O(\lg n)$ time.

For step 3, we ensured that insertion and deletion could maintain the *size* fields while still running in $O(\lg n)$ time. Ideally, a small number of changes to the data structure should suffice to maintain the additional information. For example, if we simply stored in each node its rank in the tree, the OS-SELECT and OS-RANK procedures would run quickly, but inserting a new minimum element would cause a change to this information in every node of the tree. When we store subtree sizes instead, inserting a new element causes information to change in only $O(\lg n)$ nodes.

For step 4, we developed the operations OS-SELECT and OS-RANK. After all, the need for new operations is why we bother to augment a data structure in the first place. Occasionally, rather than developing new operations, we use the additional information to expedite existing ones.

16.6 Interval trees

For an interval $i=[l,t]$, call l the low end and t the **high end** of i.

The **trichotomy** of intervals

For every pair of intervals i and j, exactly one of the following conditions holds :

1. i and j overlap
2. high[i] < low[j], *i.e.*, j is to the right of i
3. high[j] < low[i], *i.e.*, j is to the left of i

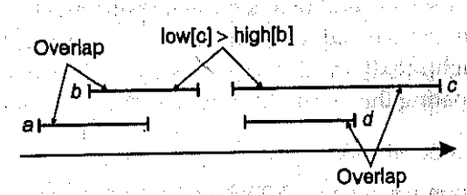


Figure 16.3

An ***interval tree*** is a red-black tree that maintains a dynamic set of elements, with each element x containing an interval *int*[x]. Interval trees support the following operations.

- **INTERVAL-INSERT**(T, x) adds the element x, whose *int* field is assumed to contain an interval, to the interval tree T.
- **INTERVAL-DELETE**(T, x) removes the element x from the interval tree T.
- **INTERVAL-SEARCH**(T, i) returns a pointer to an element x in the interval tree T such that *int*[x] overlaps interval i, or NIL if no such element is in the set.

We know that insertion into a red-black tree consists of *two* phases. The first phase goes down the tree from the root, inserting the new node as a child of an existing node. The second phase goes up the tree, changing colors and ultimately performing rotations to maintain the red-black properties.

To maintain the subtree sizes in the first phase, we simply increment *size*[x] for each node x on the path traversed from the root down toward the leaves. The new node added gets a *size* of 1. Since there are $O(\lg n)$ nodes on the traversed path, the additional cost of maintaining the *size* fields is $O(\lg n)$.

In the second phase, the only structural changes to the underlying red-black tree are caused by rotations, of which there are at most two. Moreover, a rotation is a local operation: it invalidates only the two *size* fields in the nodes incident on the link around which the rotation is performed. Referring to the code for LEFT-ROTATE (T, x) in, we add the following lines :

```
13 size[y] ← size[x]
14 size[x] ← size[left[x]] + size[right[x]] +1
```

Thus, the size has to be changed for only one node :

- the left-child of the rotated node in the case of right rotation and
- the right-child of the rotated node in the case of left rotation.

Since at most two rotations are performed during insertion into a red-black tree, only $O(1)$ additional time is spent updating *size* fields in the second phase. Thus, the total time for insertion into an n-node order-statistic tree is $O(\lg n)$-asymptotically the same as for an ordinary red-black tree.

Deletion from a red-black tree also consists of two phases: the first operates on the underlying search tree, and the second causes at most three rotations and otherwise performs no structural changes. The first phase splices out one node y. To update the sub tree sizes, we simply traverse a path from node y up to the root, decrementing the *size* field of each node on the path. Since this path has length $O(\lg n)$ in an n-node red-black tree, the additional time spent maintaining *size* fields in the first phase is $O(\lg n)$. The $O(1)$ rotations in the second phase of deletion can be handled in the same manner as for insertion.

Thus, both insertion and deletion, including the maintenance of the *size* fields, take $O(\lg n)$ time for an n-node order-statistic tree.

16.5 An Augmentation Strategy

16.5.1 How to augment a data structure

Augmenting a data structure can be broken into *four* steps :

1. choosing an **underlying data structure**,
2. determining **what kind of additional information** should be maintained in the underlying data structure,
3. **verify that the additional information can be maintained** during the execution of each basic modifying operation of the underlying data structure, and
4. developing **new operations**.

We followed these steps in to design our order-statistic trees. For step 1, we chose red-black trees as the underlying data structure. A clue to the suitability of red-black trees comes from their efficient support of other dynamic-set operations on a total order, such as MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR.

For step 2, we provided the *size* fields, which in each node x stores the size of the subtree rooted at x. Generally, the additional information makes operations more efficient. For example, we could have implemented OS-SELECT and OS-RANK using just the keys stored in the tree, but they would not have run in $O(\lg n)$ time.

For step 3, we ensured that insertion and deletion could maintain the *size* fields while still running in $O(\lg n)$ time. Ideally, a small number of changes to the data structure should suffice to maintain the additional information. For example, if we simply stored in each node its rank in the tree, the OS-SELECT and OS-RANK procedures would run quickly, but inserting a new minimum element would cause a change to this information in every node of the tree. When we store subtree sizes instead, inserting a new element causes information to change in only $O(\lg n)$ nodes.

For step 4, we developed the operations OS-SELECT and OS-RANK. After all, the need for new operations is why we bother to augment a data structure in the first place. Occasionally, rather than developing new operations, we use the additional information to expedite existing ones.

16.6 Interval trees

For an interval $i = [l, t]$, call l the **low end** and t the **high end** of i.

The **trichotomy** of intervals

For every pair of intervals i and j, exactly one of the following conditions holds :

1. i and j overlap
2. high[i] < low[j], *i.e.*, j is to the right of i
3. high[j] < low[i], *i.e.*, j is to the left of i

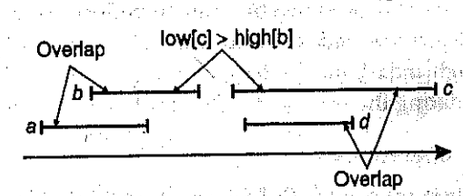


Figure 16.3

An ***interval tree*** is a red-black tree that maintains a dynamic set of elements, with each element x containing an interval *int*[x]. Interval trees support the following operations.

- **INTERVAL-INSERT**(T, x) adds the element x, whose *int* field is assumed to contain an interval, to the interval tree T.
- **INTERVAL-DELETE**(T, x) removes the element x from the interval tree T.
- **INTERVAL-SEARCH**(T, i) returns a pointer to an element x in the interval tree T such that *int*[x] overlaps interval i, or NIL if no such element is in the set.

Step 1 Underlying data structure

We choose a red-black tree in which each node x contains an interval $int[x]$ and the key of x is the low endpoint, $low[int[x]]$, of the interval. Thus, an inorder tree walk of the data structure lists the intervals in sorted order by low endpoint.

Step 2 Additional information

In addition to the intervals themselves, each node x contains a value $m = \max[x]$, which is the maximum value of any interval endpoint stored in the sub tree rooted at x. Since any interval's high endpoint is at least as large as its low endpoint, $max[x]$ is the maximum value of all right endpoints in the sub tree rooted at x.

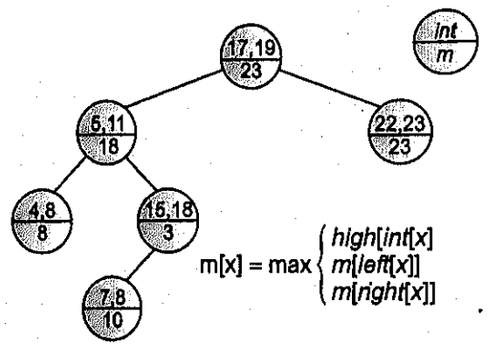


Figure 16.4

Step 3 Maintaining the information

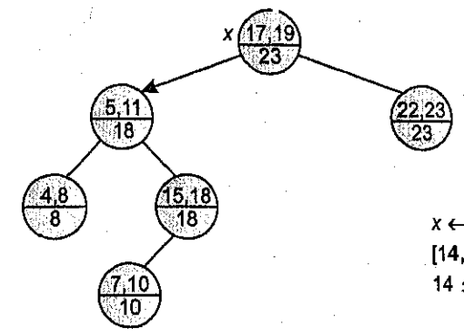
We must verify that insertion and deletion can be performed in $O(\lg n)$ time on an interval tree of n nodes. We can determine $max[x]$ given interval $int[x]$ and the max values of node x's children : $\max[x] = \max(\text{high}[\text{int}[x]], \max[\text{left}[x]], \max[\text{right}[x]])$. Thus, insertion and deletion run in $O(\lg n)$ time. In fact, updating the max fields after a rotation can be accomplished in $O(1)$ time

Step 4 Developing new operations

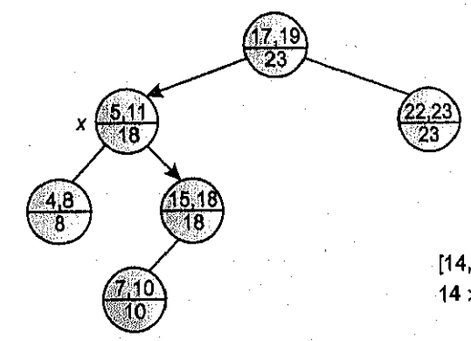
The only new operation we need is INTERVAL-SEARCH (T, i), which finds an interval in tree T that overlaps interval i. If there is no interval that overlaps i in the tree, NIL is returned.

```
INTERVAL-SEARCH (i)
x ← root[T]
while x ≠ NIL and i does not overlap int[x]
    do if left[x] ≠ NIL and max[left[x]] ≥ low[i]
       then x ← left[x]
       else x ← right[x]
return x
```

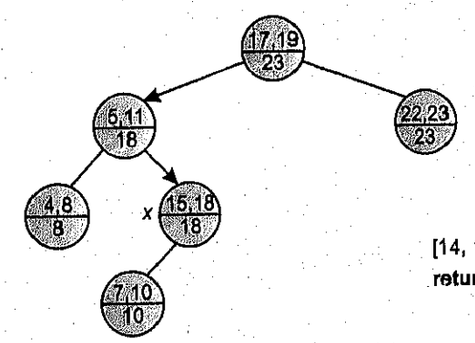

Example: INTERVAL-SEARCH ([14, 16])



$x \leftarrow root$
[14, 16] and [17, 19] don't overlap
$14 \le 18 \Rightarrow x \leftarrow left[x]$



[14, 16] and [5, 11] don't overlap
$14 > 18 \Rightarrow x \leftarrow right[x]$



[14, 16] and [15, 18] overlap
return [15, 18]

Exercise

1. Augment the following data structures.

   (a) **Link list**, where now each node maintains the number of preceding nodes.

(*b*) **Binary tree**, where now each node maintains, its own level.

(*c*) **Directed graph**, where now each node maintains its degree (in-degree and out-degree).

(*d*) **Undirected graph**, where now each node maintains its degree.

2. Show how to use an ordered-statistic tree to count the number of inversions in an array of size n in time $O(n \lg n)$.

3. Write a non-recursive version of OS-SELECT.

4. Write pseudo code for LEFT-ROTATE and RIGHT-ROTATE that operations on node in an interval tree.

5. Show how the depths of nodes in a red-black tree be efficiently maintained as fields in the nodes of the tree ?

6. Suppose each element 'e' in a red-black tree has some positive value $v[e]$ stored in it. Can we augment the nodes in the red-black tree so that each node maintains the product of values in its subtree without affecting the asymptotic running times of any standard tree operations ?

7. What are the basic steps in augmenting ? Why is augmenting of a data structure done ? Show how the dynamic set queries MINIMUM, MAXIMUM, SUCCESSOR. and PREDECESSOR can each be supported in $O(1)$ worst case time on an augmented order statistic tree.

CHAPTER

17

B-trees

17.1 Introduction

B-tree is a tree data structure that keeps data sorted and allows insertions and deletions in logarithmic amortized time. *B*-trees are balanced search trees designed to work well on magnetic disks or other direct-access secondary storage devices. It is most commonly used in databases and file systems. The B-tree's creators, **Rudolf Bayer** and **Ed McCreight**, have not explained what, if anything, the *B* stands for. The most common belief is that *B* stands for ***balanced***, as all the leaf nodes are at he same level in the tree.

In *B*-trees, internal nodes can have a variable number of child nodes within some pre-defined range. When data is inserted or removed from a node, its number of child nodes changes. In order to maintain the pre-defined range, internal nodes may be joined or split. Because a range of child nodes is permitted, *B*-trees do not need re-balancing as frequently as other self-balancing search trees, but may waste some space, since nodes are not entirely full. The lower and upper bounds on the number of child nodes are typically fixed for a particular implementation. For example, in a 2-3 *B*-tree (often simply 2-3 tree), each internal node may have only 2 or 3 child nodes.

A *B*-tree is kept balanced by requiring that all leaf nodes are at the same depth. This depth will increase slowly as elements are added to the tree, but an increase in the overall depth is infrequent, and results in all leaf nodes being one more hop further removed from the root.

Difference from Red Black Tree

B-trees differ form red-black trees in that *B*-tree nodes may have many children, from a handful to thousands. That is, the "branching factor" of a *B*-tree can be quite large, although it is usually determined by characteristics of the disk unit used. *B*-trees are similar to red-black trees in that every n-node *B*-tree has height $O(\lg n)$, although the height of a *B*-tree can be considerably less than that of a red-black tree because its branching factor can be much larger, Therefore, *B*-trees can also be used to implement many dynamic-set operations in time $O(\lg n)$. *B*-trees are similar to red-black trees but they are better at minimizing disk I/O operations. Many database systems use *B*-trees, to store information.

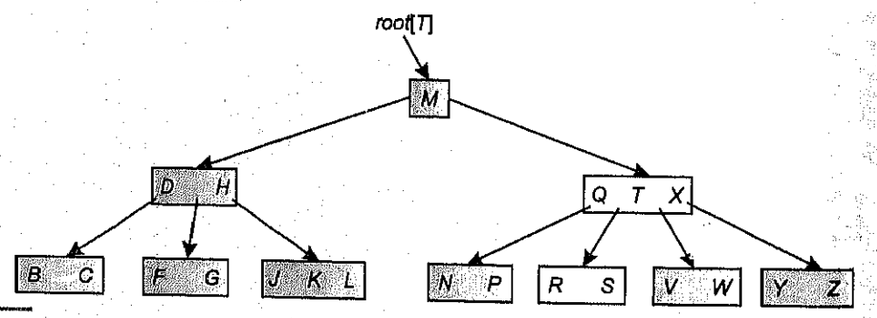


Figure 17.1

B-trees generalize binary search trees in a natural manner. If an internal *B*-tree node x contains $n[x]$ keys, then x has $n[x]+1$ children. The keys in node x are used as dividing points separating the range of keys handled by x into $n[x]+1$ sub ranges, each handled by one child of x. When searching for a key in a *B*-tree, we make an $(n[x]+1)$-way decision based on comparisons with the $n[x]$ keys stored at node x. Then structure of leaf nodes differs from that of internal nodes.

In a typical *B*-tree application, the amount of data handled is so large that all the data do not fit into main memory at once. The *B*-tree algorithms copy selected from the disk into main memory as needed and writes back onto disk, the pages that have changed. *B*-tree is designed so that only a constant number of pages are in main memory at any time; thus, the size of main memory does not limit the size of *B*-trees that can be handled.

Let x be a pointer to an object. If the object is currently in the computer's main memory, then we can refer to the fields of the object as usual ; $key[x]$, for example, If the object referred to by x resides on disk, however, then we must perform the operation. DISK-READ (x) to read object x into main memory before we can refer to its fields. (We assume that if x is already in main memory, then DISK-READ (x) requires no disk accesses ; it is a "no-op.") Similarly, the operation DISK-WRITE (x) is used to save any changes that have been made to the fields of object x. That is, the typical pattern for working with an object is as follows :

```
x ← a pointer to some object
DISK-READ(x)
operations that access and/or modify the fields of x

DISK-WRITE(x)                    // omitted if no fields of x were changed.
other operations that access but do not modify fields of x.
```


Since in most systems the running time of a *B*-tree algorithm is determined mainly by the number of DISK-READ and DISK-WRITE operations it performs, it is sensible to use these operations efficiently by having them read or write as much information as possible. Thus, a *B*-tree node is usually as large as a whole disk page. **The number of children a *B*-tree node can have is therefore limited by the size of a disk page.**

For a large *B*-tree stored on a disk, branching factors between 50 and 2000 are often used, depending on the size of a key relative to the size of a page. A large branching factor dramatically reduces both the height of the tree and the number of disk accesses required to find any key.

17.2 Definition of *B*-tree

A *B*-tree is a rooted tree (whose root is root $[T]$) having the following properties :

1. Every node x has the following fields :
   (*a*) $n[x]$, the number of keys currently stored in node x
   (*b*) the $n[x]$ keys themselves, stored in non-decreasing order, so that
   $$key_1[x] \le key_2[x] \le \ldots \le key_{n[x]}[x]$$
   (*c*) leaf$[x]$, a boolean value is TRUE if x is a leaf and FALSE if x is an internal node.
2. Each internal node x also contains $n[x]+1$ pointers $c_1[x], c_2[x], \ldots c_{n[x]+1}[x]$ to its children. Leaf nodes have no children, so their c_i fields are undefined.
3. The keys $key_i[x]$ separate the range of keys stored in each subtree : if k_i is any key stored in the subtree with root $c_i[x]$, then
   $$k_1 \le key_1[x] \le k_2 \le key_2[x] \le \ldots \le key_{n[x]}[x] \le key_{n[x]+1}$$
4. All leaves have the same depth, which is the tree's height h.
5. There are lower and upper bounds on the number of keys a node can contain. These bounds are expressed in terms of a fixed integer $t \ge 2$ called the minimum degree of the *B*-tree :
   (*a*) Every node other than the root must have at least $t-1$ keys. Every internal node other than the root has at least t children. If the tree is nonempty, the root must have at least one key.
   (*b*) Every node can contain at most $2t-1$ keys. Therefore, an internal node can have at most $2t$ children. We say that a node is full if it contains exactly $2t-1$ keys.

The simplest *B*-tree occurs when $t=2$. Every internal node then has either 2, 3, or 4 children, and we have a 2-3-4 tree. In practice, however, much larger values of t are used, typically adjusted to block size of hard disks for efficient I/O.

Theorem

If $n \ge 1$, then for any n-keys B-tree T of height h and minimum degree $t \ge 2$, then

$$h \le \log_t \frac{n+1}{2}$$

Proof. The root contains atleast one key. All other nodes contain atleast $t-1$ keys. There are atleast 2 nodes at depth 1, at least $2t$ nodes at depth 2, at least $2t^{i-1}$ nodes at depth i and $2t^{h-1}$ nodes at depth h.

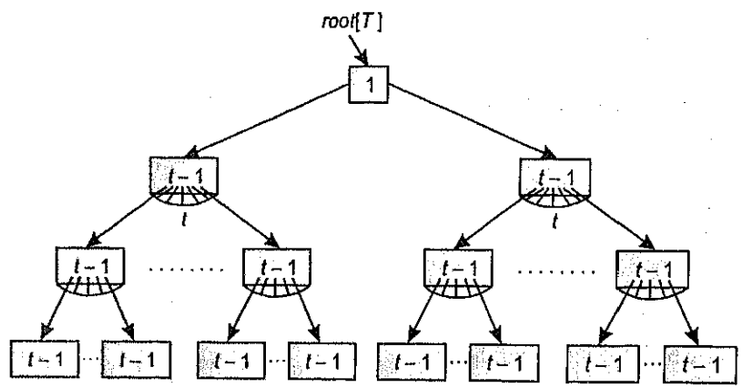


Figure 17.2 Figure shows the B-tree of height = 3

So we write $n \geq 1+(t-1)(2+2t+2t^2+2t^3+..+2t^{h-1})$

$$=1+2(t-1)(1+t+t^2+t^3...+t^{h-1})$$

$$=1+2(t-1)\left(\frac{(t^h-1)}{(t-1)}\right)$$

$$=2t^h-1$$

we get, $t^h \leq (n+1)/2$

Taking base-t logarithms of both sides, we get $h \leq \log_t (n+1)/2$.

17.3 Searching for a Key k in a B-tree

1. If x = nil, then k does not exist.
2. Compute the smallest i such that the ith key at x is greater than or equal to k.
3. If the ith key is equal to k, then the search is done.
4. Otherwise, set x to the ith child.

B-TREE-SEARCH (x, k)

```
1.  i ← 1
2.  while i ≤ n[x] and k > key_i[x]
3.      do i ← i + 1
4.  if i ≤ n[x] and k = key_i[x]
```



```
5.      then return (x, i)
6.  if leaf[x]
7.      then return NIL
8.      else DISK-READ (c_i[x])
9.          return B-TREE-SEARCH (c_i[x], k)
```

In B-TREE-SEARCH procedure, the nodes encountered during the recursion from a path down- ward from the root of the tree. The number of disk pages accessed by B-TREE-SEARCH is therefore $\theta(h)=\theta(\log_t n)$, where h is height of the tree and n is the number of keys in the tree. Since $n[x]<2t$, time taken by the while loop of lines 2-3 within each node is $O(t)$ and the total CPU time is $O(\text{th}) = O(t \log_t n)$

17.4 Creating an Empty B-Tree

To create an empty B-Tree a space for x is created and x is made the leaf node and then it is being made the root of the tree. ALLOCATE-NODE procedure allocates one disk page to be used as a new node in $O(1)$ time. We assume that it requires no DISK-READ, because there is no useful information stored on the disk for that node.

B-TREE-CREATE(T)

```
1.  x ← ALLOCATE-NODE( )
2.  leaf[x] ← TRUE
3.  n[x] ← 0
4.  DISK-WRITE(x)
5.  root[T] ← x
```

7.5 How do we search for a predecessor ?

B-TREE-PREDECESSOR (T, x, i)

```
1.  ▷ Find a pred. of key_i[x] in T
2.  if i ≥ 2 then
3.      if c_i[x] = nil then return key_{i-1}[x]
4.  ▷ if i ≥ 2 and x is a leaf
5.  ▷ return the (i - 1)st key
6.      else {
7.              ▷ if i ≥ 2 and x is not a leaf
8.              ▷ find the rightmost key in the i-th child
9.              y ← c_i[x]
10.             repeat
11.                 z ← c_{n[y]+1}
12.                 if z ≠ nil then y ← z
13.             until z = nil
14.             return key_{n[y]}[y]
15.         }
```

```
16.     else {
17.             ▷ Find y and j ≥ 1 such that
18.             ▷ x is the left most key in c_j[y]
19.                 while y ≠ root[T] and c_1[p[y]] = y do
20.                 y ← p[y]
21.             j ← 1
22.             while c_j[p[y]] ≠ y do j ← j+1
23.             if j = 1 then return "No Predecessor"
24.             return key_{j-1}[p[y]]
25.     }
```

17.6 Inserting a Key into a *B*-tree

Suppose that a key k needs to be inserted in the sub tree rooted at y in a *B*-tree T. Before inserting the key we make sure that is room for insertion, that is, not all the nodes in the sub tree are full. Since visiting all the nodes in the sub tree is very costly, we will make sure only that y is not full. If y is a leaf, insert the key. If not, find a child in which the key should go to and then make a recursive call with y set to the child.

B-TREE-INSERT (T, k)

```
1.  r ← root[T]
2.  if n[r] = 2t − 1
3.      then s ← ALLOCATE-NODE()
4.          root[T] ← s
5.          leaf[s] ← FALSE
6.          n[s] ← 0
7.          c_1[s] ← r
8.          B-TREE-SPLIT-CHILD (s, 1, r)
9.          B-TREE-INERT-NONFULL (s, k)
10.     else B-TREE-INSERT-NONFULL (r, k)
```

In *B*-TREE-INSERT line 3-9 handle the case in which the root node r is full : the root is split and a new node (having two children) becomes the root. Splitting the root is the only way to increase the height of the *B*-tree. **Unlike a binary search tree, a *B*-tree increases in height at the top instead of at the bottom.** *B*-TREE-SPLIT-CHILD introduce an operation that splits a full node y having $2t-1$ keys around its median key $key_t[y]$ into two nodes having $t-1$ keys each. The median key moves up into y's parent to identify the dividing point between the two new trees. But if y's

parent is full, it must be split before the new key can be inserted. Fig. shows this process. It takes as input a nonfull internal node x and a node y such that $y = c_i[x]$ is a full child of x. The procedure then splits this child in two and adjusts x so that it has an additional child. The tree thus grows in height by one, splitting is the only means by which tree grows.

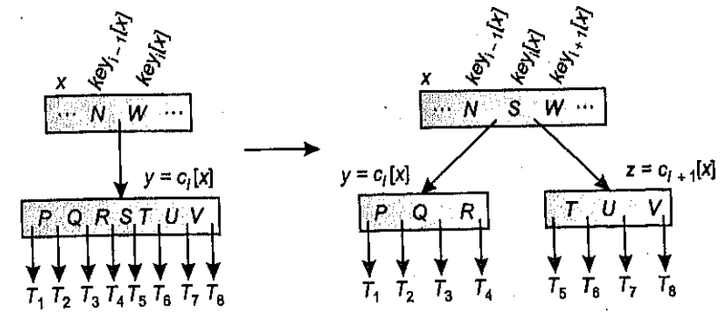


Figure 17.3

B-TREE-SPLIT-CHILD (x, i, y)

```
1.  z ← ALLOCATE-NODE()
2.  leaf[z] ← leaf[y]
3.  n[z] ← t − 1
4.  for j ← 1 to t − 1
5.      do key_j[z] ← key_{j+t}[y]
6.  if not leaf[y]
7.      then for j ← 1 to t
8.          do c_j[z] ← c_{j+t}[y]
9.  n[y] ← t − 1
10. for j ← n[x] + 1 down to i + 1
11.     do c_{j+1}[x] ← c_j[x]
12. c_{i+1}[x] ← z
13. for j ← n[x] down to i
14.     do key_{j+1}[x] ← key_j[x]
15. key_i[x] ← key_t[y]
16. n[x] ← n[x] + 1
17. DISK-WRITE(y)
18. DISK-WRITE(z)
19. DISK-WRITE(x)
```

The insertion procedure finishes by calling *B*-TREE-INSERT-NONFULL to perform the insertion of the key in the tree rooted at the nonfull root node.

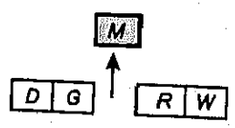
B-TREE-INSERT-NONFULL (x, k)

```
i ← n[x]
if leaf[x]
    then while i ≥ 1 and k < key_i[x]
            do key_{i+1}[x] ← key_i[x]
               i ← i − 1
         key_{i+1}[x] ← k
         n[x] ← n[x] + 1
         DISK-WRITE(x)
    else while i ≥ 1 and k < key_i[x]
    do i ← i − 1
       i ← i + 1
       DISK-READ (c_i[x])
       if n[c_i[x]] = 2t − 1
       then B-TREE-SPILIT-CHILD (x, i, c_i[x])
            if k > key_i[x]
            then i ← i + 1
    B-TREE-INSERT-NONFULL (c_i[x], k)
```

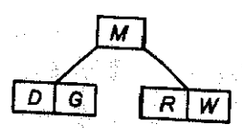
Example : *After inserting D, G, M, R, and W into a B-Tree with minimum degree 3, 2 to 5 values per node :*

$t = 3$

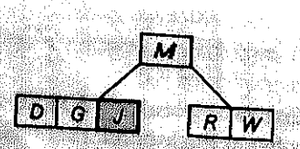
◄ To insert '*J*' : Node is full thus before insert node *j*.



The root node must be split. Move the middle value up and create two children.

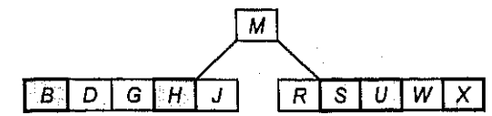


Set the child pointers.

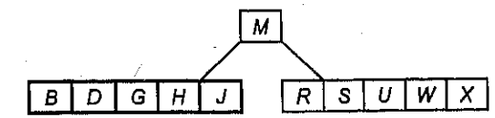


Place '*J*' in the appropriate node.

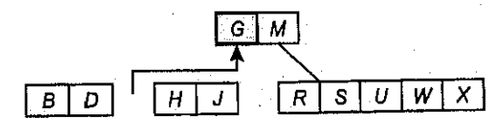
After inserting *B, H, S, U,* and X :



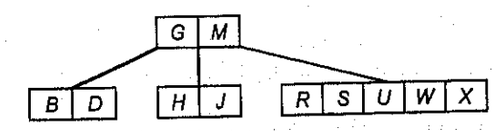
◄ To Insert '*A*' :



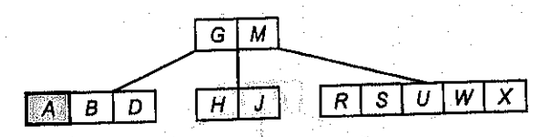
The node where '*A*' must go, is full, so it must be split before inserting node *A*.



Move the middle value, *G*, up into its parent and create 2 children.

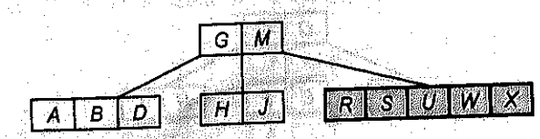


Set the children pointers.

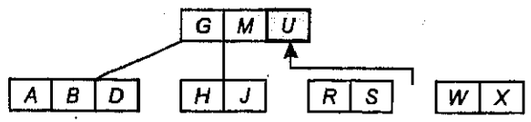


Place '*A*' in the appropriate node.

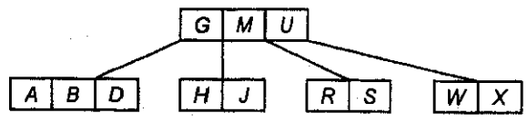
◄ To insert '*T*' :



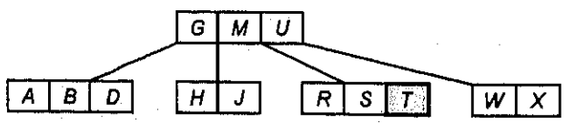
The node where 'T' goes is full, so it must be split.



Move the middle value, *U*, up to its parent and create two children.



Set the child pointers.



Place 'T' in the appropriate node.

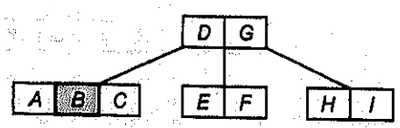
Deleting a Key from a *B*-tree

The *B*-tree delete algorithm has several different cases :

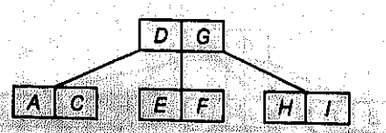
When deleting key *k* from a node *x* in a *B*-tree with $t = 3$:

Case 1. If *x* is a leaf node, then the key can just be removed.

Example : **Delete 'B'**



'B' is in a leaf node, so it can just be removed.



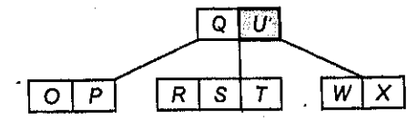
Case 2. If *x* is an internal node,

(*a*) If the key's left child has atleast *t* keys, then its largest value can be moved up to replace *k*.

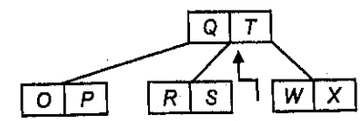
(*b*) If the key's right child has atleast *t* keys, then its smallest value can be moved up to replace *k*.

(*c*) If neither child has atleast *t* keys, then the two must be merged into one and *k* must be removed.

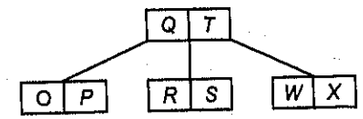
Example : **Delete 'U'**



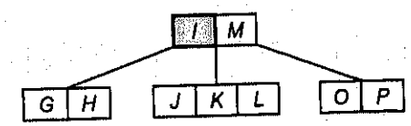
'U' is in an Internal Node.



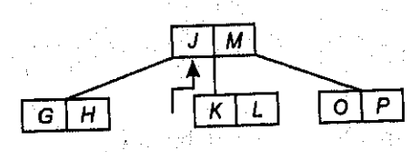
U's left child has *t* keys, so the largest value, 'T', can be moved up to replace 'U' by case 2*a*.



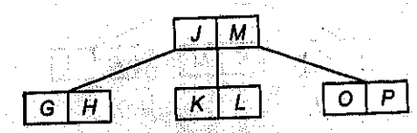
Example : **Delete 'I'**



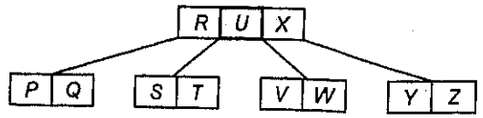
'I' is in an Internal Node.



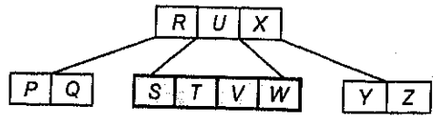
I's right child has *t* keys, so the smallest value, 'J', can be moved up to replace 'I' by case 2*b*.



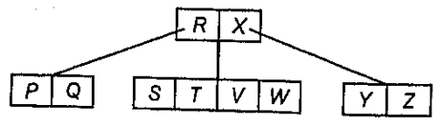
Example : **Delete 'U'**



'U's children both have $t-1$ keys.



Merge the two children into one node.



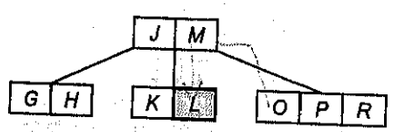
Remove 'U' and set the child pointer to the new node.

Case 3. If x has $t-1$ keys

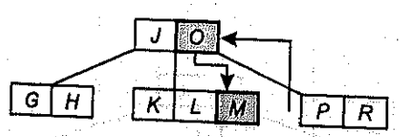
(*a*) If x has a sibling with at least t keys, move x's parent key into x, and move the appropriate extreme from x's sibling into the open slot in the parent node. Then delete the desired value.

(*b*) If x's siblings also have $t-1$ keys, merge x with one of its siblings by bringing down the parent to be the median value. Then delete the desire value.

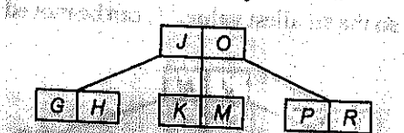
Example : **Delete 'L'**



'L' is in a node with $t-1$ keys.

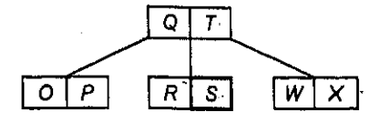


An immediate sibling has t keys, so move x parent key, 'M', into x. Move the sibling's appropriate extreme, 'O', into the parent node. All by case 3*a*.

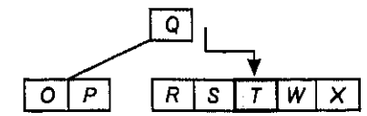


Now delete 'L' by case 1.

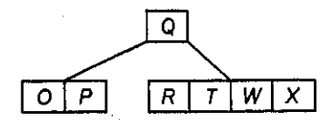
Example : **Delete 'S'**



'S' is in a node with $t-1$ keys, and its siblings also only have $t-1$ keys.



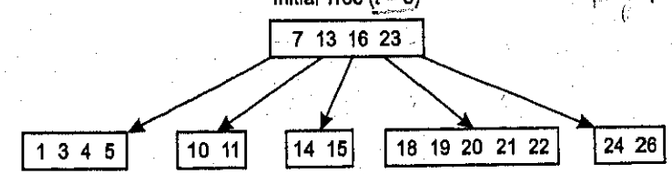
Choose one of the siblings and merge it with x by moving down the parent key to be the median for the new node, by case 3*b*



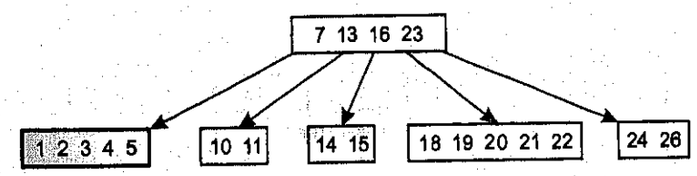
'S' can now be deleted by case 1.

Example : *Inserting a Key in a B-tree*

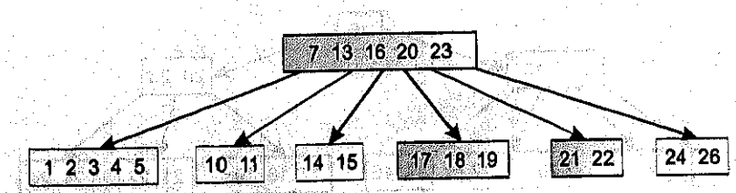
Initial Tree ($t = 3$)

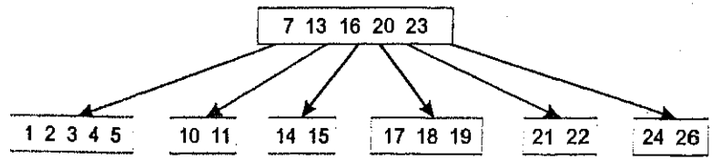


◄| Insert 2



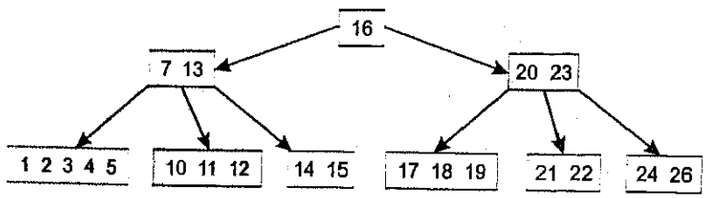
◄| Insert 17



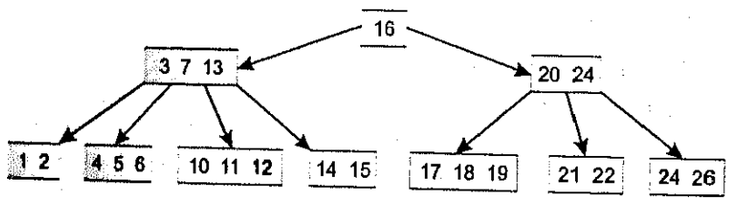


Initial tree ($t = 3$)

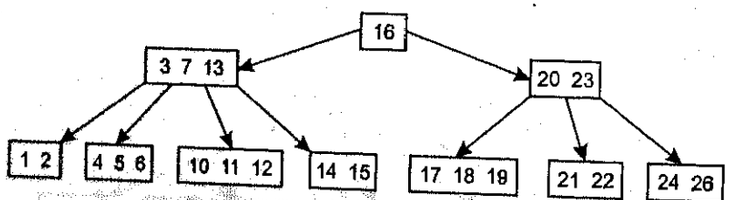
◄ Insert 12



◄ Insert 6

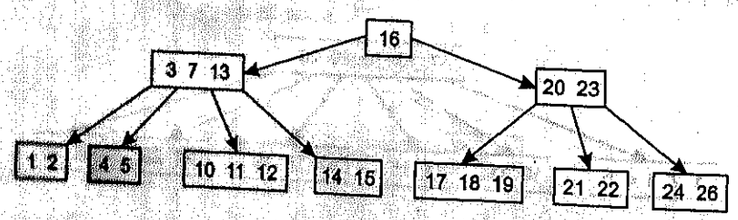


Example : *Deleting a Key in a B-tree*

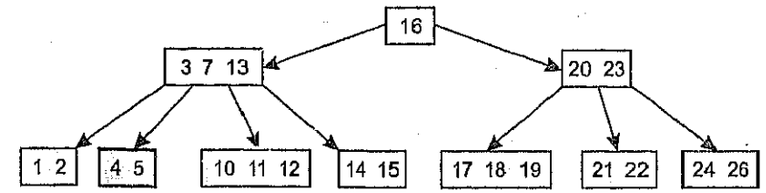


Initial Tree

◄ 6 deleted

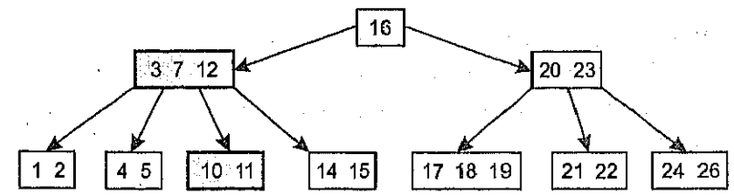


The first and simple case involves deleting the key from the leaf. $t-1$ keys remain.

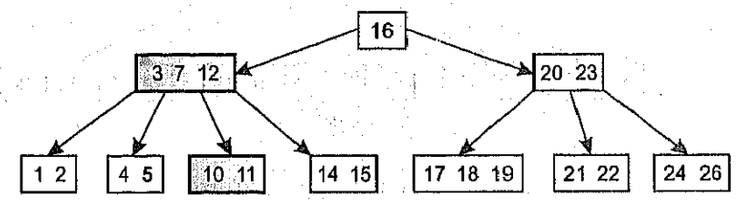


Initial tree

◄ 13 deleted

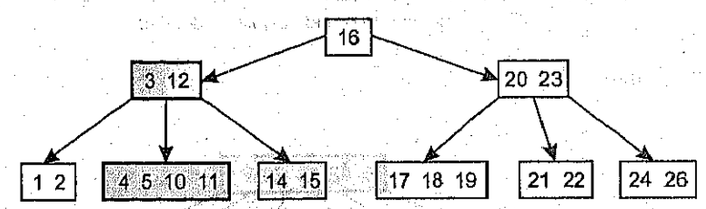


Case 2*a* is illustrated. The predecessor of 13, which lies in the preceding child of x, is moved up and takes 13*s* position. The preceding child had a key to spare in this case



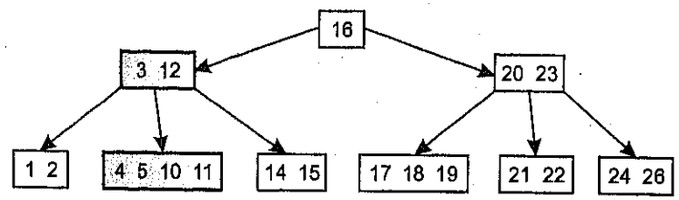
Initial tree

◄ 7 deleted



Here, both the preceding and successor children have $t-1$ keys, the minimum allowed. 7 is initially pushed down and between the children nodes to form one leaf, and is subsequently removed from the leaf.

◄ 4 deleted : Initial Tree and key 4 to be deleted in this tree.

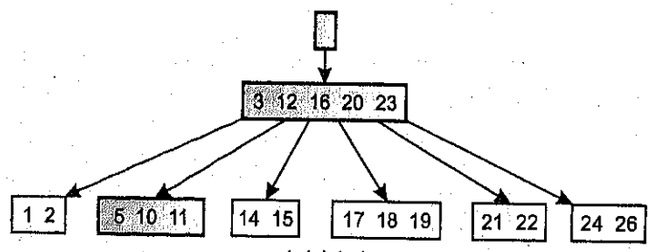


Recursion cannot descend to node 3, 12 because it has $t-1$ keys.

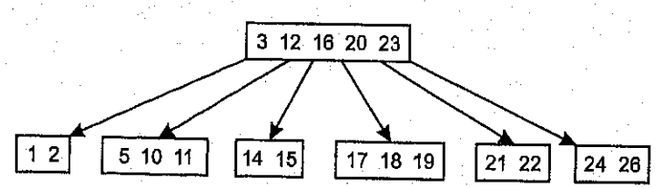
In case the two leaves to the left and right had more than $t-1$, 3, 12 could take one and 3 would be moved down.

Also, the sibling of 3, 12 has also $t-1$ keys, so it is not possible to move the root to the left and take the leftmost key from the sibling to be the new root.

Therefore the root has to be pushed down merging its own children, so that 4 can be safely deleted from the leaf.

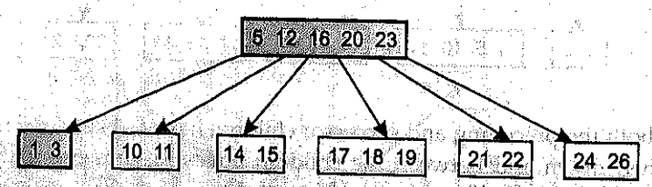


4-deleted



Outcome tree and now key 2 deleted

◄ 2 deleted



In this case 1, 2 has $t-1$ keys, but the sibling to the right has t. Recursion moves 5 to fill 3s position, 5 is moved to the appropriate leaf, and deleted from there.

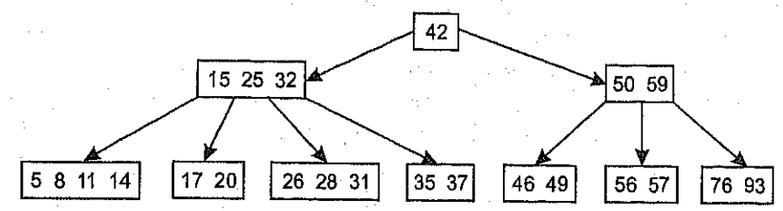
Example. *Why don't we allow a minimum degree of $t=1$.*

Solution. If we have minimum degree of t and it is not a root node, then its parent node must have $t-1$ *i.e.*, $1-1=0$ keys, but it is not possible because we cannot have a node with no keys in it so it is not possible thus $t \geq 2$.

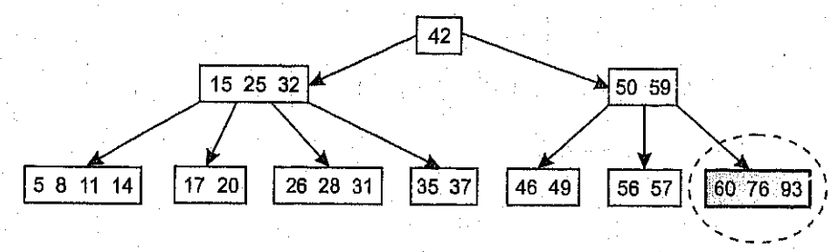
Example. *Given the following 5-way B-tree of integer data (i.e., each node can have atmost 5 children and 4 integer keys). Draw the B-Tree that results when we do each of the operations given. For each operation performed, start from the original tree i.e., these operations are not done in sequence. Circle each final answer.*

(i) Insert 60 *(ii) Remove 59*

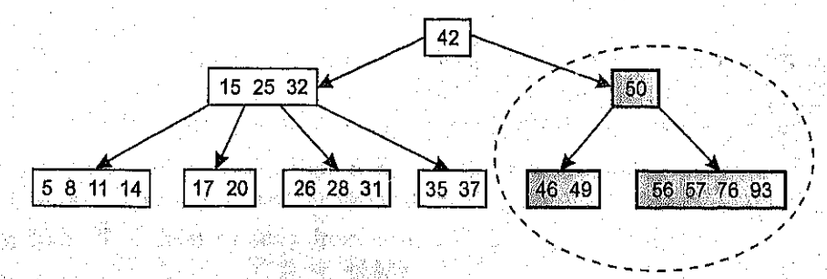
Start from B-Tree shown below



Solution. (*i*) **Insert 60.**

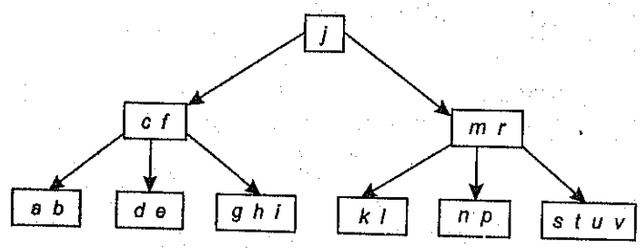


(*ii*) **Remove 59**



Exercise

1. Write procedures of operations :
   (*i*) *B*-TREE-SEARCH
   (*ii*) *B*-TREE-CREATE
   (*iii*) *B*-TREE-INSERT.
2. Show the results of inserting the keys *F, Q, P, K, A, L, R, M, N, X, Y, D, Z, E, H, T, V, W, C* in order to an empty *B*-Tree. Draw the configurations of the tree just before node splits. Also draw the final configurations.
3. Create *B*-Tree of order 5 from the following lists of data items.
   30, 20, 35, 95, 15, 60, 55, 25, 5, 65, 70, 10, 40, 50, 80, 45.
4. Create a *B*-Tree for the following list of elements $L = \langle 80, 40, 60, 20, 10, 30, 70, 50, 90, 110 \rangle$ given minimization factor $t = 3$, minimum degree = 2 and maximum degree = 5.
5. Show all legal *B*-trees of degree 2 that represent {1, 2, 3, 4, 5}.
6. Prove that maximum number of keys in a *B*-tree is $2t - 1$ if t is the minimum degree.
7. Define In-order, Pre-order and Post-order traversing of *B*-tree.
8. Draw all *B*-trees of order 5 that can be constructed from the keys {1, 2, 3, 4, 5, 6, 7, 8}.
9. Perform the deletion operation in the following *B*-tree.



delete *b, m, n, e, c* in sequence.

CHAPTER 18

Binomial Heaps

A **binomial heap** is a data structure similar to binary heap but also supporting the operation of merging two heaps quickly. This is achieved by using a special tree structure. It is important as an implementation of the **mergeable heap** abstract data type (also called meldable heap), which is a priority queue supporting merge operation.

18.1 Mergeable Heaps

A data structure known as **mergeable heaps**, which support the following *five* operations :

- **MAKE-HEAP()** creates and returns a new heap containing no elements.
- **INSERT(*H, x*)** inserts node *x*, whose *key* field has already been filled in, into heap *H*.
- **MINIMUM(*H*)** returns a pointer to the node in heap *H* whose key is minimum.
- **EXTRACT-MIN(*H*)** deletes the node from heap *H* whose key is minimum, returning a pointer to the node.
- **UNION(H_1, H_2)** creates and returns a new heap that contains all the nodes of heaps H_1 and H_2. Heaps H_1 and H_2 are "destroyed" by this operation.

In addition, the data structures also support the following *two* operations :

- **DECREASE-KEY (*H, x, k*)** assigns to node *x* within heap *H* the new key value *k*, which is assumed to be no greater than its current key value.
- **DELETE (*H, x*)** deletes node *x* from heap *H*.

18.2 Binomial Trees and Binomial Heaps

A binomial heap is a collection of binomial trees. A *binomial tree* B_k is an ordered tree defined recursively.

- The binomial tree B_0 consists of a single node.
- For $k \geq 1$, the binomial tree B_k consists of two binomial trees B_{k-1} that are *linked* together : the root of one is the leftmost child of the root of the other.

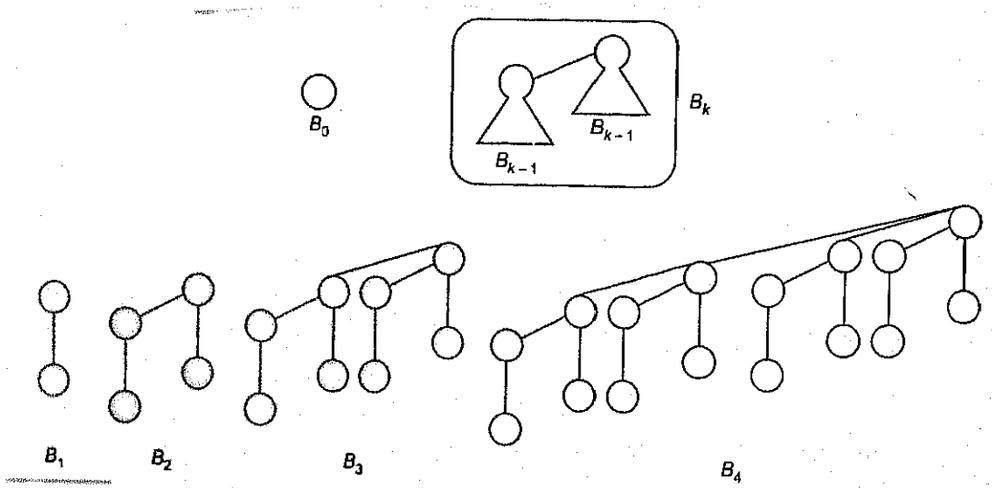


Figure 18.1

18.2.1 Properties of Binomial Trees

For the binomial tree B_k,

1. There are 2^k nodes,
2. the height of the tree is k,
3. there are exactly nodes $\binom{k}{i}$ at depth i for $i = 0, 1, \ldots, k$, and
4. the root has degree k, which is greater than that of any other node; moreover if the children of the root are numbered from left to right by $k-1, k-2, \ldots, 0$, *child i* is the root of a subtree B_i.

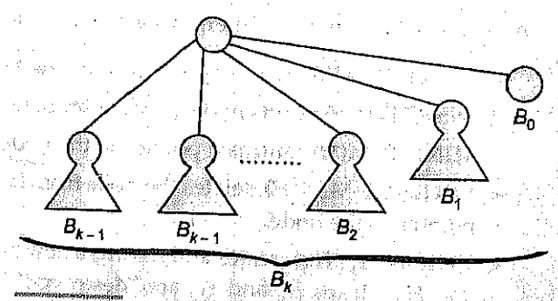


Figure 18.2

Proof : The proof is by induction on k. For each property, the basis is the binomial tree B_0. Verifying that each property holds for B_0 is trivial.

For the inductive step, we assume that this holds for B_{k-1}.

1. Binomial tree B_k consists of two copies of B_{k-1}, so B_k has $2^{k-1} + 2^{k-1} = 2^k$ nodes.
2. Because the two copies of B_{k-1} are linked to form B_k, the maximum depth of a node in B_k is one greater than the maximum depth in B_{k-1}. By the inductive hypothesis, this maximum depth is $(k-1)+1 = k$.
3. Let $D(k, i)$ be the number of nodes at depth i of binomial tree B_k. Since B_k is composed of two copies of B_{k-1} linked together, a node at depth i in B_{k-1} appears in B_k once at depth i and once at depth $i+1$. In other words, the number of nodes at depth i in B_k is the number of nodes at depth i in B_{k-1} plus the number of nodes at depth i - 1 in B_{k-1}. Thus,
$$D(k,i) = D(k-1, i) + D(k-1, i-1) = \binom{k-1}{i} + \binom{k-1}{i-1} = \binom{k}{i}$$
4. The only node with greater degree in B_k than in B_{k-1}, is the root, which has one more child than in B_{k-1}. Since the root of B_{k-1} has degree k-1, the root of B_k has degree k. Now by the inductive hypothesis, and as Figure 18.2 shows, from left to right, the children of the root of B_{k-1} are roots of $B_{k-2}, B_{k-3}, \ldots, B_0$. When B_{k-1} is linked to B_{k-1}, therefore, the children of the resulting root are roots of $B_{k-1}, B_{k-2}, \ldots, B_0$.

The term "**binomial tree**" comes from property 3 since the terms $\binom{k}{i}$ are the binomial coefficients.

i.e., ${}^kC_i$

The maximum degree of any node in an n-node binomial tree is lg n.

18.3 Binomial Heaps

A **binomial heap** H is a set of binomial trees that satisfies the following **binomial-heap properties :**

1. Each binomial tree in H is ***heap-ordered:*** the key of a node is greater than or equal to the key of its parent.
2. There is at most one binomial tree in H whose root has a given degree.

The first property ensures that the root of each binomial tree contains the smallest key in the tree, which applies to the entire heap. The second property implies that a binomial heap with n elements consists of at most $\lfloor \lg n \rfloor + 1$ binomial trees. In fact, the number of elements n uniquely determines the number and orders of these trees : **each binomial tree corresponds to digit one in the binary representation of number n.**

For example number 13 is 1101 in binary, $2^3 + 2^2 + 2^0$, and thus a binomial heap with 13 elements will consist of three binomial trees of orders 3, 2, and 0.

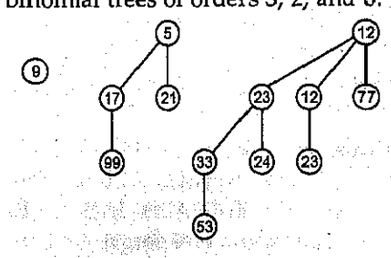


Figure 18.3

18.3.1 Representing Binomial Heaps

Each node has a *key* field and any other satellite information required by the application. In addition, each node x contains pointers $p[x]$ to its parent, *child* $[x]$ to its leftmost child, and *sibling*$[x]$ *to the sibling of* x immediately to its right. If node x is a root, then $p[x]$ = NIL. If node x has no children, then *child*$[x]$ = NIL, and if x is the rightmost child of its parent, then *sibling*$[x]$ = NIL. Each node x also contains the field *degree*$[x]$, which is the number of children of x.

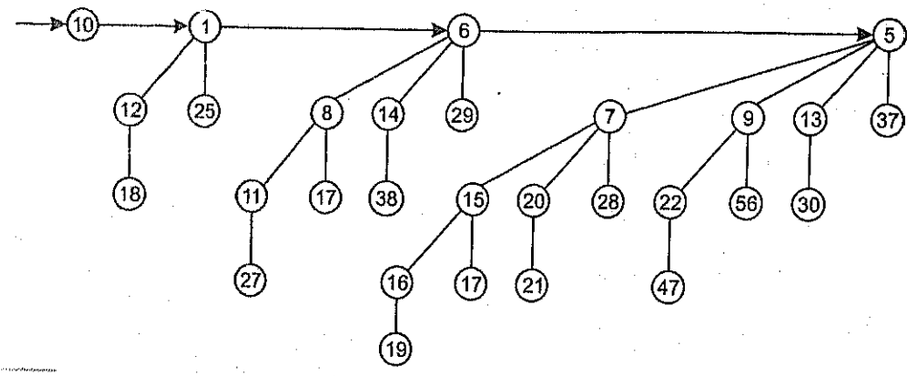


Figure 18.4

The roots of the binomial trees within a binomial heap are organized in a linked list, which we refer to as the *root list*. The degrees of the roots strictly increase as we traverse the root list. By the second binomial-heap property, in an n-node binomial heap the degrees of the roots are a subset of $\{0, 1, \ldots, \lfloor \lg n \rfloor\}$. If x is a root, then *sibling*$[x]$ points to the next root in the root list. (As usual, *sibling*$[x]$ = NIL *if* x is the last root in the root list.)

Thus, each node contains

1. a field **key** for its key,
2. a field **degree** for the number of children, degree [x]
3. a pointer **child**, which points to the leftmost-child, child[x]
4. a pointer **sibling**, which points to the right-sibling, and sibling [x]
5. pointer **p**, which points to the parent. p[x]

A given binomial heap H is accessed by the field *head*$[H]$, which is simply a pointer to the first root in the root list of H. If binomial heap H has no elements, then *head*$[H]$ = NIL.

18.3.2 Operations on Binomial Heaps

1. creation of a new heap,
2. search for the minimum key,
3. uniting two binomial heaps,
4. insertion of a node,
5. removal of the root of a tree,
6. decreasing a key, and
7. removal of a node.

18.3.2A Creating a New Binomial Heap

To make an empty binomial heap, the MAKE-BINOMIAL-HEAP procedure simply allocates and returns an object H, where *head* $[H]$ = NIL. The running time is $\theta(1)$.

18.3.2B Searching the minimum key

The procedure BINOMIAL-HEAP-MINIMUM returns a pointer to the node with the minimum key in an n-node binomial heap H. This implementation assumes that there are no keys with value ∞.

BINOMIAL-HEAP-MINIMUM (H)

```
y ← NIL
x ← head[H]
min ← ∞
while x ≠ NIL
    do if key[x] < min
        then min ← key[x]
             y ← x
    x ← sibling[x]
return y
```

Since a binomial heap is heap-ordered, the minimum key must reside in a root node. The BINOMIAL-HEAP-MINIMUM procedure checks all roots, which number at most $\lfloor \lg n \rfloor + 1$, saving the current minimum in *min* and a pointer to the current minimum in y. Because there are at most $\lfloor \lg n \rfloor + 1$ roots to check, the running time of BINOMIAL-HEAP-MINIMUM is $O(\lg n)$.

18.3.2C Uniting two binomial heaps

The BINOMIAL-HEAP-UNION procedure repeatedly links binomial trees whose roots have the same degree. The following procedure links the B_{k-1} tree rooted at node y to the B_{k-1} tree rooted at node z; that is, it makes z the parent of y. Node z thus becomes the root of a B_k tree.

BINOMIAL-LINK (y, z)

```
p[y] ← z
sibling[y] ← child[z]
child[z] ← y
degree[z] ← degree[z] + 1
```

The BINOMIAL-LINK procedure makes node y the new head of the linked list of node z's children in $O(1)$ time. It works because the left-child, right-sibling representation of each binomial tree matches the ordering property of the tree: in a B_k tree, the **leftmost child of the root is the root of a B_{k-1} tree.**

The following procedure unites binomial heaps H_1 and H_2, returning the resulting heap. It destroys the representations of H_1 and H_2 in the process. Besides BINOMIAL-LINK, the procedure uses an auxiliary procedure BINOMIAL-HEAP-MERGE that merges the root lists of H_1 and H_2 into a single linked list that is sorted by degree into monotonically increasing order.

BINOMIAL-HEAP-UNION(*H1*, *H2*)

```
H ← MAKE-BINOMIAL-HEAP( )
head[H] ← BINOMIAL-HEAP-MERGE(H1, H2)
free the objects H1 and H2 but not the lists they point to
if head[H] = NIL
    then return H
prev - x ← NIL
x ← head[H]
next-x ← sibling[x]
while next - x ≠ NIL
    do if (degree[x] ≠ degree[next - x]) or
          (sibling[next - x] ≠ NIL and degree[sibling[next - x]] = degree[x])
        then prev - x ← x                                   // Cases 1 and 2
             x ← next - x                                   // Cases 1 and 2
        else if key[x] ≤ key[next - x]
             then sibling[x] ← sibling[next - x]            // Case 3
                  BINOMIAL-LINK (next - x, x)               // Case 3
             else if prev - x = NIL                         // Case 4
                  then head[H] ← next - x                   // Case 4
                  else sibling[prev - x] ← next - x         // Case 4
                  BINOMIAL-LINK(x, next - x)                // Case 4
                  x ← next - x                              // Case 4
    next - x ← sibling[x]
return H
```

The BINOMIAL-HEAP-UNION procedure has two phases. The first phase, performed by the call of BINOMIAL-HEAP-MERGE, merges the root lists of binomial heaps H_1 and H_2 into a single linked list H that is sorted by degree into monotonically increasing order. There might be as many as two roots (but no more) of each degree, however, so the second phase links roots of equal degree until at most one root remains of each degree. Because the linked list H is sorted by degree, we can perform all the link operations quickly.

Binomial-Heap-Merge (*H1*, *H2*)

```
a ← head[H1]
b ← head[H2]
head [H1] ← Min-Degree(a, b)
if head[H1] = NIL
    return
if head[H1] = b
    then b ← a
a ← head[H1]
while b ≠ NIL
```



```
    do if sibling[a] = NIL
        then sibling[a] ← b
             return
        else if degree[sibling[a]] < degree[b]
             then a ← sibling[a]
             else c ← sibling[b]
                  sibling[b] ← sibling[a]
                  sibling[a] ← b
                  a ← sibling[a]
                  b ← c
```

The running time of BINOMIAL-HEAP-UNION is $O(\lg n)$, where n is the total number of nodes in binomial heaps H_1 and H_2.

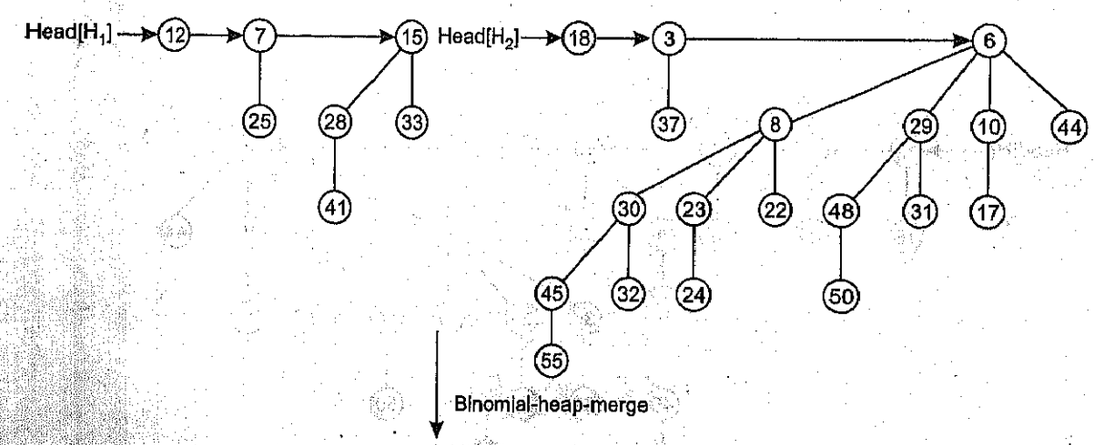
Inserting a node

The following procedure inserts node x into binomial heap H

BINOMIAL-HEAP-INSERT (*H*, *x*)

```
H' ← MAKE-BINOMIAL-HEAP( )
p[x] ← NIL
child[x] ← NIL
sibling[x] ← NIL
degree[x] ← 0
head[H'] ← x
H ← BINOMIAL-HEAP-UNION(H,H')
```

Example of Binomial Heap-Union

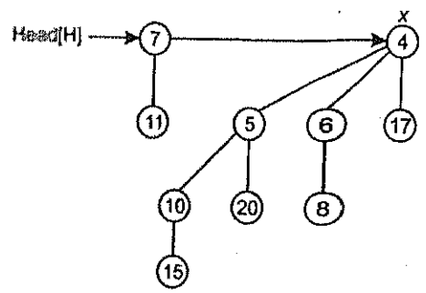


degree[x] = degree[next − x]

degree[x] ≠ degree[sibling(next − x)] and sibling(next − x) = NIL

key[x] < key(next − x)

So, case 3 apply, we get



Now, next − x = NIL, so this is the final binomial heap.

Exercise

1. How many entries does a binomial tree of rank n have ?
2. What conditions does a binomial tree satisfy ?
3. How do you join two binomial trees ? When is this a legitimate operation for binomial trees ?
4. A binomial heap has four binomial trees. Their degrees are 0, 1, 2 and 4. After you add an entry how many binomial trees will the heap have ? What are the degrees of the trees ?
5. Describe how you would use a binomial heap as an intermediate data structure in order to sort a list. Derive a tight asymptotic bound for the worst-case running time of your algorithm.
6. Can binomial tree be empty ? Why ?
7. What operations can be done efficiently on a binomial heap ? What is the advantage over a standard heap based on an almost complete binary tree ?
8. Prove that binomial tree B_k contains 2^k nodes, of which $\binom{k}{i}$ are at depth i, $0 \le i \le k$.
9. Prove that binomial heap containing n items comprises at most $\lceil \lg n \rceil$ binomial trees, the largest of which is $2^{\lfloor \lg n \rfloor}$ items.
10. Discuss the relationship between inserting into a binomial heap and incrementing a binary number and the relationship between uniting two binomial heaps and adding two binary numbers.

CHAPTER

19

Fibonacci Heaps

19.1 Introduction

Fibonacci heaps are linked lists of heap-ordered trees (children's keys are at least those of their parents) with the following characteristics :

1. The trees are not necessarily binomial.
2. Siblings are bi-directionally linked.
3. There is a pointer min [H] to the root with the minimum key.
4. The root degrees are not unique.
5. A special attribute $n[H]$ maintains the total number of nodes.
6. Each node has an additional Boolean label mark, indicating whether has lost a child since the last time it was made a child of another node.

Like a binomial heap, a Fibonacci heap is a collection of trees. Fibonacci heaps, in fact, are loosely based on binomial heaps. If neither DECREASE-KEY nor DELETE is ever invoked on a Fibonacci heap, each tree in the heap is like a binomial tree. **Fibonacci heaps differ from binomial heaps, however, in that they have a more relaxed structure, allowing for improved asymptotic time bounds.**

Fibonacci heaps offer a good example of a data structure designed with amortized analysis in mind.

Like binomial heaps, Fibonacci heaps are not designed to give efficient support to the operation SEARCH; operations that refer to a given node, therefore require a pointer to that node as part of their input.

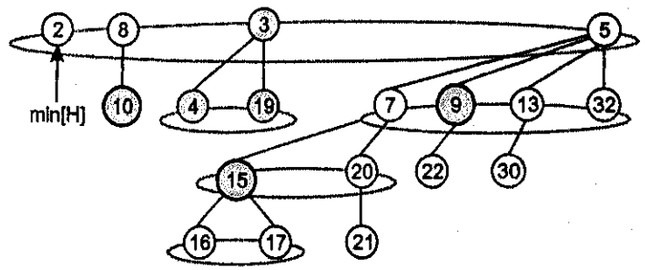


Figure 19.1

19.2 Structure of Fibonacci Heaps

Like a binomial heap, a ***Fibonacci heap*** is a collection of heap-ordered trees. The trees in a Fibonacci heap are not constrained to be binomial trees. Fig. 19.1 shows an example of a Fibonacci heap.

Unlike trees within binomial heaps, which are ordered, trees within Fibonacci heaps are rooted but unordered. Each node x contains a pointer $p[x]$ to its parent and a pointer $child[x]$ to any one of its children. The children of x are linked together in a circular, doubly linked list, which we call the ***child list*** of x. Each child y in a child list has pointers $left[y]$ and $right[y]$ that point to y's left and right siblings, respectively. If node y is an only child, then $left[y] = right[y] = y$. The order in which siblings appear in a child list is arbitrary.

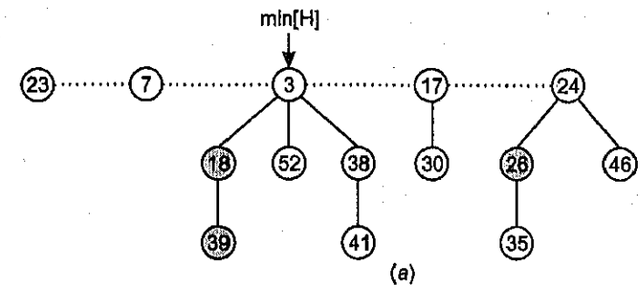
Two other fields in each node will be of use. The number of children in the child list of node x is stored in $degree[x]$. The boolean-valued field $mark[x]$ indicates whether node x has lost a child since the last time x was made the child of another node. Newly created nodes are unmarked, and a node x becomes unmarked whenever it is made the child of another node

19.2.1 Advantages of Circular, doubly linked lists use in Fibonacci heaps

Circular, doubly linked lists have two advantages for use in Fibonacci heaps. *First*, we can remove a node from a circular, doubly linked list in $O(1)$ time. *Second*, given two such lists, we can concatenate them (or "merge" them together) into one circular, doubly linked list in $O(1)$ time.

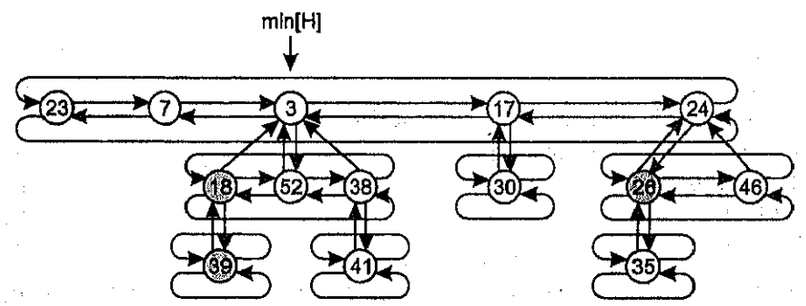
A given Fibonacci heap H is accessed by a pointer $min[H]$ to the root of the tree containing a minimum key; this node is called the **minimum node** of the Fibonacci heap. If a Fibonacci heap H is empty, then $min[H] = \text{NIL}$.

The roots of all the trees in a Fibonacci heap are linked together using their *left* and *right* pointers into a circular, doubly linked list called the **root list** of the Fibonacci heap. The pointer $min[H]$ thus points to the node in the root list whose key is minimum. The order of the trees within a root list is arbitrary. $n[H]$ is the number of nodes currently in H.



(a)

The dashed line indicates the root list. The minimum node of the heap is the node containing the key 3. The three marked nodes are blackened. The potential of this particular Fibonacci heap is $5 + 2.3 = 11$.



(b)

Figure 19.2 A Fibonacci heap consisting of five heap-ordered trees and 14 nodes. (a) Simple representation. (b) A more complete representation showing pointers p (up arrows), child (down arrows), and left and right (sideways arrows)

19.3 Potential function

To analyze the performance of Fibonacci heap operations we use the potential method. For a given Fibonacci heap H, we indicate by $t(H)$ the number of trees in the root list of H and by $m(H)$ the number of marked nodes in H. The potential of Fibonacci heap H is then defined by

$$\Phi(H) = t(H) + 2m(H)$$

For example, the potential of the Fibonacci heap shown in Fig. 19.2 is $5 + 2.3 = 11$. We assume that a Fibonacci heap application begins with no heaps. The initial potential, therefore, is 0. The potential function of a set of fibonacci heaps is the sum of individual potential function of fibonacci heaps.

19.4 Operations

19.4.1 Creating a new Fibonacci heap

To make an empty Fibonacci heap, the MAKE-FIB-HEAP procedure allocates and returns the Fibonacci heap object H, where $n[H]=0$ and $min[H]=$ NIL ; there are no trees in H. Because $t(H)=0$ and $m(H)=0$, the potential of the empty Fibonacci heap is, therefore, $\Phi(H)=0$. The amortized cost of MAKE-FIB-HEAP is thus equal to its $O(1)$ actual cost.

19.4.2 Inserting a Node

The following procedure inserts node x into Fibonacci heap H, assuming that the node has already been allocated and that $key[x]$ has already been filled in.

```
FIB-HEAP-INSERT (H, x)
degree[x] ← 0
p[x] ← NIL
child[x] ← NIL
left[x] ← x
right[x] ← x
mark[x] ← FALSE
concatenate the root list containing x with root list H
if min[H] = NIL or key[x] < key[min[H]]
        then min[H] ← x
n[H] ← n[H] + 1
```

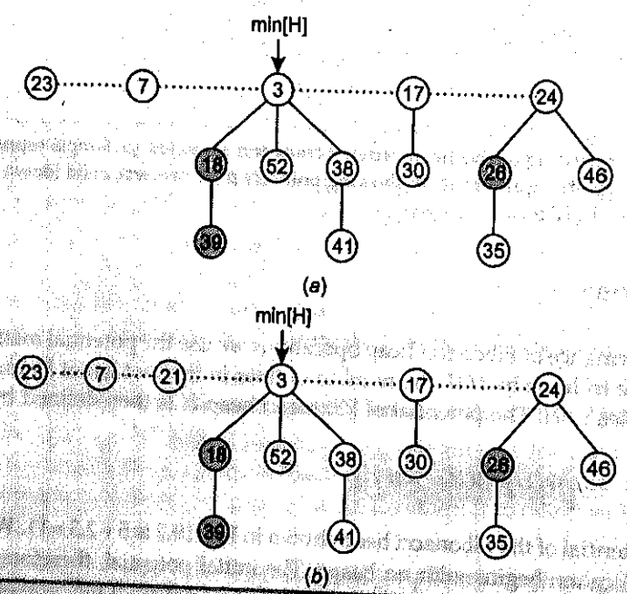


Figure 19.3 Inserting a node into a Fibonacci heap. (a) A Fibonacci heap H. (b) Fibonacci heap H after the node with key 21 has been inserted.

Unlike the BINOMIAL-HEAP-INSERT procedure, FIB-HEAP-INSERT makes no attempt to consolidate the trees within the Fibonacci heap. If k consecutive FIB-HEAP-INSERT operations occur, then k single-node trees are added to the root list.

To determine the amortized cost of FIB-HEAP-INSERT, let H be the input Fibonacci heap and H' be the resulting Fibonacci heap. Then, $t(H)=t(H)+1$ and $m(H')=m(H)$, and the increase in potential is

$$(t(H)+1)+2m(H))-(t(H)+2m(H))=1.$$

Since the actual cost is $O(1)$, the amortized cost is $O(1)+1=O(1)$.

19.4.3 Finding the Minimum Node

The minimum node of a Fibonacci heap H is always the root node given by the pointer $min[H]$, so we can find the minimum node in $O(1)$ actual time. Because the potential of H does not change, the amortized cost of this operation is equal to its $O(1)$ actual cost.

19.4.4 Uniting two Fibonacci Heaps

The following procedure unites Fibonacci heaps H_1 and H_2, destroying H_1 and H_2 in the process.

```
FIB-HEAP-UNION (H1, H2)
1.  H ← MAKE-FIB-HEAP()
2.  min[H] ← min[H1]
3.  concatenate the root list of H2 with the root list of H
4.  if (min[H1] = NIL) or (min[H2] ≠ NIL and min[H2] < min[H1])
5.      then min[H] ← min[H2]
6.  n[H] ← n[H1] + n[H2]
7.  free the objects H1 and H2
8.  return H
```

Lines 1-3 concatenate the root lists of H_1 and H_2 into a new root list H. Lines 2, 4, and 5 set the minimum node of H, and line 6 sets $n[H]$ to the total number of nodes. The Fibonacci heap objects H_1 and H_2 are freed in line 7, and line 8 returns the resulting Fibonacci heap H. As in the FIB-HEAP-INSERT procedure, no consolidation of trees occurs.

The change in potential is

$$\begin{aligned}&\Phi(H)-(\Phi(H_1)+\Phi(H_2))\\&=(t(H)+2m(H))-((t(H_1)+2m(H_1))+(t(H_2)+2m(H_2)))\\&=0\end{aligned}$$

because $t(H)=t(H_1)+t(H_2)$ and $m(H)=m(H_1)+m(H_2)$. The amortized cost of FIB-HEAP-UNION is therefore equal to its $O(1)$ actual cost.

19.4.5 Extracting the Minimum Node

The following pseudocode extracts the minimum node. The code assumes for convenience that when a node is removed from a linked list, pointers remaining in the list are updated, but pointers in the extracted node are left unchanged. It also uses the auxiliary procedure CONSOLIDATE

FIB-HEAP-EXTRACT-MIN (H)

```
1.  z ← min[H]
2.  if z ≠ NIL
3.      then for each child x of z
4.              do add x to the root list of H
5.              p[x] ←NIL
6.          remove z from the root list of H
7.          if z = right[z]
8.              then min[H] ← NIL
9.              else min[H] ← right[z]
10.                 CONSOLIDATE(H)
11.             n[H] ← n[H]−1
12. return z
```

FIB-HEAP-EXTRACT-MIN works by first making a root out of each of the minimum node's children and removing the minimum node from the root list. It then consolidates the root list by linking roots of equal degree until at most one root remains of each degree.

We start in line 1 by saving a pointer z to the minimum node; this pointer is returned at the end. If z = NIL, then Fibonacci heap H is already empty. Otherwise, as in the BINOMIAL-HEAP-EXTRACT-MIN procedure, we delete node z from H by making all of z's children roots of H in lines 3-5 (putting them into the root list) and removing z from the root list in line 6. If $z = right[z]$ after line 6, then z was the only node on the root list and it had no children, so all that remains is to make the Fibonacci heap empty in line 8 before returning z. Otherwise, we set the pointer $min[H]$ into the root list to point to a node other than z (in this case, $right[z]$).

The next step, in which we reduce the number of trees in the Fibonacci heap, is *consolidating* the root list of H; this is performed by the call CONSOLIDATE(H). Consolidating the root list consists of repeatedly executing the following steps until every root in the root list has a distinct *degree* value.

1. Find two roots x and y in the root list with the same degree, where $key[x] \le key[y]$.
2. *Link* y to x: remove y from the root list, and make y a child of x. This operation is performed by the FIB-HEAP-LINK procedure. The field $degree[x]$ is incremented, and the mark on y, if any, is cleared.

The procedure CONSOLIDATE uses an auxiliary array $A[0..D(n[H])]$; if $A[i] = y$, then y is currently a root with $degree[y] = i$.

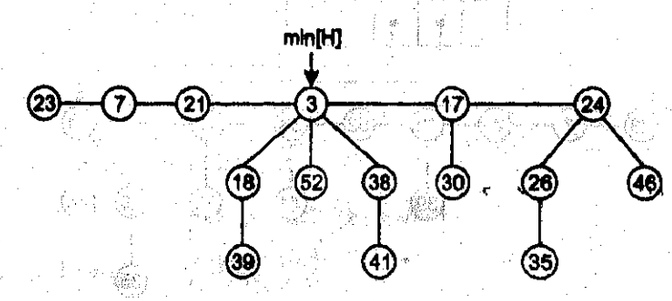
CONSOLIDATE (H)

```
for i ← 0 to D(n[H])
    do A[i] ← NIL
for each node w in the root list of H
    do x ← w
        d ← degree[x]
        while A[d] ≠ NIL
            do y ← A[d]
                if key[x] > key[y]
                    then exchange x ↔ y
                    FIB-HEAP-LINK(H, y, x)
                    A[d] ← NIL
                    d ← d + 1
                A[d] ← x
min[H] ← NIL
for i ← 0 to D(n[H])
    do if A[i] ≠ NIL
        then add A[i] to the root list of H
            if min[H] = NIL or key[A[i]] < key[min[H]]
                then min[H] ← A[i]
```

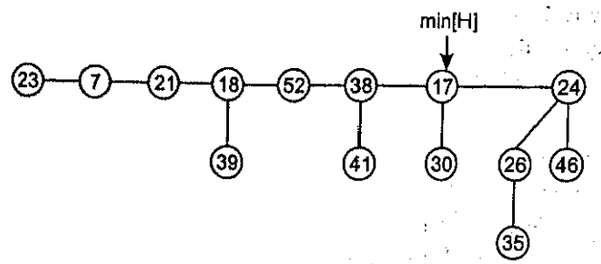
FIB-HEAP-LINK (H, y, x)

```
remove y from the root list of H
make y a child of x, incrementing degree[x]
mark[y] ← FALSE
```

Example. *Show the Fibonacci heap that results from calling FIB-HEAP-EXTRACT-MIN on the Fibonacci heap shown in the figure.*



Solution. In first step the minimum node z *i.e.*, 3 is removed from the root list and its children are added to the root list.

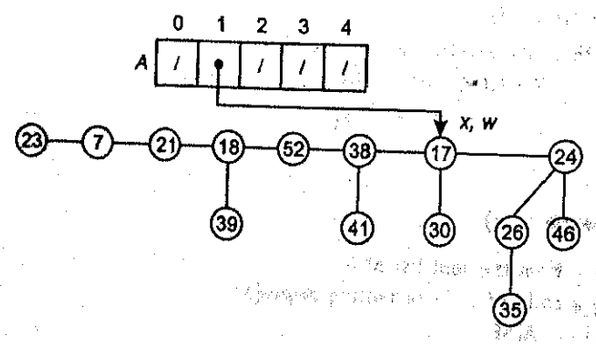


Now, min $[H] \leftarrow$ right $[z]$

Now, call consolidate (H), for this we use an auxiliary array $A[0..D(n[H])]$ and making each entry NIL. Then root list is processed by starting at the node pointed to by min$[H]$.

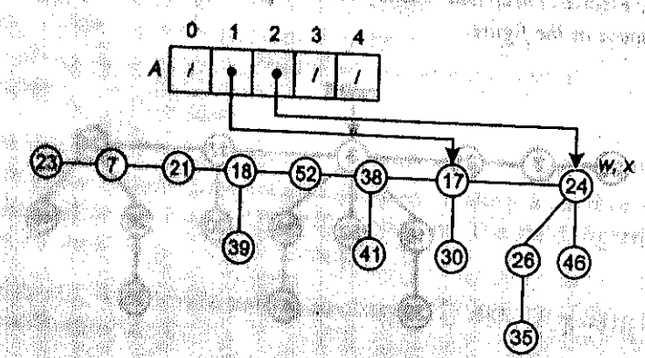
degree $[x] = 1$ $\quad\therefore\quad d = 1$

$A[d] = \text{NIL}$

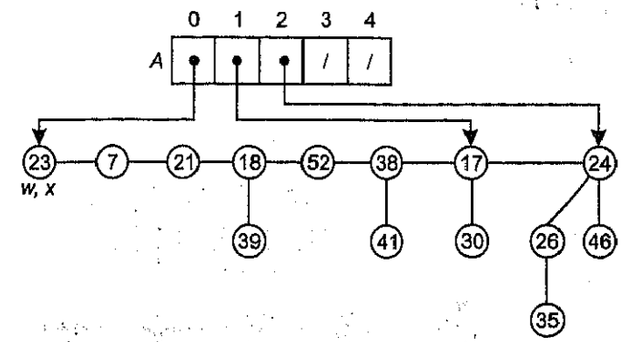


Now, w is the right node *i.e.*,

degree$[x] = 2$



degree$[x] = 0$



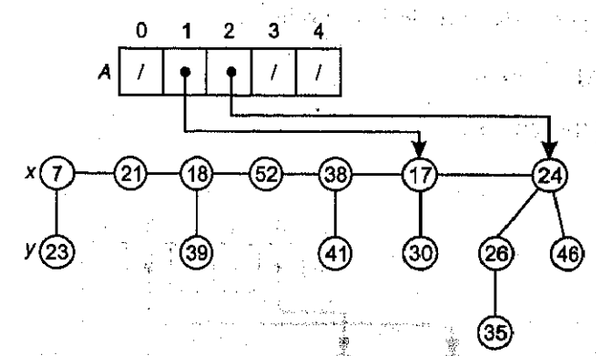
Now, w is the right node of node 23 *i.e.*, node 7.

degree$[x] = 0$

and $\quad A[0] \neq \text{NIL}$

then $\quad y = A[0]$ *i.e.*, node 23.

and key$[x]$ <key$[y]$. Now call FIB-HEAP-LINK procedure *i.e.*, remove y from the root list and make y a child of x and degree$[x]$ is incremented and the mark on y, if any, is cleared *i.e.*,



But $\quad A[1] \neq \text{NIL}$, thus $y = A[1]$

here, $\quad$ key$[x]$ < key$[y]$

so call FIB-HEAP-LINK and make y a child of x and degree of x is incremented

i.e., $\quad$ degree$[x] = 1 + 1 = 2$ so $d = 2$

But $\quad A[d] \neq \text{NIL}$, then again $y = A[d]$ *i.e.*, node 24.

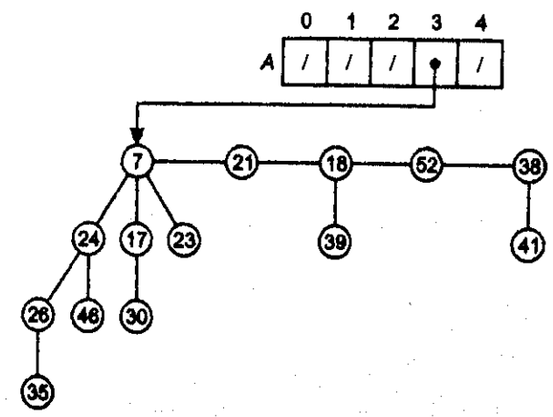
key$[x]$ < key$[y]$

Now call FIB-HEAP-LINK procedure and degree[x] is incremented *i.e.*,

now, degree[x] $=2+1=3$

$d=3$

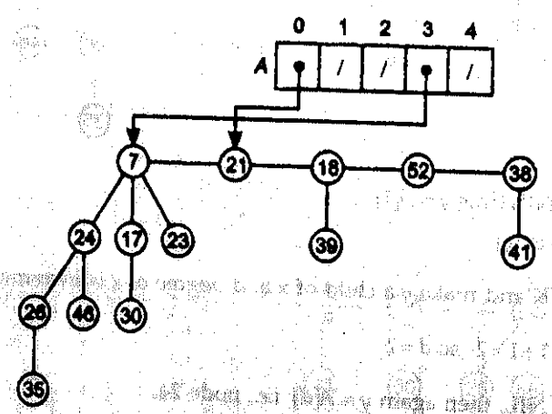
$A[3]=$ NIL



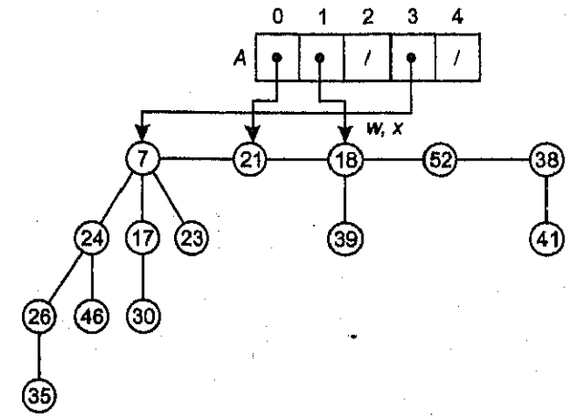
Now, w is the right node of node 7 *i.e.*, node 21.

degree [x] $=0$ so $d=0$

and $A[0]=$ NIL, so



Now, w is the node 18 and degree [x] $=1$ and $A[1]=$ NIL, so



Now, w is the node 52 and node 52 has degree 0 *i.e.*, $d=0$

$A[d] \neq$ NIL $y \leftarrow A[d]$

key[x] > key[y] so exchange $x \leftrightarrow y$.

Now x is node 21.

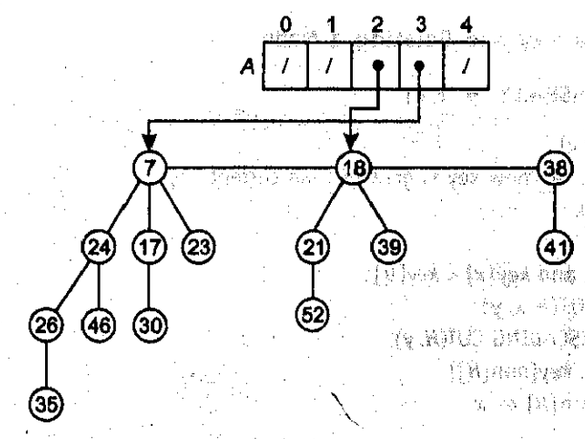
call FIB-HEAP-LINK

$A[d] \neq$ NIL again. $y \leftarrow A[d]$

key[x] > key[y] so exchange $x \leftrightarrow y$.

Now x is 18 and $d=1+1=2$.

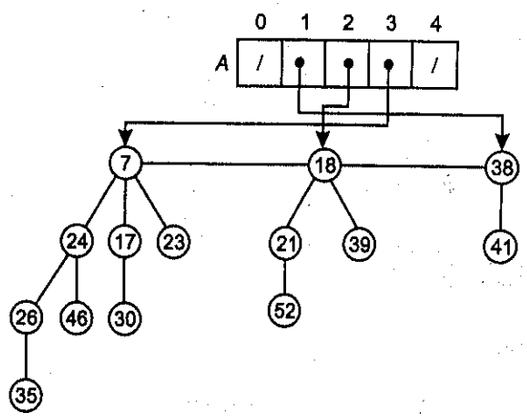
$A[d]=$ NIL



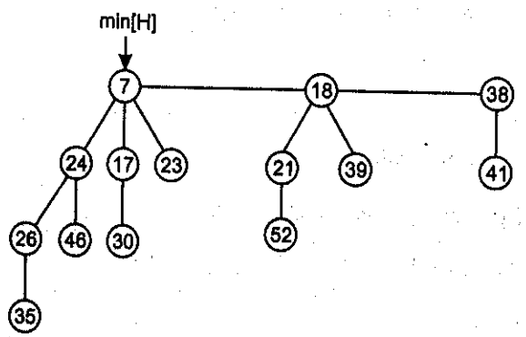
Now, w is the node 38

degree $[x] = 1$

and $A[1] = \text{NIL}$



So, final heap is



19.4.6 Decreasing a Key and Deleting a Node

46 is decreased to 15

$key[x]$

$key[x] = 15$

$y = 24$

FIB-HEAP-DECREASE-KEY (H, x, k)

```
if k > key[x]
    then error "new key is greater than current key"
key[x] ← k
y ← p[x]
if y ≠ NIL and key[x] < key[y]
    then CUT(H, x, y)
         CASCADING-CUT(H, y)
if key[x] < key[min[H]]
    then min[H] ← x
```

35 is decreased to 5

$5 > 35$

$key[x] = 5$

CUT (H, x, y)

```
remove x from the child list of y, decrementing degree[y]
add x to the root list of H
p[x] ← NIL
mark[x] ← FALSE
```

CASCADING-CUT (H, y)

```
z ← p[y]
if z ≠ NIL
    then if mark[y] = FALSE
             then mark[y] ← TRUE
             else CUT(H, y, z)
             CASCADING-CUT(H, z)
```

Marking

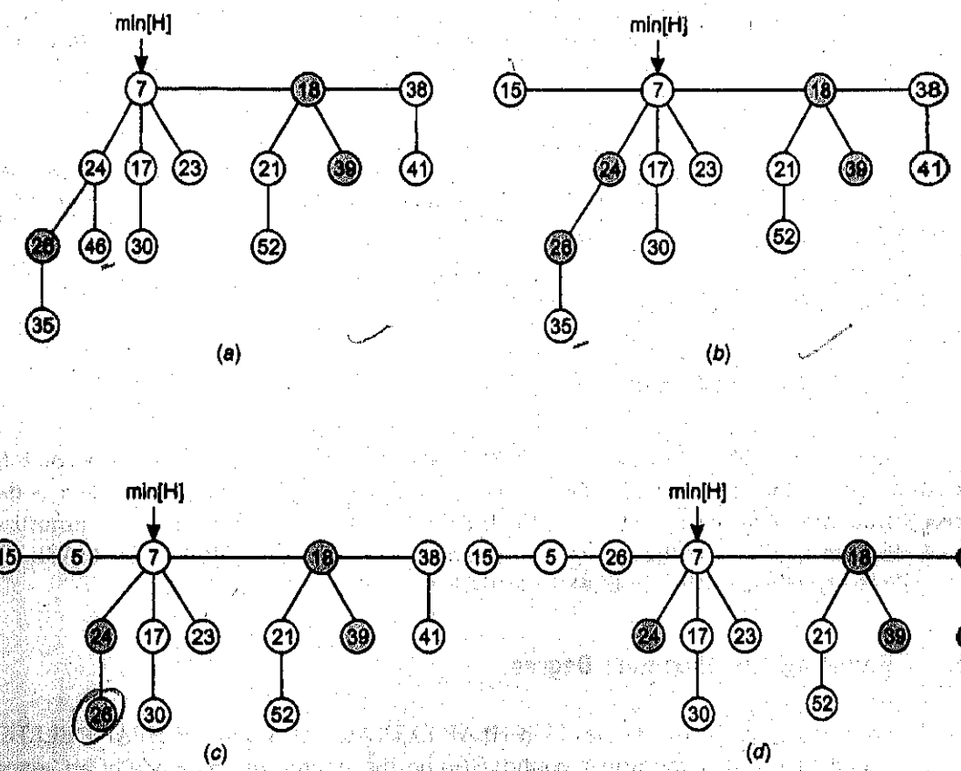


Figure 19.4

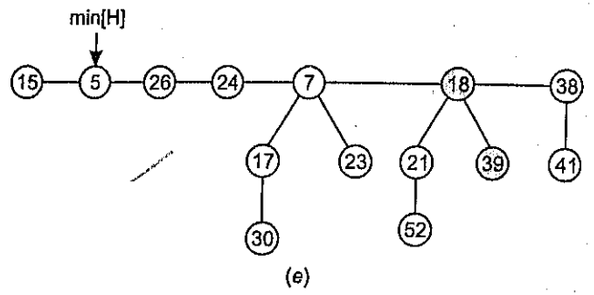


Figure. 19.4 Two calls of FIB-HEAP-DECREASE-KEY. (*a*) The initial Fibonacci heap. (*b*) The node with key 46 has its key decreased to 15. The node becomes a root, and its parent (with key 24), which had reviously been unmarked, becomes marked. (*c*)-(*e*) The node with key 35 has its key decreased to 5. In part (*c*), the node, now with key 5, becomes a root. Its parent, with key 26, is marked, so a cascading cut occurs. The node with key 26 is cut from its parent and made an unmarked root in (*d*). Another cascading cut occurs, since the node with key 24 is marked as well. This node is cut from its parent and made an unmarked root in part (*e*). The cascading cuts stop at this point, since the node with key 7 is a root. (Even if this node were not a root, the cascading cuts would stop, since it is unmarked.) The result of the FIB-HEAP-DECREASE-KEY operation is shown in part (*e*), with min[*H*] pointing to the new minimum node.

Deleting a Node

It is easy to delete a node from an n-node Fibonacci heap in $O(D(n))$ amortized time, as is done by the following pseudocode. We assume that there is no key value of $-\infty$ currently in the Fibonacci heap.

```
FIB-HEAP-DELETE (H, x)
1. FIB-HEAP-DECREASE-KEY(H, x, -∞)
2. FIB-HEAP-EXTRACT-MIN(H)
```

FIB-HEAP-DELETE is analogous to BINOMIAL-HEAP-DELETE. It makes x become the minimum node in the Fibonacci heap by giving it a uniquely small key of $-\infty$. Node x is then removed from the Fibonacci heap by the FIB-HEAP-EXTRACT-MIN procedure. The amortized time of FIB-HEAP-DELETE is the sum of the $O(1)$ amortized time of FIB-HEAP-DECREASE-KEY and the $O(D(n))$ amortized time of FIB-HEAP-EXTRACT-MIN.

19.5 Bounding the Maximum Degree

To prove that the amortized time of FIB-HEAP-EXTRACT-MIN and FIB-HEAP-DELETE is $O(\lg n)$, we must show that the upper bound $D(n)$ on the degree of any node of an n-node Fibonacci heap is $O(\lg n)$. The maximum degree in a fibonacci heap of n-nodes is indicated by $D(n)$.

Lemma 1

Let x be any node in a Fibonacci heap, and suppose that degree $[x]=k$. Let $y_1, y_2, \ldots, y_k$ denote the children of x in the order in which they were linked to x, from the earliest to the latest. Then, degree$[y_1] \geq 0$ and degree $[y_i] \geq i-2$ for $i=2,3,\ldots,k$.

Proof. It is obvious that for first child of x, y_1, degree $[y_1] \geq 0$.

For $i \geq 2$, we note that when y_1 was linked to x, all of $y_1, y_2, \ldots, y_{i-1}$ were children of x, so we must have had $degree[x] \geq i-1$. Node y_i is linked to x only if $degree[x] = degree[y_i]$, so we must have also had $degree[y_i] \geq i-1$ at that time. Since then, node y_i has lost at most one child, otherwise it would have been cut from x if it had lost two children. Thus $degree\,[y_i] \geq i-2$.

The above lemma is proved.

Recall that the nth Fibonacci sequence is defined by the recurrence

$$F_k = \begin{cases} 0 & \text{if } k=0 \\ 1 & \text{if } k=1 \\ F_{k-1}+F_{k-2} & \text{if } k \geq 2 \end{cases}$$

The following lemma gives another way to express F_k.

Lemma 2

For all integers $k \geq 0$,

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i$$

Proof. The proof is by induction on k. When $k=0$,

$$1+\sum_{i=0}^{k} F_i = 1+F_0$$

$$=1+0=1=F_2$$

We now assume the inductive hypothesis that $F_{k+1} = 1+\sum_{i=0}^{k-1} F_i$, and we have

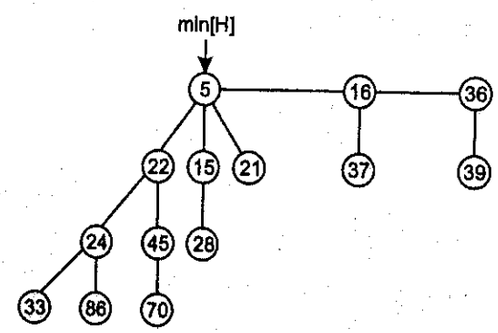
$$F_{k+2} = F_k + F_{k+1}$$

$$= F_k + \left[1+\sum_{i=0}^{k-1} F_i\right] = 1+\sum_{i=0}^{k} F_i$$

The above lemma is proved.

Exercise

1. Prove that, for a node x in a Fibonacci Heap, if c_i is the ith youngest child of x, then rank of c_i is at least $i-2$.
2. Prove that the rank of any node in a Fibonacci heap is $O(\lg n)$.
3. Suppose that a root x in a Fibonacci heap is marked. Explain how x came to be a market root. Argue that it doesn't matter to analysis that x is marked, even though it is not a root that was first linked to another node and then lost one child.
4. Show that if only the mergeable-heap operations are supported, the maximum degree $D(n)$ in an n-node Fibonacci heap is at most $\lfloor \lg n \rfloor$.
5. Suppose we generalize the cascading-cut rule to cut a node x from its parent as soon as it loses its kth child, for some integer constant k. For what values of k is $D(n) = O(\lg n)$?
6. Create a Fibonacci-heap for the following list $\langle 20,10,5,30,35,55,25,45,36,32 \rangle$.
   After creation, change the value of 45 to 23.
7. Decrease the key 45 to 34 in a Fibonacci heap shown in the Fig. Also find the resultant fibonacci heap after decreasing the key.



CHAPTER

20

Data Structures for Disjoint Sets

20.1 Introduction

An important application of the tree is the representation of sets, where "n" distinct elements are needed to be grouped into a number of disjoint sets.

A ***disjoint-set data structure*** maintains a collection $S = \{S_1, S_2, \ldots, S_k\}$ of disjoint dynamic sets. Each set is identified by a *representative*, which is some member of the set. If we have two sets S_x and S_y, $x \neq y$ such that $S_x = \langle 3,4,5,6,7 \rangle$ and $S_y = \langle 1,2 \rangle$ then these sets are called disjoint sets as there is no element which is common in both sets.

20.2 Disjoint-Set Operations

In the dynamic-set implementations, an object represents each element of a set. Letting x denote an object, we wish to support the following operations.

- **MAKE-SET(x)** creates a new set whose only member (and thus representative) is pointed to by x. Since the sets are disjoint, we require that x not already be in a set.

- UNION(x, y) unites the dynamic sets that contain x and y, say S_x and S_y into a new set that is the union of these two sets. The two sets are assumed to be disjoint prior to the operation. The representative of the resulting set is some member of $S_x \cup S_y$, although many implementations of UNION choose the representative of either S_x or S_y, as the new representative. Since we require the sets in the collection to be disjoint, we "destroy" sets S_x and S_y removing them from the collection S.
- FIND-SET(x) returns a pointer to the representative of the (unique) set containing x.

The running times of disjoint-set data structures in terms of two parameters : n, the number of MAKE-SET operations, and m, and the total number of MAKE-SET, UNION, and FIND-SET operations. Since the sets are disjoint, each UNION operation reduces the number of sets by one. After $n-1$ UNION operation, therefore, only one set remains. The number of UNION operations is thus at most $n-1$. Note also that since the MAKE-SET operations are included in the total number of operations m, we have $m \geq n$.

20.3 UNION-FIND Algorithm

A **union-find algorithm** is an algorithm that performs two useful operations on such a data structure :

- **Find.** Determine which set a particular element is in. Also useful for determining if two elements are in the same set.
- **Union.** Combine or merge two sets into a single set.

Because it supports these two operations, a disjoint-set data structure is sometimes called a ***merge-find set***.

20.4 Applications of Disjoint-set Data Structures

One of the many applications of disjoint-set data structures arises in determining the connected components of an undirected graph.

The procedure CONNECTED-COMPONENTS that follows uses the disjoint-set operations to compute the connected components of a graph. Once CONNECTED-COMPONENTS has been run as a preprocessing step, the procedure SAME-COMPONENT answers queries about whether two vertices are in the same connected component. The set of vertices of a graph G is denoted by $V[G]$, and the set of edges is denoted by $E[G]$.

```
CONNECTED-COMPONENTS(G)
for each vertex v ∈ V[G]
    do MAKE-SET(v)
for each edge (u,v) ∈ E[G]
    do if FIND-SET(u) ≠ FIND-SET(v)
        then UNION(u,v)
```



```
SAME-COMPONENT(u,v)
if FIND-SET(u) = FIND SET(v)
    then return TRUE
    else return FALSE
```

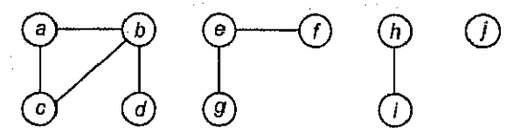


Figure 20.1 A graph with four connected components : $\{a,b,c,d\}$, $\{e,f,g\}$, $\{h,i\}$, and $\{j\}$. The collection of disjoint sets after each edge is processed.

Edge processed	*Collection of disjoint sets*									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e,g)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a,c)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h,i)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(a,b)	$\{a,b,c,d\}$				$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(e,f)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$
(b,c)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$

The procedure CONNECTED-COMPONENTS initially places each vertex v in its own set. Then, for each edge (u,v), it unites the sets containing u and v. After all the edges are processed, two vertices are in the same connected component if and only if the corresponding objects are in the same set. Thus, CONNECTED-COMPONENTS computes sets in such a way that the procedure SAME-COMPONENT can determine whether two vertices are in the same connected component.

20.5 Linked-list Representation of Disjoint Sets

A simple way to implement a disjoint-set data structure is to represent each set by a linked list. The first object in each linked list serves as its set's representative. Each object in the linked list

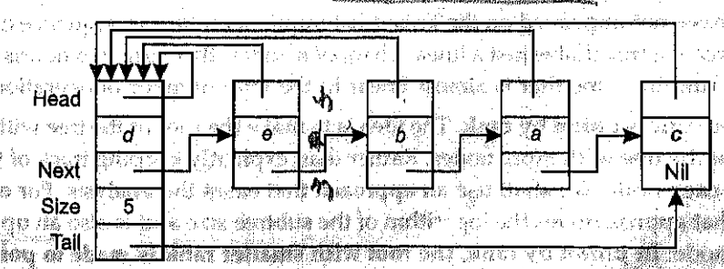


Figure 20.2

contains a set member, a pointer to the object containing the next set member, and a pointer back to the representative. Within each linked list, the objects may appear in any order (subject to our assumption that the first object in each list is the representative).

With this linked-list representation, both MAKE-SET and FIND-SET are easy, requiring $O(1)$ time. To carry out MAKE-SET(x), we create a new linked list whose only object is x. For FIND-SET (x), we just return the pointer from x back to the representative.

20.6 Disjoint-Set Forests

In a faster implementation of disjoint sets, we represent sets by rooted trees, with each node containing one member and each tree representing one set. In a ***disjoint-set forest***, each member points only to its parent. The root of each tree contains the representative and is its own parent.

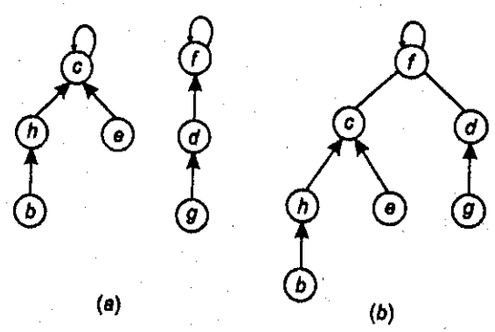


Figure 20.3 A disjoint-set forest. (*a*) Two trees representing the two sets of above Fig. The tree on the left represents the set $\{b,c,e,h\}$, with c as the representative, and the tree on the right represents the set $\{d,f,g\}$, with f as the representative. (*b*) The result of UNION(e,g).

We perform the three disjoint-set operations as follows. A MAKE-SET operation simply creates a tree with just one node. We perform a FIND-SET operation by chasing parent pointers until we find the root of the tree. The nodes visited on this path toward the root constitute the ***find path***. A UNION operation causes the root of one tree to point to the root of the other.

Heuristics to improve the running time

So far, we have not improved on the linked-list implementation. A sequence of $n-1$ UNION operations may create a tree that is just a linear chain of n nodes. By using two heuristics, however, we can achieve a running time that is almost linear in the total number of operations m.

The first heuristic is **union by rank**. The idea is to make the root of the tree with fewer nodes point to the root of the tree with more nodes. Rather than explicitly keeping track of the size of the subtree rooted at each node, we shall use an approach that eases the analysis. For each node, we maintain a ***rank*** that approximates the logarithm of the subtree size and is also an upper bound on the height of the node. **In union by rank, the root with smaller rank is made to point to the root with larger rank during a UNION operation.**

(log n)

The second heuristic, ***path compression***, is also quite simple and very effective. We use it during FIND-SET operations to make each node on the find path point directly to the root. Path compression does not change any ranks.

To implement a disjoint-set forest with the union-by-rank heuristic, we must keep track of ranks. **With each node x, we maintain the integer value *rank* [x], which is an upper bound on the height of x (the number of edges in the longest path between x and a descendant leaf).** When a singleton set is created by MAKE-SET, the initial rank of the single node in the corresponding tree is 0. Each FIND-SET operation leaves all ranks unchanged. When applying UNION to two trees, we make the root of higher rank the parent of the root of lower rank. In case of a tie, we arbitrarily choose one of the roots as the parent and increment its rank. We assign the parent of node x by $p[x]$. The LINK procedure, a subroutine called by UNION, takes pointers to two roots as inputs.

```
MAKE-SET(x)
    p[x] ← x
    rank[x] ← 0

UNION(x, y)
    LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
    if rank[x] > rank[y]
        then p[y] ← x
        else p[x] ← y
            if rank[x] = rank[y]
                then rank[y] ← rank[y] + 1
```

The FIND-SET procedure with path compression is as follows :

```
FIND-SET(x)
    if x ← p[x]
        then p[x] ← FIND-SET(p[x])
    return p[x]
```

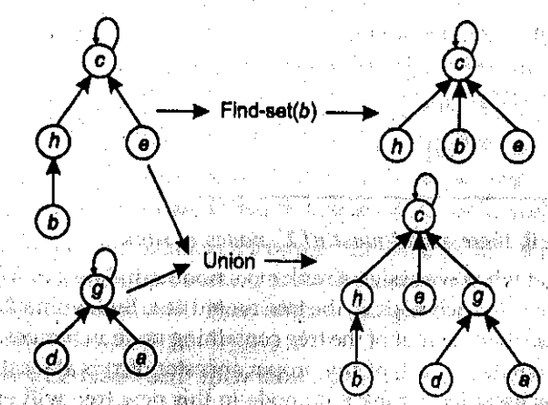


Figure 20.4

20.7 Properties of Ranks

Lemma 1

For all nodes x, we have $\text{rank}[x] \le \text{rank}[p[x]]$, if $x \ne p[x]$. The value of $\text{rank}[x]$ is initially 0 and increases through time until $x \ne p[x]$; from then on, $\text{rank}[x]$ does not change. The value of $\text{rank}[p[x]]$ is a monotonically increasing function of time.

We define $\text{size}(x)$ to be the number of nodes in the tree rooted at node x, including node x itself.

Lemma 2

For all tree roots x, $\text{size}(x) \ge 2^{\text{rank}[x]}$.

Proof. The proof is by induction on the number of LINK operations. Note that FIND-SET operations change neither the rank of a tree root nor the size of its tree.

Basis : The lemma is true before the first LINK, since ranks are initially 0 and each tree contains at least one node.

Inductive step : Assume that the lemma holds before performing the operation LINK(x, y). Let *rank* denote the rank just before the LINK, and let rank′ denote the rank just after the LINK. Define size and size′ similarly.

If $rank[x] \ne rank[y]$, assume without loss of generality that $rank[x] < rank[y]$. Node y is the root of the tree formed by the LINK operation, and

$$\begin{aligned}\text{size}'(y) &= \text{size}(x) + \text{size}(y)\\ &\ge 2^{\text{rank}[x]} + 2^{\text{rank}[y]}\\ &\ge 2^{\text{rank}[y]}\\ &= 2^{\text{rank}'[y]}\end{aligned}$$

No ranks or sizes change for any nodes other than y.

If $rank[x] = rank[y]$, node y is again the root of the new tree, and

$$\begin{aligned}\text{size}'(y) &= \text{size}(x) + \text{size}(y)\\ &\ge 2^{rank[x]} + 2^{rank[y]}\\ &= 2^{rank[y]+1}\\ &= 2^{rank'[y]}\end{aligned}$$

Lemma 3

For any integer $r \ge 0$, there are at most $n/2^r$ nodes of rank r.

Proof. Suppose that when we assign a rank r to a node x (in line 2 of MAKE-SET or in line 5 of LINK), we attach a label x to each node in the tree rooted at x. By Lemma 2, at least 2^r nodes are labeled each time. Suppose that the root of the tree containing node x changes. Lemma 1 assures us that the rank of the new root (or, in fact, of any proper ancestor of x) is at least $r+1$. Since we assign labels only when a root is assigned a rank r, no node in this new tree will ever again be labeled.

Thus, each node is labeled at most once, when its root is first assigned rank r. Since there are n nodes, there are at most n labeled nodes, with at least 2^r labels assigned for each node of rank r. If there were more than $n/2^r$ nodes of rank r, then more than $2^r \cdot (n/2^r) = n$ nodes would be labeled by a node of rank r, which is a contradiction. Therefore, at most $n/2^r$ nodes are ever assigned rank r.

Theorem

Every node has rank at most $\lfloor \lg n \rfloor$.

Proof. If we let $r > \lg n$, then there are at most $n/2^r < 1$ nodes of rank r. Since ranks are natural numbers, thus every node has rank at most $\lfloor \lg n \rfloor$ follows.

Example. *Give a non-recursive implementation of FIND-SET with path compression.*

Solution.

```
FIND-SET(x)
  y ← p[x]
  p[x] ← x
  while x ≠ y do
      z ← p[y]
      p[y] ← x
      x ← y
      y ← z
  end while
  root ← x
  y ← p[x]
  p[x] ← x
  while x ≠ y do
      z ← p[y]
      p[y] ← root
      x ← y
      y ← z
  end while
  return root
```

The first while loop traverses the path from x to the root of the tree while reversing all parent pointers along the way. The second while loop returns along this path and sets all parent pointers along the way to point to the root.

Example. *Give a sequence of m MAKE-SET, UNION and FIND-SET operations, n of which are MAKE-SET operations, that takes $\Omega(m \lg n)$ time when we use union by rank only.*

Solution. First perform the n (assume n is a power of 2) MAKE-SET operations on each of the elements $\{x_1, x_2, x_3, \ldots x_n\}$. Then perform a union on each pair (x_1, x_2), (x_3, x_4) and so on, yielding $n/2$ new sets. Continue the process until there is only a single set left. This uses a total of $n-1$ operations and produces a tree with depth $\Omega(\lg n)$.

Perform additional $k \ge n$ FIND-SET operations on the node of greatest depth.

Setting $m = n + n - 1 + k$ gives the desired running time of $\Omega(n + m \lg n) = \Omega(m \lg n)$.

Example. *Show that the running time of the disjoint-set forest with union by rank only is $O(m \lg n)$.*

Solution. We know that the rank of a node is at most the height of the subtree rooted at that node and every node has rank at most $\lfloor \lg n \rfloor$ and therefore the height h of any subtree is at most $\lfloor \lg n \rfloor$. Clearly, each call to FIND-SET (and thus UNION) takes at most $O(h) = O(\lg n)$ time.

So the resulting implementation runs in time $O(m \lg n)$.

Exercise

1. Show that after all edges are processed by CONNECTED-COMPONENTS, two vertices are in the same connected components if and only if they are in the same set.
2. How many times FIND-SET and UNION called during the execution of CONNECTED-COMPONENTS on an undirected graph $G = (V, E)$ with k connected components.
3. For each node x, how many bits are necessary to store size(x) ? How about rank[x] ?
4. Prove that the linked-list representation of disjoint sets and the weight-union heuristic, a sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, take $O(m + n \lg n)$ time.

CHAPTER

21

Dynamic Programming

21.1 Introduction

Dynamic programming is a problem solving technique that, like Divide and Conquer, solves problems by dividing them into subproblems. Dynamic programming is used when the subproblems are not independent, *e.g.* when they share the same subproblems. In this case, divide and conquer may do more work than necessary, because it solves the same subproblem multiple times.

Dynamic Programming solves each subproblem once and stores the result in a table so that it can be rapidly retrieved if needed again. It is often used in Optimization Problems ; A problem with many possible solutions for which we want to find an optimal (the best) solution. (There may be more than 1 optimal solution). Dynamic Programming is an approach developed to solve sequential, or multi-stage, decision problems ; hence, the name "dynamic" programming. But, as we shall see, this approach is equally applicable for decision problems where sequential property is induced solely for computational convenience.

Dynamic programming can be thought of as being the reverse of recursion. Recursion is a top-down mechanism – we take a problem, split it up, and solve the smaller problems that are created. Dynamic programming is a **bottom-up mechanism** – we solve all possible small problems and then combine them to obtain solutions for bigger problems.

Dynamic programming works when a problem has the following characteristics :

- **Optimal Substructure :** If an optimal solution contains optimal sub solutions, then a problem exhibits *optimal substructure.*
- **Overlapping subproblems :** When a recursive algorithm would visit the same subproblems repeatedly, then a problem has *overlapping subproblems.*

If a problem has *optimal substructure,* then we can recursively define an optimal solution. If a problem has *overlapping subproblems,* then we can improve on a recursive implementation by computing each subproblem only once.

If a problem doesn't have *optimal substructure,* there is no basis for defining a recursive algorithm to find the optimal solutions. If a problem doesn't have *overlapping subproblems,* we really don't have anything to gain by using dynamic programming.

If the space of subproblems is small enough (*i.e.* polynomial in the size of the input), dynamic programming can be much more efficient than recursion.

Applications

- Knapsack problem
- Mathematical optimization problems
- Shortest path problems
- Matrix chain multiplication
- Longest common sequence (LCS)
- Control (Cruise control, Robotics, Thermostates)
- Flight control (Balance factors that oppose one another, *e.g.,* maximize accuracy, minimize time).
- Time Sharing : Schedule user and jobs to maximize CPU usage
- Other types of scheduling.

Development of Dynamic Programming Algorithm : It can be broken into *four* steps :

1. Characterize the structure of an optimal solution
2. Recursively define the value of the optimal solution. Like divide and conquer, divide the problem into 2 or more optimal parts recursively. This helps to define what the solution will look like.
3. Compute the value of the optimal solution from the bottom up (starting with the smallest subproblem).
4. Construct the optimal solution for the entire problem from the computed values of smaller subproblems.

21.2 Common Characteristics

There are a number of characteristics that are to all dynamic programming problems. These are :

1. The problem can be divided into stages with a decision required at each stage. *For example,* in the shortest path problem, they were defined by the structure of the graph. The decision was where to go next.
2. Each stage has a number of states associated with it. The states for the shortest path problem was the node reached.
3. The decision at one stage transforms one state into a state in the next stage. The decision of where to go next defined where we arrived in the next stage.
4. Given the current state, the optimal decision for each of the remaining states does not depend on the previous states or decisions. In the shortest path problem, it was not necessary to know how we got to a node, only that we did.
5. There exists a recursive relationship that identifies the optimal decision for stage j, given that stage $j+1$ has already been solved.
6. The final stage must be solvable by itself.

The last *two* properties are tied up in the recursive relationships given above.

The big skill in dynamic programming, and the art involved, is to take a problem and determine stages and states so that all of the above hold. If we can, then the recursive relationship makes finding the values relatively easy. Because of the difficulty in identifying stages and states, we will do a fair number of examples.

21.3 Matrix Multiplication

To multiply two matrices, multiply the rows in the first matrix by the columns of the second matrix. In general :

If $A = \lfloor a_{ij} \rfloor$ is a $p \times q$ matrix, $B = \lfloor b_{ij} \rfloor$ is a $q \times r$ matrix and $C = \lfloor c_{ij} \rfloor$ is a $p \times r$ matrix

then $$AB = C \quad \text{if} \quad c_{ij} = \sum_{k=1}^{q} a_{ik} b_{kj}$$

Given following matrices $\{A_1, A_2, A_3, \ldots, A_n\}$ and we have to perform the matrix multiplication, which can be accomplished by a series of matrix multiplications

$$A_1 \times A_2 \times A_3 \times \ldots \times A_n$$

Matrix multiplication operation is **associative** in nature rather commutative. By this, we mean that we have to follow the above matrix order for multiplication but we are free to **parenthesize** the above multiplication depending upon our need.

In general, for $1 \le i \le p$ and $1 \le j \le r$

$$C[i,j] = \sum_{k=1}^{q} A[i,k]\, B[k,j]$$

It can be observed that the total entries in matrix 'C' is 'pr' as the matrix is of dimension $p \times r$ Also each entry takes $O(q)$ times to compute, thus the total time to compute all possible entries for the matrix 'C' which is the multiplication of 'A' and 'B' is proportional to the product of the dimensions $p\ q\ r$.

It is also to be noticed that we can save the number of operations by reordering the parenthesis.

Example : Consider three matrices of

$$A_{5\times4},\quad B_{4\times3},\quad C_{3\times5}$$

then the possible multiplication

$$[(AB)C] = (5\times4\times3)+(5\times3\times5) = 135$$

$$[A(BC)] = (5\times4\times5)+(4\times3\times5) = 160$$

So, we have seen that by changing the evaluation sequence cost of operations also changes.

Number of ways for parenthesizing the matrices

If we have two or three matrices to which we have to compute the multiplication then the number of possible ways for parenthesizing these matrices is not more than 3-4. In general if we have 'n' matrices then there exist $(n-1)$ places where we can break the sequence of matrices with the outermost pair of parentheses namely just after the 1st matrix, just after the 2nd matrix etc. and just after the $(n-1)$th matrix. It can be observed that after splitting the kth matrix, we are left with two parenthesized sequence of matrices : one consists 'k' matrices and other consists '$n-k$' matrices. It can be seen that for parenthesizing the choicing are independent, thus if there are X ways for parenthesizing the left sequence and Y ways for parenthesizing the right sequence. The total will be $X \cdot Y$.

$$p(n) = \begin{cases} 1 & n=1 \\ \sum_{k=1}^{n-1} p(k)\, p(n-k) & n=2 \end{cases}$$

also $p(n) = c(n-1)$ where $c(n)$ is the nth Catalon number

$$c(n) = \frac{1}{n+1}\binom{2n}{n}$$

on applying Stirling's formula we have

$$c(n) \in \Omega\left(\frac{4^n}{n^{1.5}}\right),$$

which shows that 4^n grows faster, as it is an exponential function, then $n^{1.5}$.

Dynamic programming approach

Let $A_{i..j}$ indicates the multiplication of matrices through i to j. It can be seen that the dimension of $A_{i..j}$ is $P_{i-1} \times P_j$. Our task is to break the problem into several simpler and smaller subproblems of similar structure. We consider the highest level of parenthesizing where two matrix multiplications are considered. If we consider 'k' where $1 \le k \le n-1$, then

$$A_{1..n} = A_{1..k} \times A_{k+1..n}$$

Thus we are left with *two* questions :

- How to split the sequence of matrices ?
- How to parenthesize the subsequence $A_{1..k}$ and $A_{k+1..n}$?

One possible answer to the first question for finding the best value of 'k' is to check all possible choices of 'k' and consider the best among them. But it can be observed that checking all possibilities will lead to an exponential number of total possibilities. It can also be noticed that there exists only $O(n^2)$ different sequence of matrices, in this way we do not reach to the exponential growth.

The structure of an optimal parenthesization

Let us adopt the notation $A_{i..j}$ for the matrix that results from evaluating the product $A_i\, A_{i+1} \ldots A_j$. An optimal parenthesization of the product $A_1\, A_2 \ldots A_n$. Splits the product between A_k and A_{k+1} for some integer k in the range $1 \le k \le n$ *i.e.* for some value of k, we first compute the matrices $A_{1..k}$ and $A_{k+1..n}$ and then multiply them together to produce the final product $A_{1..n}$. The cost of this optimal parenthesization is thus the cost of computing the matrix $A_{1..k}$ + the cost of computing $A_{k+1..n}$ + cost of multiplying them together.

Let $m[i,j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$, the cost of a cheapest way to compute $A_{1..n}$ would thus be $m[1..n]$.

We can define $m[i,j]$ recursively as follows :

If $i=j$ the chain consists of just one matrix $A_{i..i} = A_i$ so no scalar multiplications are necessary to compute the product . Thus $m[i,j]=0$ for $i=1,2,3,\ldots n$.

To compute $m[i,j]$, when $i<j$. Let us assume that the optimal parenthesization splits the product $A_i\, A_{i+1} \ldots A_j$ between A_k and A_{k+1} where $i \le k \le j$. Then $m[i,j]$ is equal to the minimum cost for computing the subproducts $A_{i..k}$ and $A_{k+1..j}$ + cost of multiplying them together. Since computing the matrix product $A_{i..k}$ and $A_{k+1..j}$ takes $p_{i-1}\, p_k\, p_j$ scalar multiplications, we obtain

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}\, p_k\, p_j$$

There are only $(j-1)$ possible values for 'k' namely $k = i, i+1, \ldots j-1$. Since the optimal parenthesization must use one of these values for 'k', we need only check them all to find the best . So the minimum cost of parenthesizing the product $A_i\, A_{i+1} \ldots A_j$ becomes

$$m[i,j] = \begin{cases} 0 & \text{if } i=j \\ \min_{i \le k < j} \{ m[i,k] + m[k+1,j] + p_{i-1}\, p_k\, p_j \} & \text{if } i<j \end{cases}$$

To construct an optimal solution, let us define $s[i,j]$ to be the value of 'k' at which we can split the product $A_i\, A_{i+1} \ldots A_j$ to obtain an optimal parenthesization *i.e.* $s[i,j]=k$ such that

$$m[i,j]=m[i,k]+m[k+1,\ j]+p_{i-1}\, p_k\, p_j$$

Computing optimal costs

The following pseudocode assumes that matrix A_i has dimensions $p_{i-1} \times p_i$ for $i=1,2,3,\ldots n$. The input is a sequence $\langle p_0\ p_1 \ldots p_n \rangle$ where length$[p]=n+1$. The procedure uses an auxiliary table $m[1..n,\ 1..n]$, for storing the $m[i,j]$ costs and an auxiliary table $s[1..n,\ 1..n]$ that records which index of k achieved the optimal cost in computing $m[i,j]$.

The algorithm first computes $m[i,j] \leftarrow 0$ for $i=1,2,3..n$, the minimum costs for chains of length 1.

```
MATRIX-CHAIN-ORDER(p)
n ← length[p] − 1
for i ← 1 to n
    do m[i,i] ← 0
for l ← 2 to n          ▷ l is the chain length
    do for i ← 1 to n − l + 1
        do j ← i + l − 1
            m[i, j] ← ∞
            for k ← i to j − 1
                do q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
                    if q < m[i, j]
                        then m[i, j] ← q
                            s[i, j] ← k
return m and s
```

It then uses recurrence to compute $m[i,i+1]$ for $i=1,2,\ldots n-1$ (the minimum cost of length 2) during the first execution of the loop and in the second time, it computes $m[i,i+2]\ i=1,2,\ldots n-2$ (the minimum cost of length 3) and so on.

Since we have defined $m[i,j]$ only for $i<j$, only the portion of the table m strictly above the main diagonal is used. MATRIX-CHAIN-ORDER computes the rows from bottom to top and from left to right within each row. An entry $m[i,j]$ is computed using the products $p_{i-1}\ p_k\ p_j$ for $k=i,\ i+1,\ldots j-1$.

A simple inspection of the nested loop structure of MATRIX-CHAIN-ORDER yields a running time of $O(n^3)$ for the algorithm.

Now we want to know the parenthesization that gives this minimum. To find it, we use the array, s. $s[i,j]$ records k for the optimal split in each sub array : $A_{i..j}$ becomes $(A_{i..k})$ and $(A_{k+1}..j)$. To split $A_{1..n}$, use $s[1,n]$ value for k : $A_{1..n} = A_{1..s[1,n]}\ A_{s[1,n]+1..n}$. We recursively split each of the sub sequences using the array s to find the optimal k at which to split each one.

The following recursive procedure computes the matrix-chain product $A_{i..j}$ given the matrices $A=\langle A_1\ A_2 \ldots A_n \rangle$, the s table computed by MATRIX-CHAIN-ORDER, and the indices i and j. The initial call is MATRIX-CHAIN-MULTIPLY $(A,s,1,n)$.

```
MATRIX-CHAIN-MULTIPLY (A, s, i, j)
if j > i
then X ← MATRIX-CHAIN-MULTIPLY (A, s, i, s[i, j])
     Y ← MATRIX-CHAIN-MULTIPLY (A, s, s[i, j] + 1, j)
     return MATRIX-MULTIPLY (X, Y)
else return A_i
```

Example of Finding the Multiplication Sequence

Consider $n=6$. Assume that the array $s[1..6,1..6]$ has been computed. The multiplication sequence is recovered as follows.

$s[1,6]=3 \quad (A_1\ A_2\ A_3)(A_4\ A_5\ A_6)$

$s[1,3]=1 \quad (A_1(A_2\ A_3))$

$s[4,6]=5 \quad ((A_4\ A_5)A_6)$

Hence the final multiplication sequence is

$$(A_1(A_2 A_3))((A_4 A_5)A_6)$$

Example. *We are given the sequence {4, 10, 3, 12, 20, 7}. The matrices have sizes* $4\times10, 10\times3, 3\times12, 12\times20, 20\times7$. *We need to compute* $M[i,j],\ 0\le i,\ j\le5$. *We know* $M[i,i]=0$ *for all i.*

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

We proceed, working away from the diagonal. We compute the optimal solutions for products of 2 matrices.

1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

{4, 10, 3, 12, 20, 7}

Now products of 3 matrices

$$M[1,3]=\min\begin{cases} M[1,2]+M[3,3]+p_0p_2p_3 = 120+0+4\cdot3\cdot12 = 264 \\ M[1,1]+M[2,3]+p_0p_1p_3 = 0+360+4\cdot10\cdot12 = 840 \end{cases} = \mathbf{264}$$

$$M[2,4] = \min \begin{cases} M[2,3] + M[4,4] + p_1 p_3 p_4 = 360 + 0 + 10 \cdot 12 \cdot 20 = 2760 \\ M[2,2] + M[3,4] + p_1 p_2 p_4 = 0 + 720 + 10 \cdot 3 \cdot 20 = 1320 \end{cases} = \mathbf{1320}$$

$$M[3,5] = \min \begin{cases} M[3,4] + M[5,5] + p_2 p_4 p_5 = 720 + 0 + 3 \cdot 20 \cdot 7 = 1140 \\ M[3,3] + M[4,5] + p_2 p_3 p_5 = 0 + 1680 + 3 \cdot 12 \cdot 7 = 1932 \end{cases} = \mathbf{1140}$$

k = 3
k = 4

1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

⇒

1	2	3	4	5	
0	120	264			1
	0	360	1320		2
		0	720	1140	3
			0	1680	4
				0	5

Now products of 4 matrices

$$M[1,4] = \min \begin{cases} M[1,3] + M[4,4] + p_0 p_3 p_4 = 264 + 0 + 4 \cdot 12 \cdot 20 = 1224 \\ M[1,2] + M[3,4] + p_0 p_2 p_4 = 120 + 720 + 4 \cdot 3 \cdot 20 = 1080 \\ M[1,1] + M[2,4] + p_0 p_1 p_4 = 0 + 1320 + 4 \cdot 10 \cdot 20 = 2120 \end{cases} = \mathbf{1080}$$

k = 1
k = 2
k = 3

$$M[2,5] = \min \begin{cases} M[2,4] + M[5,5] + p_1 p_4 p_5 = 1320 + 0 + 10 \cdot 20 \cdot 7 = 2720 \\ M[2,3] + M[4,5] + p_1 p_3 p_5 = 360 + 1680 + 10 \cdot 12 \cdot 7 = 2880 \\ M[2,2] + M[3,5] + p_1 p_2 p_5 = 0 + 1140 + 10 \cdot 3 \cdot 7 = 1350 \end{cases} = \mathbf{1350}$$

1	2	3	4	5	
0	120	264			1
	0	360	1320		2
		0	720	1140	3
			0	1680	4
				0	5

⇒

1	2	3	4	5	
0	120	264	1080		1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

Now products of 5 matrices

$$M[1,5] = \min \begin{cases} M[1,4] + M[5,5] + p_0 p_4 p_5 = 1080 + 0 + 4 \cdot 20 \cdot 7 = 1544 \\ M[1,3] + M[4,5] + p_0 p_3 p_5 = 264 + 1680 + 4 \cdot 12 \cdot 7 = 2016 \\ M[1,2] + M[3,5] + p_0 p_2 p_5 = 120 + 1140 + 4 \cdot 3 \cdot 7 = 1344 \\ M[1,1] + M[2,5] + p_0 p_1 p_5 = 0 + 1350 + 4 \cdot 10 \cdot 7 = 1630 \end{cases} = \mathbf{1344}$$

k = 1
k = 2
k = 3
k = 4

1	2	3	4	5	
0	120	264	1080		1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

⇒

1	2	3	4	5	
0	120	264	1080	1344	1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

To print the optimal parenthesization, we use the PRINT-OPTIMAL-PARENS procedure.

PRINT-OPTIMAL-PARENS (s, i, j)

```
if i = j
    then print "A"
    else print "("
        PRINT-OPTIMAL-PARENS (s, i, s[i, j])
        PRINT-OPTIMAL-PARENS (s, s[i, j] + 1, j)
        print ")"
```

Now for optimal parenthesization , Each time we find the optimal value for M[i,j] we also store the value of k that we used. If we did this for the example, we would get

1	2	3	4	5	
0	120/1	264/2	1080/2	1344/2	1
	0	360/2	1320/2	1350/2	2
		0	720/3	1140/4	3
			0	1680/4	4
				0	5

The k value for the solution is 2, so we have $((A_1\ A_2)(A_3\ A_4\ A_5))$. The first half is done. The optimal solution for the second half comes from entry $M[3,5]$.The value of k here is 4, so now we have $((A_1\ A_2)((A_3\ A_4)\ A_5))$.Thus the optimal solution is to parenthesize $((A_1\ A_2)((A_3\ A_4)\ A_5))$.

21.4 Memoization

So far we have talked about implementing dynamic programming in a bottom-up fashion.

Dynamic programming can also be implemented using **memoization**. With memoization, we implement the algorithm recursively, but we keep track of all of the sub solutions. If we encounter a subproblem that we have seen, we look up the solution. If we encounter a subproblem that we have not seen, we compute it, and add it to the list of sub solutions we have seen. Each subsequent time that the subproblem is encountered, the value stored in the table is simply looked up and returned.

MEMOIZED-MATRIX-CHAIN (p)

```
n ← length[p] − 1
for i ← 1 to n do
    for j ← 1 to n do
        m[i, j] ← infinity (∞)
return LOOKUP-CHAIN(p, 1, n)
```

LOOKUP-CHAIN (p, i, j)

```
if m[i, j] < infinity then
    return m[i, j]
if i = j then
    m[i, j] ← 0
else for k ← i to j − 1 do
    q ← LOOKUP-CHAIN(p, i, k) + LOOKUP-CHAIN (p, k + 1, j) + p_{i−1} p_k p_j
    if q < m[i, j] then
        m[i, j] ← q
return m[i, j]
```

Memoization offers the efficiency of dynamic programming. It maintains the **top-down recursive strategy.**

Comparison with dynamic programming

Dynamic programming algorithm usually outperforms a top-down memoized algorithm by constant factor, because there is no over-head for recursion and fewer overheads for maintaining the table. In situations where not every subproblem is computed, memoization only solves those that are needed but dynamic programming solves all the subproblems.

In summary, the matrix-chain multiplication problem can be solved in $O(n^3)$ time by either a top-down, memorized algorithm or a bottom-up dynamic-programming algorithm.

21.5 Longest common subsequence (LCS)

A subsequence of a given sequence is just the given sequence with some elements left out. Given two sequences X and Y, we say that a sequence Z is a common sequence of X and Y if Z is a subsequence of both X and Y.

In the longest common subsequence problem, we are given two sequences $X = \langle x_1\, x_2 \ldots x_m \rangle$ and $Y = \langle y_1\, y_2 \ldots y_n \rangle$ and wish to find a maximum length common subsequence of X and Y. LCS problem can be solved using dynamic programming.

Characterizing a longest common subsequence

A brute-force approach to solving the LCS problem is to specify all subsequences of X and check each subsequence to see if it is also a subsequence of Y, keeping track of the longest subsequence found. Each subsequence of X corresponds to a subset of the indices $\{1,2,\ldots m\}$ of X, there are 2^m subsequences of X, so this approach requires exponential time.

The LCS problem has an optimal-substructure property. Given a sequence $X = \langle x_1\, x_2 \ldots x_m \rangle$, we define the ith prefix of X, for $i = 0,1,2\ldots m$ as $X_i = \langle x_1\, x_2 \ldots x_i \rangle$. For example, if $X = \langle A, B, C, B, C, A, B, C \rangle$ then $X_4 = \langle A, B, C, B \rangle$.

Theorem (Optimal substructure of an LCS)

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be the sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

The above theorem implies that there are either one or two subproblems to examine when finding an LCS of $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$. If $x_m = y_n$ we must find an LCS of X_{m-1} and Y_{n-1}. If $x_m \neq y_n$, then we must solve two subproblems finding as LCS of X_{m-1} and Y and finding an LCS of X and Y_{n-1}. Whenever of these LCS's is longer is an LCS of X and Y. But each of these subproblems has the subproblem of finding the LCS of X_{m-1} and Y_{n-1}.

Let us define $c[i, j]$ to be the length of an LCS of the sequence X_i and Y_j. If either $i = 0$ or $j = 0$, one of the sequences has length 0, so the LCS has length 0. The optimal substructure of the LCS problem gives the recurrence formula

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1,j-1]+1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(c[i,j-1], c[i-1,j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Now, we will develop an exponential recursive algorithm **LCS-LENGTH** to compute an LCS of two strings.

Procedure LCS-LENGTH takes two sequences X and Y as inputs. It stores the $c[i, j]$ values in a table $c[0\ldots m, 0\ldots n]$ whose entries are computed in row-major order, that is first row of c is filled from left to right, then the second row and so on. It also maintains the table $b[1..m, 1..n]$ to simplify construction of an optimal solution. The procedure returns the b and c tables. $c[m, n]$ contains the length of LCS of X and Y.

LCS-LENGTH (X, Y)

```
m ← lenght[X]
n ← length[Y]
for i ← 1 to m
    do c[i,0] ← 0
for j ← 0 to m
    do c[0, j] ← 0
for i ← 1 to m
    do for j ← 1 to n
        do if x_i = y_j
            then c[i, j] ← c[i − 1, j − 1] + 1
                b[i, j] ← " ↖"
            else if c[i − 1, j] ≥ c[i, j − 1]
                then c[i, j] ← c[i − 1, j]
                    b[i, j] ← "↑"
                else c[i, j] ← c[i, j − 1]
                    b[i, j] ← "←"
return c and b.
```

Example. *Given two sequences $X[1..m]$ and $Y[1..n]$. Find the longest subsequence common to both. Note : not substring, subsequence.*

So if $x : A\ B\ C\ B\ D\ A\ B$

$y : B\ D\ C\ A\ B\ A$

the longest subsequence turns out to be $B\ C\ B\ A$. There could be other subsequences of the longest length, the algorithm will get one of them.

Here $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$

$m = \text{length}[X]$ and $n = \text{length}[Y]$

$m = 7$ and $n = 6$

Now, filling in the $m \times n$ table with the values of $c[i, j]$ and the appropriate arrow for the value of $b[i, j]$. Initialize top row and left column to 0 which takes $\theta(m+n)$ time.

Work across the rows starting at the top. Any time $x_i = y_j$ fill in the diagonal neighbour + 1 and mark the box with the wingding'↖' otherwise fill in the box with the max of the box above and the box to the left. That is, the entry of $c[i, j]$ depends only on whether $x_i = y_j$ and the values in entries $c[i-1, j]$, $c[i, j-1]$ and $c[i-1, j-1]$ which are computed before $c[i, j]$. The max length is the lower right hand corner. In $c[i-1, j]$ and $c[i, j-1]$ entries if $c[i-1, j] \geq c[i, j-1]$ then $b[i, j]$ entry is '↑' otherwise "←".

Then, to reconstruct the elements of an LCS, follow the $b[i, j]$ arrows from the lower right-hand corner. Each '↖' on the path corresponds to an entry for which $x_i = y_j$ is a member of LCS.

Here

$x_1 = x[1] = A$	$y_1 = y[1] = B$
$x_2 = B$	$y_2 = D$
$x_3 = C$	$y_3 = C$
$x_4 = B$	$y_4 = A$
$x_5 = D$	$y_5 = B$
$x_6 = A$	$y_6 = A$
$x_7 = B$	

Now, fill the values of $c[i, j]$ in $m \times n$ table.

Initially, for $i = 1$ to 7 $c[i, 0] = 0$

for $j = 0$ to 6 $c[0, j] = 0$

That is

i		j	0	1	2	3	4	5	6
			y_j	B	D	C	A	B	A
0	x_i		0	0	0	0	0	0	0
1	A		0						
2	B		0						
3	C		0						
4	B		0						
5	D		0						
6	A		0						
7	B		0						

Now, for $i = 1$ and $j = 1$

x_1 and y_1 we get $x_1 \neq y_1$ *i.e.,* $A \neq B$

and $c[i-1, j] = c[0,1] = 0$

$c[i, j-1] = c[1,0] = 0$

That is, $c[i-1, j] = c[i, j-1]$ so $c[1,1] = 0$ and $b[1,1] =$ '↑'

Now, $i = 1$ and $j = 2$

x_1 and y_2 we get $x_1 \neq y_2$ *i.e.,* $A \neq D$.

$c[i-1, j] = c[0,2] = 0$

$c[i, j-1] = c[1,1] = 0$

That is, $c[i-1, j] = c[i, j-1]$ and $c[1,2] = 0$ $b[1,2] =$ '↑'

Now $i = 1$ and $j = 3$

x_1 and y_3 we get $A \neq C$ *i.e.,* $x_1 \neq y_3$.

$c[i-1, j] = c[0,3] = 0$

$c[i, j-1] = c[1,2] = 0$

So, $c[1,3] = 0$ $b[1,3] =$ "↑"

Now $i = 1$ and $j = 4$

x_1 and y_4 we get $x_1 = y_4$ *i.e.,* $A = A$

$c[1,4] = c[1-1, 4-1] + 1$

$= c[0,3] + 1$

$= 0 + 1 = 1$

$c[1,4] = 1$

$b[1,4] =$ '↖'

Now $i = 1$ and $j = 5$

x_1 and y_5 we get $x_1 \neq y_5$

$c[i-1, j] = c[0,5] = 0$

$c[i, j-1] = c[1,4] = 1$

Thus, $c[i, j-1] > c[i-1, j]$ *i.e.,* $c[1,5] = c[i, j-1] = 1$. So, $b[1,5] =$ '←'

Now for $i = 1$ and $j = 6$

x_1 and y_6 we get $x_1 = y_6$

$c[1,6] = c[1-1, 6-1] + 1$

$= c[0,5] + 1 = 0 + 1 = 1$

$c[1,6] = 1$

$b[1,6] =$ '↖'

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

Now $i=2$ and $j=1$

we get x_2 and y_1 $B=B$ *i.e.*, $x_2=y_1$

$$c[2,1]=c[2-1,1-1]+1$$
$$=c[1,0]+1$$
$$=0+1=1$$
$$c[2,1]=1 \text{ and } b[2,1]=\text{'}\nwarrow\text{'}$$

Similarly, we fill the all values of $c[i,j]$ and we get

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	←1	↖ 1
2	B	0	↖ 1	←1	←1	↑ 1	↖ 2	←2
3	C	0	↑ 1	↑ 1	↖ 2	←2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	←3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Figure 21.1

The entry 4 in $c[7,6]$ is the length of the LCS $\langle B,C,B,A\rangle$ for X and Y.

To find, the elements of an LCS, follow the $b[i,j]$ arrows from the lower right-hand corner. In Fig. 21.1 the path is shaded.

Each '↖' on the path corresponds to an entry for which $x_i=y_j$ is a member of an LCS.

Constructing an LCS

The b table returned by LCS-LENGTH can be used to quickly construct an LCS of $X=\langle x_1\, x_2 \ldots x_m\rangle$ and $Y=\langle y_1\, y_2 \ldots y_n\rangle$. The following recursive procedure prints out an LCS of X and Y in proper, forward order.

```
PRINT-LCS (b,X,i,j)
if i = 0 or j = 0
    then return
if b[i, j] = "↖"
    then PRINT-LCS (b,X,i−1, j−1)
        print x_i
elseif b[i, j] = "↑"
    then PRINT-LCS (b,X,i−1, j)
else PRINT-LCS (b,X,i, j−1)
```

For the b table in the Fig 21.1, this procedure prints "*BCBA*". The procedure takes time $O(m+n)$, since atleast one of i and j is decremented in each stage of the recursion.

Example. *What are the basic four steps of dynamic programming ?*

Solution. The development of a DP algorithm consists of the following *four* steps :

1. Characterize the structure of an optimal solution
2. Define the value of an optimal solution recursively
3. Compute the value of an optimal solution in a bottom-up fashion
4. Construct an optimal solution from the computed information

Example. *Determine the LCS of* $\langle 1,0,0,1,0,1,0,1\rangle$ *and* $\langle 0,1,0,1,1,0,1,1,0\rangle$.

Solution. Let $X=\langle 1,0,0,1,0,1,0,1\rangle$ and $Y=\langle 0,1,0,1,1,0,1,1,0\rangle$. We know

$$c[i,j]=\begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ c[i-1,j-1]+1 & \text{if } i,j>0 \text{ and } x_i=y_j \\ \max(c[i,j-1],c[i-1,j]) & \text{if } i,j>0 \text{ and } x_i\neq y_j \end{cases}$$

We are looking for $c[8,9]$. The following table is built.

$x = \langle 1,0,0,1,0,1,0,1 \rangle \qquad y = \langle 0,1,0,1,1,0,1,1,0 \rangle$

		0	1	2	3	4	5	6	7	8	9
		y_j	0	1	0	1	1	0	1	1	0
0	x_i	0	0	0	0	0	0	0	0	0	0
1	1	0	↑ 0	↖ 1	← 1	↖ 1	↖ 1	← 1	↖ 0	↖ 1	← 1
2	0	0	↖ 1	↑ 1	↖ 2	← 2	← 2	↖ 2	← 2	← 2	↖ 2
3	0	0	↖ 1	↑ 1	↖ 2	↑ 2	↑ 2	↖ 3	← 3	← 3	↖ 3
4	1	0	↑ 1	↖ 2	↑ 2	↖ 3	↖ 3	↑ 3	↖ 4	↖ 4	← 4
5	0	0	↖ 1	↑ 2	↖ 3	↑ 3	↑ 3	↖ 4	↑ 4	↑ 4	↖ 5
6	1	0	↑ 1	↖ 2	↑ 3	↖ 4	↖ 4	↑ 4	↖ 5	↖ 5	↑ 5
7	0	0	↖ 1	↑ 2	↖ 3	↑ 4	↑ 4	↖ 5	↑ 5	↑ 5	↖ 6
8	1	0	↑ 1	↖ 2	↑ 3	↖ 4	↖ 5	↑ 5	↖ 6	↖ 6	↑ 6

From the table, we can deduce that LCS = 6. There are several such sequences, for instance $\langle 1,0,0,1,1,0 \rangle$, $\langle 0,1,0,1,0,1 \rangle$ and $\langle 0,0,1,1,0,1 \rangle$.

Example. *Show how to reconstruct an LCS from the completed c table and the original sequence* $X = \langle x_1 x_2 \ldots x_m \rangle$ *and* $Y = \langle y_1 y_2 \ldots y_n \rangle$ *in* $O(m+n)$ *time, without using the b table.*

Solution. It is easy to see that it takes $O(m+n)$ time if the b table is used. We now eliminate the b table. Each $c[i,j]$ entry depends on only three other entries : $[i-1,j]$, $c[i,j-1]$ and $c[i-1,j-1]$. Given the value of $c[i,j]$, we can determine in $O(1)$ time which of these three values was used to compute $c[i;j]$, without inspecting table b. Thus, we can reconstruct LCS in $O(m+n)$ time using the c table, X, and Y.

```
PRINT-LCS(C, X, Y, i, j)
if i = 0 or j = 0
    then return
if x_i = y_j and c[i, j] = 1 + c[i − 1, j − 1]          ▸ ↖
    then PRINT-LCS(c, X, Y, i − 1, j − 1)
    print x_i
else if x_i ≠ y_j and c[i, j] = c[i, j − 1]             ▸ ↑
    then PRINT-LCS(c, X, Y, i, j − 1)
else if x_i ≠ y_j and c[i, j] = c[i − 1, j]             ▸ ←
    then PRINT-LCS(c, X, Y, i − 1, j)
else
    error
```

Initial call is PRINT-LCS(c, X, Y, m, n)

Example. *Give an* $O(n^2)$ *time algorithms to find a longest decreasing subsequence of a sequence of n numbers.*

Solution. Let $a_1, a_2, a_3, \ldots, a_n$ be the n numbers. We follow the four steps of devising a dynamic programming algorithm in the following.

Step 1. *Characterize the structure of an optimal solution*

Let $X_i = a_1, a_2, \ldots, a_i$ the ith prefix of the sequence and $l(i)$ be the longest decreasing subsequence of X_i $1 \le i \le n$. Then, the problem is to find $\max_{1 \le i \le n} \{l(i)\}$.

Step 2. *Define the value of an optimal solution recursively*

$$l(1) = 1$$

$$l(i) = \max_{1 \le j \le n} \{l(j) + 1 : a_j \ge a_i\}$$

Step 3. Compute l (index) $= \max_{1 \le i \le n} \{l(i)\}$

Algorithm Longest-decreasing-sequence (a, n)

```
l[1] ← 1 ; S[1] ← 0
for i ← 2 to n do
    l_max ← 0, S[i] ← 0
    for j ← 1 to i − 1 do
        if (a[j] > a[i]) and (l[j] + 1 > l_max) then
            l_max ← l[j] + 1
            S[i] ← j
        l[i] ← l_max
        l_max ← 0 ; index ← 0
    for i ← 1 to n do
        if (l[i] > l_max) then
        l_max ← l[i]
        index ← i
```

Step 4. *Construct an optimal solution from the computed information.*

Algorithm Find-Sequence (S, index)

```
if S[index] = 0 then exist ;
call Find-Sequence (S, S[index]) ;
print a_index ;
```

The running time complexity of algorithm longest decreasing sequence is

$$\sum_{i=1}^{n} \sum_{j=1}^{i-1} c \le cn^2$$

Thus, its running time is $O(n^2)$, where c is a constant.

Example. *"Give an $O(n^2)$ time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers."*

Solution. Here is one suggestion :

1. Let $X = \langle x_1, \ldots, x_n \rangle$ be the sequence.
2. $Y \leftarrow$ MERGE-SORT (X)
3. $\langle c, b \rangle \leftarrow$ LCS-LENGTH (X, Y)
4. PRINT-LCS (b, X, n, n)

Example. *"Give a memoized version of LCS-LENGTH that runs in $O(mn)$ time."*

Solution.

```
MEMO-LCS-LENGTH (X,Y,i,j)
    for k ← 0 to length (X)
        for l ← 0 to length [Y]
            m[k,l] = ∞
    return MEMO-LCS-LENGTH' (X,Y,i,j)

MEMO-LCS-LENGTH' (X,Y,i,j)
    if m[i,j] ≠ ∞
        then return m[i,j]
    if i = 0 or j = 0
        then m[i,j] = 0
            return m[i,j]
    if x_i = y_j
        then m[i,j] = MEMO-LCS-LENGTH' [i-1, j-1] +1
            return m[i,j]
    m[i,j] = max (MEMO-LCS-LENGTH' (i, j-1), MEMO-LCS-LENGTH' (i-1, j))
    return m[i,j]
```

Exercise

1. Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is $\langle 5,10,3,12,5,50,6 \rangle$
2. Show that a full parenthesization of an n-element expression has exactly $n-1$ pairs of parentheses.
3. Determine an LCS of $\langle A,B,C,D,B,A,C,D,F \rangle$ and $\langle C,B,A,F \rangle$.
4. Show how to compute the length of an LCS using only 2 $\min(m,n)$ entries in the c table plus $O(1)$ additional space. Then, show how to do this using $\min(m,n)$ entries plus $O(1)$ additional space.
5. Find the longest common subsequence from the given two sequence of characters.
   (a) $P = \langle A,B,C,D,B,C,D,C,D,D \rangle$; $Q = \langle B,C,D,C,D \rangle$
   (b) $P = \langle 1,0,0,1,0,1,1,0,1,1,0,1 \rangle$; $Q = \langle 0,1,1,0 \rangle$
6. Compute the optimal sequence for the multiplication operation for the following matrices.
   (a) $P_{6\times5}\, Q_{5\times7}\, R_{7\times3}\, T_{3\times9}$ (b) $A_{5\times3}\, B_{3\times7}\, C_{7\times2}\, D_{2\times5}$

CHAPTER

22

Greedy Algorithms

22.1 Introduction

Greedy algorithms solve problems by making the choice that seems best at the particular moment. Many optimization problems can be solved using a greedy algorithm. Some problems have no efficient solution, but a greedy algorithm may provide a solution that is close to optimal. A greedy algorithm works if a problem exhibits the following *two* properties :

1. *Greedy choice property.* A globally optimal solution can be arrived at by making a locally optimal solution. In other words, an optimal solution can be obtained by making "greedy" choices.
2. *Optimal substructure.* Optimal solutions contain optimal sub solutions. In other words, solutions to sub problems of an optimal solution are optimal.

22.2 An Activity-Selection Problem

Our first example is the problem of scheduling a resource among several competing activities. We shall find that a greedy algorithm provides a well-designed and simple method for selecting a maximum-size set of mutually compatible activities.

Suppose $S = \{1, 2, \ldots, n\}$ is the set of n proposed activities. The activities share a resource, which can be used by only one activity at a time *e.g.*, a Tennis Court, a Lecture Hall etc . Each activity i has a **start time** s_i and a **finish time** f_i, where $s_i \le f_i$. If selected, activity i takes place during the half-open time interval $[s_i, f_i)$. Activities i and j are **compatible** if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap (*i.e.*, i and j are compatible if $s_i \ge f_j$ or $s_j \ge f_i$). The **activity-selection problem** selects the maximum-size set of mutually compatible activities.

In this strategy we first select the activity with minimum duration $(f_i - s_i)$ and schedule it. Then, we skip all activities that are not compatible to this one, which means we have to select compatible activity having minimum duration and then we have to schedule it. This process is repeated until all the activities are considered. It can be observed that the process of selecting the activity becomes faster if we assume that the input activities are in order by **increasing finishing time** : $f_1 \le f_2 \le f_3 \le \ldots \le f_n$

The running time of algorithm GREEDY-ACTIVITY-SELECTOR is $\theta(n \log n)$, as sorting can be done in $O(n \log n)$. There are $O(1)$ operations per activity, thus total time is

$$O(n \log n) + n \cdot O(1) = O(n \log n).$$

The pseudo code of GREEDY-ACTIVITY-SELECTOR is as follows :

```
GREEDY-ACTIVITY-SELECTOR (s, f)
n ← length[s]
A ← {1}
j ← 1
for i ← 2 to n
        do if s_i ≥ f_j
                then A ← A ∪ {i}
                     j ← i
return A
```

Theorem

The GREEDY-ACTIVITY-SELECTOR algorithm gives an optimal solution to the activity selection problem.

Proof. Let $S = \{1, 2, \ldots, n\}$ be the set of activities to schedule. Since we are assuming that the activities are in order by finish time, activity 1 has the earliest finish time. We wish to show that, there is an optimal solution that begins with a greedy choice (with activity 1, which as the earliest finish time).

Suppose that $A \subseteq S$ is an optimal solution to the given instance of the activity-selection problem, and let us order the activities in A by finish time. Suppose further that the first activity in A is activity k.

- If $k = 1$, then schedule A begins with a greedy choice.
- If $k \ne 1$, we want to show that there is another optimal solution B to S that begins with the greedy choice, activity 1.

Let $B = A - \{k\} \cup \{1\}$. Because $f_1 \le f_k$, the activities in B are disjoint, and since B has the same number of activities as A, it is also optimal. Thus, B is an optimal solution for S that contains the greedy choice of activity 1.

Once the greedy choice of activity 1 is made, the problem reduces to finding an optimal solution for the activity-selection problem over those activities in S that are compatible with activity 1. That is, if A is an optimal solution to the original problem S, then $A' = A - \{1\}$ is an optimal solution to the activity-selection problem $S' = \{i \in S : S_i \le f_1\}$.

Why ? If we could find a solution B' to S' with more activities than A', *adding activity* 1 *to* B' would yield a solution B to S with more activities than A, thereby contradicting the optimality of A. Therefore, after each greedy choice is made, we are left with an optimization problem of the same form as the original problem. By induction on the number of choices made, making the greedy choice at every step produces an optimal solution.

Example. *Given 10 activities along with their start and finish time as*

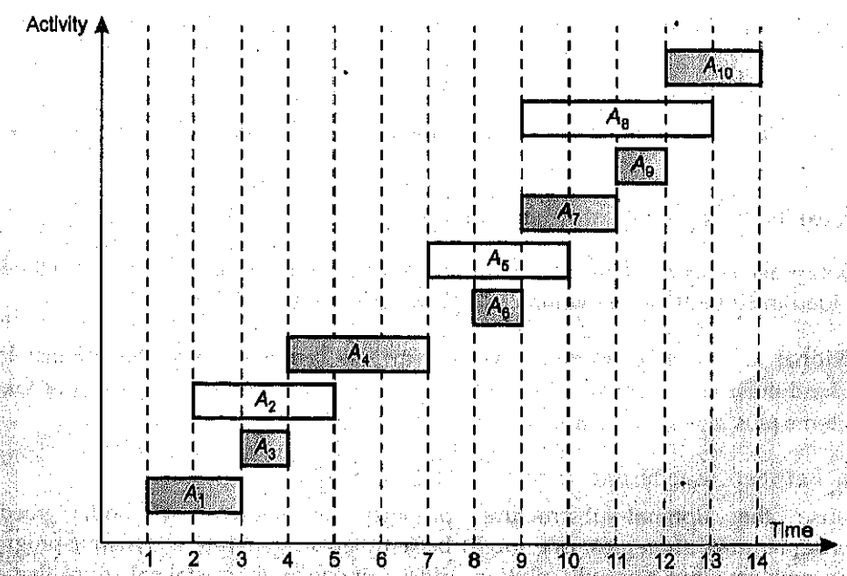
$$S = \langle A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8, A_9, A_{10} \rangle$$
$$S_i = \langle 1,2,3,4,7,8,9,9,11,12 \rangle$$
$$f_i = \langle 3,5,4,7,10,9,11,13,12,14 \rangle$$

Compute a schedule where the largest number of activities takes place.

Solution. The solution for the above activity scheduling problem using greedy strategy is illustrated below.

Arranging the activities in increasing order of finish time.

Activity	A_1	A_3	A_2	A_4	A_6	A_5	A_7	A_9	A_8	A_{10}
Start	1	3	2	4	8	7	9	11	9	12
Finish	3	4	5	7	9	10	11	12	13	14



Now, schedule A_1

Next, schedule A_3, as A_1 and A_3 are non-interfering

Next, Skip A_2 as it is interfering.

Next, schedule A_4 as A_1, A_3 and A_4 are non-interfering, then next, schedule A_6 as A_1, A_3, A_4 and A_6 are non-interfering.

Skip A_5 as it is interfering.

Next, schedule A_7 as A_1, A_3, A_4, A_6 and A_7 are non-interfering.

Next, schedule A_9 as A_1, A_3, A_4, A_6, A_7 and A_9 are non-interfering.

Skip A_8, as it is interfering.

Next, schedule A_{10} as $A_1, A_3, A_4, A_6, A_7, A_9$ and A_{10} are non-interfering.

Thus, the final activity schedule is

$$\langle A_1, A_3, A_4, A_6, A_7, A_9, A_{10} \rangle$$

22.3 Knapsack Problems

We want to pack n items in your luggage

- The ith item is worth v_i dollars and weighs w_i pounds
- Take as valuable a load as possible, but cannot exceed W pounds
- v_i, w_i, W are integers

0-1 Knapsack Problem

- each item is taken or not taken.
- cannot take a fractional amount of an item or take an item more than once

Fractional Knapsack Problem

- fractions of items can be taken rather than having to make a binary (0-1) choice for each item

Both exhibit the optimal-substructure property

0-1 knapsack problem. Consider a optimal solution. If item j is removed from the load, the remaining load must be the most valuable load weighing at most $W - w_j$.

Fractional knapsack problem. If w of item j is removed from the optimal load, the remaining load must be the most valuable load weighing at most $W - w$ that can be taken from other $n-1$ items plus $w_j - w$ of item j.

Difference Between Greedy and Dynamic Programming

Because the optimal-substructure property is shown by both greedy and dynamic-programming strategies, one might be tempted to generate a dynamic-programming solution to a problem when a greedy solution suffices, or one might mistakenly think that a greedy

solution works when in fact a dynamic-programming solution is required. The most important difference between greedy algorithms and dynamic programming is that we don't solve every optimal sub-problem with greedy algorithms. In some cases, greedy algorithms can be used to produce sub-optimal solutions. That is, solutions which aren't necessarily optimal, but are perhaps very close.

In dynamic programming, we make a choice at each step, but the choice may depend on the solutions to sub-problems. In a greedy algorithm, we make whatever choice seems best at the moment and then solve the sub-problems arising after the choice is made. The choice made by a greedy algorithm may depend on choices so far, but it cannot depend on any future choices or on the solutions to sub-problems. Thus, unlike dynamic programming, which solves the sub-problems bottom up, a greedy strategy usually progresses in a top-down fashion, making one greedy choice after another, interactively reducing each given problem instance to a smaller one.

Fractional knapsack problem can be solvable by the greedy strategy whereas the 0-1 problem is not. To solve the fractional problem

- Compute the value per pound v_i / w_i for each item
- Obeying a greedy strategy, we take as much as possible of the item with the greatest value per pound.
- If the supply of that item is exhausted and we can still carry more, we take as much as possible of the item with the next value per pound, and so forth until we cannot carry any more.
- sorting the items by value per pound, the greedy algorithm runs in $O(n \lg n)$ time .

Fractional Knapsack (Array *v*, Array *w*, int *W*)

```
for i = 1 to Size(v)
    do p[i] = v[i] / w[i]
Sort-Descending(p)
i ← 1
while (W > 0)
    do amount = min(W, w[i])
    solution[i] = amount
    W = W-amount
    i ← i + 1
return solution
```

0-1 knapsack problem cannot be solved by the greedy strategy because it is unable to fill the knapsack to capacity, and the empty space lowers the effective value per pound of the load and we must compare the solution to the sub-problem in which the item is included with the solution to the sub-problem in which the item is excluded before we can make the choice.

Example. *Consider 5 items along their respective weights and values*

$$I = \langle I_1, I_2, I_3, I_4, I_5 \rangle$$
$$w = \langle 5, 10, 20, 30, 40 \rangle$$
$$v = \langle 30, 20, 100, 90, 160 \rangle$$

The capacity of knapsack $W = 60$. *Find the solution to the fractional knapsack problem.*

Solution. Initially,

Item	w_i	v_i
I_1	5	30
I_2	10	20
I_3	20	100
I_4	30	90
I_5	40	160

Taking value per weight ratio *i.e.*, $p_i = v_i / w_i$

Item	w_i	v_i	$p_i = v_i / w_i$
I_1	5	30	6.0
I_2	10	20	2.0
I_3	20	100	5.0
I_4	30	90	3.0
I_5	40	160	4.0

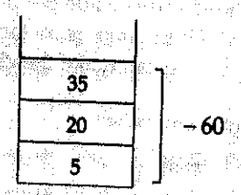
Now, arrange the value of p_i in decreasing order.

Item	w_i	v_i	$p_i = v_i / w_i$
I_1	5	30	6.0
I_3	20	100	5.0
I_5	40	160	4.0
I_4	30	90	3.0
I_2	10	20	2.0

Now, fill the knapsack according to the decreasing value of p_i.

First we choose item I_1 whose weight is 5, then choose item I_3 whose weight is 20. Now the total weight in knapsack is $5+20=25$.

Now, the next item is I_5 and its weight is 40, but we want only 35. So we choose fractional part of if *i.e.*,



The value of fractional part of I_5 is

$$\frac{160}{40} \times 35 = 140$$

Thus the maximum value is

$$= 30 + 100 + 140 = 270$$

22.4 Huffman Codes

Data can be encoded efficiently using Huffman codes. It is a widely used and very effective technique for compressing data ; savings of 20% to 90% are typical, depending on the characteristics of the file being compressed. Huffman's greedy algorithm uses a table of the frequencies of occurrence of each character to build up an optimal way of representing each character as a binary string.

Suppose we have 10^5 characters in a data file. Normal storage: 8 bits per character (ASCII)- 8×10^5 bits in file. But, we want to compress the file and store it compactly. Suppose only 6 characters appear in the file :

	a	*b*	*c*	*d*	*e*	*f*	Total
Frequency	45	13	12	16	9	5	100

How can we represent the data in a compact way ?

(i) **Fixed Length code.** Each letter represented by an equal number of bits. With a fixed length code, at least 3 bits per character :

For example :

a	000
b	001
c	010
d	011
e	100
f	101

For a file with 10^5 characters, we need 3×10^5 bits.

(ii) A **variable-length code** can do considerably better than a fixed-length code, by giving frequent characters short code words and infrequent characters long code words.

For example,

a	0
b	101
c	100
d	111
e	1101
f	1100

Number of bits $= (45*1+13*3+12*3+16*3+9*4+5*4)*1000$

$= 2.24 \times 10^5$ bits.

Thus, 224,000 bits to represent the file, a saving of approximately 25%. In fact, this is an optimal character code for this file.

Let us denote the characters by $C_1, C_2, \ldots, C_n$ and denote their frequencies by $f_1, f_2, \ldots, f_n$. Suppose there is an encoding E in which a bit string S_i of length s_i represents C_i, the length of the file compressed by using encoding E is

$$L(E, F) = \sum_{i=1}^{n} s_i \cdot f_i$$

22.5 Prefix Codes

The prefixes of an encoding of one character must not be equal to a complete encoding of another character *e.g.* 1100 and 11001 are not valid codes because 1100 is a prefix of 11001. This constraint is called the **prefix constraint**. Codes in which no codeword is also a prefix of some other code word are called **prefix codes.**

Shortening the encoding of one character may lengthen the encoding of others. To find an encoding E that satisfies the prefix constraint and minimizes $L(E, F)$.

Prefix codes are desirable because they simplify encoding (compression) and decoding. Encoding is always simple for any binary character code; we just concatenate the code words representing each character of the file. Decoding is also quite simple with a prefix code. Since no codeword is a prefix of any other, the codeword that begins an encoded file is unambiguous. We can simply identify the initial codeword, translate it back to the original character, remove it from the encoded file, and repeat the decoding process on the remainder of the encoded file.

The decoding process needs a convenient representation for the prefix code so that the initial codeword can be easily picked off. A binary tree whose leaves are the given characters provides one such representation. We interpret the binary codeword for a character as the path from the root to that character, where 0 means "go to the left child" and 1 means "go to the right child." Note that these are not binary search trees, since the leaves need not appear in sorted order and internal nodes do not contain character keys.

An optimal code for a file is always represented by a full binary tree, in which every non-leaf node has two children. The fixed-length code in our example is not optimal since its tree, because it is not a full binary tree : there are code words beginning 10 . . . , but none beginning 11 . . . . Since we can now restrict our attention to full binary trees, we can say that if C is the alphabet from which the characters are drawn, then the tree for an optimal prefix code has exactly $|C|$ *leaves, one for each letter of the alphabet, and exactly* $|C|-1$ internal nodes.

Given a tree T corresponding to a prefix code, it is a simple matter to compute the number of bits required to encode a file. For each character c in the alphabet C, let $f(c)$ denote the frequency of c in the file and let $d_T(c)$ denote the depth of c's leaf in the tree. Note that $d_T(c)$ is also the length of the codeword for character c. The number of bits required to encode a file is thus

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

which we define as the *cost* of the tree T.

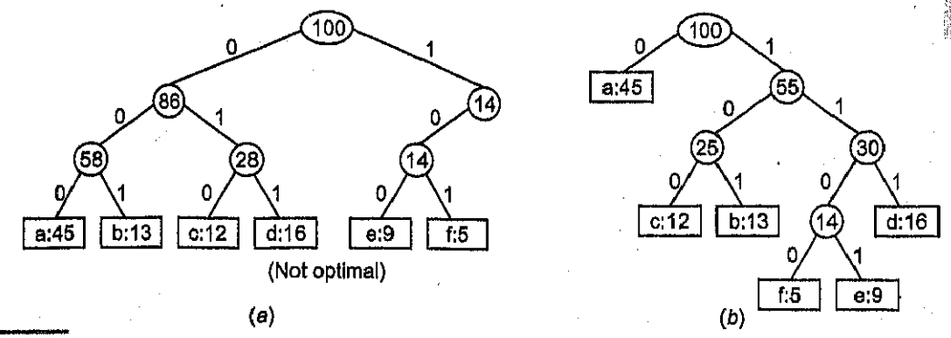
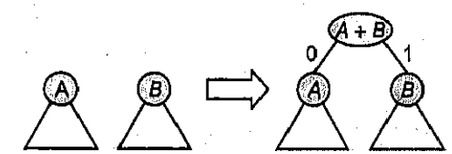


Figure 22.1

Fig. 22.1, Trees corresponding to the coding schemes. Each leaf is labeled with a character and its frequency of occurrence. Each internal node is labeled with the sum of the weights of the leaves in its subtree. (*a*) The tree corresponding to the fixed-length code $a = 000, \ldots, f = 100$ (*b*) The tree corresponding to the optimal prefix code $a = 0$, $b = 101$, ..., $f = 1100$.

22.6 Greedy Algorithm for Constructing a Huffman Code

Huffman invented a greedy algorithm that constructs an optimal prefix code called a ***Huffman code.***



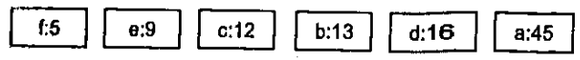
The algorithm builds the tree T corresponding to the optimal code in a bottom-up manner. It begins with a set of $|C|$ leaves and performs a sequence of $|C|-1$ "merging" operations to create the final tree.

In the pseudo code HUFFMAN(C), we assume that C is a set of n characters and that each character $c \in C$ is an object with a defined frequency $f[c]$. A priority queue Q keyed on f, is used to identify the two least-frequent objects to merge together. The result of the merger of two objects is a new object whose frequency is the sum of the frequencies of the two objects that were merged.

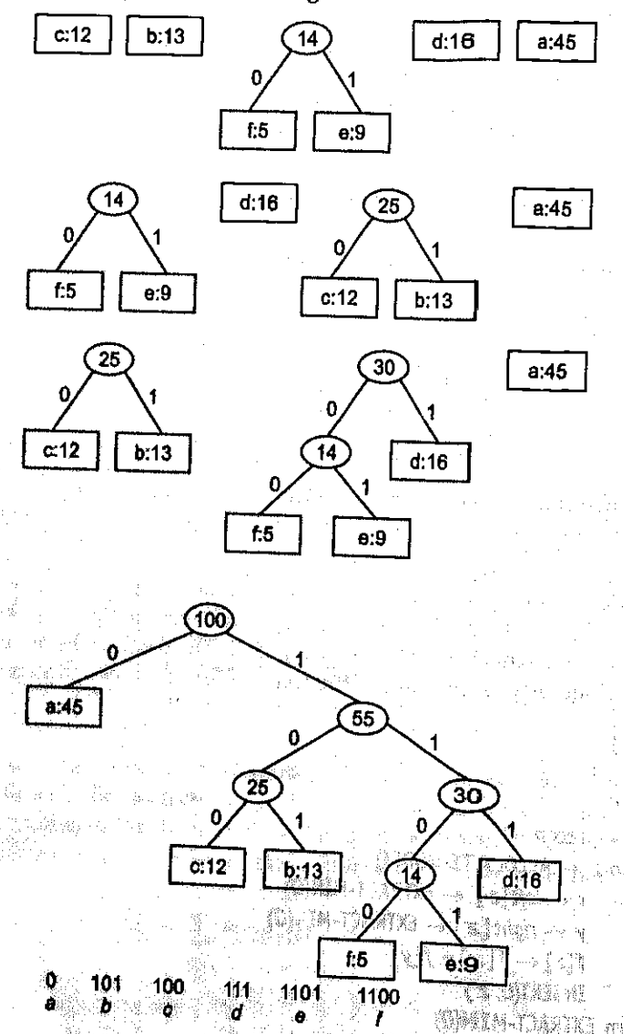
```
HUFFMAN(C)
n ← |C|
Q ← C
for i ← 1 to n − 1
    do z ← ALLOCATE-NODE()
        x ← left[z] ← EXTRACT-MIN(Q)
        y ← right[z] ← EXTRACT-MIN(Q)
        f[z] ← f[x] + f[y]
        INSERT(Q, z)
return EXTRACT-MIN(Q)
```

The analysis of the running time of Huffman's algorithm assumes that Q is implemented as a binary heap. For a set C of n characters, the initialization of Q in line 2 can be performed in $O(n)$ time using the BUILD-HEAP procedure. The for loop in lines 3-8 is executed exactly $|n|-1$ times, and since each heap operation requires time $O(n \lg n)$, the loop contributes $O(n \lg n)$ to the running time. Thus, the total running time of HUFFMAN on a set of n characters is $O(n \lg n)$.

Example :



The algorithm is based on a reduction of a problem with n characters to a problem with $n-1$ characters. A new character replaces two existing ones.



Using Huffman codes

- Each message has a different tree. The tree must be saved with the message.
- Huffman codes are effective for long files where the savings in the message can offset the cost for storing the tree.
- Decode files by starting at root and proceeding down the tree according to the bits in the message (0 = left, 1 = right). When a leaf is encountered, output the character at that leaf and restart at the root.
- Huffman codes are also effective when the tree can be pre-computed and used for a large number of messages (*e.g.*, a tree based on the frequency of occurrence of characters in the English language).
- Huffman codes are not very good for random files (each character about the same frequency).

22.7 Activity or Task Scheduling Problem

This is the problem of optimally scheduling unit-time tasks on a single processor, where each task has a dead line and a penalty that must be paid if the dead line is missed.

A **unit time task** is a job such as a program to be run on a computer that requires exactly one unit of time to complete. Given a finite set S of unit-time tasks, a schedule for S is a permutation of S specifying the order in which these tasks are to be performed. The first task in the schedule begins at time 0 and finishes at time 1, the second task begins at time 1 and finishes at time 2 and so on.

The problem of scheduling unit time tasks with dead lines and penalties for a single processor has the following inputs :

- a set $S = \{1, 2, 3, \ldots n\}$ of n-unit-time tasks.
- a set of n integer dead lines $d_1, d_2, d_3, \ldots d_n$ such that each d_i satisfies $1 \le d_i \le n$ and task i is supposed to finish by time d_i and
- a set of n non-negative weights or penalties $w_1, w_2, \ldots w_n$ such that a penalty w_i is incurred if task i is not finished by time d_i and no penalty is incurred if a task finishes by its dead line.

Here we find a schedule for S that minimizes the total penalty incurred for missed dead lines.

A task is **late** in this schedule if it finished after its dead line. Otherwise, the task is **early** in the schedule. An arbitrary schedule can always be put into **early-first form**, in which the early tasks precede the late tasks, *i.e.*, if some early task x follows some late task y, then we can switch the positions of x and y without affecting x being early or y being late.

An arbitrary schedule can always be put into **canonical form**, in which the early tasks precede the late tasks and the early tasks are scheduled in order of non-decreasing dead lines.

The search for an optimal schedule reduces to finding a set A of tasks that are to be early in the optimal schedule. Once A is determined, we can create the actual schedule by listing the elements of A in order of non-decreasing dead line, then listing the late tasks (*i.e.*, $S - A$) in any order, producing a canonical ordering of the optimal schedule.

A set A of tasks is **independent** if there exists a schedule for these tasks such that no tasks are late. So, the set of early tasks for a schedule forms an independent set of tasks 'I' denote the set of all independent sets of tasks.

For any set of tasks A, A is independent if for $t = 0,1,2,\ldots n$ we have $N_t(A) \le t$ where $N_t(A)$ denote the number of tasks in A whose dead line is t or earlier, *i.e.*, if the tasks in A are scheduled in order of monotonically increasing dead lines, then no task is late.

Example. *Find the optimal schedule for the following task with given weights (penalties) and dead lines.*

	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

Solution. According to Greedy algorithm we sort the jobs in decreasing order of their penalties so that minimum of penalties will be charged.

In this problem, we can see that maximum time for which uniprocessor machine will run is 6 units because it is the maximum dead line.

Let T_i represents the tasks where $i = 1$ to 7

T_2	T_3	T_4	T_1	T_7	T_5	T_6

0 1 2 3 4 5 6 7

T_5 and T_6 cannot be accepted after T_7 so penalty is

$$w_5 + w_6 = 30 + 20 = 50. \qquad \langle 2\ 3\ 4\ 1\ 7\ 5\ 6 \rangle$$

other schedule is

T_2	T_4	T_1	T_3	T_7	T_5	T_6

0 1 2 3 4 5 6 7 $\langle 2\ 4\ 1\ 3\ 7\ 5\ 6 \rangle$

There can be many other schedules but $\langle 2\ 4\ 1\ 3\ 7\ 5\ 6 \rangle$ is optimal.

Note *In this smaller dead line come first and complete the task earlier than larger dead line tasks.*

Example. *Let* $n = 4\,(P_1, P_2, P_3, P_4) = (100, 10, 15, 27)$ *and* $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$ *where* P_i *are profits on processes or job and* d_i *are dead line of completion. Find the optimal schedule.*

Solution. Max. dead line is 2 so max. number of processes that are scheduled is 2.

Feasible Solution	Processing Sequence	Value
(1, 2)	(2, 1)	10 + 100 = 110
(1, 3)	(1, 3) or (3, 1)	100 + 15 = 115
(1, 4)	(4, 1)	27 + 100 = 127
(2, 3)	(2, 3)	25
(3, 4)	(4, 3)	42
(1)	1	100
(2)	2	10
(3)	3	15
(4)	4	27

Thus, the optimal schedule is (4, 1) and profit 127.

22.8 Travelling Sales Person Problem

In this a salesman needs to visit 'n' cities in such a manner that all cities must be visited at once and in the end he returns to the city from where he started with minimum cost.

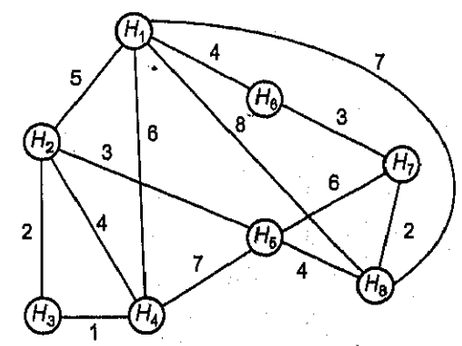
Suppose the cities are $x_1, x_2, \ldots x_n$ where cost c_{ij} denotes the cost of travelling from city x_i to x_j. The travelling salesman problem is to find a route starting and ending at x_1 that will take in all the cities with the minimum cost.

Greedy Strategy for Travelling Salesman Problem

Start with any arbitrary city called x_1 then choose the minimum cost city from x_1. The method is represented until all the cities are visited and at every step we have to select the city with the minimum weight.

Example. *A newspaper agent daily drops the newspaper to the area assigned in such a manner that he has to cover all the houses in the respective area with minimum travel cost. Compute the minimum travel cost.*

The area assigned to the agent where he has to drop the newspaper is shown in the Fig.



Solution. The cost-adjacency matrix of graph G is as follows :

$\text{cost}_{ij} =$

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	4	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

The tour starts from area H_1 and then select the minimum cost area reachable from H_1.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	4	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Mark area H_6 because it is the minimum cost area reachable from H_1 then select minimum cost area reachable from H_6.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	4	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Mark area H_7 because it is the minimum cost area reachable from area H_6 and then select minimum cost area reachable from H_7.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	4	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Mark area H_8 and select minimum cost area reachable from H_8.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	4	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Mark area H_5 and select minimum cost area reachable from H_5.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	4	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Mark area H_2 and select minimum cost area reachable from H_2.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	4	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Mark area H_3 and select minimum cost area reachable from H_3.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	4	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Mark area H_4 and select minimum cost area reachable from H_4 it is H_1. So, using the greedy strategy we get the following

$$H_1 \xrightarrow{4} H_6 \xrightarrow{3} H_7 \xrightarrow{2} H_8 \xrightarrow{4} H_5 \xrightarrow{3} H_2 \xrightarrow{2} H_3 \xrightarrow{1} H_4 \xrightarrow{6} H_1$$

Thus, the minimum travel cost

$$= 4+3+2+4+3+2+1+6 = 25$$

22.9 Matroids

A **matroid** is an ordered pair $M(S, I)$ satisfying the following conditions.

1. S is a finite nonempty set.
2. I is a nonempty family of subsets of S, called the **independent** subsets of S, such that if $B \in I$ and $A \subseteq B$, then $A \in I$. We say that I is **hereditary** if it satisfies this property. Note that the empty set ϕ is necessarily a member of I.
3. If $A \in I$, $B \in I$ and $|A| < |B|$, then there is some element $x \in B - A$ such that $A \cup \{x\} \in I$. We say that M satisfies the **exchange property.**

We say that a matroid $M(S, I)$ is ***weighted*** if there is an associated weight function w that assigns a strictly positive weight $w(x)$ to each element $x \in S$. The weight function w extends to subsets of S by summation :

$$w(A) = \sum_{x \in A} w(x)$$

For any $A \subseteq S$.

For example, if we let $w(e)$ denote the length of an edge e in a graphic matroid M_G, then $w(A)$ is the total length of the edges in edge set A.

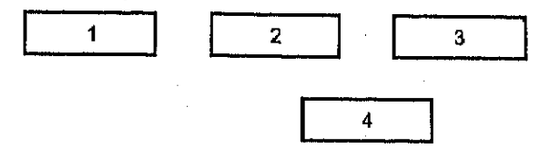
22.10 Minimum Spanning Tree

This is yet another example of greedy technique where we compute the minimum spanning tree in a given graph. This problem can be comprehensively studied by including Kruskal's and Prim's minimum spanning tree algorithm which are discussed in detail in the coming chapters.

Example. *"Not just any greedy approach to the activity-selection problem produces a maximum-size set of mutually compatible activities. Give an example to show that the approach of selecting the activity of least duration from those that are compatible with previously selected activities does not work. Do the same for the approaches of always selecting the compatible activity that overlaps the fewest other remaining activities and always selecting the compatible remaining activity with the earliest start time".*

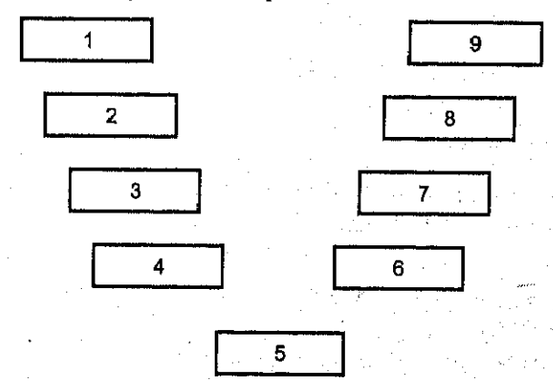
Solution.

◄ Choose the next activity with the shortest duration :



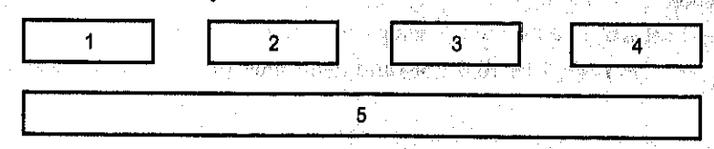
This gives us $A = \{1, 4\}$. The original algorithm returns $A = \{1, 2, 3\}$

◄ Choose the next activity that overlaps the last with the remaining activities.



This gives us $A = \{1, 5, 9\}$ (since they only overlap two others). The original algorithm returns $A = \{1, 4, 6, 9\}$

◄ Choose the next activity with the earliest start time.



If we go by earliest start time, we have to pick 5 first, thus excluding all other ones. The original greedy algorithm returns $A = \{1, 2, 3, 4\}$

Example. *Consider the problem of making change for N cents using the least number of coins. Describe a greedy algorithm to make change consisting of quarters, dimes, nickels and pennies.*

(*Hint. a quarter* $=25c$, *a dime* $=10c$, *a nickel* $=5c$ *and a penny* $=1c$)

Solution.

Step 1. Let $f(N)$ be the minimum number of change coins needed for n cents.

Step 2. Define the recurrence. Then we have

$$f(N)=\begin{cases} 1 \text{ if } N=1 \\ 1 \text{ if } N=5 \\ 1 \text{ if } N=10 \\ 1 \text{ if } N=25 \\ \min\{f(N-25)+1, f(N-10)+1, f(N-5)+1, f(N-1)+1\} \text{ otherwise} \end{cases}$$

Step 3. Compute $f(N)$

Step 4. If we need to find all different coins, another array is needed to keep the intermediate computation.

Exercise

1. Given 10 activities along with their start and finish time as

$$S=(A_1,A_2,A_3,A_4,A_5,A_6,A_7,A_8,A_9,A_{10})$$
$$S_i=(1,2,3,4,5,6,7,8,9,10)$$
$$f_i=(5,3,4,6,7,8,11,10,12,13)$$

Compute a schedule where the largest number of activities take place.

2. Find an optimal solution for the knapsack Instances

$n=7$, $M=15$ $(P_1,P_2,\ldots,P_7)=(10,5,15,7,6,18,3)$ and $(w_1,w_2,\ldots w_7)=(2,3,5,7,1,4,1)$

3. Consider the knapsack instance $n=3(w_1,w_2,w_3)=(2,3,4)$ and $(P_1,P_2,P_3)=(2,3,5)$ $M=5$. Find the optimal solution.
4. Show how to solve fractional knapsack problem in $\theta(n)$ time.
5. Generalize Huffman's algorithm for ternary code words and prove that it yields optimal ternary codes.
6. Let $C=\{0,1,\ldots,n-1\}$ be a set of characters. Show that any optimal prefix code on C can be represented by a sequence of $2n-1+n\lceil \lg n \rceil$.
7. Find the optimal schedule for the following jobs with $n=7$ profits

$(P_1,P_2,\ldots P_7)=(3,5,18,20,6,1,38)$ and dead lines $(d_1,d_2,d_3,\ldots d_7)=(1,3,3,4,1,2,1)$?

CHAPTER

23

Backtracking

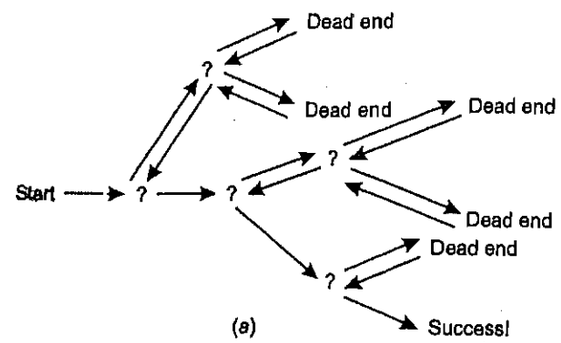
23.1 Introduction

Have you ever seen poor blind people walking in roads? If they find any obstacles in their way, they would just move backward. Then they will proceed in other direction. How a blind person could move backward when he finds obstacles ? Simple answer...by intelligence! Similarly, If an algorithm backtracks with intelligence, it is called *Backtracking algorithm.*

Suppose we have to make a series of *decisions*, among various *choices*, where we don't have enough information to know what to choose and each decision leads to a new set of choices. Some sequence of choices (possibly more than one) may be a solution to our problem. **Backtracking** is a methodical way of trying out various sequences of decisions, until we find one that "works".

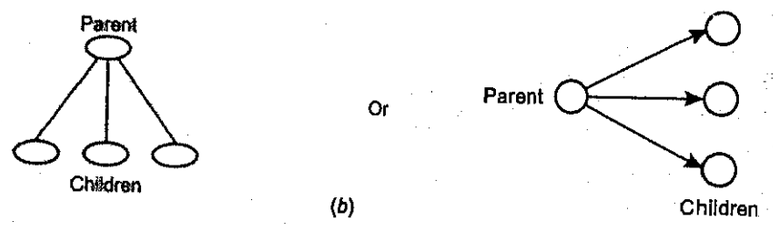
In the following Figure,

- Each non-leaf node in a tree is a parent of one or more other nodes (its children)
- Each node in the tree, other than the root, has exactly one parent



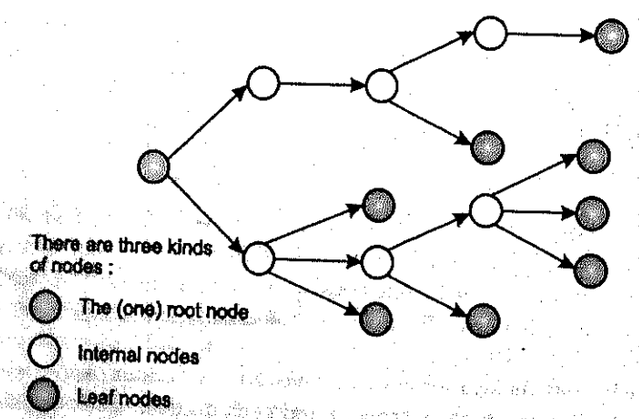
(a)

Usually, however, we draw our trees *downward*, with the root at the top.



(b)

A tree is composed of nodes



(c)

Figure 23.1

Backtracking can be thought of as searching a tree for a particular "goal" leaf node.

There is a type of data structure called a tree, but we are **not** using it here. Actually, if we diagram the sequence of choices we make, the diagram looks like a tree. Our backtracking algorithm "sweeps out a tree" in "problem space". Backtracking is really quite simple–we "explore" each node, as follows :

To "explore" node N :

1. If N is a goal node, return "success"
2. If N is a leaf node, return "failure"
3. For each child C of N,

   Explore C

   If C was successful, return "success"
4. Return "failure"

The basic idea of backtracking is to build up a vector one component at a time and to test whether the vector being formed has any chance of success. The major advantage of back tracking algorithm is that if it is realized that the partial vector generated does not lead to an optimal solution then that vector may be ignored.

Backtracking algorithm determines the solution by systematically searching the solution space for the given problem. Backtracking is a **depth first search** with some bounding function. All solutions using backtracking are required to satisfy a complex set of constraints. The constraints may be explicit or implicit.

Explicit constraints are rules, which restrict each vector element to be chosen from the given set. **Implicit constraints** are rules, which determine which of the tuples in the solution space, actually satisfy the criterion function.

23.2 Recursive Maze Algorithm

Recursive maze algorithm is one of the good examples for backtracking algorithms. In fact Recursive maze algorithm is one of the most available solutions for solving maze.

Maze

Maze is an area surrounded by walls; in between we have a path from starting position to ending position. We have to start from the starting point and travel towards the ending point.

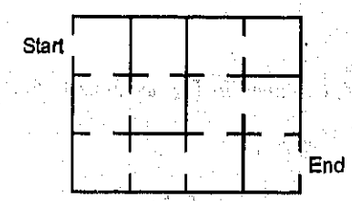


Figure 23.2

Principle of Maze

As explained above, in maze we have to travel from the starting point to ending point. The problem is to choose the path. If we find any dead-end before ending point, we have to backtrack and change the direction. The direction for traversing is North, East, West and South. We have to continue "move and backtrack" until we reach the ending point.

Assume that we are having a two-dimensional maze cell [WIDTH][HEIGHT]. Here cell[x][y]=1 denotes wall and cell[x][y]=0 denotes free cell in the particular location x, y in the maze. The directions we can move in the array are North, East, West and South. The first step is to make the boundary of the two-dimensional array as 1 so that we won't go out of the maze, and always reside inside the maze at any time.

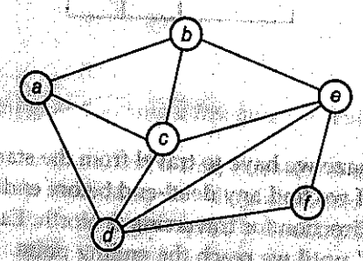
Example Maze						
1	1	1	1	1	1	1
1	0	0	0	1	1	1
1	1	1	0	1	1	1
1	1	1	0	0	0	1
1	1	1	1	1	0	1
1	1	1	1	1	1	1

Now start moving from the starting position (since the boundary is filled by 1) and find the next free cell, then move to the next free cell and so on. If we reach a dead-end, we have to backtrack and make the cells in the path as 1(wall). Continue the same process till the ending point is reached.

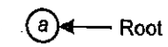
23.3 Hamiltonian Circuit Problem

Given a graph $G=(V, E)$, we have to find the ***Hamiltonian Circuit*** using Backtracking approach. We start our search from any arbitrary vertex, say 'a'. This vertex 'a' becomes the root of our implicit tree. The first element of our partial solution is the first intermediate vertex of the Hamiltonian Cycle that is to be constructed. The next adjacent vertex is selected on the basis of alphabetical (or numerical) order. If at any stage any arbitrary vertex makes a cycle with any vertex other than vertex 'a' then we say that **dead end** is reached. In this case we backtrack one step, and again the search begins by selecting another vertex and backtrack the element from the partial solution must be removed. The search using backtracking is successful if a Hamiltonian cycle is obtained.

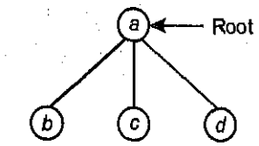
Example. *Consider a graph $G=(V, E)$ shown in Fig. we have to find a Hamiltonian circuit using Back tracking method.*



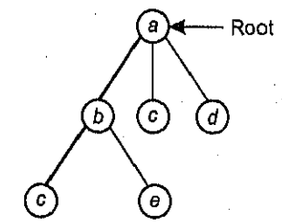
Solution. Firstly, we start our search with vertex 'a', this vertex 'a' becomes the root of our implicit tree.



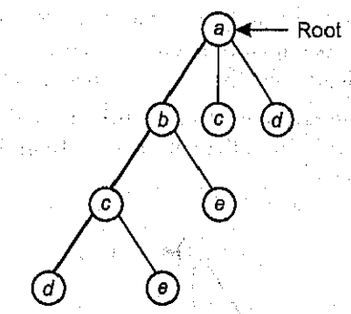
Next, we choose vertex 'b' adjacent to 'a' as it comes first in lexicographical order (b, c, d).



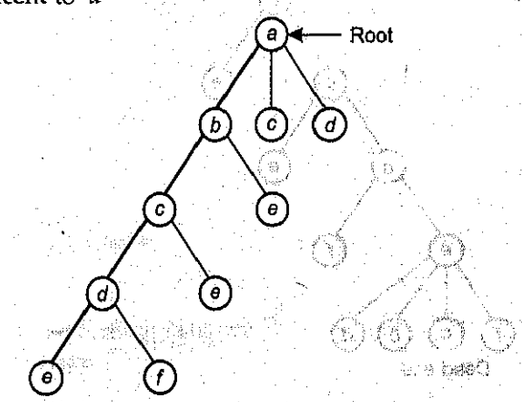
Next, we select 'c' adjacent to 'b'



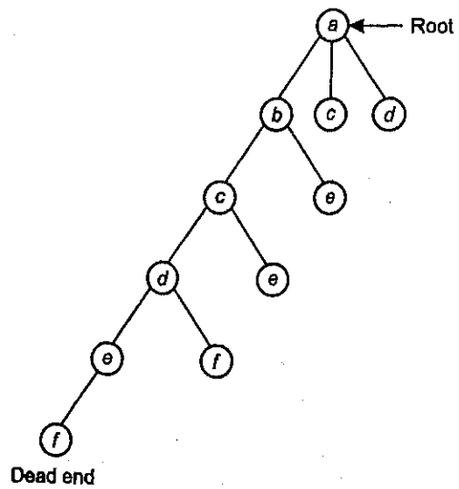
Next, we select 'd' adjacent to 'c'



Next, we select 'e' adjacent to 'd'

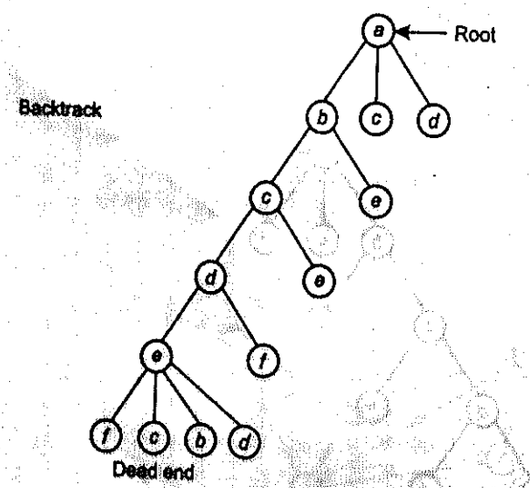


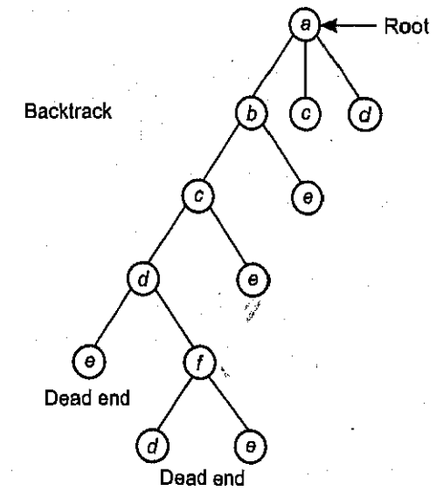
Next, we select vertex 'f' adjacent to 'e'. The vertex adjacent to 'f' are d and e but they have already visited. Thus, we get the dead end and we back track one step and remove the vertex 'f' from partial solution.



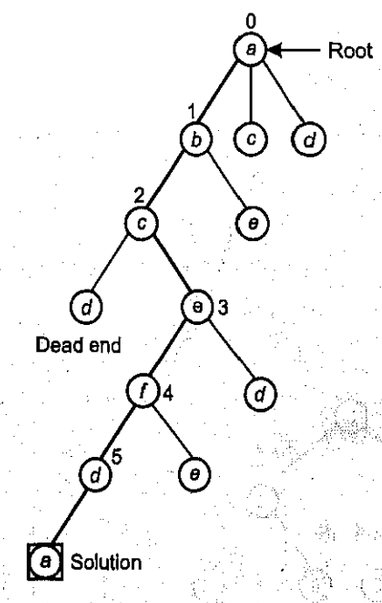
From back tracking, the vertex adjacent to 'e' are b, c, d, f from which vertex 'f' has already been checked and b, c, d have already visited. So, again we back track one step. Now, the vertex adjacent to d are e, f from which e has already been checked and adjacent of 'f' are d and e. If 'e' vertex visited then again we get dead state. So again we backtrack one step.

Now, adjacent to c is 'e' and adjacent to 'e' is 'f' and adjacent to 'f' is 'd' and adjacent to 'd' is 'a' Here, we get the Hamiltonian cycle as all the vertex other than the start vertex 'a' is visited only once. ($a-b-c-e-f-d-a$).





Again back track



Here we have generated one Hamiltonian circuit but other Hamiltonian circuit can also be obtained by considering other vertex.

23.4 Subset-sum Problem

In the subset-sum problem we have to find a subset s' of the given set $S=\{S_1, S_2, S_3, \ldots S_n\}$ where the elements of the set S are n positive integers in such a manner that $s' \in S$ and sum of the elements of subset 's' ' is equal to some positive integer 'X'.

The subset-sum problem can be solved by using the backtracking approach. In this implicit tree is a binary tree. The root of the tree is selected in such a way that it represents that no decision is yet taken on any input. We assume that the elements of the given set are arranged in an increasing order :

$$S_1 \le S_2 \le S_3 \ldots \le S_n$$

The left child of the root node indicates that we have to include 'S_1' from the set 'S' and the right child of the root node indicates that we have to exclude 'S_1' . Each node stores the sum of the partial solution elements. If at any stage the number equals to 'X' then the search is successful and terminates.

The dead end in the tree occurs only when either of the two inequalities exist :

- The sum of s' is too large *i.e.*

$$s' + S_i + 1 > X$$

- The sum of s' is too small *i.e.*

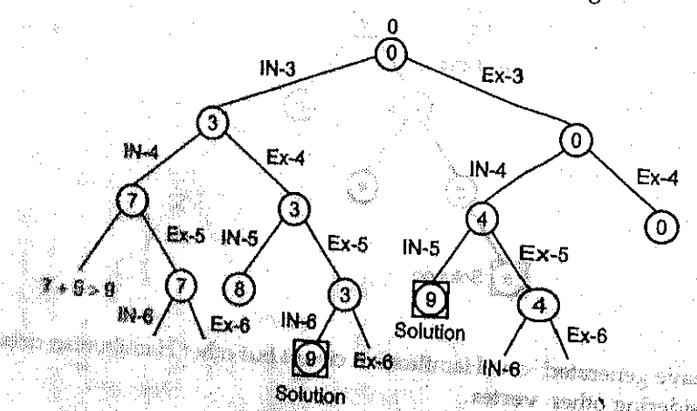
$$s' + \sum_{i=i+1}^{n} S_i < X$$

Example. *Given a set $S=\{3,4,5,6\}$ and $X=9$. Obtain the subset sum using Back tracking approach.*

Solution. Initially $S=\{3,4,5,6\}$ and $X=9$

$$S' = \{\phi\}$$

The implicit binary tree for subset sum problem is shown in Fig.



The number inside a node is the sum of the partial solution elements at a particular level.

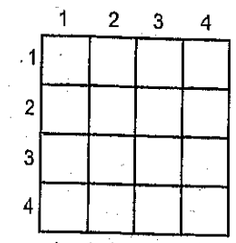
Thus, if our partial solution elements sum is equal to the positive integer 'X' then at that time search will terminates, or it continues if all the possible solutions need to be obtained.

23.5 *N*-Queens Problem

N-queens problem is to place n-queens in such a manner on an $n \times n$ chessboard that no two queens attack each other by being in the same row, column or diagonal.

It can be seen that for $n=1$, the problem has a trivial solution, and no solution exists for $n=2$ and $n=3$. So first we will consider the 4-queens problem and then generalise it to n-queens problem.

Given, a 4×4 chessboard and number the rows and column of the chessboard 1 through 4.



4 × 4 chessboard

Figure 23.3

Since we have to place 4 queens such as q_1, q_2, q_3 and q_4 on a chessboard, such that no two queens attack each other. In such a condition each queen must be placed on a different row, *i.e.*, we place queen "i" on row "i".

Now, we place queen q_1 in the very first acceptable position (1, 1). Next, we place queen q_2 so that both these queens do not attack each other. We find that if we place q_2 in column 1 and 2 then the dead end is encountered. Thus the first acceptable position for q_2 is column 3 *i.e.*, (2, 3) but then no position is left for placing queen 'q_3' safely. So we backtrack one step and place the queen 'q_2' in (2, 4), the next best possible solution. Then we obtain the position for placing 'q_3' which is (3, 2). But later this position also leads to dead end and no place is found where 'q_4' can be placed safely. Then we have to backtrack till 'q_1' and place it to (1, 2) and then all the other queens are placed safely by moving q_2 to (2, 4), q_3 to (3, 1) and q_4 to (4, 3). That is, we get the solution $\langle 2, 4, 1, 3 \rangle$. This is one possible solution for 4-queens problem. For other possible solution the whole method is repeated for all partial solutions. The other solution for 4-queens problem is $\langle 3, 1, 4, 2 \rangle$ *i.e.*,

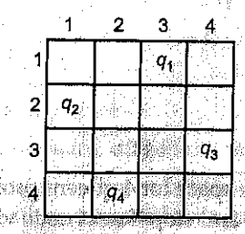


Figure 23.4

The implicit tree for 4-queen problem for solution $\langle 2,4,1,3 \rangle$ is as follows :

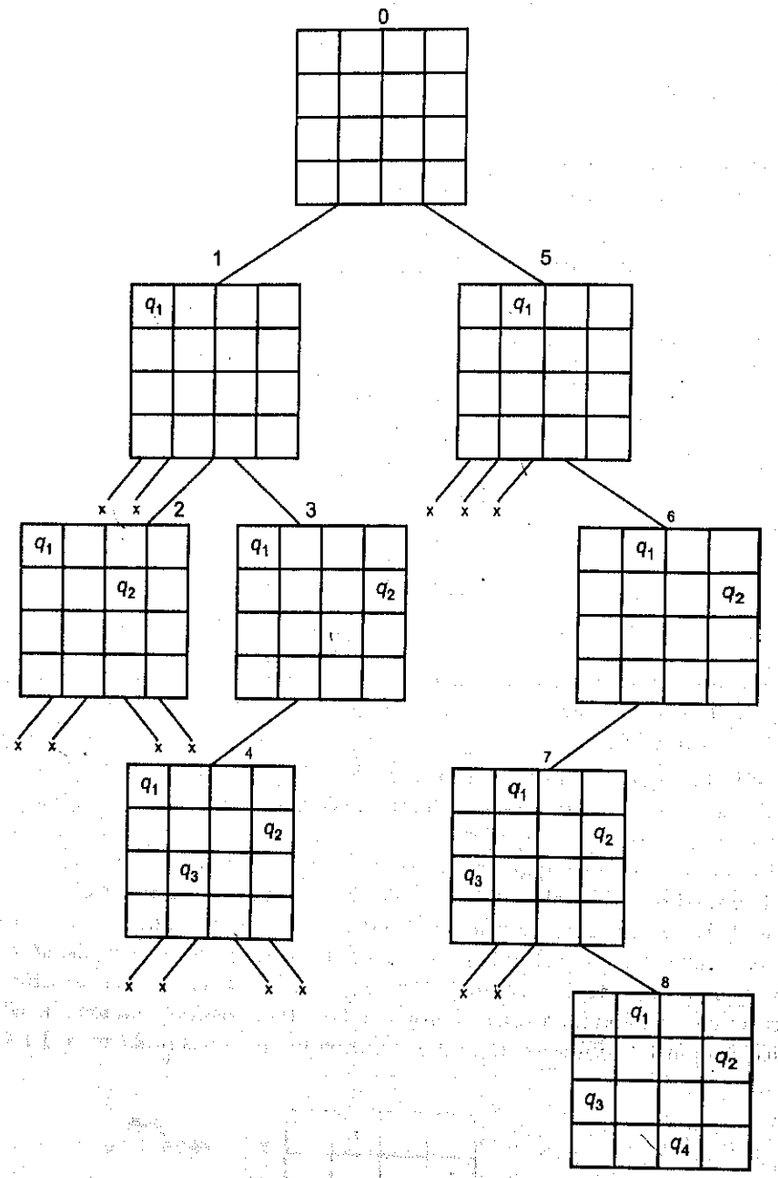


Figure 23.5

Fig.23.6 shows the complete state space for 4-queens problem. But we can use backtracking method to generate the necessary node and stop if next node violates the rule *i.e.*, if two queens are attacking.

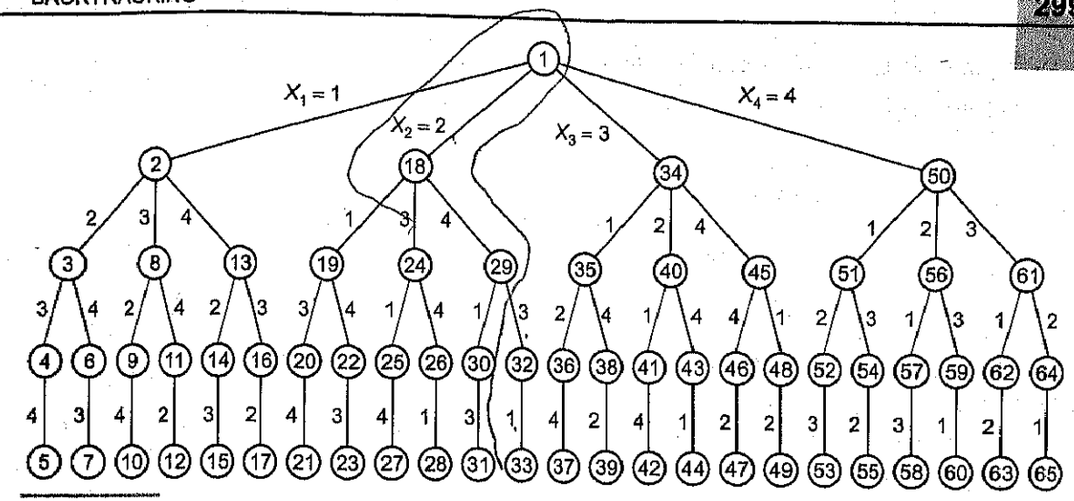


Figure 23.6 **4-queens solution space with nodes numbered in DFS**

It can be seen that all the solutions to the 4-queens problem can be represented as 4-tuples (x_1, x_2, x_3, x_4) where x_i represents the column on which queen "q_i" is placed.

One possible solution for 8 queens problem is shown in the Fig. 23.7

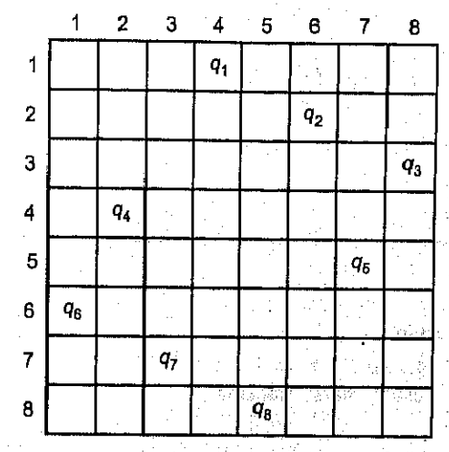


Figure 23.7

Thus, solution for 8-queen problem for $\langle 4,6,8,2,7,1,3,5 \rangle$.

If two queens are placed at positions (i, j) and (k, l).

Then, they are on the same diagonal only if $(i-j) = k-l$ or $i+j = k+l$.

Then *first* equation implies that $j-l = i-k$

The *second* equation implies that $j-l = k-i$

Therefore, two queens lie on the same diagonal if and only if $|j-l| = |i-k|$

Place (k, i) returns a boolean value that is true if the kth queen can be placed in column i. It tests both whether i is distinct from all previous values $x_1, x_2 \ldots x_{k-1}$ and whether there is no other queen on the same diagonal.

Using place, we give a precise solution to the n-queens problem.

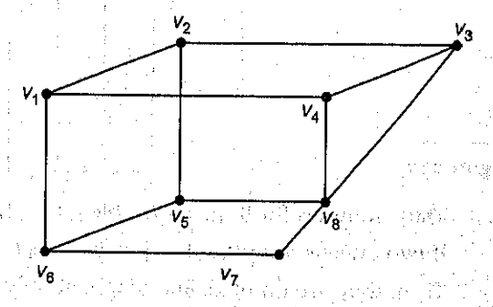
```
Place (k,i)
    {
        For j ← 1 to k − 1
            do if ( x[j] = i)
                or (Abs (x[j]) − i) = (Abs (j − k))
            then return false ;
        return true ;
    }
```

Place (k, i) returns true if a queen can be placed in the kth row and ith column otherwise returns false.

$x[\,]$ is a global array whose first $k-1$ values have been set. $Abs(r)$ returns the absolute value of r.

```
N-Queens (k, n)
  {
      for i ← 1 to n
          do if Place (k,i) then
              { x[k] ← i ;
                  if (k = n) then
                      write (x[1..n)) ;
                  else
                      N-Queens (k + 1, n) ;
              }
  }
```

Exercise



1. For a graph shown, find the Hamiltanian circuit using back tracking technique.
2. Given a set $S = \{1,3,4,5\}$ and $X = 8$, we have to find subset-sum using back tracking approach.
3. Find all the possible solutions for the
   (a) 4×4 chessboard, 4 queens problem
   (b) 8×8 chessboard, 8 queens problem.
4. Run the n-queens program for $n = 8, 9$ and 10. Find the number of solutions your program finds for each value of n.
5. Design a back tracking algorithm for the Hamiltanian cycle problem.
6. Design an algorithm for the n-colouring problem, considering back tracking technique.
7. For which values of n does the n queens problem have no solutions ? Prove your answer.

CHAPTER

24

Branch and Bound

24.1 Introduction

Branch and bound is a systematic method for solving optimization problems. Branch and bound is a rather general optimization technique that applies where the greedy method and dynamic programming fail. However, it is much slower. Indeed, it often leads to exponential time complexities in the worst case. On the other hand, if applied carefully, it can lead to algorithms that run reasonably fast on average. This method was first proposed by **A. H. Land** and **A. G. Doig** in 1960 for linear programming .

Branch-and-bound procedure requires two tools. The first one is a way of covering the feasible region by several smaller feasible sub-regions (ideally, splitting into sub-regions). This is called **branching**, since the procedure may be repeated recursively to each of the sub regions and all produced sub-regions naturally form a tree structure, called search tree or branch-and-bound-tree or something similar. Its nodes are the constructed sub-regions. Another tool is **bounding**, which is a fast way of finding upper and lower bounds for the optimal solution within a feasible sub-region.

The Branch and bound technique like **Backtracking** explores the implicit graph and deals with the optimal solution to a given problem. In this technique at each stage we calculate the bound for a particular node and check whether this bound will be able to give the solution or not .If we find that at any node the solution so obtained is appropriate but the remaining solution is not leading to a best case then we leave this node without exploring. Depth-first search or breadth-first

search is used for calculating the bound. At each stage we have for each node a bound – lower bound for minimization problem and upper bound for maximization problem. For each node, bound is calculating by means of partial solution. The calculated bound is checked with previous best result and if found that new partial solution results leads to worse case, than bound with the best solution so far, is selected and we leave this part without exploring it further. Otherwise the checking is done with the previous best result obtained so far, every partial solution while exploring.

The backtracking algorithm is effective for decision problem, but it is not designed for optimization problems, where we also have a cost function $f(x)$ that we wish to minimize or maximize. The branch and bound design has all the elements of backtracking, except that rather than simply stopping the entire search process any time a solution is found, we continue processing until the best solution is found. In addition, the algorithm has a scoring mechanism to always choose the most promising configuration to explore in each iteration. Because of this approach, branch and bound is sometimes called a **best-first search** strategy. Starting by considering the root problem (the original problem with the complete feasible region), the lower-bounding and upper-bounding procedures are applied to the root problem. If the bounds match, then an optimal solution has been found and the procedure terminates. Otherwise, the feasible region is divided into two or more regions ; these sub-problems partition the feasible region. The algorithm is applied recursively to the sub-problems. If an optimal solution is found to a sub-problem, it is a feasible solution to the full problem, but not necessarily globally optimal. If the lower bound for a node exceeds the best-known feasible solution, no globally optimal solution can exist in the subspace of the feasible region represented by the node. Therefore, the node can be removed from consideration. The search proceeds until all nodes have been solved or pruned, or until some specified threshold is met between the best solution found and the lower bounds on all unsolved sub-problems.

The core of this approach is a simple observation that (for a minimization task) if the lower bound for a sub-region A from the search tree is greater than the upper bound for any other (previously examined) sub region B, then A may be safely discarded from the search. This step is called **pruning**. It is usually implemented by maintaining a global variable 'm' that records the minimum upper bound seen among all sub-regions examined so far ; any node whose lower bound is greater than 'm' can be discarded.

It may happen that the upper bound for a node matches its lower bound ; that value is then the minimum of the function within the corresponding sub-region. Sometimes there is a direct way of finding such a minimum. In both these cases it is said that the node is solved. Note that this node may still be pruned as the algorithm progresses.

Ideally the procedure stops when all nodes of the search tree are either pruned or solved. At that point, all non-pruned sub-regions will have their upper and lower bounds equal to the global minimum of the function. In practice the procedure is often terminated after a given time ; at that point, the minimum lower bound and the maximum upper bound, among all non-pruned sections, define a range of values that contains the global minimum. Alternatively, within an overriding time constraint, the algorithm may be terminated when some error criterion such as (max – min)/(min + max) falls below a specified value.

The efficiency of the method depends critically on the effectiveness of the branching and bounding algorithms used; bad choices could lead to repeated branching, without any pruning,

until the sub-regions become very small. In that case the method would be reduced to an exhaustive enumeration of the domain, which is often impractically large. There is no universal bounding algorithm that works for all problems, and there is little hope that one will ever be found ; therefore the general paradigm needs to be implemented separately for each application, with branching and bounding algorithms that are specially designed for it.

Branch and bound methods may be classified according to the bounding methods and according to the ways of creating/inspecting the search tree nodes. This approach is used for a number of NP-hard problems, such as

- Knapsack problem
- Assignment problem
- Integer programming
- Nonlinear programming
- Travelling salesman problem (TSP) etc.

Backtracking algorithms determine problem solutions by systematically searching the solution space for the given problem instance. This search is facilitated by using a tree **organization** for the **solution** space. For a given solution space many tree organizations may be possible. Figure 23.6 shows a possible tree organization. A tree such as this is called a **permutation tree**.

At this point it is useful to develop some terminology regarding tree organizations of solution spaces. Each node in this tree defines a **problem state**. All paths from the root to other nodes define the **state space** of the problem. **Solution states** are those problem states 'S' for which the path from the root to 's' defines a tuple in the solution space. In Figure 23.6 only leaf nodes are solution states.

Answer states are those solution states 's' for which the path from the root to 'S' defines a tuple that is a member of the set of solutions. The tree organization of the solution space is referred to as the **state space tree**.

24.2 Live Node, Dead Node and Bounding Functions

Once a state space tree has been conceived of for any problem, the problem can be solved by systematically generating the problem states, determining which of these are solution states, and finally determining which solution states are answer states.

There are two fundamentally different ways to generate the problem states. Both of these begin with the root node and generate other nodes.

A node which has been generated and all of whose children have not yet been generated is called a **live node**. The live node whose children are currently being generated is called the E-node (node being expanded). A ***dead node*** is a generated node which is not to be expanded further or all of whose children have been generated. In both methods of generating problem states, we have a list of live nodes. In the first of these two methods as soon as a new child C of the current E-node R is generated, this child will become the new E-node. Then R will become the E-node again when the subtree C has been fully explored. This corresponds to a depth first generation of the problem states. In the second state generation method, the E-node remains the E-node until it is dead. In both methods, ***bounding functions*** are used to kill live nodes without generating all their children.

This is done carefully enough that at the conclusion of the process at least one answer node is always generated or all answer nodes are generated if the problem requires us to find all solutions. Depth first node generation with bounding functions is called *backtracking*. State generation methods in which the *E*-node remains the *E*-node until it is dead lead to **branch-and-bound** methods.

The term branch and bound refers to all state space search methods in which all children of the E-node are generated before any other live node can become the E-node. In branch-and-bound terminology, a BFS-like state space search will be called FIFO (First In First Out) search as the list of live nodes is a first-in-first-out list (or queue). A D-search like state space search will be called LIFO (Last In First Out) search as the list of live nodes is a last-in-first-out list (or stack). As in the case of backtracking, bounding functions are used to help avoid the generation of subtrees that do not contain an answer node.

24.3 FIFO Branch-and-Bound Algorithm

Let us see how a FIFO branch-and-bound algorithm would search the state space tree for the 4-queens problem.

- Initially, there is only one live node, node 1. This is the case when no queen has been placed on the chessboard.
- This node becomes the *E*-node.
- Expand and generate all its children, nodes 2, 18, 34 and 50 are generated.
- These nodes represent a chessboard with queen 1 in row 1 and columns 1, 2, 3 and 4 respectively.
- The only live nodes now are nodes 2, 18, 34 and 50. If the nodes are generated in this order, then the next *E*-node is node 2.
- It is expanded and nodes 3, 8 and 13 are generated.
- Node 3 is immediately killed using the bounding function. As a bounding function, we use the obvious criteria that if $(x_1, x_2, \ldots\ldots x_i)$ is the path to the current *E*-node, then all children nodes with parent-child labelings x_{i+1} are such that $(x_1, x_2, \ldots\ldots x_{i+1})$ represents a chessboard configuration in which no two queens are attacking.
- Nodes 8 and 13 are added to the queue of live nodes and node 18 becomes the next *E*-node.
- Nodes 19, 24 and 29 are generated. Nodes 19 and 24 are killed as a result of the bounding functions.
- Node 29 is added to the queue of live odes.
- The *E*-node is node 34.
- Figure 24.1 shows the portion of the tree of Figure 23.6 that is generated by a FIFO branch and bound search.
- In Figure, nodes that are killed as a result of the bounding function have a "*B*" under them and the numbers outside the nodes give the order in which the nodes are generated by FIFO branch and bound.
- Compare with backtracking algorithm indicates that backtracking is a superior method for this search problem.

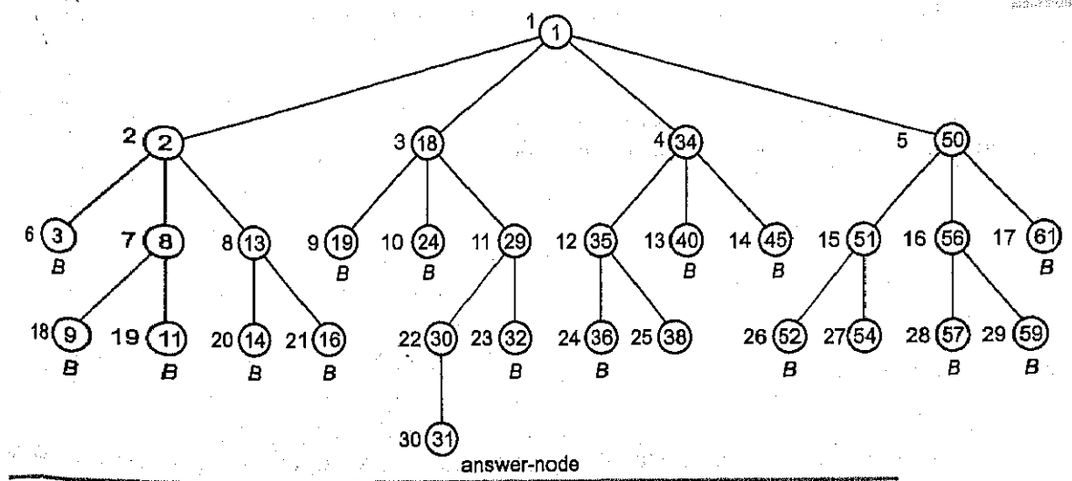


Figure 24.1 Portion of 4-queens state space tree generated by FIFO branch and bound.

24.4 Least Cost (LC) Search

In both LIFO and FIFO branch-and-bound the selection rule does not give preference to nodes that will lead to answer quickly but just queues those behind the current live nodes. In 4-queen problem, if three queens have been placed on the board, it is obvious that the answer may be reached in one more move. The rigid selection rule requires that other live nodes be expanded and then, the current node be tested.

The search for an answer node can often be speeded by using an "intelligent" ranking function $\hat{C}(.)$ for live nodes. The next *E*-node is selected on the basis of this ranking function.

The ideal way to assign ranks would be on the basis of the additional computational effort (or cost) needed to reach an answer node from the life node. For any node 'x', this cost could be :

1. The number of nodes in the subtree x that need to be generated before an answer node is generated.
2. The number of levels the nearest answer node is from 'x'.

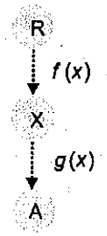
Using cost measure 2, the cost of the root of the tree of figure 24.1 to 4 (node 31 is four levels from node 1) and the costs of nodes 18 and 34, 29 and 35 and 30 and 38 are respectively 3, 2, and 1. The costs of all remaining nodes on levels 2, 3 and 4 are respectively greater than 3, 2 and 1. Using these costs as a basis to select the next *E*-node, the *E*-nodes are nodes 1, 18, 29 and 30 (in that order). The only other nodes to get generated are nodes 2, 34, 50, 19, 24, 32 and 31. If cost measure 1 is used, then the search would always generate the minimum number of nodes every branch-and-bound type algorithm must generate. If cost measure 2 is used then the only nodes to become E-nodes are the nodes on the path from the root to the nearest answer node.

The difficulty with using either of these ideal cost functions is that computing the cost of a node usually involves a search of the subtree x for an answer node.

Let x be a node in the state space tree and the cost function of x is

$$C(x) = f(x) + g(x)$$

where $f(x)$ is the real cost of traversing from the root to x and is a non-decreasing function and $g(x)$ is an estimated cost from node x to an answer node.



Let $\hat{g}(x)$ be an estimate of the additional effort needed to reach an answer node from x. Node x is assigned a rank using a function $\hat{c}(.)$ such that

$$\hat{c}(x) = f(h(x)) + \hat{g}(x)$$

where $h(x)$ is the cost of reaching x from the root and $f(.)$ is any non-decreasing function.

A search strategy that uses a cost function $\hat{c}(x) = f(h(x)) + \hat{g}(x)$ to select the next E-node would always choose for **its next E-node a live node with least $\hat{c}(.)$**. Hence, such a search strategy is called an LC-search (Least Cost search). It is interesting to note that BFS and D-search are special cases of LC-search. If we use $\hat{g}(x) \equiv 0$ and $f(h(x)) =$ level of node x, then a LC-search generates nodes by levels. This is essentially the same as a BFS. If $f(h(x)) \equiv 0$ and $\hat{g}(x) \geq \hat{g}(y)$ whenever y is a child of x, then the search is essentially a D-search. An LC-search coupled with **bounding functions** is called an LC branch-and-bound search.

24.5 The 15-Puzzle : An Example

The 15-puzzle consists of 15 numbered tiles on a square frame with a capacity of 16 tiles. We are given an initial arrangement of the tiles and the objective is to transform this arrangement into the goal arrangement through a series of legal moves. The only legal moves are ones in which a tile adjacent to the empty spot (ES) is moved to ES.

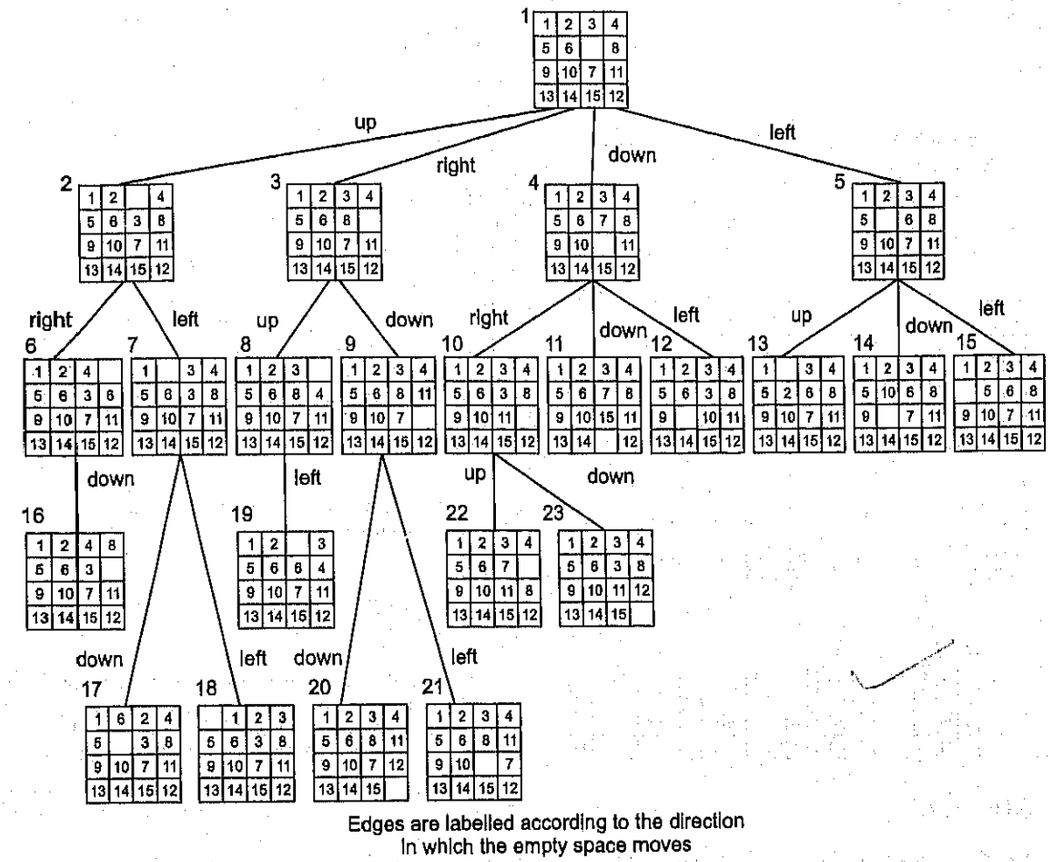
1	3	4	15
2		5	12
7	6	11	14
8	9	10	13

(*a*) An arrangement

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

(*b*) Goal arrangement

From the initial arrangement four moves are possible. We can move any one of the tiles numbered 2, 3, 5 or 6 to the empty spot. Following this move, other moves can be made.



Edges are labelled according to the direction in which the empty space moves

Figure 24.2 Part of the state space tree for the 15-puzzle.

Branch-and-Bound Method

- $c(x) = f(x) + g(x)$, where $f(x)$ is the length of the path from the root to node x and $g(x)$ is an estimate of the length of a shortest path from x downward to a goal node.
- $g(x) =$ number of non-blank tiles not in their goal position.
- Initially, the bound is set to infinity $B \leftarrow \infty$.
- Then whenever an answer is reached, the bound is compared.
- If the cost of the new answer is less than the bound, then the bound is replaced by the new smaller cost.
- If the estimated cost $c(x) < B$, then x is bounded. This is because the cost traversing from the root via x to goal node must be greater than or equal to B. Thus, x cannot lead to better solution than B.

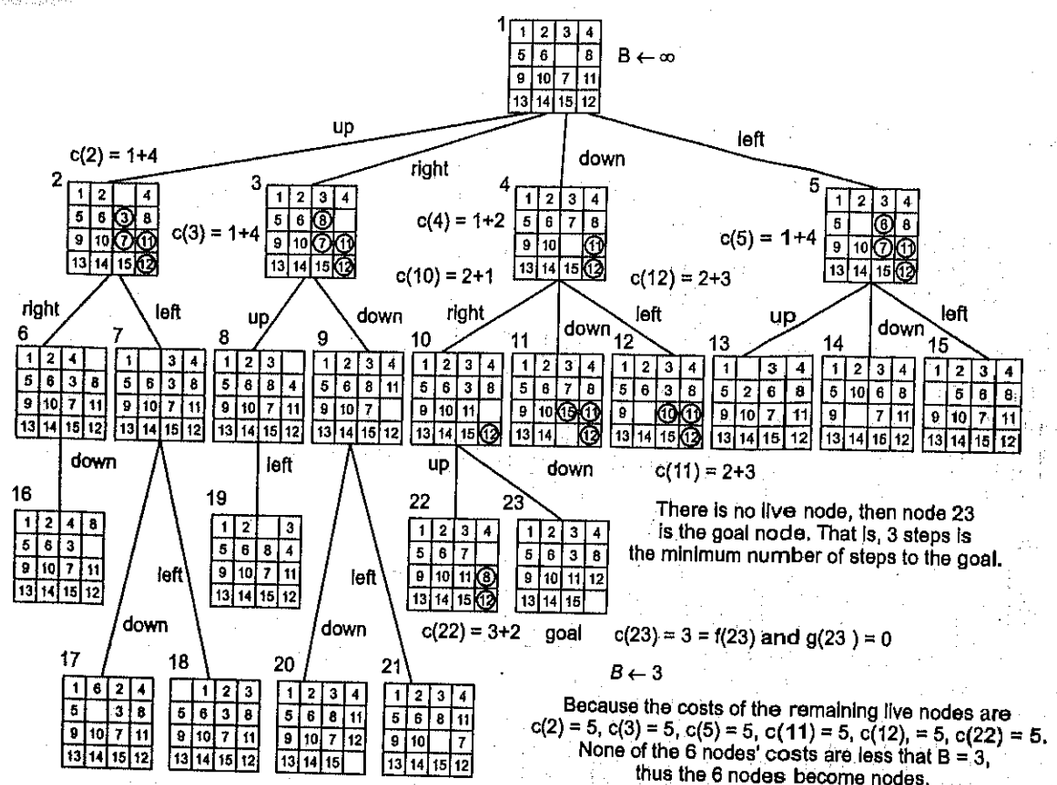


Figure 24.3

24.6 Branch-and-Bound Algorithm for TSP

Now, let us consider how branch and bound can be applied to solve the optimization version of the travelling salesman (TSP) problem. In the optimization version of this problem, we are given a graph G with a cost function $c(e)$ defined for each edge e in G, and we want to find the smallest cost that visits every vertex in G and returning back to its starting vertex.

We can design an algorithm for TSP by computing for each edge $e=(u,v)$, the minimum-cost path that begins at u and ends at v while visiting all other vertices in G along the path. To find such a path, we apply branch and bound technique. We generate the path from u to v in $G-\{e\}$ by augmenting a current path by one vertex in each loop of the branch-and-bound algorithm.

- After we have build a partial path P starting with vertex u, we only consider augmenting P with vertices not in P.
- We can classify a partial path P with "dead end" if the vertices not in P are disconnected in $G-\{e\}$.
- To define lower bound *i.e.*, lb, we can use total cost of the edges in $P+c(e)$.

This will certainly be a lower bound for any tour that will be built from e and p.

After we have run the algorithm for one edge e in G, we can use the best path found so far over all tested edges than restarting the current best solution.

24.7 Knapsack Problem

In knapsack problem, we have to fill the knapsack of capacity W, with a given set of items $I_1, I_2, \ldots I_n$ having weight $w_1, w_2, w_3, \ldots w_n$ in such a manner that the total weight of the items should not exceed the knapsack capacity and the maximum possible value can be obtained.

By using Branch and Bound technique, we have a bound that none of the items can have total sum more than the capacity of knapsack and must give the maximum possible value. The implicit tree for this problem is constructed as a binary tree, where left branch signifies the inclusion of the item and the right branch signifies exclusion. The node structure is as follows it contains three parts, the first part indicates the total weight of the item, the second part indicates the value of the current item and the third part indicates the upper bound for the node *i.e.*,

wt.	value
ub	
node	

The upper bound ub of the node can be computed as

$$ub = v + (W - w)(v_{i+1} / w_{i+1})$$

where v is the value of the current node.

W is the total weight *i.e.*, knapsack capacity

w is the wt. of current node.

Example. *Consider three items along with their respective weights and values as*

$$I = \langle I_1, I_2, I_3 \rangle; \quad w = \langle 5,4,3 \rangle; \quad v = \langle 6,5,4 \rangle$$

the knapsack has the maximum capacity $W=7$, we have to pack this knapsack using the branch and bound technique so as to give the maximum possible value while considering all constraints.

Solution.

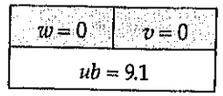
Items	w_i	v_i	v_i / w_i
I_1	5	6	1.2
I_2	4	5	1.25
I_3	3	4	1.3

i.e., calculate value per weight ratio. Now, we arrange the table according to its descending values.

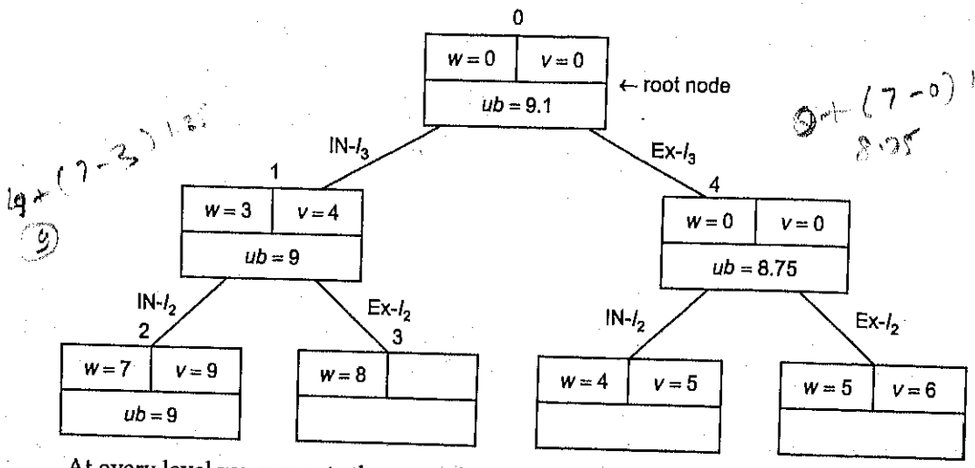
Items	w_i	v_i	v_i / w_i
I_3	3	4	1.3
I_2	4	5	1.25
I_1	5	6	1.2

Now, we start with root node, the upper bound for the root node can be computed as

$$ub = 0 + (7-0) * 1.3 \qquad (v=0,\ w=0,\ v_1 / w_1 = 1.3)$$
$$= 9.1$$



Next, we include item I_3, which is indicated by the left branch and exclude item I_3 which is indicated by right branch.



At every level we compute the upper bound and explore the node while selecting the item. Finally the node with maximum upper bound is selected as an optimum solution.

In this example node with I_3 and I_2 gives the optimum solution *i.e.*, maximum value is 9.

24.8 Assignment Problem

In assignment problem 'n' people are assigned 'n' tasks. We have to find the minimum total cost of assignment where each person has exactly one task to perform. For solving this problem we use n by n cost matrix.

Given cost matrix C, where C_{ij} refers to the minimum cost for person 'P_i' to perform task 't_j', $1 \le i \le n$ and $1 \le j \le n$, then the problem is to assign person to tasks so as to minimize the total cost of performing the 'n' tasks *i.e.*, choose the element from the matrix in such a manner that no two elements are in the same column and total cost should be minimum.

Example. *Given a cost matrix for three persons P_1, P_2, P_3 which are assigned following tasks.*

$$C = \begin{array}{c|ccc} & T_1 & T_2 & T_3 \\ \hline P_1 & 6 & 9 & 5 \\ P_2 & 4 & 8 & 3 \\ P_3 & 5 & 11 & 6 \end{array}$$

Obtain the minimum total cost of assignment using Branch and Bound technique where each person has exactly one task to perform.

Solution. It is noticeable that the cost of any solutions, including the optimal one, cannot be smaller than the sum of the minimum elements present in each row of the cost matrix.

In this example the sum of minimum elements present in each row is $5+3+5=13$. This sum becomes the lower bound for our example.

In branch and bound, the node currently having the best possible solution is selected and explored. We select the node for exploring by comparing the lower bounds. The node with the minimum lower bound is selected and explored.

We start from the root which signifies that no assignment is yet done. The node structure contains two parts. In first part the information about the assignments is stored and in the second part lower bound stores.

For instance,

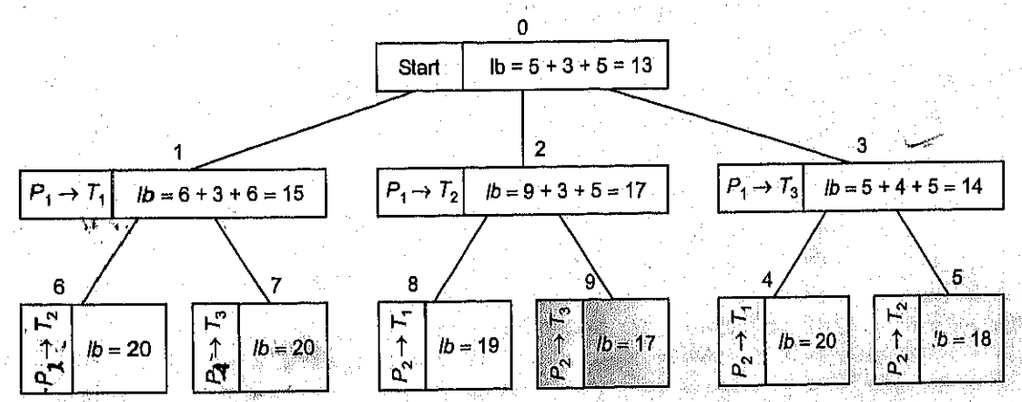
$P_1 \to T_3$	$lb = 14$

$$lb = 5+4+5 = 14$$

In the above node, P_1 is assigned task T_3 and possible lower bound is 14 *i.e.*, $5+4+5$.

The first level of the tree corresponds to 3 tasks. Let us assume that these tasks are assigned to person P_1. Then, we calculate the possible lower bound and select the node having minimum lower bound.

Thus, we select the node where task T_2 is assigned to P_1



Implicit graph for the instance of assignment problem.

Now, we explore this node and can see that P_2 can be assigned T_1 or T_2. It can be seen that lower bounds of these nodes exceed the lower bounds of other nodes. So we explore it and calculate the optimal lower bound. After examining each node, we obtain node labeled '9' having the optimal solution *i.e.*,

P_1 is assigned T_2, P_2 is assigned T_3 and P_3 is assigned T_1.

Thus the total cost $= 9+3+5 = 17$.

Exercise

1. Use Branch and Bound to solve the assignment problems with the following cost matrices :

	1	2	3	4
a	94	1	54	68
b	74	10	88	82
c	62	88	8	76
d	11	74	81	21

2. Devise algorithms for the following using Branch and Bound techniques :
   (*a*) Assignment Problem
   (*b*) Knapsack Problem
   (*c*) Travelling Sales person problem.
3. Solve travelling sales person problem by using Branch and Bound technique.
4. Solve the following instance of the knapsack problem by using Branch and Bound technique.

Items	*w*	*v*
I_1	9	15
I_2	6	6
I_3	7	5
I_4	2	1

CHAPTER

25

Amortized Analysis

25.1 Introduction

An ***amortized analysis*** is any strategy for analyzing a sequence of operations to show that the average cost per operation is small, even though a single operation within the sequence might be expensive. It refers to finding the average running time per operation over a worst-case sequence of operations

This is not an algorithmic technique. It's just a more *accurate* technique for analyzing algorithms *under some conditions.*

Amortized analysis differs from average-case performance in that probability is not involved ; amortized analysis guarantees the time per operation over worst-case performance. The basic idea is that a worst case operation can alter the state in such a way that the worst case cannot occur again for a long time, thus "amortizing" its cost.

Notice that average-case analysis and probabilistic analysis are not the same thing as amortized analysis. In average-case analysis, we are averaging over all possible inputs ; in probabilistic analysis, we are averaging over all possible random choices ; in amortized analysis, we are averaging over a sequence of operations. Amortized analysis assumes worst-case input and typically does not allow random choices.

There are several techniques used in amortized analysis :

- **Aggregate analysis** determines the upper bound $T(n)$ on the total cost of a sequence of n operations, then calculates the average cost to be $T(n)/n$.
- **Accounting method** determines the individual cost of each operation, combining its immediate execution time and its influence on the running time of future operations. Usually, many short-running operations accumulate a "debt" of unfavorable state in small increments, while rare long-running operations decrease it drastically.
- **Potential method** is like the accounting method, but overcharges operations early to compensate for undercharges later.

25.2 Aggregate Analysis

The aggregate method, though simple, lacks the exactness of the other two methods. In particular, the accounting and potential methods allow a specific ***amortized cost*** to be allocated to each operation.

EXAMPLE : *Stack operations with MULTIPOP*

The push operation inserts an element to the top of the stack and POP operation deletes an element from the top of the stack. Here, we are augmenting the stack data structure with a new operation "MULTIPOP"

```
MULTIPOP (S, k)
    While not Stack empty (S) and k ≠ 0
        do pop (S)
        k ← k − 1
```

Here a stack S and number k, let's add the operation MULTIPOP which removes the top k objects on the stack. Multipop just calls Pop either k times or until the stack is empty.

Example. *What is the running time of Multipop(S, k) on a stack of x objects ?*

Solution. The cost is $\min(s, k)$ pop operations.

If there are n stack operations, in the worst case, a single Multipop can take $O(n)$ time. Let's analyze a sequence of n push, pop, and multipop operations on an initially empty stack. The worst case cost of a multipop operation is $O(n)$ since the stack size is at most n, so the worst case time for any operation is $O(n)$. Hence a sequence of n operations costs $O(n^2)$.

This analysis is technically correct, but overly pessimistic while some of the multipop operations can take $O(n)$ time, not all of them. We need some way to average over the entire sequence of n operations. In fact, the total cost of n operations on an initially empty stack is $O(n)$. Why ? Because each object can be popped at most once for each time that it is pushed. Hence the number of times POP (including calls within multipop) can be called on a nonempty stack is at most the number of push operations which is $O(n)$.

In aggregate analysis, for any value of n, any sequence of n Push, Pop and Multipop operations on an initially empty stack takes $O(n)$ time. The average cost of an operation is thus $O(n)/n = O(1)$. Thus all stack operations have an amortized cost of $O(1)$.

25.3 Accounting Method or Taxation Method

In this we assign different charge to different operations. The operations may be charged more or less than their actual cost. The charging amount of the operation is referred as its *amortized cost*. When amortized cost > actual cost the difference is saved in specific objects as *'Credits'*. These credits are used when the operation charge is less than the actual cost. The total credit store in the data structure is not allowed to be negative.

The sum of the amortized costs for any sequence of operations must be upper bound for the actual total cost of these operations.

If we denote the actual cost of the ith operation by c_i and its amortized cost by $\hat{c}_i$

$$\sum_{i=1}^{n} c_i \le \sum_{i=1}^{n} \hat{c}_i$$

for all sequence of n operation.

EXAMPLE : *Stack Operations*

The actual costs of the operations are :

Push : 1
Pop : 1
Multipop : $\min(k, S)$ where k is the argument supplied to multipop and 'S' is the stack size when it is called.

Let us assign the following amortized costs :

Push \$2
Pop \$0
Multipop \$0

Note that the amortized cost of Multipop is a constant (0), whereas the actual cost is variable.

Then each time we do a push, we spend one dollar of the tax to pay for the push and then save the other dollar of the tax to pay for the inevitable pop or multipop of that item. Note that if we do n operations, the total amount of taxes we collect is then $2n$.

When we do a push, we use 1 dollar of the tax to pay for the push and then store the extra dollar with the item that was just pushed on the stack. Then all items on the stack will have one dollar stored with them. Whenever we do a pop, we can use the dollar stored with the item popped to pay for the cost of that pop. Morever, whenever we do a multipop, for each item that we pop off in the multipop, we can use the dollar stored with that item to pay for the cost of popping that item.

We've shown that we can use the 2 tax on each item pushed to pay for the cost of all pops, pushes and multipops. Morever we know that this taxation scheme collects at most $2n$ dollars in taxes over n stack operation. Hence we've shown that the amortized cost per operation is $O(1)$.

Thus, for any sequence of n operations of push, pop and multipop the total amortized cost is an upper bound on the total actual cost. Since total amortized cost is $O(n)$, so it is the total actual cost.

25.4 Potential Method

Instead of representing prepaid work, as credit stored with specific objects in the data structure, the potential method of amortized analysis represent the prepaid work as *'potential energy'* or *'potential'*, that can be released to pay for future operations. The stored potential is associated with the entire data structure rather than specific objects within the data structure.

Notations

- D_0 is the initial data structure on which n operations are performed
- D_i is the data structure after the ith operation.
- c_i is the actual cost of the ith operation.
- Φ is a potential function maps each D_i value to its potential value $\Phi(D_i)$ such that $\Phi(D_0)=0$ and $\Phi(D_i)\geq 0$ for all i.

The **amortized cost** of the ith operation with respect to Φ is defined as

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

$$\hat{c}_i = c_i + \underbrace{\Phi(D_i) - \Phi(D_{i-1})}_{\textit{potential difference } \Delta\Phi_i}$$

The total amortized cost for n operation is

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1}))$$

Summing both sides

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1}))$$

$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

$$\geq \sum_{i=1}^{n} c_i \text{ since } \Phi(D_n)\geq 0 \text{ and } \Phi(D_0)=0$$

Example. *"If the set of stack operations included a MULTIPUSH operation, which pushes k items onto the stack, would the O(1) bound on the amortized cost of stack operations continue to hold ?"*

Solution. No, because one could perform n numbers of MULTIPUSH-k element-operations, costing $O(k)$ per operation, which totals $O(nk)$.

Example. *A sequence of n operations is performed on a data structure. The ith operation costs i if i is an exact power of 2 and 1 otherwise, use aggregate analysis to determine the amortized cost per operation.*

Solution. The ith operation costs i if $i=2^k$, otherwise 1.

Operation	*Real Cost*
1	1
2	2
3	1
4	4
5	1
6	1
7	1
8	8
9	1
⋮	⋮

$$T(n)=1+2+1+4+1+1+1+8+1+1+\ldots$$

$$=n+\sum_{i=0}^{\lfloor \lg n \rfloor} 2^i - \lfloor \lg n \rfloor$$

$$=n+2^{\lfloor \lg n \rfloor+1}-1-\lfloor \lg n \rfloor$$

$$<n+2n\ldots$$

$$<3n.$$

$$\text{The amortized cost per operation} = \frac{\text{total cost}}{\text{number of operations}}$$

$$< \frac{3n}{n} = 3$$

We can therefore claim that each operation is $O(1)$.

Exercise

1. A sequence of stack operations is performed on stack whose size never exceeds k. After every k operations, a copy of the entire stack is made for back up purpose. Show the cost of n stack operations, including copying the stack is $\theta(n)$ by assigning suitable amortized cost to various stack operations.
2. Explain the amortized cost analysis in splaying ?
3. Find the amortized cost of 8-bit Binary counter using potential method.
4. Obtain the amortized cost using accounting method and potential method for
   (*a*) Depth first search.
   (*b*) Breadth first search.
   (c) Minimum spanning tree.

CHAPTER

26

Elementary Graphs Algorithms

26.1 Introduction

Many problems are naturally formulated in terms of objects and the connections between them. *For example,*

- *Airline Routes.* What is the fastest (or cheapest) way to get from one city to another ?
- *Electrical circuits.* Circuit elements are wired together. How does the current flow ?
- *Job Scheduling.* The objects are tasks that need to be performed. The connections indicate which jobs should be done before which other jobs.

A **graph** is a mathematical object that describes such situations. A **graph** is a collection of vertices, V, and edges, E. An edge connects *two* vertices.

A **simple path** is a path with no vertex repeated. A **simple cycle** is a simple path except the first and last vertex is repeated and, for an undirected graph, number of vertices >=3. A **tree** is a graph with no cycles. A **complete graph** is a graph in which all edges are present. A **sparse graph** is a graph with relatively few edges. A **dense graph** is a graph with many edges. An **undirected**

graph has no specific direction between the vertices. A **directed graph** has edges that are "one way". We can go from one vertex to another, but not vice versa. A **weighted graph** has weights associated with each edge. The weights can represent distances, costs, etc.

26.2 How is a Graph Represented ?

26.2.1 Representing Graphs as an Adjacency List

An **adjacency list** is an array, which contains a list for each vertex. The list for a given vertex contains all the other vertices that are connected to the first vertex by a single edge.

26.2.2 Representing Graphs as an Adjacency Matrix

An **adjacency matrix** is a matrix with a row and column for each vertex. The matrix entry is 1 if there is an edge between the row vertex and the column vertex. The diagonal may be zero or one. Each edge is represented twice.

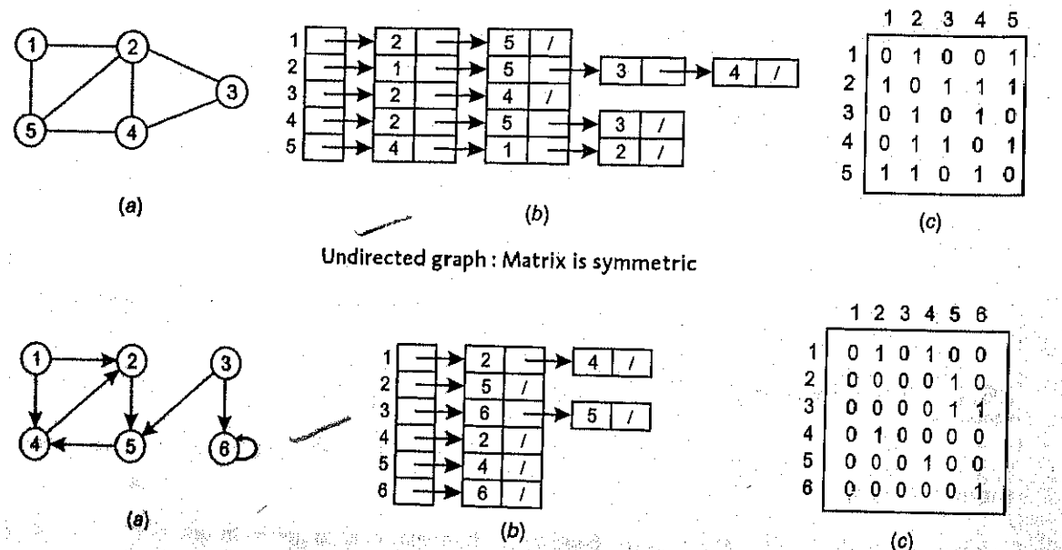


Figure 26.1

26.2.3 Lists vs. Matrices

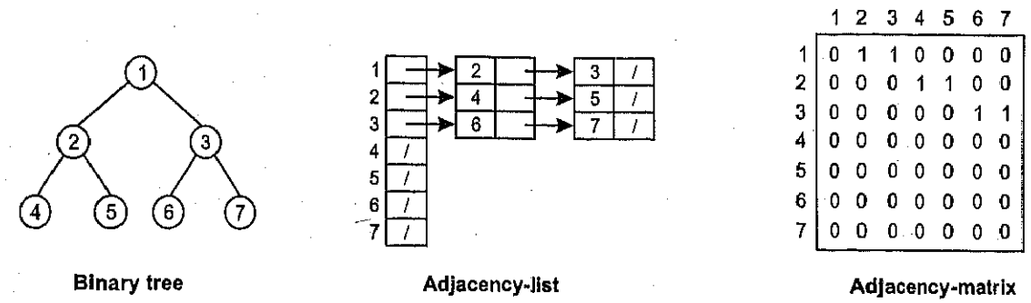
Speed. An adjacency matrix provides the fastest way to determine whether or not a particular edge is present in the graph. For an adjacency list, there is no faster way than to search the entire list for the presence of the edge.

Memory. Normally, an adjacency matrix requires more space (n^2 entries). However, if n is small and the graph is unweighted, an adjacency matrix only needs 1 bit per entry. The adjacency list requires at least 1 word per entry (for the address of the next node of the list). This may offset the advantage of fewer entries.

Example. *Give an adjacency-list representation for a complete binary tree on 7 vertices. Give an equivalent adjacency-matrix representation. Assume that vertices are numbered from 1 to 7 as in a binary heap.*

Solution.

Adjancency Matrix



Example. *Given an adjacency-list representation of a directed graph, how long does it take to compute the out-degree of every vertex ? How long does it take to compute the in-degree ?*

Solution. Given the adjacency-list representation of a graph the out and in-degree of every node can easily be computed in $O(E+V)$ time.

Example. *The transpose of a directed graph $G^T = (V, E^T)$, where $E^T = \{(v,u) \in V X V : (u,v) \in E\}$. Thus, G^T is G with all its edges reversed. Describe efficient algorithm for computing G^T from G, for both the adjacency-list and adjacency-matrix representations of G. Analyze the running times of your algorithms.*

Solution. The transpose of a directed graph G^T can be computed as follows : If the graph is represented by an adjacency-matrix A simply compute the transpose of the matrix A^T in time $O(V^2)$. If the graph is represented by an adjacency-list a single scan through this list is sufficient to construct the transpose. The time used is the $O(E+V)$.

Example. *The square of a directed graph $G=(V, E)$ is the graph $G^2 = (V, E^2)$ such that $(u,w) \in E^2$ if and only if for some $v \in V$, both $(u,v) \in E$ and $(v,w) \in E$. That is, G^2 contains an edge between u and w whenever G contains a path with exactly two edges between u and w. Describe efficient algorithms for computing G^2 from G for both the adjacency-list and adjacency-matrix representations of G. Analyze the running times of your algorithms.*

Solution. The square of a graph can be computed as follows: If the graph is represented as an adjacency matrix A simply compute the product A^2 where multiplication and addition is exchanged by and 'ing and or 'ing. Using the trivial algorithm yields a running time of $O(n^3)$. Using Strassen algorithm improves the bound to $O(n^{\lg 7}) = O(n^{2.81})$. If we are using the adjacency-list representation we can simply append lists eliminating duplicates. Assuming the lists are sorted we can proceed as follows : For each node v in some list replace the v with $Adj[v]$ and merge this into the list. Each list can be at most V long and therefore each merge operation takes at most $O(V)$ time. Thus the total running time is $O((E+V)V) = O(E+V^2)$.

```
for i = 1 to V
    for j = 1 to V
    {
        G²[i][j] = 0
        for k = 1 to V
            if (g[i][k] == 1 && g[k][j] == 1)
                {
                    G²[i][j] == 1 ;
                    break ;
                }
    }
```

For many applications, one must systematically search through a graph to determine the structure of the graph.

Two common elementary algorithms for tree searching are :

- **Breadth-first search (BFS)**, and
- **Depth-first search (DFS).**

Both of these algorithms work on directed or undirected graphs. Many advanced graph algorithms are based on the ideas of BFS or DFS. Each of these algorithms traverses edges in the graph, discovering new vertices as it proceeds. The difference is in the order in which each algorithm discovers the edge

26.3 Breadth First Search

Breadth first search trees have a nice property: Every edge of G can be classified into one of three groups. Some edges are in T themselves. Some connect two vertices at the same level of T. And the remaining ones connect two vertices on two adjacent levels. It is not possible for an edge to skip a level.

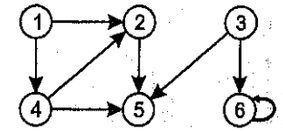
Therefore, the breadth first search tree really is a shortest path tree starting from its root. Every vertex has a path to the root, with path length equal to its level (just follow the tree itself), and no path can skip a level so this really is a shortest path

Analysis

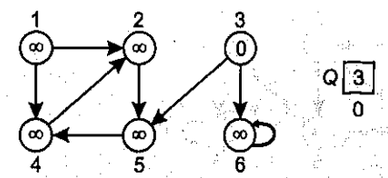
We use aggregate analysis. After initialization, no vertex is ever whitened, and thus the test in line 13 ensures that each vertex is enqueued at most once, and hence dequeued at most once. The operations of enqueuing and dequeuing take $O(1)$ time, so the total time devoted to queue operations is $O(V)$. Because the adjacency list of each vertex is scanned only when the vertex is dequeued, each adjacency list is scanned at most once. Since the sum of the lengths of all the adjacency list is $Q(E)$, the total time spent in scanning adjacency list is $O(E)$. The overhead for initialization is $O(V)$, and thus total running time of BFS is $O(V + E)$. Thus, BFS runs in linear time in the size of the adjacency-list representation of G.


```
BFS (G, s)
1.  for each vertex u ∈ V[G] − {s}
2.      do color[u] ← WHITE
3.          d[u] ← ∞
4.          π[u] ← NIL
5.  color[s] ← GRAY
6.  d[s] ← 0
7.  π[s] ← NIL
8.  Q ← ϕ
9.  ENQUEUE(Q, s)
10. while Q ≠ ϕ
11.     do u ← DEQUEUE(Q)
12.         for each v ∈ Adj[u]
13.             do if color[v] = WHITE
14.                 then color[v] ← GRAY
15.                     d[v] ← d[u] + 1
16.                     π[v] ← u
17.                     ENQUEUE(Q, v)
18.             color[u] ← BLACK
```

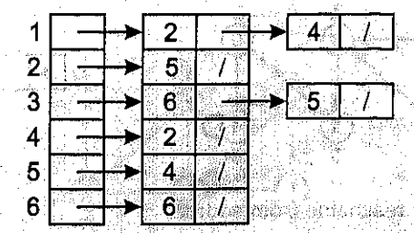
Example. *Consider the graph G in Fig. Describe the whole process of breadth first search. Using vertex 3 as the source.*



Solution.



First, we create adjacency-list representation.



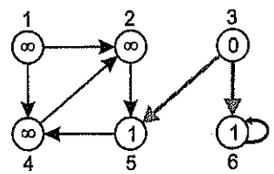
So, adj[4] = {6, 5} so color [6] = GRAY

color [5] = GRAY

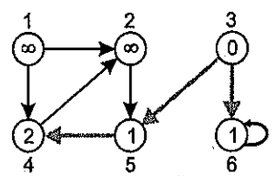
and $d[6] = 1$ $d[5] = 1$

$\pi[6] \leftarrow 3$ $\pi[5] \leftarrow 3$

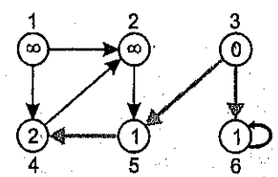
Now



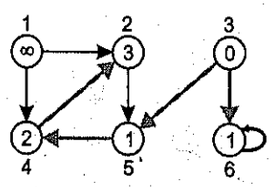
Q | 5 | 6
1 1



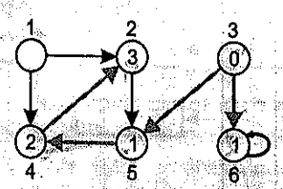
Q | 6 | 4
1 2



Q | 4 |
2



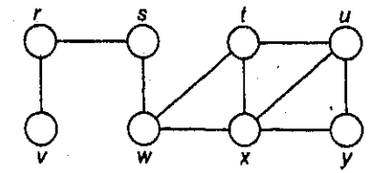
Q | 2 |
3



Q | ϕ |

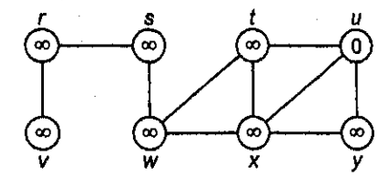
Thus vertex 1 cannot be reachable from source.

Example. *Consider the graph G shown in the Fig. Describe the whole process of BFS.*

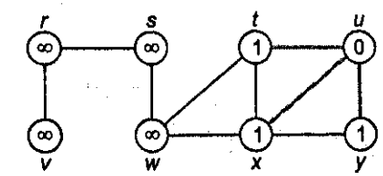


Using vertex u as the source.

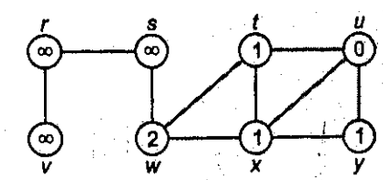
Solution.



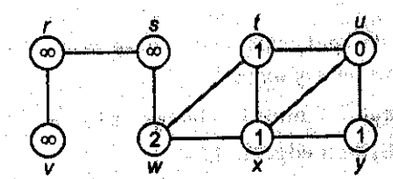
Q | u |
0



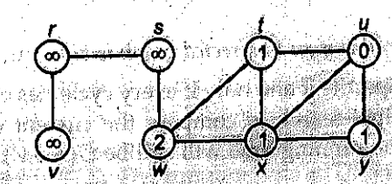
Q | t | x | y |
1 1 1



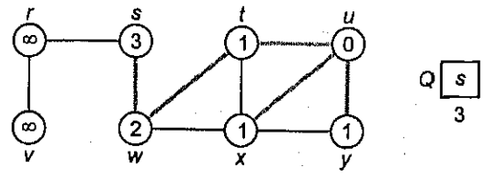
Q | x | y | w |
1 1 2

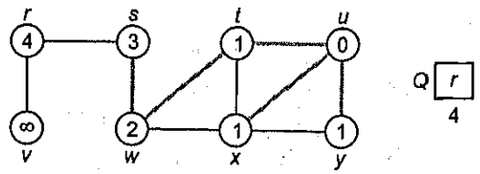


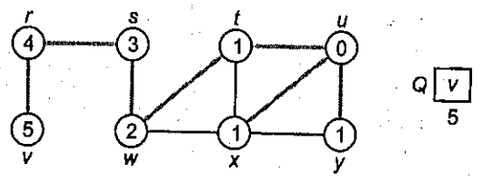
Q | y | w |
1 2

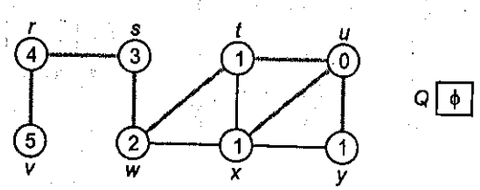


Q | w |
2









Example. *What is the running time of BFS if its input graph is represented by an adjacency matrix and the algorithm is modified to handle this form of input ?*

Solution. The line "for each $v \in \text{Adj}[u]$ ... " must be changed to

"for each $v \in V[G]$ do if edge$[u, v]$..."

and so the time used scanning for neighbor can increase to $O(V^2)$, yielding a total running time of $O(V^2 + V)$.

Example. *Give an efficient algorithm to determine if an undirected graph is bipartite.*

Solution. we know that a graph is bipartite if and only if every cycle has even length. Using this fact we can simply modify breadth-first-search to compare the current distance with the distance of an encountered gray vertex. The running time will still be $O(E+V)$.

26.4 Depth First Search

In depth-first search, edges are explored out of the most recently discovered vertex v that still has unexplored edges leaving it. When all of v's edges have been explored, the search backtracks to explore edges leaving the vertex from which v was discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex. If any undiscovered vertices remain, then one of them is selected as a new source and the search is repeated from that source. This entire process is repeated until all vertices are discovered.

As in the BFS, whenever a vertex v is discovered during a scan of the adjacency list of an already discovered vertex u, DFS records this by setting v's predecessor field $\pi[v]$ to u. In DFS predecessor subgraph is composed of several trees, because the search may be repeated from multiple sources.

In DFS, each vertex v has two timestamps: the first timestamp $d[v]$ *i.e.* discovery time records when v is first discovered *i.e.* grayed, and the second timestamp $f[v]$ *i.e.* finishing time records when the search finishes examining v's adjacency list *i.e.* blacked. For every vertex $d[u] < f[u]$.

```
DFS(G)
1.  for each vertex u ∈ V[G]
2.      do color[u] ← WHITE
3.          π[u] ← NIL
4.  time ← 0
5.  for each vertex u ∈ V[G]
6.      do if color[u] = WHITE
7.          then DFS-VISIT(u)
```

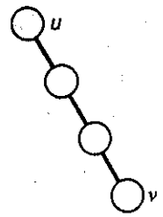
```
DFS-VISIT(u)
1.  color[u] ← GRAY                ▷ White vertex u has just been discovered.
2.  time ← time + 1
3.  d[u] ← time
4.  for each v ∈ Adj[u]            ▷ Explore edge (u,v)
5.      do if color[v] = WHITE
6.          then π[v] ← u
7.              DFS-VISIT (v)
8.  color[u] ← BLACK               ▷ Blacken u, it is finished.
9.  f[u] ← time ← time + 1
```

Analysis

In this we use aggregate analysis. The loops on lines 1-3 and lines 5-7 of DFS take time $\theta(V)$, exclusive the calls to DFS-VISIT. The DFS-Visit calls exactly once for each vertex $v \in V$. During an execution of DFS-VISIT, the loop on lines 4-7 is executed $|\text{Adj}[v]|$ times. The total cost of executing lines 4-7 of DFS-VISIT is $\theta(E)$. The running time of DFS is therefore $\theta(V+E)$.

An important property is that :

The **white path property** states that v is a descendant of u if and only if, at the time of discovery of u, there is a white path from u to v.



If v is a descendant of u, then at time $d[u]$, there must be at least one white path from u to v for the DFS algorithm to follow. Conversely, if there is a white path from u to v at time $d[u]$, then the last node on that path is a neighbor of v which has not yet been processed.

One useful aspect of the DFS algorithm is that it traverses connected components one at a time, and thus it can be used to identify the connected components in a given graph. As an example, if a handwriting sample is scanned into a pixelized image and DFS is performed on that image, all connected components below a certain threshold size (which represent unwanted dirt in the sample) can be removed.

Parenthesis Theorem

In any DFS of a graph $G=(V, E)$, for any *two* vertices u and v exactly one of the following holds :

- The intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ are entirely disjoint, and u nor v is a descendant of the other in the depth-first forest,
- The interval $[d[u], f[u]]$ is contained entirely within the interval $[d[v], f[v]]$, and u is a descendant of v in a depth-first tree, or
- The interval $[d[v], f[v]]$ is contained entirely within the interval $[d[u], f[u]]$, and v is a descendant of u in a depth-first tree.

Classification of Edges

DFS can be used to classify the edges of the input graph $G=(V, E)$. We can define *four* edge types in terms of the depth-first forest.

- **Tree edges.** edges in the depth-first forest.
- **Back edges.** (u,v) which v is an ancestor of u in a depth-first forest. Self-loop are also considered as back edges.
- **Forward edges.** nontree edges (u, v) which v is an descendant of u in depth-first forest.
- **Cross edges.** all other edges. They can go between vertices in the same depth-first tree or in different depth-first trees.

Edge (u,v) can be classified by the color of v

WHITE. tree edge
GRAY. back edge
BLACK. forward or cross edge

26.5 BFS and DFS in Directed Graphs

Although we have discussed them for undirected graphs, the same search routines work essentially unmodified for directed graphs. The only difference is that when exploring a vertex v, we only want to look at edges (v, w) going out of v ; we ignore the other edges coming into v.

For directed graphs, too, we can prove nice properties of the BFS and DFS tree that help to classify the edges of the graph. For BFS in directed graphs, each edge of the graph connects two vertices at the same level, goes down exactly one level, or goes up any number of levels. For DFS, each edge either connects an ancestor to a descendant, a descendant to an ancestor, or one node to a node in a previously visited sub tree. It is not possible to get "forward edges" connecting a node to a sub tree visited later than that node. We'll use this property to test if a directed graph is strongly connected (every vertex can reach every other one).

Example. *Make a 3-by-3 chart with row and column labels WHITE, GRAY, and BLACK. In each cell (i, j), indicate whether, at any point during a depth-first search of a directed graph, there can be an edge from a vertex of color i to a vertex of color j. For each possible edge, indicate what edge it can be. Make a second such chart for depth first search of an undirected graph.*

Solution. In the following table we indicate if there can be an edge (i, j) with the specified colors during a depth-first search. If the entry in the table is present we also indicate which type the edge might be: Tree, Forward or Back edge. For the directed case :

(i,j)	WHITE	GRAY	BLACK
WHITE	TBFC	BC	C
GRAY	TF	TFB	TFC
BLACK		B	TFBC

T : Tree edge ; F : Forward edge ; B : Black edge ; C ; Cross edge.

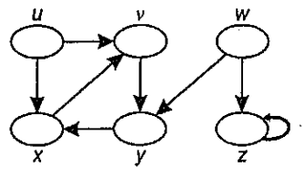
For undirected graph

(i,j)	WHITE	GRAY	BLACK
WHITE	TB	TB	
GRAY	TB	TB	TB
BLACK		TB	TB

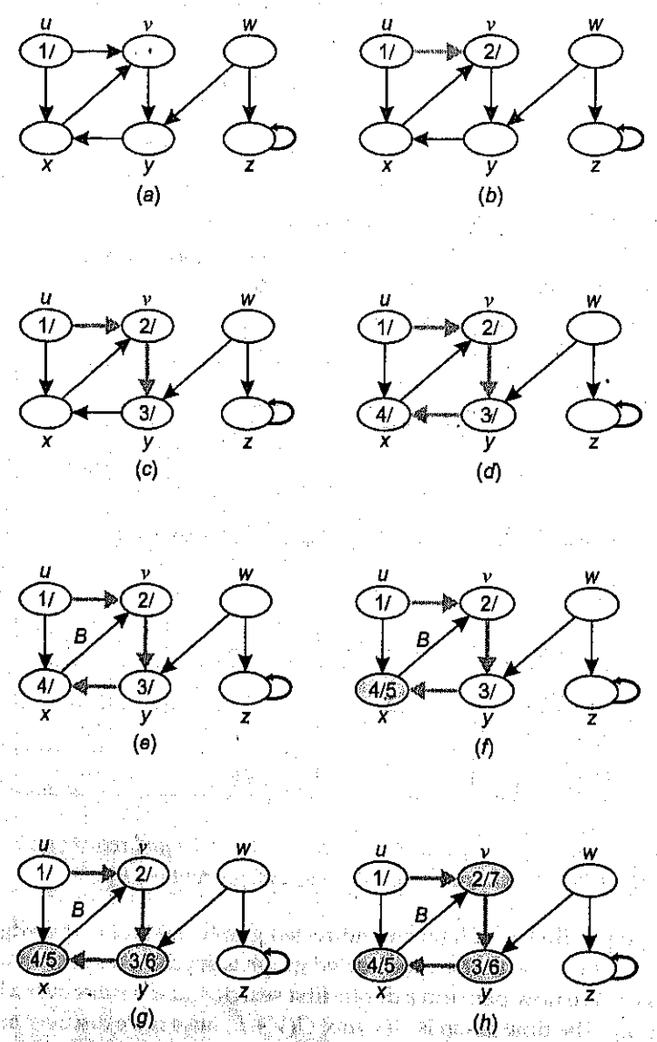
Example. *Give an algorithm that determines whether or not a given undirected graph $G=(V, E)$ contains a cycle, Algorithm should run in $O(V)$ time, independent of $|E|$.*

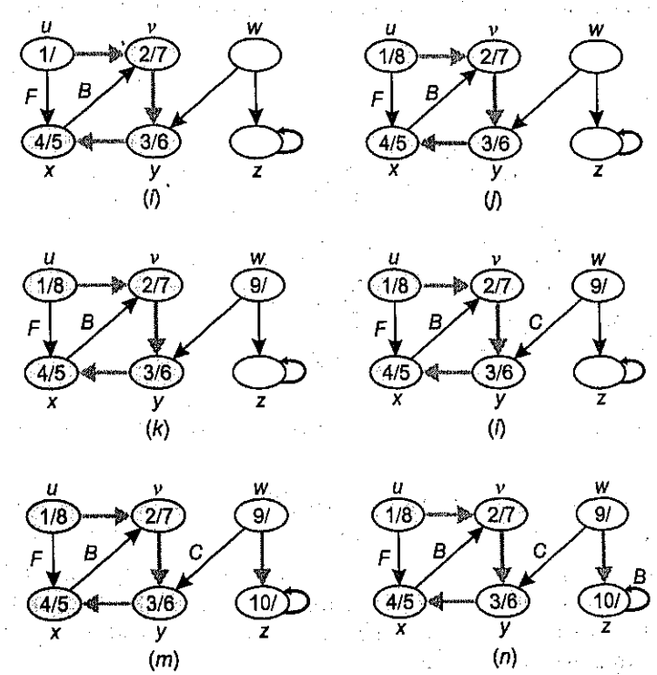
Solution. A depth-first search on an undirected graph yields only tree edges and back edges as shown in the above table. So that an undirected graph is acyclic if and only if a depth-first search yields no back edges. We now perform a depth-first search and if we discover a back edge then the graph must be acyclic. The time taken is $O(V)$ not $O(V+E)$ since if we discover more than V distinct edges the graph must be acyclic and we will have seen a back edge.

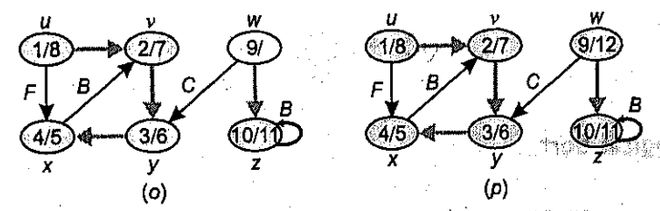
Example. *Show the progress of the depth-first-search (DFS) algorithm on a directed graph.*



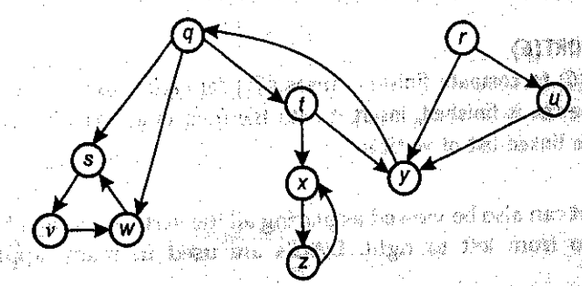
Solution.





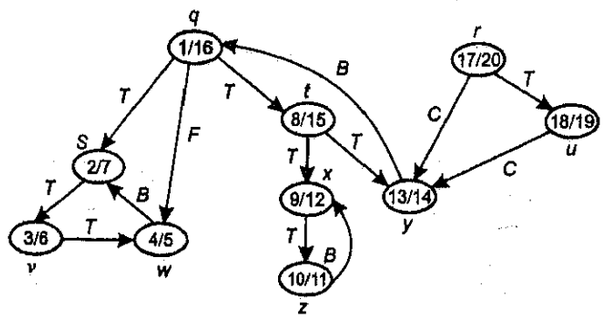


Example. *Show how DFS works on the graph.*



Assume that the for loop of lines 5-7 of the DFS procedure considers the vertices in alphabetical order, and assume that each adjacency list is ordered alphabetically. Show the discovery and finishing times for each vertex, and show the classification of each edge.

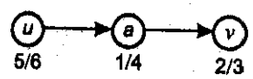
Solution. Shown below are the assignments of $d[\,]$ and $f[\,]$ fields and the classification of edges.



T : Tree edges *B* : Back edges *F* : Forward edges *C* : Cross edges

Example. *Give a counter example to the conjecture that if there is a path from u to v in a directed graph G, and if d [u] < d [v] in a DFS of G, then v is a descendant of u in the depth-first forest produced.*

Solution. The statement is not true since one may get different results depending on which order DFS-VISIT visits the vertices. A counter example is shown in the fig. If DFS- VISIT chooses to visit '*u*' first, '*v*' will be visited in the next step. Let 's say that DFS-VISIT (u) is called next. Then $d[v] < d[u]$ but u is not a descendant of v.



26.6 Topological Sort

Directed acyclic graphs or DAG's are used for topological sorts. A topological sort of a directed acyclic graph $G=(V, E)$ is a linear ordering of $v \in V$ such that if $(u, v) \in E$ then u appears before v in this ordering. If G is cyclic, no such ordering exists .

TOPOLOGICAL-SORT(G)

1. call DFS (G) to compute finishing times $f[v]$ for each vertex v
2. as each vertex is finished, insert it onto the front of a linked list
3. return the linked list of vertices

Topological sort can also be viewed as placing all the vertices along a horizontal line so that all directed edges go from left to right. DAG's are used in many applications to indicate precedence.

We can perform a topological sort in time $\theta(V+E)$, since DFS takes $\theta(V+E)$ time and it takes $O(1)$ time to insert each of the $|V|$ vertices onto the front of the linked list.

Example. *Another way to perform topological sorting on a directed acyclic graph $G=(V, E)$ is to repeatedly find a vertex of in-degree 0 , output it , and remove it and all of its outgoing edges from the graph. Explain how to implement this idea so that it runs in time $O(V+E)$. What happens to this algorithm if G has cycles ?*

Solution.

```
TOPOLOGICAL-SORT (G)
for each vertex u ∈ V
    do in-degree[u] ← 0
for each vertex u ∈ V
    do for each v ∈ Adj[u]
          do in-degree[v] ← in-degree[v] +1
Q ← ϕ
for each vertex u ∈ V
    do if in-degree[u] = 0
          then ENQUEUE(Q, u)
while Q ≠ ϕ
    do u ← DEQUEUE(Q)
          output u
          for each v ∈ Adj[u]
                do in-degree[v] ← in-degree[v] − 1
                      if in-degree[v] = 0
                      then ENQUEUE(Q, v)
do if in-degree[u] ≠ 0
          then report that there is a cycle
```

Another way to check the cycles would be to count the vertices that are output and report a cycle if that number is $<|V|$.

To find vertices with in-degree 0 we store these in a table. This takes $\theta(V+E)$ time. ($|V|$) visited adjacency lists, a total of $|E|$ visited edges and $\theta(1)$ time per edge.) We store vertices with in-degree 0 on a stack or in a queue so that they can be retrieved in $\theta(1)$ time.

To initialize the queue takes $\theta(V)$ time. When we have printed a vertex we remove its out-going edges from the graph by determining the in-degree on all its neighbors. Those vertices whose in-degree dropped to 0 are inserted into the queue. The total running time is $O(V+E)$. If the graph contains no cycles, all the vertices will be printed.

26.7 Strongly Connected Components

A directed graph is called **strongly connected** if for every pair of vertices u and v there is a path from u to v and a path from v to u. The **strongly connected components (SCC)** of a directed graph are its maximal strongly connected subgraphs. These form a partition of the graph.

Kosaraju's algorithm efficiently computes the strongly connected components of a directed graph.

Kosaraju's algorithm

Strongly-connected Components (G)

1. call DFS(G) to compute finishing times $f[u]$ for each vertex u.
2. compute G^T
3. call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing $f[u]$
4. produce as output the vertices of each tree in the DFS forest formed in line 3 as a separate SCC.

Analysis

A linear-time algorithm, if the graph is represented as an adjacency list $\theta(V+E)$, as a matrix it is an $O(V^2)$ algorithm, computes the strongly connected components of a directed graph $G=(V,E)$ using two Depth-first searches (DFSs), one on G, and one on G^T, the transpose graph. Equivalently, Breadth-first search (BFS) can be used instead of DFS.

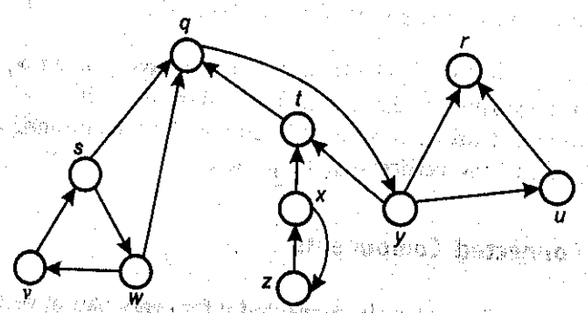
A transpose graph is a graph with all of the edges reversed. It is named because its adjacency matrix is the transpose of the adjacency matrix of the original graph.

Example. *Find the strongly connected components in the Fig. shown in the previous question.*

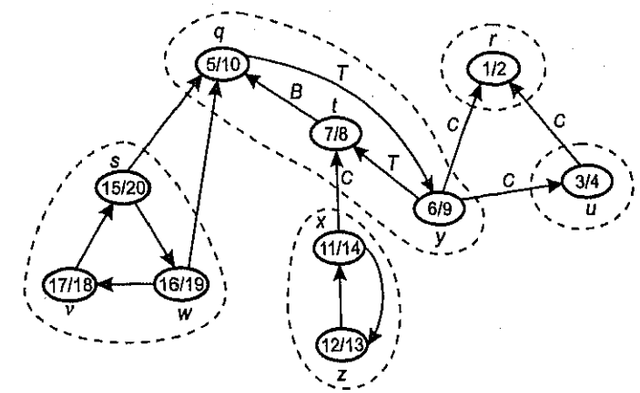
Solution. The first DFS gives the list

r, u, q, t, y, x, z, s, v, w (reverse order of turning black)

Now, G^T as follows.



Call DFS (G^T)



Strongly connected components are $\{s, w, v\}$, $\{q, y, t\}$, $\{x, z\}$, $\{r\}$, $\{u\}$.

Example. *How can the number of strongly connected components of a graph change if a new edge is added ?*

Solution. The number of strongly connected components can be reduced, when the new added edge makes two strongly connected components combined into one. Otherwise, the number will be the same.

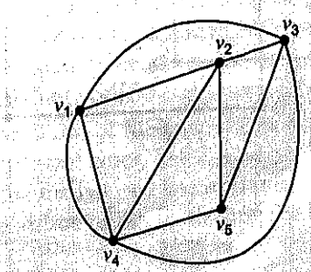
Example. *Prove that for any directed graph G, we have* $((G^T)^{SCC})^T = G^{SSC}$. *That is, the transpose of the component graph of* G^T *is the same as the component graph of G.*

Solution. Since the strongly connected relationship is an equivalence relation, G and G^T will always have the same strongly connected components. If two vertices X and Y are not strongly connected, then there is a unique path from X to Y in G iff there is a unique path from Y to X in G^T.

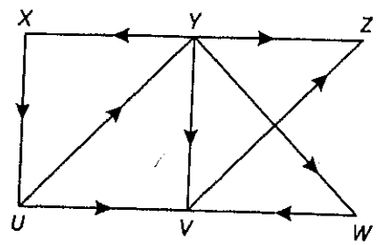
A similar property will also hold for the component graph and transpose of the component graph, *i.e.*, if two vertices X and Y are not strongly connected in G, then here is a unique path from SCC(G,X) to SCC(G,Y) in $(G)^{SCC}$ iff there is a unique path from SCC(G^T,Y) to SCC(G^T, X) in $(G^T)^{SCC}$.

Exercise

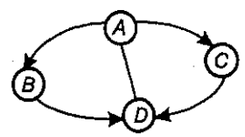
1. For the graph shown below find the following :
   (*a*) adjacency list representation. (*b*) adjacency matrix representation.



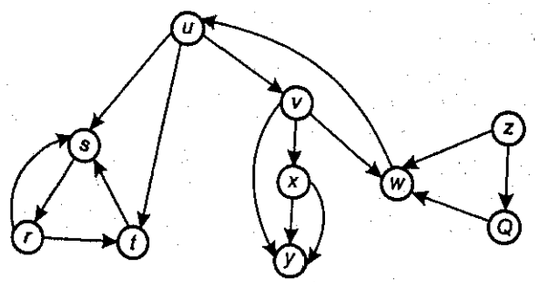
2. Let G be an undirected graph G with n vertices and m edges. Describe an algorithm running in $O(n+m)$ time traverses each edge of G exactly once in each direction.
3. Show that algorithm to find a cycle using BFS or DFS must be $O(n)$.
4. Find the adjacency matrix for the following graph in Fig. Find in-degree and out-degree of all the nodes.



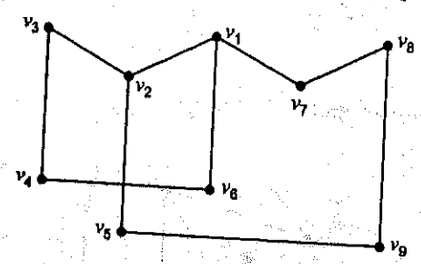
5. Consider the following graph in Fig. State, if this graph is strongly connected. List all the simple paths.



6. Show the parenthesis structure of the following graph.



7. Starting from vertex v_4 apply BFS and DFS in the following graph.



CHAPTER

27

Minimum Spanning Tree

27.1 Spanning Tree

A **spanning tree** of a graph is just a sub graph that contains all the vertices and is a tree. A graph may have many spanning trees. We can also assign a *weight* to each edge, which is a number representing how unfavorable it is, and use this to assign a weight to a spanning tree by computing the sum of the weights of the edges in that spanning tree. A **minimum spanning tree** or **minimum weight spanning tree** is then a spanning tree with weight less than or equal to the weight of every other spanning tree. More generally, any undirected graph has a **minimum spanning forest**.

To explain further upon the Minimum Spanning Tree (MST), and what it applies to, let's consider a couple of real-world examples :

1. One practical application of a MST would be in the design of a network. For instance, a group of individuals, who are separated by varying distances, wish to be connected together in a telephone network. Although MST cannot do anything about the distance from one connection to another, it can be used to determine the least costly paths with no cycles in this network, thereby connecting everyone at a minimum cost.
2. Another useful application of MST would be finding airline routes. The vertices of the graph would represent cities, and the edges would represent routes between the cities. Obviously, the further one has to travel, the more it will cost, so MST can be applied to optimize airline routes by finding the least costly paths with no cycles.

To explain how to find a Minimum Spanning Tree, we will look at two algorithms: the Kruskal algorithm and the Prim algorithm. Both algorithms differ in their methodology, but both eventually end up with the MST. ***Kruskal's algorithm uses edges, and Prim's algorithm uses vertex connections in determining the MST.*** Both algorithms are greedy algorithms that run in polynomial time. At each step of an algorithm, one of several possible choices must be made.

27.2 Kruskal's Algorithm

Kruskal's algorithm is an algorithm that finds a minimum spanning tree for a connected weighted graph. It finds a safe edge to add to the growing forest by finding, of all the edges that connect any two trees in the forest, an edge (u,v) of least weight. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. **If the graph is not connected, then it finds a minimum spanning forest (a minimum spanning tree for each connected component).**

```
MST-KRUSKAL (G, w)
A ← ϕ
for each vertex v ∈ V[G]
    do MAKE-SET(v)
sort the edges of E into nondecreasing order by weight w
for each edge (u,v) ∈ E, taken in nondecreasing order by weight
    do if FIND-SET(u) ≠ FIND-SET(v)
        then A ← A ∪ {(u,v)}
            UNION (u,v)
return A
```

→ u & v belong to diff ~~forests~~ trees
no creation of cycles.

Analysis

Where E is the number of edges in the graph and V is the number of vertices, Kruskal's algorithm can be shown to run in $O(E \log E)$ time, or equivalently, $O(E \log V)$ time, all with simple data structures. These running times are equivalent because :

- E is at most V^2 and $\log V^2 = 2 \times \log V$ is $O(\log V)$.
- If we ignore isolated vertices, which will each be their own component of the minimum spanning tree anyway, $V \le 2E$, so $\log V$ is $O(\log E)$.

We can achieve this bound as follows: first sort the edges by weight using a comparison sort in $O(E \log E)$ time ; Next, we use a disjoint-set data structure to keep track of which vertices are in which components. We need to perform $O(E)$ operations, two find operations and possibly one union for each edge. Even a simple disjoint-set data structure such as disjoint-set forests with union by rank can perform $O(E)$ operations in $O(E \log V)$ time. Thus the total time is

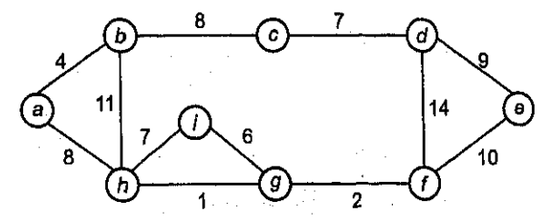
$$O(E \log E) = O(E \log V)$$

Example. *Let (u,v) be a minimum weight edge in a graph G. Show that (u,v) belongs to some minimum spanning tree of G.*

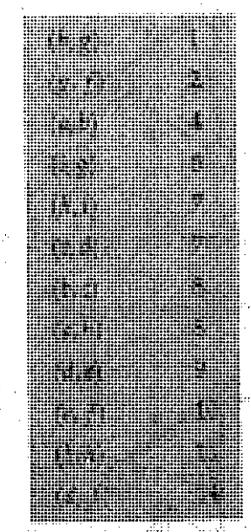
Solution. Proof by contradiction : assume that (u,v) doesn't belong to some MST of G. If we would add (u,v) to the MST we would get a cycle because MST is a tree. But if we then removed another edge in the cycle we would end up with a MST that is at least as cheap as the original.

Hence, (u,v) does belong to some minimum spanning tree of G.

Example. *Find the minimum spanning tree of the following graph using kruskals algorithm.*

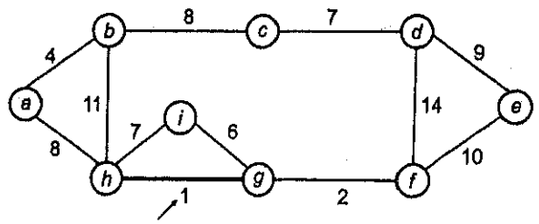


Solution. First we initialize the set A to the empty set and create $|v|$ trees, one containing each vertex with MAKE-SET procedure. Then sort the edges in E into order by non-decreasing weight, *i.e.*,

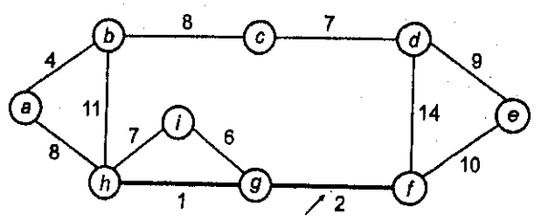


Now, check for each edge (u,v), whether the end points u and v belong to the same tree. If they do, then the edge (u,v) cannot be added. Otherwise, the two vertices belong to different trees and the edge (u,v) is added to A and the vertices in the two trees are merged in by UNION procedure.

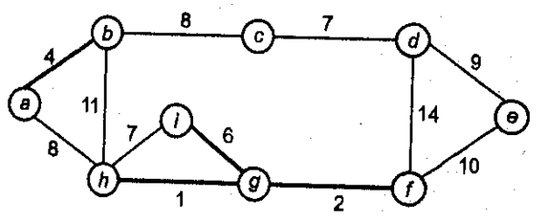
So, first take (h,g) edge



the (g, f) edge.

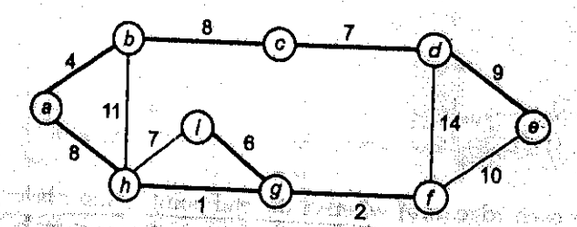


then (a,b) and (i,g) edges are considered and forest becomes.



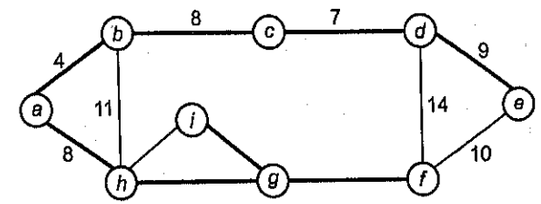
Now, edge (h,i). Both h and i vertices are in same set, thus it creates a cycle. So this edge is discarded.

Then edge (c,d), (b,c), (a,h), (d,e), (e,f) are considered and forest becomes.

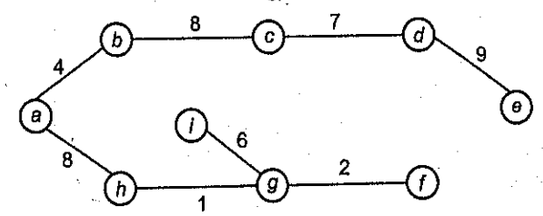


In (e, f) edge both end points e and f exist in same tree so discarded this edge. Then (b,h) edge, it also creates a cycle.

After that edge (d, f) and the final spanning tree is shown as in dark lines.



or



Minimum cost MST

27.3 Prim's Algorithm

The main idea of Prim's algorithm is similar to that of Dijkstra's algorithm (discussed later) for finding shortest path in a given graph. Prim's algorithm has the property that the edges in the set A always form a single tree. We begin with some vertex v in a given graph $G=(V, E)$, defining the initial set of vertices A. Then, in each iteration, we choose a minimum-weight edge (u,v) connecting a vertex v in the set A to the vertex u outside of set A. Then vertex u is brought in to A. This process is repeated until a spanning tree is formed. Like Kruskal's algorithm, here too, the important fact about MSTs is we always choose the smallest-weight edge joining a vertex inside set A to the one outside the set A. The implication of this fact is that it adds only edges that are safe for A; therefore when the algorithm terminates, the edges in set A form a MST.

```
MST-PRIM(G, w, r)
for each u ∈ V[G]
    do key[u] ← ∞
       π[u] ← NIL
key[r] ← 0
Q ← V[G]
while Q ≠ ϕ
    do u ← EXTRACT-MIN(Q)
       for each v ∈ Adj[u]
           do if v ∈ Q and w(u,v) < key[v]
              then π[v] ← u
                   key[v] ← w(u,v)
```

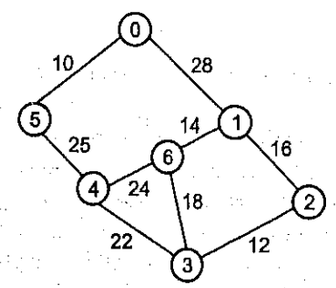
Example. *Suppose that all edge weights in a graph are integers in the range from 1 to $|V|$. How fast can you make Prim's algorithm run ? What if the edge weights are integers in the range from 1 to W for some constant W ?*

Solution. The running time of Prims algorithm is composed :

$O(V)$ initialization.
$O(V)$-time for EXTRACT-MIN.
$O(E)$-time for DECREASE-KEY.

If the edges are in the range 1.....$|V|$ the priority queue can speed up EXTRACT_MIN and DECREASE-KEY to $O(\lg\lg V)$ thus yielding a total running time of $O(V \lg\lg V + E \lg\lg V) = O(E \lg\lg V)$. If the edges are in the range from 1 to W we can implement the queue as an array $[1...W+1]$ where the ith slot holds a doubly linked list of the edges with weight i. The $W+1$st slot contains ∞. EXTRACT_MIN now runs in $O(W) = O(1)$ time since we can simply scan for the first nonempty slot and return the first element of that list. DECREASE-KEY runs in $O(1)$ time as well since it can be implemented by moving an element from one slot to another.

Example. *Generate minimum cost spanning tree for the following graph using Prim's algorithm.*



Solution. In Prim's algorithm, first we initialize the priority queue Q, to contain all the vertices and the key of each vertex to ∞ except for the root, whose key is set to 0. Suppose 0 vertex is the root, *i.e.*, r. By EXTRACT-MIN(Q) produre, now $u = r$ and Adj$[u] = \{5,1\}$.

Removing u from the set Q and adds it to the set $V - Q$ of vertices in the tree. Now, update the key and π fields of every vertex v adjacent to u but not in the tree

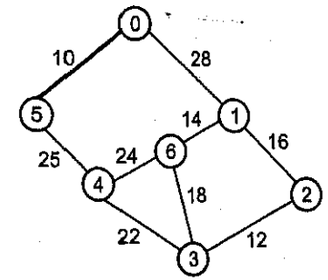
$key\,[5] = \infty$

$w[0,5) = 10$ *i.e.,* $w(u,v) < key\,[5]$

so, $\pi[5] = 0$ and $key\,[5] = 10$.

and $key\,[1] = \infty$

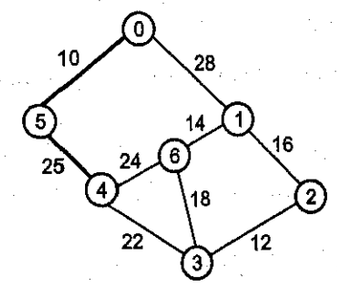
$w(0,1) = 28$ *i.e.,* $w(u,v) < key\,[5]$ so $\pi[1] = 0$ and $key\,[1] = 28$.



Now, by EXTRACT_MIN(Q) Remove 5 because key[5] = 10 which is minimum so $u = 5$.

$Adj\,[5] = \{4\}$

$w(u,v) < key\,[v]$ then $key\,[4] = 25$



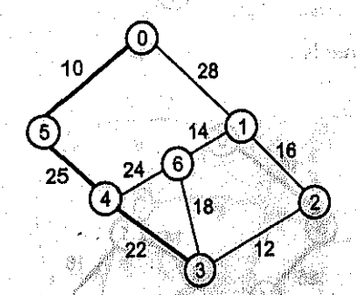
Now remove 4 because $key\,[4] = 25$ which is minimum so $u = 4$.

$Adj\,[4] = \{6, 3\}$

$w(u,v) < key\,[v]$ then $key\,[6] = 24$

$key\,[3] = 22$

Now remove 3 because $key\,[3] = 22$ is minimum so $u = 3$.



$Adj\,[3] = \{4, 6, 2\}$

$4 \neq Q$ $key\,[6] = 24$ now becomes $key\,[6] = 18$.

$key\,[2] = 12$

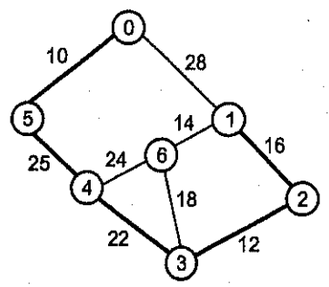
Now, in Q. $key[2] = 12$, $key[6] = 18$, $key[1] = 28$

By EXTRACT_MIN(Q) Remove 2, because $key[2] = 12$ is minimum.

$Adj[2] = \{3, 1\}$

$3 \neq Q$.

Now $key[1] = 16$.

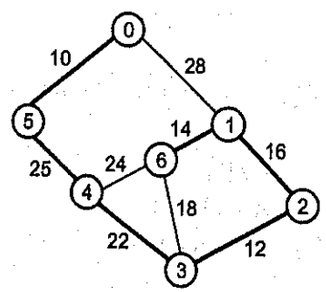


By EXTRACT_MIN(Q) Remove 1 because $key[1] = 16$ is minimum.

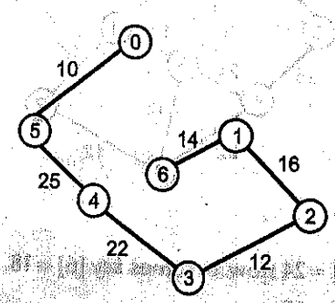
$Adj[1] = \{0, 6, 2\}$

$\{0, 2\} \notin Q$. $key[6] = 14$.

Now becomes Q contains only one vertex 6. By EXTRACT_MIN remove it



Thus, the final spanning tree is

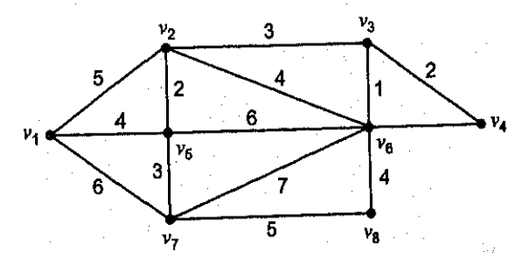


Example. *There are two well-known algorithms for finding minimum spanning trees. Point out the difference between Prim's algorithm and Kruskal's algorithm in terms of the construction of the MST tree.*

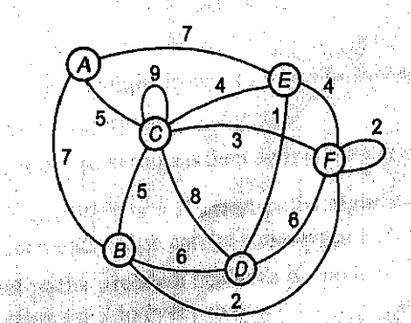
Solution. The construction of an MST tree by Prim's algorithm is started from a single node, and the tree is expanded until it covers all the nodes in the graph. In other words, there is only one partial tree during the construction of the MST tree from the very beginning to the end. However, in the construction of an MST tree using Kruskal's algorithm, the algorithm starts from a n-tree forest. Each time it merges two trees in the forest into one, the algorithm continues until there is only one tree left in the forest.

Exercise

1. Show that if all the weights in a connected weighted graph G are distinct, then there is exactly one minimum spanning tree for G.
2. For the graph shown below obtain the following :
   (*a*) MST by Kruskal's Method
   (*b*) MST by Prim's Method.

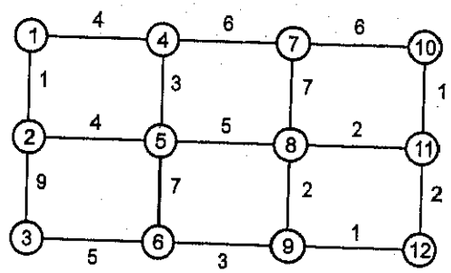


3. Discuss Kruskal and Prim algorithms and write their pseudo codes.
4. Find out the MST of the following graph.



5. Thre is a network given in the Fig. below as a highway map and the number recorded next to each arc as the maximum elevation enountered in traversing the arc.

A traveller plans to drive from node 1 to 12 on this highway. The traveller dislikes high altitudes and so would like to find a path connecting node 1 to 12 that minimizes the maximum altitude. Find the best path for the traveller using a minimum spanning tree.



CHAPTER

28

Single-Source Shortest Paths

28.1 Introduction

In a **shortest-paths problem**, we are given a weighted, directed graph $G=(V, E)$, with weight function $w : E \to \mathbf{R}$ mapping edges to real-valued weights. The weight of path $p=\langle v_0, v_1, \ldots, v_k \rangle$ is the sum of the weights of its constituent edges :

$$w(P)=\sum_{i=1}^{k} w(v_{i-1}, v_i)$$

We define the **shortest-path weight** from u to v by $\delta(u, v)=\min(w(p) : u \rightsquigarrow v)$, if there is a path from u to v, and $\delta(u, v)=\infty$, otherwise.

A **shortest path** from vertex s to vertex t is then defined as any path p with weight $w(p)=\delta(s, t)$.

The *breadth-first-search algorithm* is a shortest-paths algorithm that works on unweighted graphs, that is, graphs in which each edge can be considered to have unit weight.

Single-source shortest-paths problem : Given a graph $G=(V, E)$, we want to find a shortest path from a given source vertex $s \in V$ to every vertex $v \in V$.

Variants

But, the algorithm for the single-source problem, including the following variants, can solve many other problems.

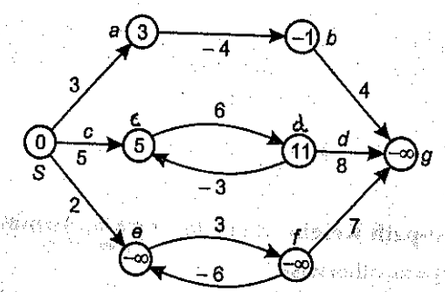
- **Single-destination shortest-paths problem.** Find a shortest path to a given ***destination*** vertex t from every vertex v. By reversing the direction of each edge in the graph, we can reduce this problem to a single-source problem.
- **Single-pair shortest-path problem.** Find a shortest path from u to v for given vertices u and v. If we solve the single-source problem with source vertex u, we solve this problem also. Moreover, no algorithms for this problem are known that run asymptotically faster than the best single-source algorithms in the worst case.
- **All-pairs shortest-paths problem.** Find a shortest path from u to v for every pair of vertices u and v. Running a single-source algorithm once from each vertex can solve this problem ; but it can usually be solved faster, and its structure is of interest in its own right.

28.2 Shortest Path : Existence

If some path from s to v contains a negative cost cycle then, there does not exist a shortest path. Otherwise, there exists a shortest $s-v$ that is simple.

Cost of $C < 0$

If there is a negative-weight cycle on some path from s to v, we define $\delta(s,v) = -\infty$.



There are infinitely many paths from s to c: $\langle s,c\rangle$, $\langle s,c,d,c\rangle$, $\langle s,c,d,c,d,c\rangle$, and so on. Because the cycle $\langle c,d,c\rangle$ has weight $6+(-3)=3>0$, the shortest path from s to c is $\langle s,c\rangle$, with weight $\delta(s,c)=5$. Similarly, there are infinitely many paths from s to e: $\langle s,e\rangle$, $\langle s,e,f,e\rangle$, $\langle s,e,f,e,f,e\rangle$, and so on. Since the cycle $\langle e,f,e\rangle$ has weight $3+(-6)=-3<0$, however, there is no shortest path from s to e. By traversing the negative-weight cycle $\langle e,f,e\rangle$ arbitrarily many times, we can find paths from s to e with arbitrarily large negative weights, and so $\delta(s,e)=-\infty$.

Some shortest-paths algorithms, such as **Dijkstra's algorithm**, assume that all edge weights in the input graph are nonnegative, as in the road-map example. Others, such as the **Bellman-Ford algorithm**, allow negative-weight edges in the input graph and produce a correct answer as long as no negative-weight cycles are reachable from the source. Typically, if there is such a negative-weight cycle, the algorithm can detect and report its existence.

28.3 Representing Shortest Paths

Given a graph $G=(V,E)$, we maintain for each vertex $v \in V$ a ***predecessor*** $\pi[v]$ that is either another vertex or NIL. During the execution of a shortest-paths algorithm, however, the π values need not indicate shortest paths. As in breadth-first search, we shall be interested in the ***predecessor subgraph*** $G_\pi = (V_\pi, E_\pi)$ induced by the values π. Here again, we define the vertex set V_π, to be the set of vertices of G with non-NIL predecessors, plus the source s:

$$\mathbf{V}_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\}\}$$

The directed edge set E_π is the set of edges induced by the π values for vertices in $\mathbf{V}_\pi$:

$$\mathbf{E}_\pi = \{(\pi[v], v) \in E : v \in \mathbf{V}_\pi - \{s\}\}$$

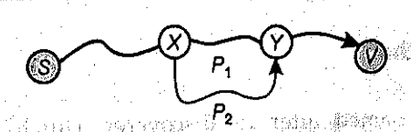
A ***shortest-paths tree*** rooted at s is a directed sub graph $G' = (V', E')$, where $V' \subseteq V$ and $E' \subseteq E$, such that

1. V' is the set of vertices reachable from s in G,
2. G' forms a rooted tree with root s, and
3. for all $v \in V'$, the unique simple path from s to v in G' is a shortest path from s to v in G.

Shortest paths are not necessarily unique, and neither are shortest-paths trees.

28.4 Shortest Path : Properties

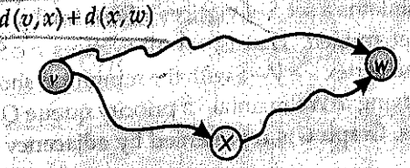
- **Optimal substructure property.** All sub-paths of shortest paths are shortest paths.



Let P_1 be $x-y$ sub-path of shortest $s-v$ path P. Let P_2 be any $x-y$ path. Then cost of $P_1 \leq$ cost of P_2, otherwise P not shortest $s-v$ path.

- **Triangle inequality.** Let $d(v,w)$ be the length of the shortest path from v to w. Then,

$$d(v,w) \leq d(v,x) + d(x,w)$$



Relaxation

The single-source shortest-paths algorithms are based on a technique known as **relaxation**, a method that repeatedly decreases an upper bound on the actual shortest-path weight of each vertex until the upper bound equals the shortest-path weight. For each vertex $v \in V$, we maintain an attribute $d[v]$, which is an upper bound on the weight of a shortest path from source s to v. We call $d[v]$ a **shortest-path estimate**. We initialize the shortest-path estimates and predecessors by the following procedure.

```
INITIALIZE-SINGLE-SOURCE (G, s)
for each vertex v ∈ V[G]
    do d[v] ← ∞
       π[v] ← NIL
d[s] ← 0
```

After initialization, $\pi[v] = \text{NIL}$ for all $v \in V$, $d[v] = 0$ for $v = s$, and $d[v] = \infty$ for $v \in V - \{s\}$.

The process of **relaxing** an edge (u, v) consists of testing whether we can improve the shortest path to v found so far by going through u and, if so, updating $d[v]$ and $\pi[v]$. A relaxation step may decrease the value of the shortest-path estimate $d[v]$ and update v's predecessor field $\pi[v]$. The following code performs a relaxation step on edge (u, v).

```
RELAX (u, v, w)
if d[v] > d[u] + w(u, v)
    then d[v] ← d[u] + w(u, v)
         π[v] ← u
```

In **Dijkstra's algorithm** and the shortest-paths algorithm for directed acyclic graphs, each edge is relaxed exactly once. In the **Bellman-Ford algorithm**, each edge is relaxed several times.

28.5 Dijkstra's Algorithm

Dijkstra's algorithm, named after its discoverer, Dutch computer scientist **Edsger Dijkstra**, is a greedy algorithm that solves the single-source shortest path problem for a directed graph $G = (V, E)$ with nonnegative edge weights *i.e.* we assume that $w(u, v) \geq 0$ for each edge $(u, v) \in E$.

Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. That is, for all vertices $v \in S$, we have $d[v] = \delta(s, v)$. The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, inserts u into S, and relaxes all edges leaving u. We maintain a priority queue Q that contains all the vertices in $v - s$, keyed by their d values. Graph G is represented by adjacency lists.


```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE (G, s)
S ← φ
Q ← V[G]
while Q ≠ φ
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX (u, v, w)
```

Because Dijkstra's algorithm always chooses the "lightest" or "closest" vertex in $V - S$ to insert into set S, we say that it uses a **greedy strategy**.

Dijkstra's algorithm bears some similarity to both **breadth-first search** and **Prim's algorithm** for computing minimum spanning trees. It is like breadth-first search in that set S corresponds to the set of black vertices in a breadth-first search; just as vertices in S have their final shortest-path weights, so black vertices in a breadth-first search have their correct breadth-first distances. Dijkstra's algorithm is like Prim's algorithm in that both algorithms use a priority queue to find the "lightest" vertex outside a given set (the set S in Dijkstra's algorithm and the tree being grown in Prim's algorithm), insert this vertex into the set, and adjust the weights of the remaining vertices outside the set accordingly.

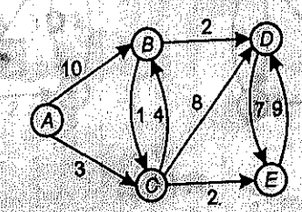
Analysis

The running time of Dijkstra's algorithm on a graph with edges E and vertices V can be expressed as a function of $|E|$ and $|V|$ using the Big-O notation. The simplest implementation of the Dijkstra's algorithm stores vertices of set Q in an ordinary linked list or array, and operation Extract-Min (Q) is simply a linear search through all vertices in Q. In this case, the running time is $O(|V|^2 + |E| = O(V^2)$.

For sparse graphs, that is, graphs with many fewer than $|V|^2$ edges, Dijkstra's algorithm can be implemented more efficiently by storing the graph in the form of adjacency lists and using a binary heap or Fibonacci heap as a priority queue to implement the Extract-Min function. With a binary heap, the algorithm requires $O((|E| + |V|)\log|V|)$ time (which is dominated by $O(|E|\log|V|)$ assuming every vertex is connected), and the Fibonacci heap improves this to $O(|E| + |V|\log|V|)$.

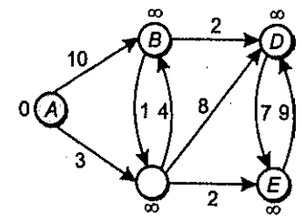
A C E B D

Example :



A	B	C	D	E
0	∞	∞	∞	∞
	10	[3]	∞	∞
	7	3	11	[5]
	[7]	3	11	5
	7	3	[9]	5
	7	3		5

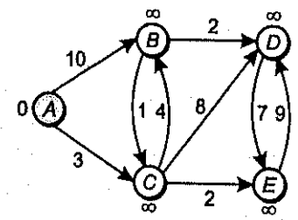
Initialize :



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

$S: \{\}$

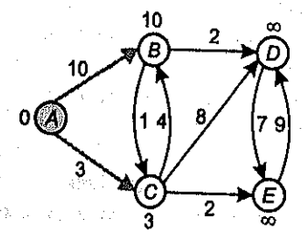
"A" ← EXTRACT-MIN(Q) :



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

$S: \{A\}$

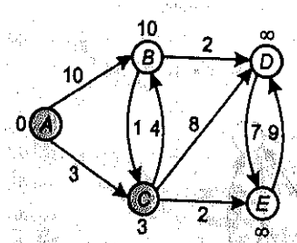
Relax all edges leaving A:



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	–	–

$S: \{A\}$

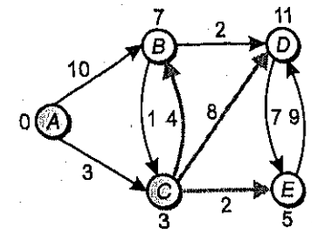
"C" ← EXTRACT-MIN(Q) :



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	–	–

$S: \{A, C\}$

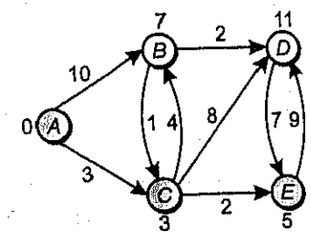
Relax all edges leaving C:



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	–	–
		7		11	5

$S: \{A, C\}$

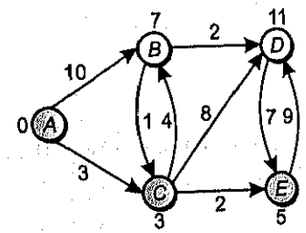
"E" ← EXTRACT-MIN(Q) :



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	–	–
		7		11	5

$S: \{A, C, E\}$

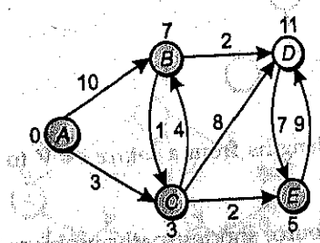
Relax all edges leaving E:



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

$S: \{A, C, E\}$

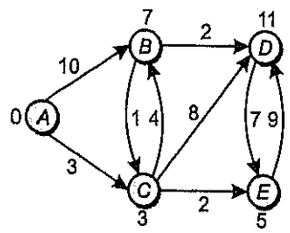
"B" ← EXTRACT-MIN(Q) :



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

$S: \{A, C, E, B\}$

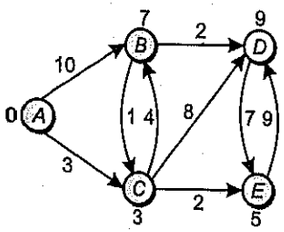
Relax all edges leaving B :



Q :	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

$S : \{A, C, E, B\}$

"D" ← EXTRACT-MIN(Q) :



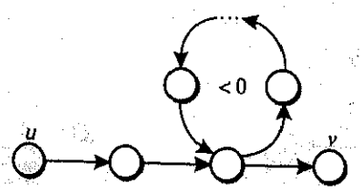
Q :	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

$S : \{A, C, E, B, D\}$

28.6 The Bellman-Ford Algorithm

If a graph $G=(V, E)$ contains a negative weight cycle, then some shortest paths may not exist.

Example :



Bellman-Ford algorithm finds all shortest-path lengths from a *source* $s \in V$ to all $v \in V$ or determines that a negative-weight cycle exists.

Thus, ***Bellman-Ford algorithm*** solves the single-source shortest-paths problem in the more general case in which edge weights can be **negative**. Given a weighted, directed graph $G=(V, E)$ with source s and weight function $w : E \to R$, the Bellman-Ford algorithm returns a **boolean value**

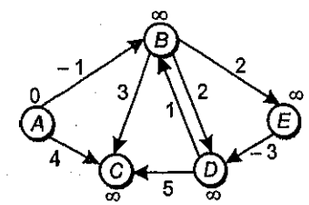
indicating whether or not there is a negative-weight cycle that is reachable from the source. If there is such a cycle, the algorithm indicates that **no solution** exists. If there is no such cycle, the algorithm produces the shortest paths and their weights. The algorithm returns TRUE if and only if the graph contains no negative-weight cycles that are reachable from the source.

BELLMAN-FORD(G, w, s)

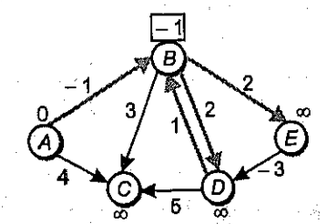
```
INITIALIZE-SINGLE-SOURCE(G, s)
for i ← 1 to |V[G]| − 1
    do for each edge (u,v) ∈ E[G]
        do RELAX(u,v,w)
for each edge (u,v) ∈ E[G]
    do if d[v] > d[u] + w(u,v)
        then return FALSE
return TRUE
```

Example :

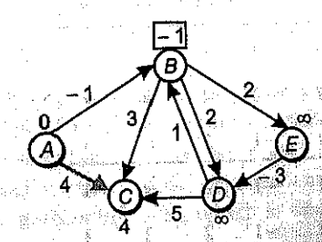
Order of edges : $(B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D)$



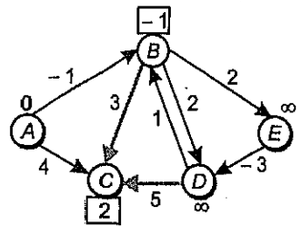
A	B	C	D	E
0	∞	∞	∞	∞



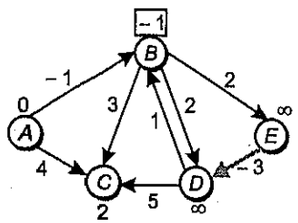
A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞



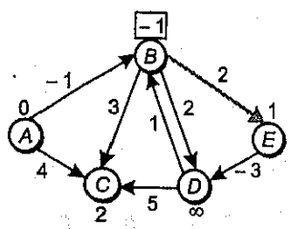
A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞



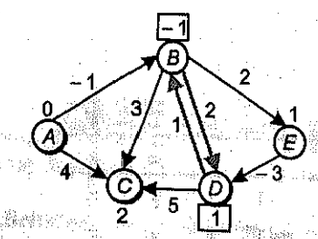
A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞



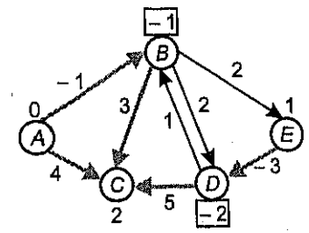
A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞



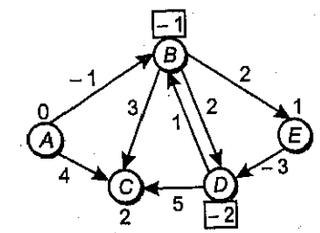
A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞
0	−1	2	∞	1



A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞
0	−1	2	∞	1
0	−1	2	1	1



A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞
0	−1	2	∞	1
0	−1	2	1	1
0	−1	2	−2	1



A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞
0	−1	2	∞	1
0	−1	2	1	1
0	−1	2	−2	1

Note *Values decrease monotonically.*

28.7 Single-source shortest paths in directed acyclic graphs

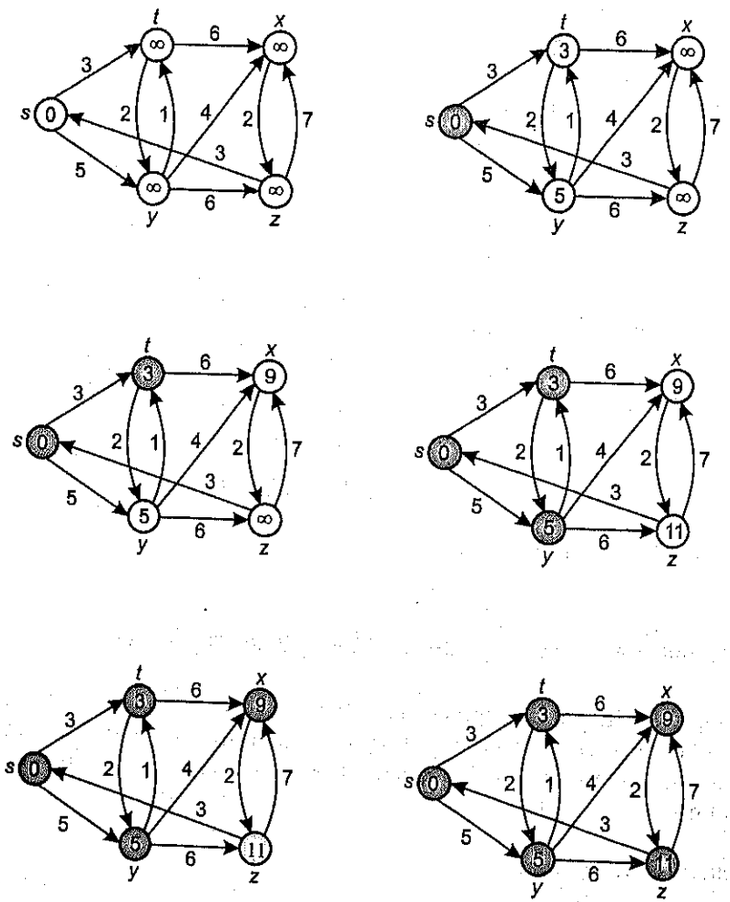
By relaxing the edges of a weighted dag (directed acyclic graph) $G=(V, E)$ according to a topological sort of its vertices, we can compute shortest paths from a single source in $\theta(V+E)$ time. Shortest paths are always well defined in a dag, since even if there are negative-weight edges, no negative-weight cycles can exist.

```
DAG-SHORTEST-PATHS (G, w, s)
1.  Topologically sort the vertices of G
2.  INITIALIZE-SINGLE-SOURCE (G, s)
3.  for each vertex u taken in topologically sorted order
4.      do for each vertex v ∈ Adj[u]
5.          do RELAX(u, v, w)
```

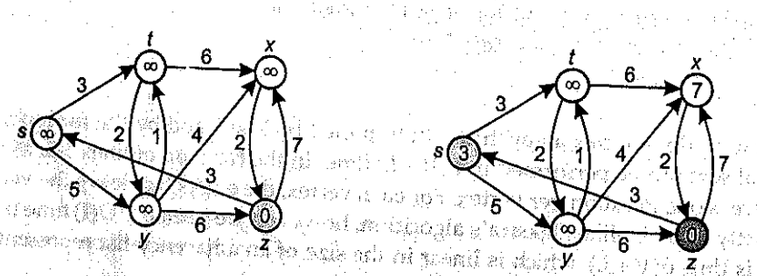
The running time of this algorithm is determined by line 1 and by the **for** loop of lines 3-5. The topological sort can be performed in $\theta(V+E)$ time. In the **for** loop of lines 3-5, as in Dijkstra's algorithm, there is one iteration per vertex. For each vertex, the edges that leave the vertex are each examined exactly once. Unlike Dijkstra's algorithm, however, we use only $O(1)$ time per edge. The running time is thus $\theta(V+E)$, which is linear in the size of an adjacency-list representation of the graph.

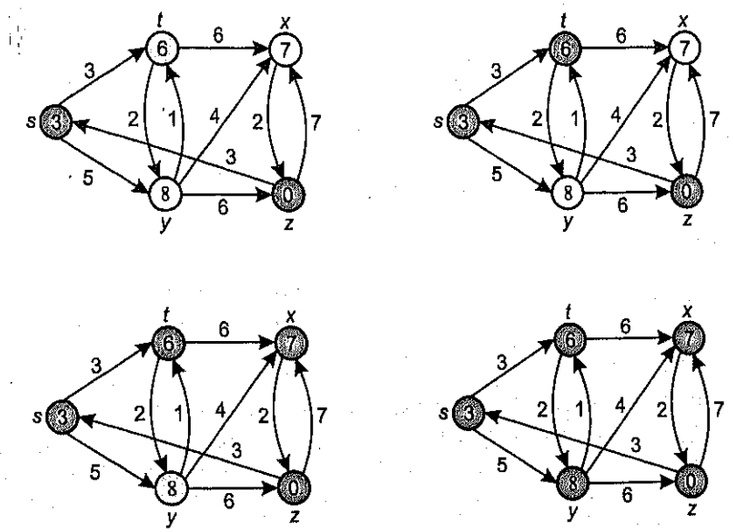
Example. *Run Dijkstra's algorithm on the directed graph of Fig. , First using vertex s as the source and then using vertex z as the source.*

Solution. First using vertex s as the source.



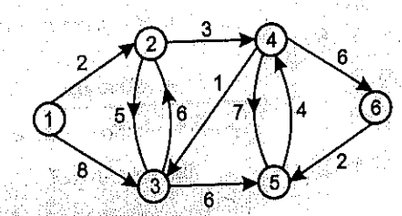
Then using vertex z as the source



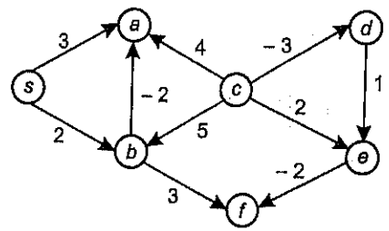


Exercise

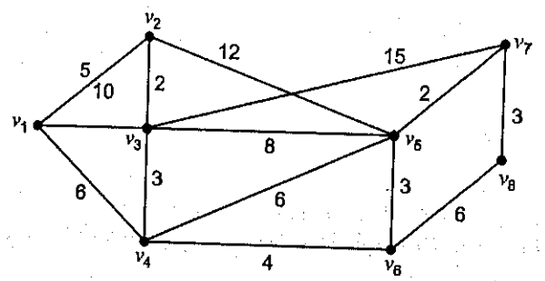
1. We would like to solve, as efficiently as possible, the single source shortest path problems in each of the following graphs. For each graph, state which algorithm would be best to use, and give its running time :
   (*i*) A weighted directed acyclic graph.
   (*ii*) A weighted directed graph where all edges weights are non-negative, the graph contains a directed cycle.
   (*iii*) A weighted directed graph in which some, but not all of the edges have negative weights, the graph contains a directed cycle.
2. Solve the shortest path problems using Dijkstra's algorithm. Count the number of distance updates.



3. Find shortest path using BELLMANFORD algorithm from S to F of the following Fig.



4. Draw a simple, connected, weighted graph with 8 vertices and 16 edges, each with unique edge weights. Identify one vertex as a "start" vertex and illustrate a running of Dijkstra's algorithm on this graph.
5. Show how to modify Dijkstra's algorithm for the case when the graph is directed and we want to compute shortest directed paths from the source vertex to all other vertices.
6. Draw a simple (directed) weighted graph G with 10 vertices and 18 edges, such that G contains a minimum-weight cycle with at least 4 edges. Show that the Bellman-Ford algorithm will find this cyle.
7. Find shortest path from V_1 to V_8.



CHAPTER

29

All-Pairs Shortest Paths

29.1 Introduction

The all-pairs shortest path problem can be considered the mother of all routing problems. It aims to compute the shortest path from each vertex v to every other u. Using standard single-source algorithms, we can expect to get a naive implementation of $O(n^3)$ if we use Dijkstra for example, *i.e.* running a $O(n^2)$ process n times. Likewise, if we use the Bellman-Ford-Moore algorithm on a *dense graph*, it'll take about $O(n^4)$, but handle negative arc-lengths too.

Storing all the paths explicitly can be very memory expensive indeed, as we need one spanning tree for each vertex. This is often impractical in terms of memory consumption, so these are usually considered as all-pairs shortest distance problems, which aim to find just the distance from each to each node to another. We typically want the output in tabular form : the entry in u's row and v's column should be the weight of a shortest path from u to v.

Three approaches for improvement :

algorithm	*cost*
matrix multiplication	$O(V^3 \lg V)$
Floyd-Warshall	$O(V^3)$
Johnson	$O(V^2 \lg V + VE)$

Unlike the single-source algorithms, which assume an adjacency-list representation of the graph, most of the algorithms use an adjacency-matrix representation. (Johnson's algorithm for sparse graphs uses adjacency lists.) The input is an $n \times n$ matrix W representing the edge weights of an n-vertex directed graph $G=(V, E)$. That is, $W=(w_{ij})$, where

$$w_{ij} = \begin{cases} 0 & \text{if } i=j, \\ w(i,j) & \text{if } i \neq j \text{ and } (i,j) \in E, \\ \infty & \text{if } i \neq j \text{ and } (i,j) \notin E. \end{cases}$$

Negative-weight edges are allowed, but we assume for the time being that the input graph contains no negative-weight cycles.

The tabular output of the all-pairs shortest-paths algorithms is an $n \times n$ matrix $D=(d_{ij})$, where entry d_{ij} contains the weight of a shortest path from vertex i to vertex j. That is, if we let $\delta(i,j)$ denote the shortest-path weight from vertex i to vertex j, then $d_{ij} = \delta(i,j)$ at termination.

To solve the all-pairs shortest-paths problem on an input adjacency matrix, we need to compute not only the shortest-path weights but also a ***predecessor matrix*** $\Pi=(\pi_{ij})$, where π_{ij} is NIL if either $i=j$ or there is no path from i to j, and otherwise π_{ij} is some predecessor of j on a shortest path from i. For each vertex $i \in V$, we define the ***predecessor sub graph*** of G for i as $G_{\pi,i} = (V_{\pi,i}, E_{\pi,i})$, where

$$V_{\pi,i} = \{j \in V : \pi_{ij} \neq \text{NIL}\} \cup \{i\} \quad \text{and} \quad E_{\pi,i} = \{(\pi_{ij}, j) : j \in V_{\pi,i} \text{ and } \pi_{ij} \neq \text{NIL}\}$$

29.2 Matrix Multiplication

Let us define the $V \times V$ matrix

$$D^{(m)} = (d_{ij}^{(m)})$$

$$d_{ij}^{(m)} = \text{the length of the shortest path from } i \text{ to } j \text{ with} \leq m \text{ edges.}$$

When $m=0$, there is a shortest path from i to j with no edges if and only if $i=j$.

$$d_{ij}^{(0)} = \begin{cases} 0 & \text{if } i=j \\ w_{ij} & \text{if } i \neq j, (i,j) \in E, \\ \infty & \text{otherwise} \end{cases}$$

29.3 The Floyd-Warshall Algorithm

Floyd–Warshall algorithm (sometimes known as the **Roy–Floyd algorithm**, since Bernard Roy described this algorithm in 1959) is a graph analysis algorithm for finding shortest paths in a weighted, directed graph. A single execution of the algorithm will find the shortest path between all pairs of vertices. It does so in $\theta(V^3)$ time, where V is the number of vertices in the graph. Negative-weight edges may be present, but we shall assume that there are no negative-weight cycles.

The algorithm considers the *"intermediate"* vertices of a shortest path, where an ***intermediate*** vertex of a simple path $p=\langle v_1, v_2, \ldots, v_m \rangle$ is any vertex of p other than v_1 or v_m, that is, any vertex in the set $\{v_2, v_3, \ldots, v_{m-1}\}$

The Floyd-Warshall algorithm is based on the following observation. Let the vertices of G be $V=\{1,2,\ldots,n\}$, and consider a subset $\{1,2,\ldots,k\}$ of vertices for some k. For any pair of vertices $i, j \in V$, consider all paths from i to j whose intermediate vertices are all drawn from $\{1,2,\ldots,k\}$, and let p be a minimum-weight path from among them. (Path p is simple, since we assume that G contains no negative-weight cycles.) The Floyd-Warshall algorithm exploits a relationship between path p and shortest paths from i to j with all intermediate vertices in the set $\{1,2,\ldots k-1\}$. The relationship depends on whether or not k is an intermediate vertex of path p.

If k is not an intermediate vertex of path p, then all intermediate vertices of path p are in the set $\{1,2,\ldots k-1\}$. Thus, a shortest path from vertex i to vertex j with all intermediate vertices in the set $\{1,2,\ldots k-1\}$ is also a shortest path from i to j with all intermediate vertices in the set $\{1,2,\ldots,k\}$.

If k is an intermediate vertex of path p, then we break p down into $i \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} j$.

Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, \ldots, k\}$.

A recursive definition is given by

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k=0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{if } k \geq 1) \end{cases}$$

```
FLOYD-WARSHALL (W)
1.  n ← rows[W]
2.  D^(0) ← W
3.  for k ← 1 to n
4.      do for i ← 1 to n
5.          do for j ← 1 to n
6.              do d_ij^(k) ← min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
7.  return D^(n)
```

The strategy adopted by the Floyd-Warshall algorithm is **dynamic programming.**

The running time of the Floyd-Warshall algorithm is determined by the triply nested for loops of lines 3-6. Each execution of line 6 takes $O(1)$ time. The algorithm thus runs in time $\theta(n^3)$.

29.3.1 Constructing a Shortest Path

There are a variety of different methods for constructing shortest paths in the Floyd-Warshall algorithm. One way is to compute the matrix D of shortest-path weights and then construct the predecessor matrix Π from the D matrix. This method can be implemented to run in $O(n^3)$ time. Given predecessor matrix Π, the PRINT-ALL-PAIRS-SHORTEST-PATH procedure can be used to print the vertices on a given shortest path.

We can compute the predecessor matrix Π just as the Floyd-Warshall algorithm computes the matrices $D(k)$. Specifically, we compute a sequence of matrices $\Pi^{(0)}, \Pi^{(1)}, \ldots, \Pi^{(n)}$, where $\Pi = \Pi^{(n)}$ and is defined to be the predecessor of vertex j on a shortest path from vertex i with all intermediate vertices in the set $\{1, 2, \ldots, k\}$.

The recursive formulation of $\pi_{ij}^{(k)}$. When $k=0$, a shortest path from i to j has no intermediate vertices at all. Thus,

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{if } i=j \text{ or } w_{ij}=\infty \\ i & \text{if } i=j \text{ and } w_{ij}<\infty \end{cases}$$

For $k \geq 1$, if we take the path $i \rightsquigarrow k \rightsquigarrow j$, where $k \neq j$ then the predecessor of j we choose is the same as the predecessor of j we chose on a shortest path from k with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. Otherwise, we choose the same predecessor of j that we chose on a shortest path from i with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. Formally, for $k \geq 1$,

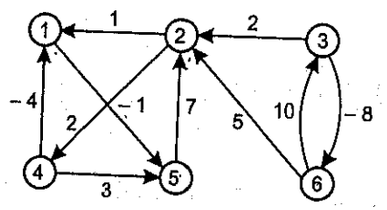
$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

Example. *How can the output of the Floyd-Warshall algorithm be used to detect the presence of a negative weight cycle ?*

Solution. Check for every edge (i, j) whether

$$w_{ij} + d_{ji} < 0$$

Example. *Apply Floyd-Warshall algorithm for constructing shortest path. Show the matrix $D^{(k)}$ that results each iteration.*



Solution.

$$D^{(0)} = \begin{bmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & \infty & \infty \\ \infty & 2 & 0 & \infty & \infty & -8 \\ -4 & \infty & \infty & 0 & 3 & \infty \\ \infty & 7 & \infty & \infty & 0 & \infty \\ \infty & 5 & 10 & \infty & \infty & 0 \end{bmatrix}$$

We know

$$d_{ij}^{(k)} = \min[d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}]$$

k	i	j	$d_{ij}^{(k-1)}$	$d_{ik}^{(k-1)}$	$d_{kj}^{(k-1)}$	d_{ij}^{k}
1	1	1	0	0	0	0
1	1	2	∞	0	∞	∞
1	1	3	∞	0	∞	∞
1	1	4	∞	0	∞	∞
1	1	5	−1	0	−1	−1
1	1	6	∞	0	∞	∞
1	2	1	1	1	0	1
1	2	2	0	1	∞	0
1	2	3	∞	1	∞	∞
1	2	4	2	1	∞	2
1	2	5	∞	1	−1	0
1	2	6	∞	1	∞	∞
1	3	1	∞	∞	0	∞
1	3	2	2	∞	∞	2
1	3	3	0	∞	∞	0
1	3	4	∞	∞	∞	∞
1	3	5	∞	∞	−1	∞
1	3	6	−8	∞	∞	−8
1	4	1	−4	−4	0	−4
1	4	2	∞	−4	∞	∞
1	4	3	∞	−4	∞	∞
1	4	4	0	−4	∞	0
1	4	5	3	−4	−1	−5
1	4	6	∞	−4	∞	∞
1	5	1	∞	∞	0	∞
1	5	2	7	∞	∞	7
1	5	3	∞	∞	∞	∞
1	5	4	∞	∞	∞	∞
1	5	5	0	∞	−1	0
1	5	6	∞	∞	∞	∞
1	6	1	∞	∞	0	∞
1	6	2	5	∞	∞	5
1	6	3	10	∞	∞	10
1	6	4	∞	∞	∞	∞
1	6	5	∞	∞	−1	∞
1	6	6	0	∞	∞	0

$D^{(1)}$	1	2	3	4	5	6
1	0	∞	∞	∞	−1	∞
2	1	0	∞	2	0	∞
3	∞	2	0	∞	∞	−8
4	−4	∞	∞	0	−5	∞
5	∞	7	∞	∞	0	∞
6	∞	5	10	∞	∞	0

Similarly, we get

$D^{(2)}$	1	2	3	4	5	6
1	0	∞	∞	∞	−1	∞
2	1	0	∞	2	0	∞
3	3	2	0	4	2	−8
4	−4	∞	∞	0	−5	∞
5	8	7	∞	9	0	∞
6	6	5	10	7	5	0

$D^{(3)}$	1	2	3	4	5	6
1	0	∞	∞	∞	−1	∞
2	1	0	∞	2	0	∞
3	3	2	0	4	2	−8
4	−4	∞	∞	0	−5	∞
5	8	7	∞	9	0	∞
6	6	5	10	7	5	0

$D^{(4)}$	1	2	3	4	5	6
1	0	∞	∞	∞	−1	∞
2	−2	0	∞	2	−3	∞
3	0	2	0	4	−1	−8
4	−4	∞	∞	0	−5	∞
5	5	7	∞	9	0	∞
6	3	5	10	7	2	0

$D^{(5)}$	1	2	3	4	5	6
1	0	6	∞	8	−1	∞
2	−2	0	∞	2	−3	∞
3	0	2	0	4	−1	−8
4	−4	2	∞	0	−5	∞
5	5	7	∞	9	0	∞
6	3	5	10	7	2	0

and

$D^{(6)}$	1	2	3	4	5	6
1	0	6	∞	8	−1	∞
2	−2	0	∞	2	−3	∞
3	−5	−3	0	−1	−6	−8
4	−4	2	∞	0	−5	∞
5	5	7	∞	9	0	∞
6	3	5	10	7	2	0

The shortest path from 3 to 1 is

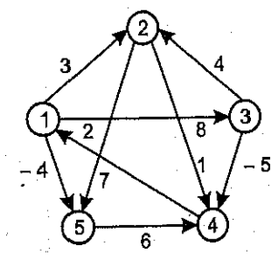
$3 \to 6 \to 2 \to 4 \to 1$ and is -5 units.

Example. *"As it appears above, the Floyd-Warshall algorithm requires $\Theta(n^3)$ space, since we compute $d_{ij}^{(k)}$ for $i, j, k = 1,2,3,\ldots,n$. Show that the following procedure, which simply drops all superscripts, is correct and thus only $\Theta(n^2)$ space is required".*

```
FLOYD-WARSHALL'(W)
n ← rows(W)
D ← W
for k ← 1 to n
    do for i ← 1 to n
        do for j ← 1 to n
            do d_ij ← min(d_ij, d_ik + d_kj)
return D.
```

Solution. Since $d_{ik}^{(k)} = d_{ik}^{(k-1)}$ and $d_{kj}^{(k-1)} = d_{kj}^{(k)}$, no entry with either subscript changes during iteration. Therefore we can perform the computation with only copy of D.

Example. *Apply Floyd-Warshall algorithm for constructing shortest path. Show the matrices $D^{(k)}$ and $\pi^{(k)}$ computed by the Floyd-Warshall algorithm for the graph.*



Solution. $d_{ij}^{(k)} = \min[d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}]$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \le d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \pi^{(0)} = \begin{bmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \pi^{(1)} = \begin{bmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \pi^{(2)} = \begin{bmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \pi^{(3)} = \begin{bmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \pi^{(4)} = \begin{bmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{bmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \pi^{(5)} = \begin{bmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{bmatrix}$$

29.4 Transitive Closure

The transitive closure of a graph G is defined as $G^* = (V, E^*)$ where

$$E^* = \{(i,j) \text{ there is a path from vertex } i \text{ to vertex } j \text{ in } G\}$$

For $i, j, k = 1,2,\ldots n$ we define $t_{ij}^{(k)}$ to be 1 if there exists a path in graph G from vertex i to vertex j with all intermediate vertices in the set $\{1,2,\ldots,k\}$ and 0 otherwise we construct the transitive closure $G^* = (V, E^*)$ by putting edge (i,j) into E^* if and only if $t_{ij}^{(n)} = 1$.

The recursive definition of $t_{ij}^{(k)}$ is

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{if } i \neq j \text{ and } (i,j) \notin E \\ 1 & \text{if } i = j \text{ or } (i,j) \in E \end{cases}$$

and for $k \geq 1$

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$$

TRANSITIVE-CLOSURE (G)

```
n ← |V[G]|
for i ← 1 to n
    do for j ← 1 to n
```



```
        do if i = j or (i, j) ∈ E[G]
            then t_ij^(0) ← 1
            else t_ij^(0) ← 0
for k ← 1 to n
    do for i ← 1 to n
        do for j ← 1 to n
            do t_ij^(k) ← t_ij^(k-1) ∨ (t_ik^(k-1) ∧ t_kj^(k-1))
```

Example. *Give an $O(VE)$-time algorithm for computing the transitive closure of a directed graph $G = (V, E)$.*

Solution.

1. Construct a new graph $G^* = (V, E^*)$
2. E^* is initially empty.
3. For each vertex v traverse the graph G adding edges for every node encountered in E^*.

   This takes $O(VE)$-time.

29.5 Johnson's Algorithm

Johnson's algorithm finds shortest paths between all pairs in $O(V^2 \lg V + VE)$ time. The algorithm returns a matrix of shortest path weight for all pairs or reports that the input graph contains a negative-weight cycle.

Johnson's algorithm use the technique of "**reweighting**". If all edge weights w in a graph $G = (V, E)$ are non-negative, we can find shortest paths between all pairs of vertices by running Dijkstra's algorithm once from each vertex. If G has negative-weight edges, we simply compute a new-set of non-negative edge weights that allows us to use the same method. The new set of edges weights $\hat{w}$ must satisfy *two* important properties :

1. For all pairs of vertices $u, v \in V$, a shortest path from u to v using weight function w is also a shortest path from u to v using weight function $\hat{w}$.
2. For all edges (u,v), the new weight $\hat{w}(u,v)$ is non-negative.

Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to R$ and let $h : v \to R$ be any function mapping vertices to real number.

For each edge $(u,v) \in E$ define

$$\hat{w}(u,v) = w(u,v) + h(u) - h(v)$$

where $h(u) =$ label of u

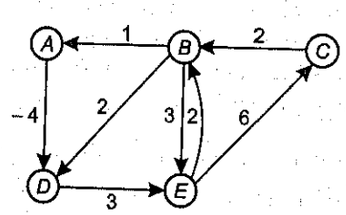
$h(v) =$ label of v

JOHNSON(G)

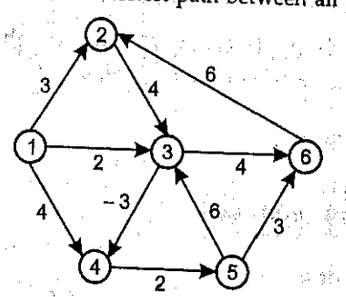
1. Compute G' where $V[G'] = V[G] \cup \{S\}$ and
   $E[G'] = E[G] \cup \{(s,v) : v \in V[G]\}$
2. If BELLMAN-FORD (G', w, s) = FALSE
   then "input graph contains a negative wt. cycle"
   else
   for each vertex $v \in V[G']$
   do $h(v) \leftarrow \delta(s,v)$
   computed by Bellman-Ford algorithm.
   for each edge $(u,v) \in E[G']$
   do $\hat{w}(u,v) \leftarrow w(u,v) + h(u) - h(v)$
   for each vertex $u \in V[G]$
   do run DIJKSTRA $(G, \hat{w}, u)$ to compute
   $\hat{\delta}(u,v)$ for all $v \in V[G]$
   for each vertex $v \in V[G]$
   do $d_{uv} \leftarrow \hat{\delta}(u,v) + h(v) - h(u)$
   return D.

Exercise

1. Write any algorithm to find all-pair shortest path. Drive its complexity.
2. For the graph (directed, weighted)
   Apply Floyd-Warshall algorithm for constructing shortest path. Show the matrix $D^{(k)}$ that results each iteration.



3. Use Johnson's algorithm to find the shortest path between all pairs of vertices in the graph.



4. Modify the Floyd-Warshall algorithm to find the negative weight cycle.

Maximum Flow

Some real-life problems, like those involving the flow of liquids through pipes, current through wires, and delivery of goods, can be modeled using flow networks. Other problems which seem unrelated to networks can also be modeled using flow networks. We discuss the basics of flow networks, several network flow problems, and develop a few algorithms to solve the maximum network flow problem.

30.1 Flow Networks and Flows

A **flow network** is a directed graph $G=(V,E)$ such that

1. For edge $(u,v) \in E$, we associate a nonnegative **capacity** $c(u,v) \geq 0$. If $(u,v) \notin E$, we assume that $c(u,v)=0$.
2. There are two distinguished points, the **source** s and the **sink** t;
3. For every vertex $v \in V$, there is a path from s to t containing v.

Let $G=(V,E)$ be a flow network. Let s be the source of the network, and let t be the sink. A *flow* in G is a real-valued function $f : V \times V \rightarrow \mathbf{R}$ such that the following properties hold :

- **Capacity constraint.** For all $u,v \in V$, we require $f(u,v) \leq c(u,v)$.
- **Skew symmetry.** For all $u,v \in V$, we require $f(u,v) = -f(v,u)$.

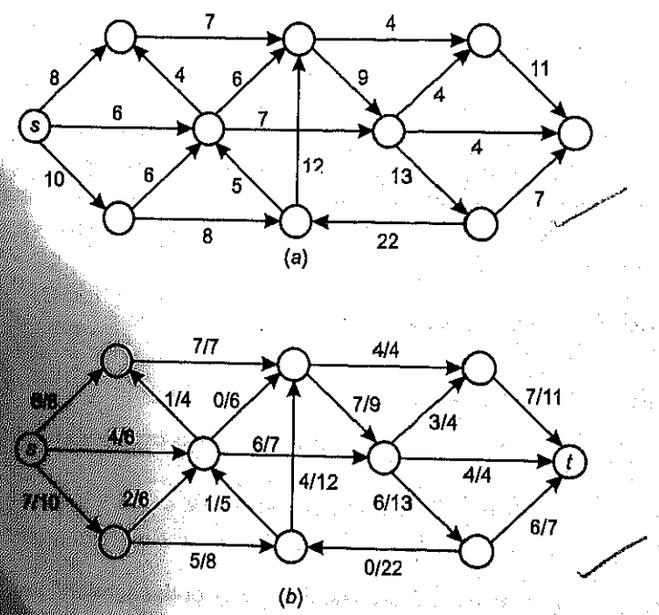
◂| **Flow conservation.** For all $u \in V-\{s,t\}$, we require

$$\sum_{v \in V} f(u,v) = \sum_{u \in V} f(u,v) = 0$$

The quantity $f(u,v)$, which can be positive or negative, is called the ***net flow*** from vertex u to vertex v. In the ***maximum-flow problem***, we are given a flow network G with source s and sink t, and we wish to find a flow of maximum value from s to t.

The *three* properties can be described as follows :

1. **Capacity constraint** makes sure that the flow through each edge is not greater than the capacity.
2. **Skew symmetry** simply means that the flow from u to v is the negative of the flow from v to u.
3. The **flow-conservation property** says that the total net flow out of a vertex other than the source or sink is 0. In other words, the amount of flow into a v is the same as the amount of flow out of v for every vertex $v \in V-\{s,t\}$



rk $G=(V,E)$ and Fig. 30.1(*b*) a flow f in G. Only the positive flows are shown. beled with *flow/capacity*. For example, 3/4 means the flow ity is 4.

The **value** of the flow is the net flow from the source,

$$|f| = \sum_{v \in V} f(s,v)$$

The **positive net flow** entering a vertex v is defined by

$$\sum_{\{u \in V:\, f(u,v)>0\}} f(u,v)$$

The positive net flow leaving a vertex is defined symmetrically. One interpretation of the flow-conservation property is that the positive net flow entering a vertex other than the source or sink must equal the positive net flow leaving the vertex.

A flow f is said to be **integer-valued** if $f(u,v)$ is an integer for all $(u,v) \in E$. Clearly the value of the flow is an integer in an integer-valued flow.

Implicit summation notation

For convenience, we will omit the set braces when it simplifies notation. For instance, we will write $V-s-t$ instead of $V-\{s,t\}$ The following notation will greatly simplify the writing of proofs involving functions like the net flow f.

If X and Y are sets, and g is a 2-variable function, we define

$$g(X,Y) = \sum_{x \in X} \sum_{y \in Y} g(x,y)$$

For example, the flow-conservation constraint can be expressed as the condition that $f(u,V)=0$ for all $u \in V-s-t$.

Lemma

If $X \subseteq V-s-t$, then $f(V,X) = f(X,V) = 0$.

Proof. Since $X \subseteq V-s-t$,,

$$f(V,X) = \sum_{x \in X} f(V,x) = \sum_{x \in X} 0 = 0$$

The proof is similar for $f(X,V)=0$.

The following identities will also be useful. Let $G=(V,E)$ be a flow network, and let f be a flow in G. Then, for $X,Y,Z \subseteq V$ and $Y \cap Z = \phi$,

◂| $f(X,X)=0$.

◂| $f(X,Y) = -f(Y,X)$.

◂| $f(X \cup Y, Z) = f(X,Z) + f(Y,Z)$ and

◂| $f(Z, X \cup Y) = f(Z,X) + f(Z,Y)$

We can prove that the value of a flow is the total net flow into the sink ; that is,

$$|f| = f(V,t)$$

◄ **Flow conservation.** For all $u \in V-\{s,t\}$, we require

$$\sum_{v \in V} f(u,v) = \sum_{u \in V} f(u,v) = 0$$

The quantity $f(u,v)$, which can be positive or negative, is called the ***net flow*** from vertex u to vertex v. In the ***maximum-flow problem,*** we are given a flow network G with source s and sink t, and we wish to find a flow of maximum value from s to t.

The *three* properties can be described as follows :

1. **Capacity constraint** makes sure that the flow through each edge is not greater than the capacity.
2. **Skew symmetry** simply means that the flow from u to v is the negative of the flow from v to u.
3. The **flow-conservation property** says that the total net flow out of a vertex other than the source or sink is 0. In other words, the amount of flow into a v is the same as the amount of flow out of v for every vertex $v \in V-\{s,t\}$

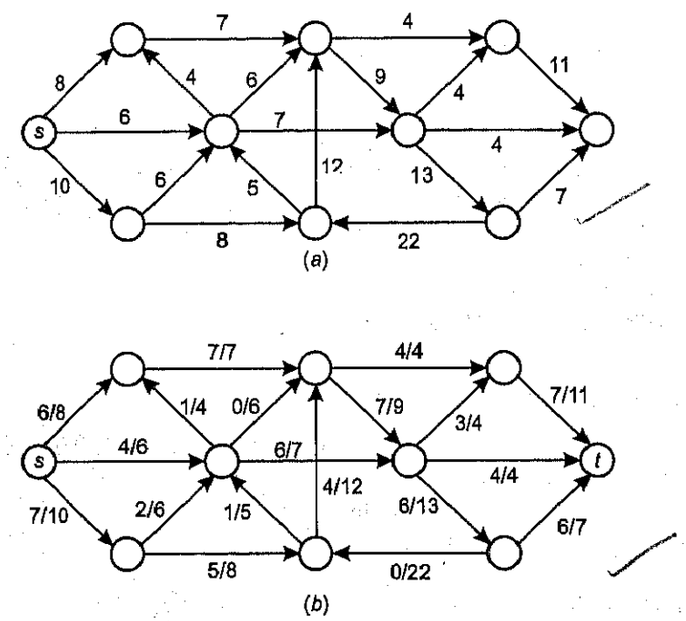


Figure 30.1 (*a*) A flow network $G=(V,E)$ and Fig. 30.1(*b*) a flow f in G. Only the positive flows are shown. Each edge is labeled with *flow/capacity*. For example, 3/4 means the flow is 3 and the capacity is 4.

The **value** of the flow is the net flow from the source,

$$|f| = \sum_{v \in V} f(s,v)$$

The **positive net flow** entering a vertex v is defined by

$$\sum_{\{u \in V:\, f(u,v)>0\}} f(u,v)$$

The positive net flow leaving a vertex is defined symmetrically. One interpretation of the flow-conservation property is that the positive net flow entering a vertex other than the source or sink must equal the positive net flow leaving the vertex.

A flow f is said to be **integer-valued** if $f(u,v)$ is an integer for all $(u,v) \in E$. Clearly the value of the flow is an integer in an integer-valued flow.

Implicit summation notation

For convenience, we will omit the set braces when it simplifies notation. For instance, we will write $V-s-t$ instead of $V-\{s,t\}$ The following notation will greatly simplify the writing of proofs involving functions like the net flow f.

If X and Y are sets, and g is a 2-variable function, we define

$$g(X,Y) = \sum_{x \in X} \sum_{y \in Y} g(x,y)$$

For example, the flow-conservation constraint can be expressed as the condition that $f(u,V)=0$ for all $u \in V-s-t$.

Lemma

If $X \subseteq V-s-t$, then $f(V,X) = f(X,V) = 0$.

Proof. Since $X \subseteq V-s-t$,,

$$f(V,X) = \sum_{x \in X} f(V,x) = \sum_{x \in X} 0 = 0$$

The proof is similar for $f(X,V)=0$.

The following identities will also be useful. Let $G=(V,E)$ be a flow network, and let f be a flow in G. Then, for $X,Y,Z \subseteq V$ and $Y \cap Z = \phi$,

◄ $f(X,X)=0$.

◄ $f(X,Y) = -f(Y,X)$.

◄ $f(X \cup Y, Z) = f(X,Z) + f(Y,Z)$ and

◄ $f(Z, X \cup Y) = f(Z,X) + f(Z,Y)$

We can prove that the value of a flow is the total net flow into the sink ; that is,

$$|f| = f(V,t)$$

This is intuitively true, since all vertices other than the source and sink have a net flow of 0 by flow conservation, and thus the sink is the only other vertex that can have a nonzero net flow to match the source's nonzero net flow. Our formal proof goes as follows :

$$
\begin{aligned}
|f| &= f(s,V) && \textit{(by definition)}\\
&= f(V,V)-f(V-s,V) && \textit{(by Lemma)}\\
&= f(V,V-s) && \textit{(by Lemma)}\\
&= f(V,t)+f(V,V-s-t) && \textit{(by Lemma)}\\
&= f(V,t) && \textit{(by flow conservation)}
\end{aligned}
$$

30.2 Network Flow Problems

The most obvious flow network problem is the following.

Problem 1. *Given a flow network $G=(V,E)$, the **maximum flow problem** is to find a flow with maximum value.*

Problem 2. *The **multiple source and sink maximum flow problem** is similar to the maximum flow problem, except there is a set $\{s_1,s_2,s_3\ldots s_n\}$ of sources and a set $\{t_1,t_2,t_3\ldots t_n\}$ of sinks.*

Fortunately, this problem is no harder than ordinary maximum flow. Given a multiple source and sink flow network G, we define a new flow network G′ by adding

- A super source s,
- A super sink t,
- For each s_i, add edge (s,s_i) with capacity ∞, and
- For each t_i, add edge (t_i,t) with capacity ∞

Figure shows a multiple source and sinks flow network and an equivalent single source and sink flow network.

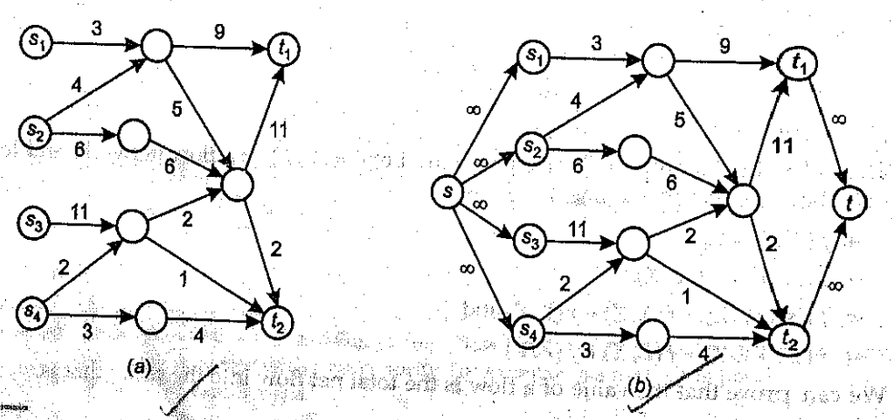


Figure 30.2

30.3 Residual Networks, Augmenting Paths, and Cuts

The residual network consists of edges that can admit more net flow. More formally, suppose that we have a flow network $G=(V,E)$ with source s and sink t. Let f be a flow in G, and consider a pair of vertices $u,v \in V$. The amount of **additional net flow** we can push from u to v before exceeding the capacity $c(u,v)$ is the **residual capacity** of (u,v), given by

$$c_f(u,v)=c(u,v)-f(u,v).$$

When the net flow $f(u,v)$ is negative, the residual capacity $c_f(u,v)$ is greater than the capacity $c(u,v)$. For example, if $c(u,v)=16$ and $f(u,v)=-4$, then the residual capacity $c_f(u,v)$ is 20.

Given a flow network $G=(V,E)$ and a flow f, the ***residual network*** of G induced by f is $G_f=(V,E_f)$, where

$$E_f=\{(u,v)\in V\times V : c_f(u,v)\geq 0\}$$

That is, each edge of the residual network, or ***residual edge***, can admit a strictly positive net flow.

Lemma

Let $G=(V,E)$ be a flow network with flow f. G_f be the residual network of G induced by f, and let f_r be a flow in G_f. Let $f'=f+f_r$ that is for each pair $u,v\in V$, $f'(u,v)=f(u,v)+f_r(u,v)$. Then f' is a flow in G with value $|f'|=|f|+|f_r|$.

Proof. We need to verify that the 3 properties of a flow are satisfied

1. By definition

$$f_r(u,v)\leq c_f(u,v)=c(u,v)-f(u,v) \text{ for all } u,v\in V.$$

Given this, we can demonstrate capacity constraint as follows

$$
\begin{aligned}
f'(u,v) &= f(u,v)+f_r(u,v)\\
&\leq f(u,v)+c(u,v)-f(u,v)\\
&= c(u,v)
\end{aligned}
$$

2. To see that f' has skew symmetry, notice that $u,v\in V$.

$$
\begin{aligned}
f'(u,v) &= f(u,v)+f_r(u,v)\\
&= -f(v,u)-f_r(v,u)\\
&= -(f(v,u)+f_r(v,u))\\
&= -f'(v,u)
\end{aligned}
$$

3. f' has flow conservation, since for all $u \in V-s-t$

$$\sum_{v\in V} f'(u,v) = \sum_{v\in V}(f(u,v)+f_r(u,v))$$

$$= \sum_{v\in V} f(u,v) + \sum_{v\in V} f_r(u,v)$$

$$=0+0=0$$

Finally we can easily see that

$$|f'| = \sum_{v\in V} f'(s,v)$$

$$= \sum_{v\in V}(f(s,v)+f_r(s,v)$$

$$= \sum_{v\in V} f(s,v) + \sum_{v\in V} f_r(s,v)$$

$$=|f|+|f_r|.$$

Given a flow network $G=(V,E)$ and a flow f, an **augmenting path** p is a simple path from s to t in the residual network G_f. By the definition of the residual network, each edge (u,v) on an augmenting path admits some additional positive net flow from u to v without violating the capacity constraint on the edge.

Let $G=(V,E)$ be a flow network with flow f. The **residual capacity** of an augmenting path p is

$$c_f(p) = \min\{c_f(u,v):(u,v) \text{ is on } p\}.$$

The residual capacity is the maximal amount of flow that can be pushed through the augmenting path. Notice that if there is an augmenting path, then each edge on it has positive capacity. We will use this fact to compute a maximum flow in a flow network.

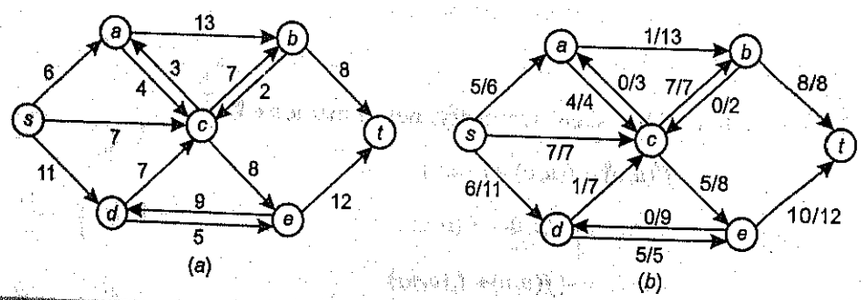


Figure 30.3 (a) A flow network $G=(V,E)$ (b) A flow in f in G

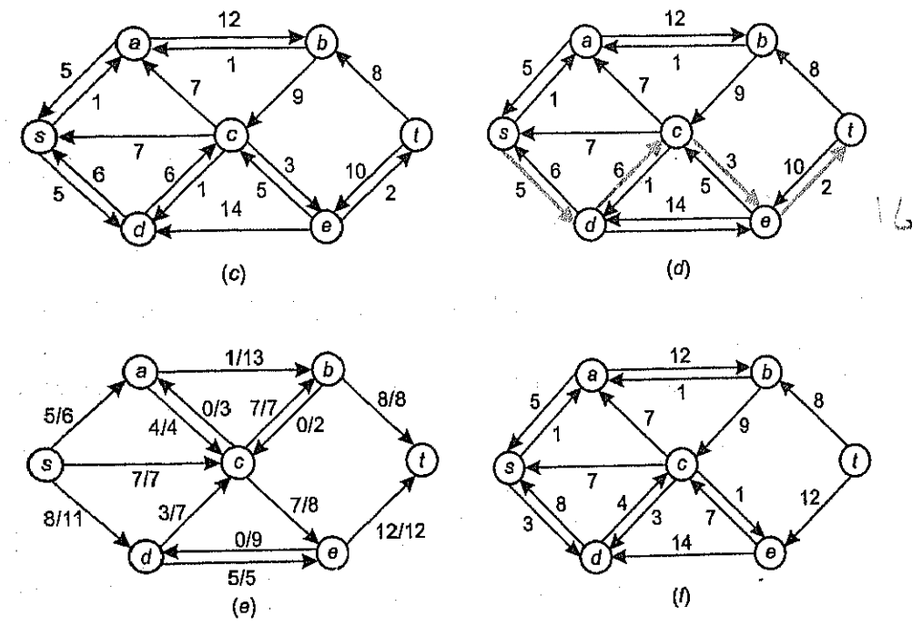


Figure 30.3 (c) The residual network G_f. (d) The grey edges form an augmenting path with capacity 2. (e) A new flow $f'=f+f_p$ (f) The residual network G_f.

Corollary. Let $G=(V,E)$ be a flow network, f a flow in G, and p an augmenting path in G_f. Then $f'=f+f_p$ is a flow in G with value $|f'|=|f|+|f_p|>|f|$.

A **cut** (S,T) of flow network $G=(V,E)$ is a partition of V into S and $T=V-S$ such that $s\in S$ and $t\in T$. If f is a flow, then the **net flow** across the cut (S,T) is defined to be $f(S,T)$. The **capacity** of the cut (S,T) is $c(S,T)$.

For example, consider the flow network $G=(V,E)$ with flow f from Fig 30.3(b). Let $(\{s,a,b,c,d\},\{e,t\})$ be a cut in G. Then the flow across the cut is

$$f(b,t)+f(c,e)+f(d,e)+f(e,d)=8+5+5+0=18$$

and the capacity is

$$c(b,t)+c(c,e)+c(d,e)=8+8+5=21$$

Observe that the net flow across a cut can include negative net flows between vertices, but that the capacity of a cut is composed entirely of non-negative values.

Lemma

Let f be a flow in a flow network G with source s and sink t, and let (S,T) be a cut of G. Then, the net flow across (S,T) is $f(S,T)=|f|$.

Proof. It is straight forward to see that

$$f(S,T) = f(S,V) - f(S,S)$$
$$= f(S,V)$$
$$= f(s,V) + f(S-s,V)$$
$$= f(s,V)$$
$$= |f|.$$

Corollary. *Let $G=(V,E)$ be a flow network. Then the value of any flow in is bounded above by the capacity of any cut.*

Proof. Let (S, T) be any cut of G and let f be any flow . Using above lemma and the capacity constraints

$$|f| = f(S,T)$$
$$= \sum_{u \in S} \sum_{v \in T} f(u,v)$$
$$\le \sum_{u \in S} \sum_{v \in T} c(u,v)$$
$$= c(S,T).$$

30.4 Max-flow min-cut Theorem

If f is a flow in a flow network $G=(V,E)$ with source s and sink t, then the following conditions are equivalent :

1. f is a maximum flow in G.
2. The residual network G_f contains no augmenting paths.
3. $|f| = c(S,T)$ for some cut (S, T) of G.

Proof. **(1) ⇒ (2) :** Suppose f is a maximum flow in G but that G_f has an augmenting path p. Then, by Corollary, the flow sum $f' = f + f_p$, where f_p is a flow in G with value strictly greater than $|f|$, contradicting the assumption that f is a maximum flow.

(2) ⇒ (3) : Suppose that G_f has no augmenting path, that is, that G_f contains no path from s to t. Define

$$S = \{v \in V : \text{There exists a path from s to } v \text{ in } G_f\}$$

and $T = V - S$. The partition (S, T) is a cut: we have $s \in S$ and $t \notin S$ because there is no path from s to T in G_f. For each pair of vertices u and v such that $u \in S$ and $v \in T$, we have $f(u,v) = c(u,v)$, since otherwise $(u,v) \in E_f$ and v is in set S. By Lemma, therefore, $|f| = f(S,T) = c(S,T)$.

(3) ⇒ (1) : By Corollary, $|f| \le c(S,T)$ for all cuts (S, T). The condition $|f| = c(S,T)$ thus implies that f is a maximum flow.

30.5 Ford-Fulkerson Algorithm

The Ford-Fulkerson method is iterative. We start with $f(u,v)=0$ for all $u,v \in V$, giving an initial flow of value 0. At each iteration, we increase the flow value by finding an "augmenting path," which we can think of simply as a path from the source s to the sink t along which we can push more flow, and then augmenting the flow along this path. We repeat this process until no augmenting path can be found. The max-flow min-cut theorem will show that upon termination, this process yields a maximum flow.

```
FORD-FULKERSON-METHOD (G, s, t)

1.  initialize flow f to 0
2.  while there exists an augmenting path p
3.      do augment flow f along p
4.  return f
```

The FORD-FULKERSON algorithm simply expands on the FORD-FULKER-SON-METHOD pseudo code given above. Lines 1-3 initialize the flow f to 0. The **while** loop of lines 4-8 repeatedly finds an augmenting path p in G_f and augments flow f along p by the residual capacity $c_f(p)$. When no augmenting paths exist, the flow f is a maximum flow.

```
FORD-FULKERSON (G, s, t)

1.  for each edge (u,v) ∈ E[G]
2.  do f[u,v] ← 0
3.  f[v,u] ← 0
4.  while there exists a path p from s to t in the residual network G_f
5.  do c_f(p) ← min {c_f(u,v) : (u,v) is in p}
6.  for each edge (u,v) in p
7.      do f[u,v] ← f[u,v] + c_f(p)
8.      f[v,u] ← −f[u,v]
```

Analysis of Ford-Fulkerson

The running time of FORD-FULKERSON depends on how the augmenting path p in line 4 is determined. If it is chosen poorly, the algorithm might not even terminate: the value of the flow will increase with successive augmentations, but it need not even converge to the maximum flow value. If the augmenting path is chosen by using a breadth-first search, the algorithm runs in polynomial time.

Example. *Using the definition of a flow, prove that if $(u,v) \notin E$ and $(v,u) \notin E$ then*

$$f(u,v) = f(v,u) = 0$$

Solution. Assume $(u,v) \notin E$ and $(v,u) \notin E$ then by capacity constraint $f(u,v) \le 0$ and $f(v,u) \le 0$. By skew symmetry $f(u,v) = f(v,u) = 0$.

Example. *Given a flow network $G=(V,E)$. Let f_1 and f_2 be functions from $V\times V$ to R. The flow sum f_1+f_2 is the function from $V\times V$ to R defined by*

$$(f_1+f_2)(u,v)=f_1(u,v)+f_2(u,v) \qquad \forall \quad u,v\in V$$

If f_1 and f_2 are flows in G, which of the three flow properties must the flow sum f_1+f_2 satisfy, and which might it violate ?

Solution. Capacity constraint. May clearly be violated

Skew symmetry. We have

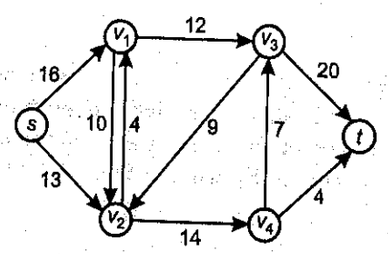
$$\begin{aligned}(f_1+f_2)(u,v)&=f_1(u,v)+f_2(u,v)\\&=-f_1(v,u)-f_2(v,u)\\&=-(f_1(v,u)+f_2(v,u))\\&=-(f_1+f_2)(v,u)\end{aligned}$$

Flow conservation. Let $u\in V-s$ be given. Then

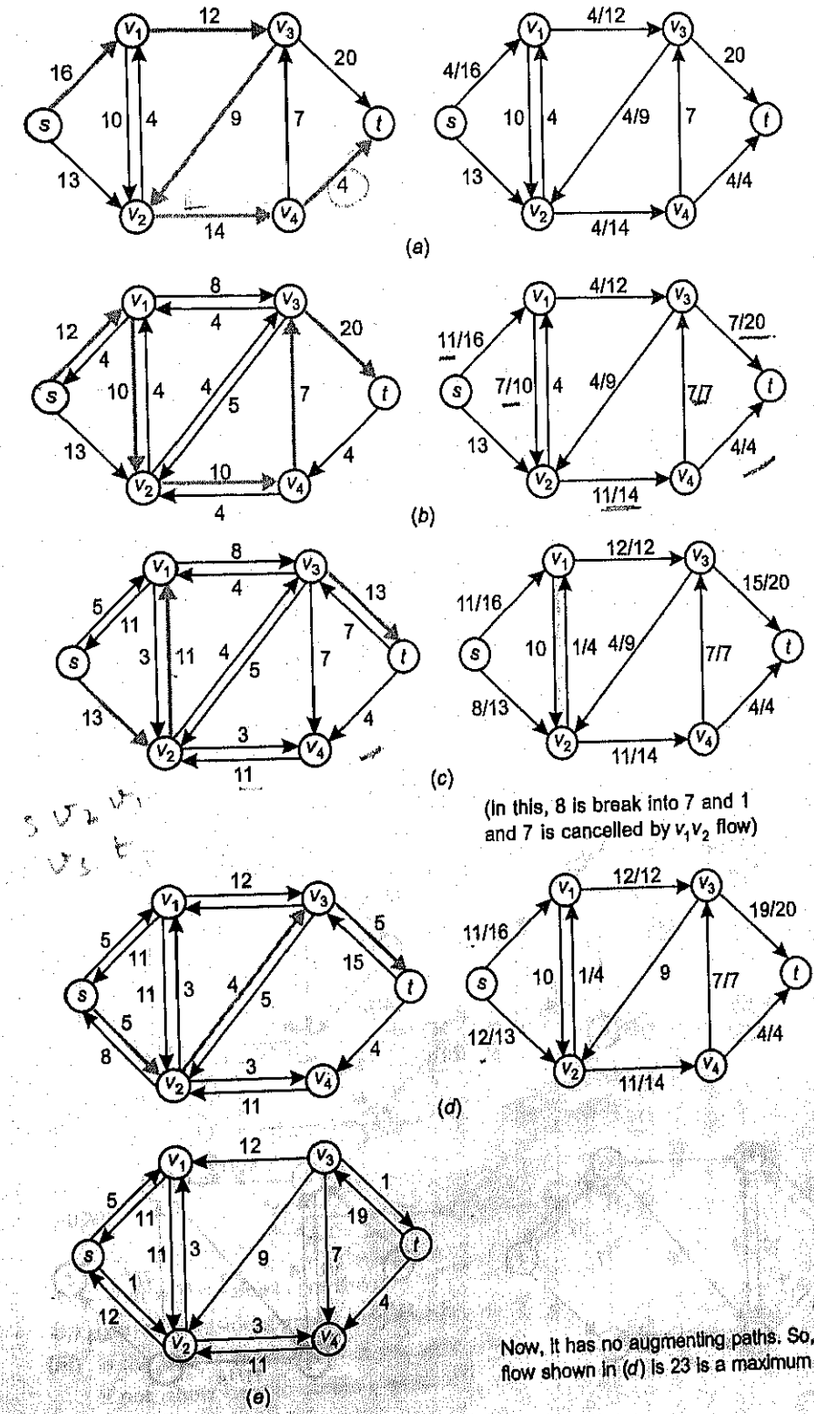
$$\sum_{v\in V}(f_1+f_2)(u,v)=\sum_{v\in V}(f_1(u,v)+f_2(u,v))$$

$$=\sum_{v\in V}f_1(u,v)+\sum_{v\in V}f_2(u,v)$$

Example. *In the flow network illustrated below, each directed edge is labelled with its capacity. Use the Ford-Fulkerson algorithm to find the maximum flow.*

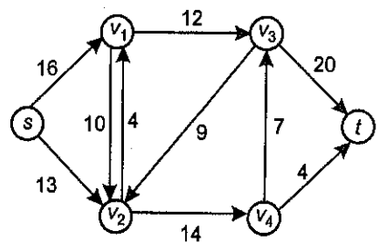


Solution. The left side of each part shows the residual network G_f with a shaded augmenting path p and the right side of each part shows the net flow f.

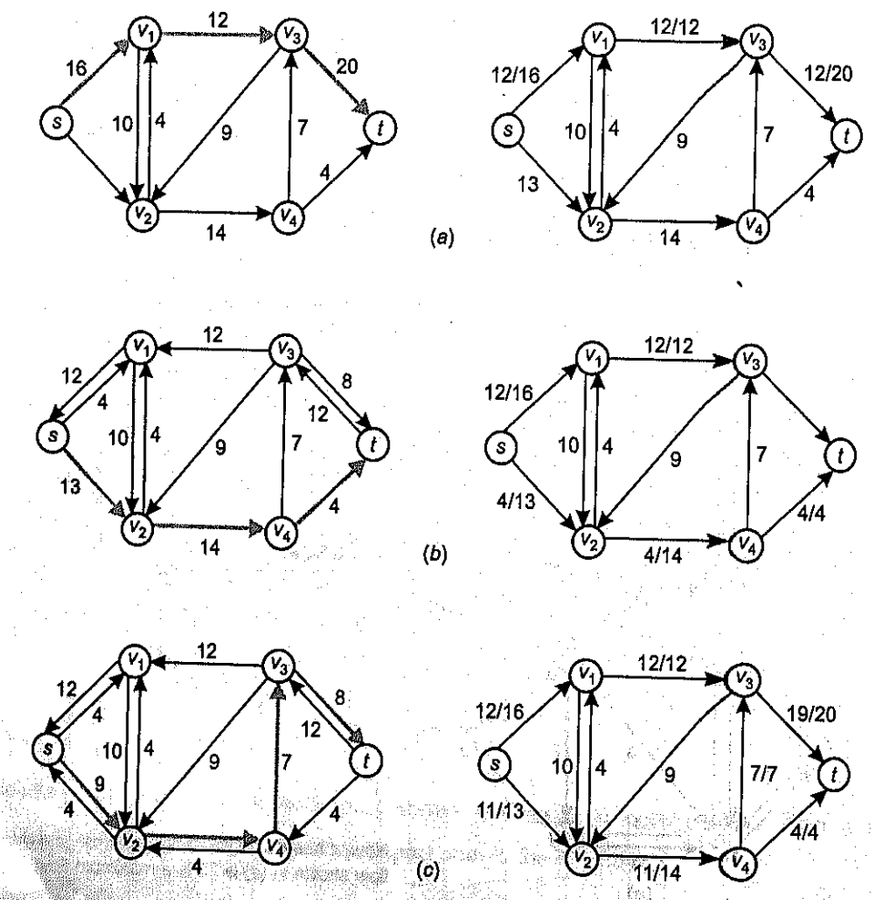


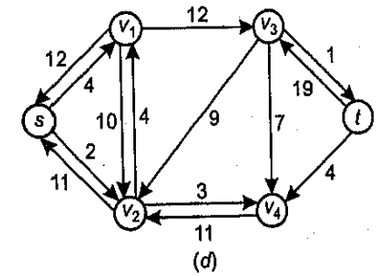
Now, it has no augmenting paths. So, the maximum flow shown in (d) is 23 is a maximum flow.

Example. *Show the execution of the Edmonds-Karp algorithm on the flow network shown in the figure.*



Solution. The bound on FORD-FULKERSON can be improved if we implement the computation of the augmenting path p with a breadth-first search, that is, if the augmenting path is a *shortest path* from s to t in the residual network. We call the Ford-Fulkerson method so implement the Edmonds-Karp algorithm. This algorithm runs in $O(VE^2)$ time.





It has no augmenting paths. So the flow f shown in (c) is therefore a maximum flow =23

Example. *What is the worst-case running time of the Ford-Fulkerson algorithm if all edge capacities are bounded by a constant ?*

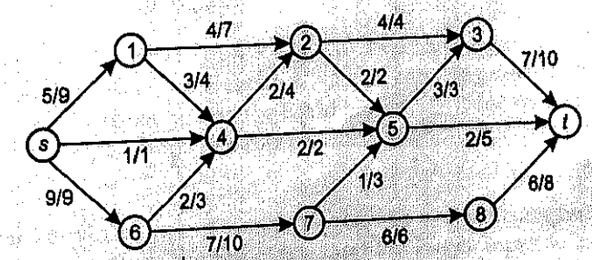
Solution. If every edge capacity is bounded by a constant m, then the maximum flow must be bounded by $O(m)$ and the Ford-Fulkerson algorithm would run in $O(m^2)$ in worst-case.

Example. *Let N be a flow network with n vertices and m edges. Show how to compute an augmenting path with the largest residual capacity in $O((n+m)\log n)$ time.*

Solution. To compute an augmenting path with the largest residual capacity, we use maximum spanning tree algorithm, which is just like a minimum spanning tree algorithm with all the weights multiplied by (−1).

Exercise

1. We are given a flow f in the following flow network. On each edge, the label f/c represents the flow f and the capacity c on the edge :

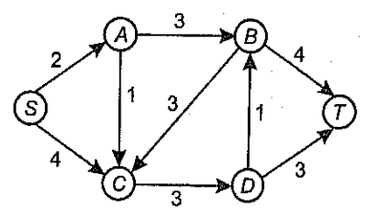


(*i*) What is the value $|f|$ of the flow ?

(*ii*) Draw the residual graph corresponding to f.

(*iii*) Is the flow f maximum ? If yes, state why. If not, show how to obtain a large flow.

2. In the flow network illustrated below, each directed edge is labelled with its capacity. Use the Ford-Fulkerson algorithm to find the maximum flow. The first augmenting path is S-A-C-D-T and the second augmenting path S-A-B-C-D-T.



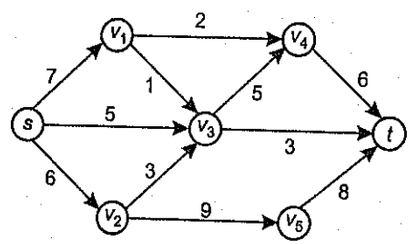
   (*i*) Draw the residual network after we have updated the flow using these two augmenting paths.

   (*ii*) What is the numerical value of the maximum flow. Draw a dotted line through the original graph to represent the minimum cut.

3. Draw a flow network with 9 vertices and 12 edges. Illustrate an execution of the Ford-Fulkerson algorithm on it.

4. Show that, given a maximum flow in a network with m edges, a minimum cut of N can be computed in $O(m)$ time.

5. Illustrate the execution of the Ford-Fulkerson algorithm in the flow network.



CHAPTER 31

Sorting Networks

31.1 Comparison Networks

A comparison network is made of wires and comparators. A **comparator**, is a device with two inputs, x and y, and two outputs, x' and y', where

$$x' = \min(x, y)$$
$$y' = \max(x, y)$$

Comparison networks differ from RAM's in two important respects. *First*, they can only perform comparisons. Thus, an algorithm such as counting sort cannot be implemented on a comparison network. *Second*, unlike the RAM model, in which operations occur serially - that is, one after another-operations in a comparison network may occur at the same time, or "in parallel."

In comparison networks inputs appear on the left and outputs on the right, with the smaller input value appearing on the top output and the larger input value appearing on the bottom output. We can thus think of a comparator as sorting its two inputs. We shall assume that each comparator operates in $O(1)$ time. In other words, we assume that the time between the appearance of the input values x and y and the production of the output values x' and y' is a constant.

A wire transmits a value from place to place. A comparison network contains n input wires $a_1, a_2, \ldots, a_n$, through which the values to be sorted enter the network, and n output wires $b_1, b_2, \ldots, b_n$ which produce the results computed by the network.

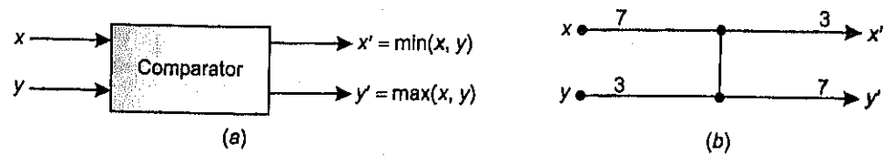


Figure 31.1

Comparison network is a set of comparators interconnected by wires. Under the assumption that each comparator takes unit time, we can define the "running time" of a comparison network. Running time of comparator can define in terms of **depth**. More formally, we define the **depth** of a wire as follows. An input wire of a comparison network has depth 0. Now, if a comparator has two input wires with depths d_x and d_y, then its output wires have depth $\max(d_x, d_y)+1$.

A ***sorting network*** is a comparison network for which the output sequence is monotonically increasing (that is, $b_1 \le b_2 \le \ldots \le b_n$) for every input sequence. Thus not every comparison network is a sorting network.

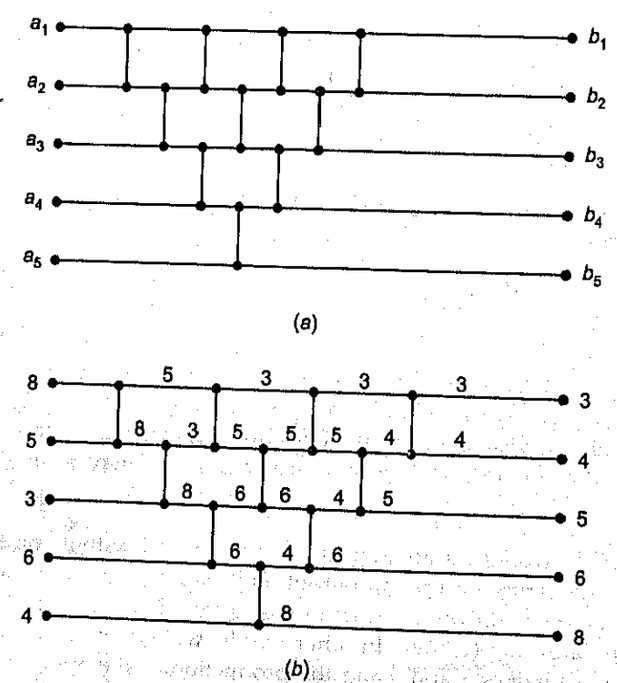


Figure 31.2 A sorting network based on insertion sort.

A comparison network is like a procedure in that it specifies how comparisons are to occur, but it is unlike a procedure in that its physical size depends on the number of inputs and outputs. Therefore, we shall actually be describing "families" of comparison networks. Any sorting network on n inputs has depth at least lg n the number of comparators in any sorting network is at least $\Omega(n \lg n)$.

The zero-one Principle

The *zero-one principle* says that if a sorting network works correctly when each input is drawn from the set {0,1}, then it works correctly on arbitrary input numbers. (The numbers can be integers, reals, or, in general, any set of values from any linearly ordered set.) The proof of the zero-one principle relies on the notion of a monotonically increasing function.

Lemma

If a comparison network transforms the input sequence a = <a1, a2, . . . , an> into the output sequence $b = \langle b_1, b_2, \ldots, b_n \rangle$, *then for any monotonically increasing function f, the network transforms the input sequence* $f(a) = \langle f(a_1), f(a_2), \ldots, f(a_n) \rangle$ *into the output sequence* $f(b) = \langle f(b_1), f(b_2), \ldots, f(b_n) \rangle$.

Proof. If f is a monotonically increasing function, then a single comparator with inputs $f(x)$ and $f(y)$ produces outputs $f(\min(x,y))$ and $f(\max(x,y))$. Let us consider a comparator whose input values are x and y. The upper output of the comparator is $\min(x,y)$ and the lower output is

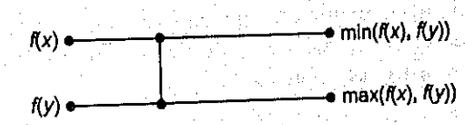


Figure 31.3

$\max(x,y)$. Suppose we now apply $f(x)$ and $f(y)$ to the inputs of the comparator, as is shown in Fig. 31.3. The operation of the comparator yields the value $\min(f(x), f(y))$ on the upper output and the value $\max(f(x), f(y))$ on the lower output. Since f is monotonically increasing, $x \le y$ implies $f(x) \le f(y)$. Thus, we have the identities

$$\min(f(x), f(y)) = f(\min(x,y))$$
$$\max(f(x), f(y)) = f(\max(x,y))$$

Thus, the comparator produces the values $f(\min(x,y))$ and $f(\max(x,y))$ when $f(x)$ and $f(y)$ are its inputs, which completes the proof.

(Zero-one Principle)

·k with n inputs sorts all 2^n possible sequences of 0's and 1's correctly, and then ·ary numbers correctly.

he contradiction i.e. the network sorts all zero-one sequences, but there ry numbers that the network does not correctly sort. That is, there exists $\ldots, a_n\rangle$ containing elements a_i and a_j such that $a_i < a_j$, but the network utput sequence. We define a monotonically increasing function f as

$$f(x)=\begin{cases}0 & \text{if } x \le a_i \\ 1 & \text{if } x > a_i\end{cases}$$

y monotonically increasing function f, the network transforms the input $a_2),\ldots, f(a_n)\rangle$ into the output sequence $f(b)=\langle f(b_1), f(b_2),\ldots, f(b_n)\rangle$. By work places $f(a_j)$ before $f(a_i)$ in the output sequence when input. But since $f(a_j)=1$ and $f(a_i)=0$, we obtain the contradiction that he zero-one sequence $\langle f(a_1), f(a_2),\ldots, f(a_n)\rangle$ correctly. Hence theorem is

ıg Network

ther monotonically increases and then monotonically decreases, or else ınd then monotonically increases is called bitonic sequence. For example, ⟩ and ⟨8,7,5,2,4,6⟩ are both bitonic. The **bitonic sorter** is a comparison sequences of 0's and 1's.

comprised of several stages, each of which is called a **half-cleaner**. Each on network of depth 1 in which input line i is compared with line $1+n/2$ ume that n is even.)

quence of 0's and 1's is applied as input to a half-cleaner, the half-cleaner uence in which smaller values are in the top half, larger values are in the halves are bitonic. In fact, at least one of the halves is clean-consisting of and it is from this property that we derive the name "**half-cleaner.**" (Note es are bitonic.)

combining half-cleaners, we can build a bitonic sorter, which is a network uences. The first stage of BITONIC-SORTER[n] consists of HALF-CLEANER[n], bitonic sequences of half the size such that every element in the top half is at ry element in the bottom half. Thus, we can complete the sort by using two SORTER [$n/2$] to sort the two halves recursively.

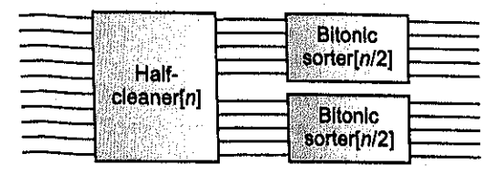


Figure 31.4

The depth D(n) of BITONIC-SORTER[n] is given by the recurrence whose solution is $D(n)=\lg n$.

31.3 Merging Network

We modify BITONIC-SORTER[n] to create the merging network MERGER[n]. Merging networks can merge two sorted input sequences into one sorted output sequence. The merging network is based on the following intuition. Given two sorted sequences, if we reverse the order of the second sequence and then concatenate the two sequences, the resulting sequence is bitonic. For example, given the sorted zero-one sequences $X=00000111$ and $Y=00001111$, we reverse Y to get $Y^R=11110000$. Concatenating X and Y^R yields 0000011111110000, which is bitonic.

The sorting network SORTER[n] uses the merging network to implement a parallel version of merge sort. The first stage of SORTER[n] consists of $n/2$ copies of MERGER[2] that work in parallel to merge pairs of 1-element sequences to produce sorted sequences of length 2. The second stage consists of $n/4$ copies of MERGER [4] that merge pairs of these 2-element sorted sequences to produce sorted sequences of length 4. In general, for $k=1,2,\ldots,\lg n$, stage k consists of $n/2^k$ copies of MERGER[2^k] that merge pairs of the 2^k-1-element sorted sequences to produce sorted sequences of length 2^k. At the final stage, one sorted sequence consisting of all the input values is produced. This sorting network can be shown by induction to sort zero-one sequences, and consequently, by the zero-one principle, it can sort arbitrary values.

The depth of SORTER[n] is given by the recurrence

$$D(n)=\begin{cases}0 & \text{if } n=1 \\ D(n/2)+\lg n & \text{if } n=2^k \text{ and } k\ge 1\end{cases}$$

whose solution is $D(n)=\theta(\lg^2 n)$. Thus, we can sort n numbers in parallel in $O(\lg^2 n)$ time.

Exercise

1. Prove that any sorting network, on n input has a depth alleast $\lg n$
2. Prove that the number of comparators in any sorting network is $\Omega(n\lg n)$.

3. Let n be an exact power of 2. Show how to construct an n-input, n-output comparison network of depth $\lg n$ in which the top output wire always carries the minimum input value and the bottom output wire always carries the maximum input value.
4. State and prove an analog of the zero-one principle for a decision-tree model.
5. How many zero-one bitonic sequences of length n are there ?
6. How many different zero-one input sequences must be applied to the input comparison network to verify that it is merging network ?
7. Prove that any merging network, regardless of the order of inputs, requires $\Omega(n \lg n)$ comparators.
8. Show that any network that can merge 1 item with $n-1$ sorted items to produce a sorted sequence of length n must have depth atleast $\lg n$.

CHAPTER

32

Algorithms for Parallel Computers

32.1 Parallel Computing

Traditionally, software has been written for **serial** computation :

- To be run on a single computer having a single Central Processing Unit (CPU) ;
- A problem is broken into a discrete series of instructions.
- Instructions are executed one after another.
- Only one instruction may execute at any moment in time.

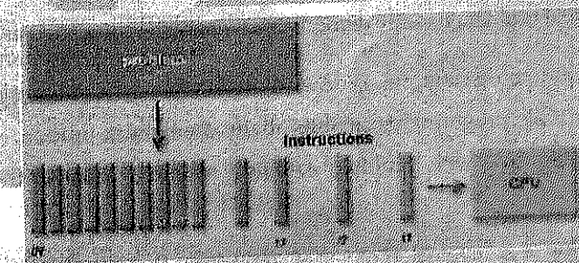


Figure 32.1

In the simplest sense, ***parallel computing*** is the simultaneous use of multiple compute resources to solve a computational problem.

- To be run using multiple CPUs
- A problem is broken into discrete parts that can be solved concurrently
- Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different CPUs

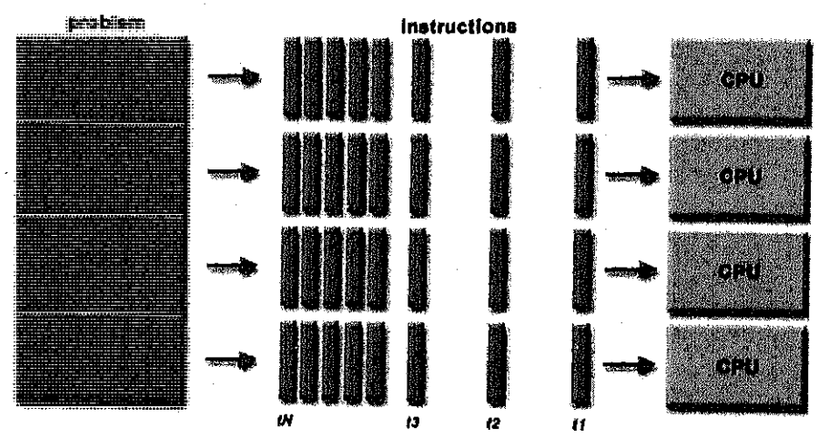


Figure 32.2

The compute resources can include :

- A single computer with multiple processors ;
- An arbitrary number of computers connected by a network ;
- A combination of both.

The computational problem usually demonstrates characteristics such as the ability to be :

- Broken apart into discrete pieces of work that can be solved simultaneously ;
- Execute multiple program instructions at any moment in time ;
- Solved in less time with multiple compute resources than with a single compute resource.

Parallel computing is an evolution of serial computing that attempts to emulate what has always been the state of affairs in the natural world : many complex, interrelated events happening at the same time, yet within a sequence.

Some examples :

- Planetary and galactic orbits
- Weather and ocean patterns
- Automobile assembly line
- Daily operations within a business
- Building a shopping mall

32.2 Why Use Parallel Computing ?

The primary reasons for using parallel computing :

- Save time
- Solve larger problems
- Provide concurrency (do multiple things at the same time)

Other reasons might include :

- *Taking advantage of non-local resources.* using available compute resources on a wide area network, or even the Internet when local compute resources are scarce.
- *Cost savings.* using multiple "cheap" computing resources instead of paying for time on a supercomputer.
- *Overcoming memory constraints.* single computers have very finite memory resources. For large problems, using the memories of multiple computers may overcome this obstacle.

Limits to serial computing both physical and practical reasons pose significant constraints to simply building ever faster serial computers :

- *Transmission speeds.* the speed of a serial computer is directly dependent upon how fast data can move through hardware. Absolute limits are the speed of light (30 cm/nanosecond) and the transmission limit of copper wire (9 cm/nanosecond). Increasing speeds necessitate increasing proximity of processing elements.
- *Limits to miniaturization.* processor technology is allowing an increasing number of transistors to be placed on a chip. However, even with molecular or atomic-level components, a limit will be reached on how small components can be.
- *Economic limitations.* it is increasingly expensive to make a single processor faster. Using a larger number of moderately fast commodity processors to achieve the same (or better) performance is less expensive.

32.3 von Neumann Architecture

For over 40 years, virtually all computers have followed a common machine model known as the von Neumann computer. Named after the Hungarian mathematician **John von Neumann.** A von Neumann computer uses the stored-program concept. The CPU executes a stored program that specifies a sequence of read and write operations on the memory.

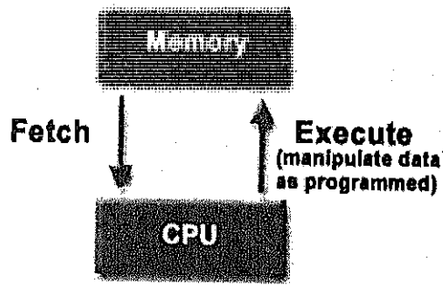


Figure 32.3

Basic design :

- Memory is used to store both program and data instructions
- Program instructions are coded data which tell the computer to do something
- Data is simply information to be used by the program
- A central processing unit (CPU) gets instructions and/or data from memory, decodes the instructions and then *sequentially* performs them.

32.4 Flynn's Classical Taxonomy

- There are different ways to classify parallel computers. One of the more widely used classifications, in use since 1966, is called Flynn's Taxonomy.
- Flynn's taxonomy distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of **Instruction** and **Data**. Each of these dimensions can have only one of two possible states : **Single** or **Multiple.**
- The matrix below defines the 4 possible classifications according to Flynn.

S I S D Single Instruction, Single Data	**S I M D** Single Instruction, Multiple Data
M I S D Multiple Instruction, Single Data	**M I M D** Multiple Instruction, Multiple Data

Single Instruction, Single Data (SISD)

- A serial (non-parallel) computer
- *Single instruction.* only one instruction stream is being acted on by the CPU during any one clock cycle
- *Single data.* only one data stream is being used as input during any one clock cycle
- Deterministic execution
- This is the oldest and until recently, the most prevalent form of computer Examples : mainframes most PCs, single CPU workstations

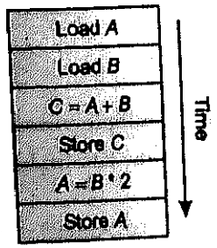


Figure 32.4

Single Instruction, Multiple Data (SIMD)

- A type of parallel computer
- *Single instruction.* All processing units execute the same instruction at any given clock cycle
- *Multiple data.* Each processing unit can operate on a different data element
- This type of machine typically has an instruction dispatcher, a very high-bandwidth internal network, and a very large array of very small-capacity instruction units.
- Best suited for specialized problems characterized by a high degree of regularity,such as **image processing.**
- Synchronous (lockstep) and deterministic execution
- *Two varieties.* Processor Arrays and Vector Pipelines
- *Examples.*
  Processor Arrays. Connection Machine CM-2, Maspar MP-1, MP-2
  Vector Pipelines. IBM 9000, Cray C90, Fujitsu VP, NEC SX-2, Hitachi S820

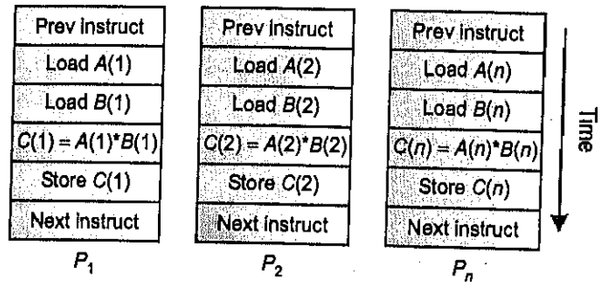


Figure 32.5

Multiple Instruction, Single Data (MISD)

- A single data stream is fed into multiple processing units.
- Each processing unit operates on the data independently via independent instruction streams.
- Few actual examples of this class of parallel computer have ever existed. One is the experimental Carnegie-Mellon C.mmp computer (1971).
- Some conceivable uses might be :

  multiple frequency filters operating on a single signal stream

  multiple cryptography algorithms attempting to crack a single coded message

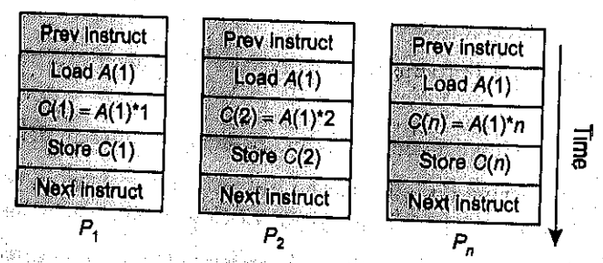


Figure 32.6

Multiple Instruction, Multiple Data (MIMD)

- Currently, the most common type of parallel computer. Most modern computers fall into this category.
- *Multiple Instruction*, every processor may be executing a different instruction stream.
- *Multiple Data*, every processor may be working with a different data stream.

- Execution can be synchronous or asynchronous, deterministic or non-deterministic.
- *Examples*. most current supercomputers, networked parallel computer "grids" and multi-processor SMP computers - including some types of PCs.

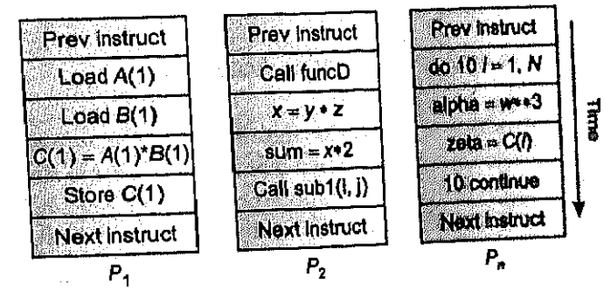


Figure 32.7

32.5 Parallel Algorithms

Parallel algorithms are the algorithms that perform more than one operation at a time. The study of parallel algorithms we must choose an appropriate model for parallel computing. The random-access machine, or RAM is serial rather than parallel. The parallel models are sorting **networks and circuits or PRAM** (pronounced "PEE-ram"). Many parallel algorithms for arrays, lists, trees, and graphs can be easily described in the PRAM model. The basic architecture of the parallel random-access machine (PRAM) as follows :

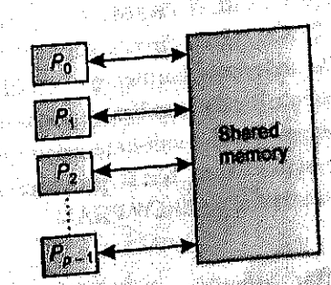


Figure 32.8

There are p ordinary (serial) processors $P_0, P_1, \ldots, P_{p-1}$ that have as storage a shared, global memory. All processors can read from or write to the global memory "in parallel" (at the same time). The processors can also perform various arithmetic and logical operations in parallel.

The key assumption in the PRAM model is that running time can be measured as the number of parallel memory accesses an algorithm performs. The running time of a parallel

algorithm depends on the number of processors executing the algorithm as well as the size of the problem input. Therefore, we take both time and processor count when analyzing PRAM algorithms; this contrasts with serial algorithms, in whose analysis we have focused mainly on time.

32.6 Concurrent Versus Exclusive Memory Accesses

A **concurrent-read** algorithm is a PRAM algorithm during whose execution multiple processors can read from the same location of shared memory at the same time. An **exclusive-read** algorithm is a PRAM algorithm in which no two processors ever read the same memory location at the same time. Similar we can divide PRAM algorithms into **concurrent-write** and **exclusive-write** algorithms.

Commonly used abbreviations for the types of algorithms are

EREW exclusive read and exclusive write,
CREW concurrent read and exclusive write,
ERCW exclusive read and concurrent write, and
CRCW concurrent read and concurrent write.

(These abbreviations are usually pronounced not as words but rather as strings of letters.)

A PRAM that supports only EREW algorithms is called an **EREW PRAM**, and one that supports CRCW algorithms is called a **CRCW PRAM**. A CRCW PRAM can, of course, execute EREW algorithms, but an EREW PRAM cannot directly support the concurrent memory accesses required in CRCW algorithms. The underlying hardware of an EREW PRAM is relatively simple, and therefore fast, because it needn't handle conflicting memory reads and writes. A CRCW PRAM requires more hardware support if the unit-time assumption is to provide a reasonably accurate measure of algorithmic performance, but it provides a programming model that is arguably more straightforward than that of an EREW PRAM.

32.7 List Ranking by Pointer Jumping

We can store a list in a PRAM much as we store lists in an ordinary RAM. To operate on list objects in parallel, however, it is convenient to assign a "responsible" processor to each object. We shall assume that there are as many processors as list objects, and that the ith processor is responsible for the ith object. For example, shows a linked list consisting of the sequence of objects $\langle 2,4,6,1,8,5\rangle$. Since there is one processor per list object, every object in the list can be operated on by its responsible processor in $\Theta(1)$ time.

Suppose that we are given a singly linked list L with n objects and wish to compute, for each object in L, its distance from the end of the list. More formally, if next is the pointer field, we wish to compute a value $d[i]$ for each object i in the list such that

$$d[i] = \begin{cases} 0 & \text{if next}[i] = \text{NIL} \\ d[\text{next}[i]] + 1 & \text{if next}[i] \neq \text{NIL} \end{cases}$$

We call the problem of computing the d values the **list-ranking problem.**

One solution to the list-ranking problem is simply to propagate distances back from the end of the list. This method takes $\Theta(n)$ time, since the kth object from the end must wait for the $k-1$ objects following it to determine their distances from the end before it can determine its own. This solution is essentially a serial algorithm.

An efficient parallel solution, requiring only $O(\lg n)$ time, is given by the following pseudo code.

```
LIST-RANK (L)
1.  for each processor i, in parallel
2.      do if next[i] = NIL
3.          then d [i] ← 0
4.          else d [i] ← 1
5.  while there exists an object i such that next[i] ≠ NIL
6.      do for each processor i, in parallel
7.          do if next[i] ≠ NIL
8.              then d [i] ← d [i] + d[next[i]]
9.                  next[i] ← next[next[i]]
```

The idea implemented by line 9, in which we set next[i] ← next[next[i]] for all non-nil pointers next[i], is called **pointer jumping.** Note that the pointer fields are changed by pointer jumping, thus destroying the structure of the list. If the list structure must be preserved, then we make copies of the next pointers and use the copies to compute the distances.

Parallel prefix on a list

A **prefix computation** is defined in terms of a binary, associative operator $\otimes$. The computation takes as input a sequence $\langle x_1, x_2, \ldots, x_n\rangle$ and produces as output a sequence $\langle y_1, y_2, \ldots, y_n\rangle$ such that $y_1 = x_1$ and

$$\begin{aligned} y_k &= y_{k-1} \otimes x_k \\ &= x_1 \otimes x_2 \otimes \ldots \otimes x_k \end{aligned}$$

For $k = 2,3,\ldots,n$. In other words, each y_k is obtained by "multiplying" together the first k elements of the sequence of x_k-hence, the term "prefix."

```
LIST-PREFIX(L)
for each processor i, in parallel
    do y[i] ← x[i]
while there exists an object i such that next[i] ≠ NIL
    do for each processor i, in parallel
        do if next[i] ≠ NIL
            then y[next[i]] ← y[i] ⊗ y[next[i]]
                 next[i] ← next[next[i]]
```

In List-Rank, processor i updates $d[i]$ - its own d value - whereas in LIST-PREFIX, processor i updates $y[\text{next}[i]]$ - another processor's y value.

32.8 The Euler-Tour Technique

It is used for finding the depth of a node in a binary tree. To store binary trees in a PRAM, we use a simple binary-tree representation. Each node i has fields parent[i], left[i], and right[i], which point to node i's parent, left child, and right child, respectively. Let us assume that each node is identified by a non-negative integer.

An **Euler tour** of a graph is a cycle that traverses each edge exactly once, although it may visit a vertex more than once. A connected, directed graph has an Euler tour if and only if for all vertices v, the in-degree of v equals the out-degree of v. Since each undirected edge (u,v) in an undirected graph maps to two directed edges (u,v) and (v,u) in the directed version, the directed version of any connected, undirected graph – and therefore of any undirected tree – has an Euler tour.

To compute the depths of nodes in a binary tree T, we first form an Euler tour of the directed version of T (viewed as an undirected graph). The tour corresponds to a walk of the tree and is represented by a linked list running through the nodes of the tree. Rules for connecting various processors of a node to a another node is as follows :

- "A" processor of a node will point to "A" processor of its left node, if it exists, and otherwise points to a "B" processor of the same node.
- "B" processor of a node will point to "A" processor of its right child, if it exists, and otherwise points to a "C" processor of the same node.
- "C" processor of a node will point to the "B" processor of its parent if it is a left child and to the "C" processor of its parent if it is a right child. The root's C processor points to NIL.

Thus, the head of the linked list formed by the Euler tour is the root's A processor, and the tail is the root's C processor. Given the pointers composing the original tree, an Euler tour can be constructed in $O(1)$ time and is represented in the Fig. 32.9.

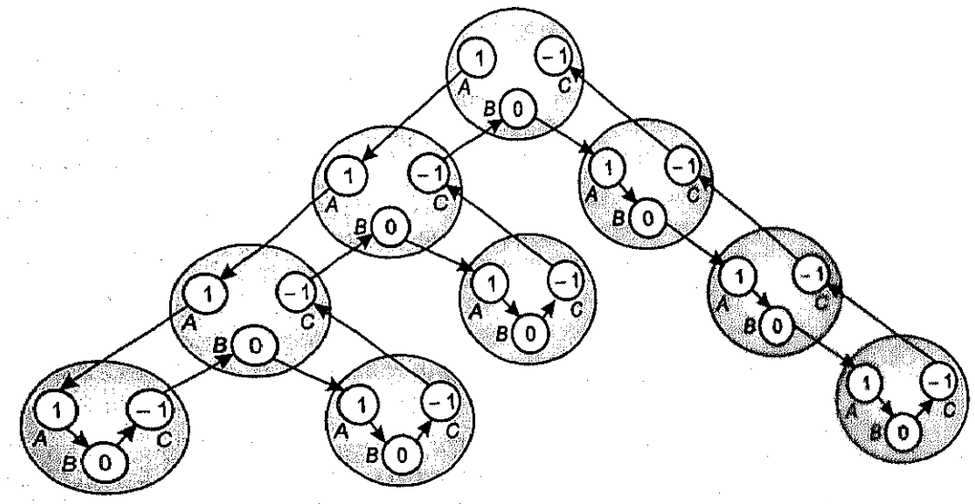


Figure 32.9 Using the Euler-Tour technique to compute the depth of each node in a binary tree.

Once we have the linked list representing the Euler tour of T, we place a 1 in each A processor, a 0 in each B processor, and –1 in each C processor. We then perform a parallel prefix computation using ordinary addition as the associative operation, which is represented in Fig. 32.10.

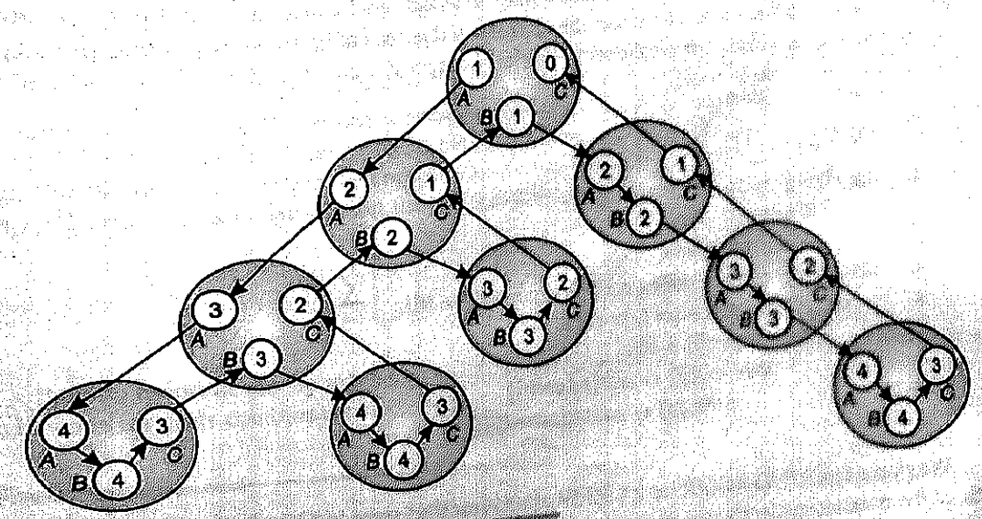


Figure 32.10 Result of the parallel prefix computation.

We claim that after performing the parallel prefix computation, the depth of each node resides in the node's C processor. The numbers are placed into the A, B, and C processors in such a way that the net effect of visiting a subtree is to add 0 to the running sum. The A processor of each node i contributes 1 to the running sum in i's left subtree, reflecting the depth of i's left child being one greater than the depth of i. The B processor contributes 0 because the depth of node i's left child equals the depth of node i's right child. The C processor contributes -1, so that from the perspective of node i's parent, the entire visit to the subtree rooted at node i has no effect on the running sum.

The list representing the Euler tour can be computed in $O(1)$ time. It has $3n$ objects, and thus the the parallel prefix computation takes only $O(\lg n)$ time. Thus, the total amount of time to compute all node depths is $O(\lg n)$. Because no concurrent memory accesses are needed, the algorithm is an EREW algorithm.

32.9 CRCW Algorithms Versus EREW Algorithms

The debate about whether or not concurrent memory accesses should be provided by the hardware of a parallel computer is a messy one. Some argue that hardware mechanisms to support CRCW algorithms are too expensive and used too infrequenly to be justified. Others complain that EREW PRAM's provide too restrictive a programming model. The answer to this debate probably lies somewhere in the middle, and various compromise models have been proposed.

We shall show that there are problems on which a CRCW algorithm outperforms the best possible EREW algorithm. For the problem of finding the identities of the roots of trees in a forest, concurrent reads allow for a faster algorithm. For the problem of finding the maximum element in an array, concurrent writes permit a faster algorithm.

Suppose we are given a forest of binary trees in which each node i has a pointer parent[i] to its parent, and we wish each node to find the identity of the root of its tree. Associating processor i with each node i in a forest F, the following pointer–jumping algorithm stores the identity of the root of each node i's tree in root[i].

```
FIND-ROOTS (F)

for each processor i, in parallel
    do if parent[i] = NIL
        then root[i] ← i
while there exists a node i such that parent[i] ≠ NIL
    do for each processor i, in parallel
        do if parent[i] ≠ NIL
            then root[i] ← root[parent[i]]
                 parent[i] ← parent[parent[i]]
```

We claim that FIND-ROOTS is a CREW algorithm that runs in $O(\lg d)$ time, where d is the depth of the maximum-depth tree in the forest.

We know that CRCW algorithms can solve some problems more quickly than can EREW algorithms. Moreover, any EREW algorithm can be executed on a CRCW PRAM. Thus, the CRCW model is strictly more powerful than the EREW model.

32.10 Brent's Theorem

Brent's theorem shows how we can efficiently simulate a combinational circuit by a PRAM. A ***combinational circuit*** is an acyclic network of ***combinational elements***. Each combinational element has one or more inputs, and assumes that each element has exactly one output. The number of inputs is the ***fan-in*** of the element, and the number of places to which its output feeds is its ***fan-out***. The ***size*** of a combinational circuit is the number of combinational elements that it contains. The number of combinational elements on a longest path from an input of the circuit to an output of a combinational element is the element's ***depth***. The ***depth*** of the entire circuit is the maximum depth of any of its elements.

Theorem

Any depth-d, size-n combinational circuit with bounded fan-in can be stimulated by a p-processor CREW algorithm in $O(n/p+d)$ time.

Proof. An element cannot be simulated until the outputs from any elements that feed it have been computed. Concurrent reads are employed whenever several combinational elements being simulated in parallel require the same value.

Since all elements at depth 1 depend only on circuit inputs, they are the only ones that can be simulated initially. Once they have been simulated, all elements at depth 2 can be simulated, and so forth, until we finish with all elements at depth d. The key idea is that if all elements from depths 1 to i have been simulated, we can simulate any subset of elements at depth $i+1$ in parallel, since their computations are independent of one another.

For $i=1,2,\ldots,d$, let n_i be the number of elements at depth i in the circuit. Thus,

Consider the n_i combinational elements at depth i. By grouping them into $\lceil n_i/p \rceil$ groups, where the first $\lfloor n_i/p \rfloor$ groups have p elements each and the leftover elements, if any, are in the last group, the PRAM can simulate the computations performed by these combinational elements in $O(\lceil n_i/p \rceil)$ time. The total simulation time is therefore on the order of

$$\sum_{i=1}^{d} \left\lceil \frac{n_i}{p} \right\rceil \le \sum_{i=1}^{d} \left(\frac{n_i}{p} + 1 \right) = \frac{n}{p} + d$$

Brent's theorem can be extended to EREW simulations when a combinational circuit has $O(1)$ fan-out for each combinational element.

Theorem

Any depth-d, size-n combinational circuit with bounded fan-in and fan-out can be simulated on a p-processor EREW PRAM in $O(n/p+d)$ time.

Proof. We perform a simulation similar to that in the proof of Brent's theorem. The only difference is in the simulation of wires, which is where requires concurrent reading. For the EREW simulation, after the output of a combinational element is computed, it is not directly read by processors requiring its value. Instead, the output value is copied by the processor simulating the element to the $O(1)$ inputs that require it. The processors that need the value can then read it without interfering with each other.

Exercise

1. Discuss Brent's Theorem in detail.
2. Give an EREW algorithm to perform prefix computation on an array $A[1..n]$. Do not use pointers but use index property.
3. Merging of two sorted sequences each of length m can be computed in $O(\lg m)$ time using m CREW PRAM processors.
4. Given an $O(\lg n)$ EREW alogorithm that determines for each object in an n-object list whether it is the middle ($\lfloor n/2 \rfloor$) object.

CHAPTER

33

Matrix Operations

33.1 Introduction

A **matrix** is a rectangular array of numbers.

For example, $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$

is a 3×3 matrix $A = (a_{ij})$ where for $i = 1,2,3$ and $j = 1,2,3$ the element of the matrix in row i and column j is a_{ij}. We use uppercase letters to denote matrices and corresponding subscripted lowercase letters to denote their elements. The set of all $m \times n$ matrices with real-valued entries is denoted $R^{m \times n}$. In general, the set of $m \times n$ matrices with entries drawn from a set S is denoted $S^{m \times n}$.

The **transpose** of a matrix A is the matrix A^T obtained by exchanging the rows and columns of A.

$$(A^T)_{ij} = A_{ji}$$

$$\begin{bmatrix} 0 & 4 \\ 7 & 0 \\ 3 & 1 \end{bmatrix}^T = \begin{bmatrix} 0 & 7 & 3 \\ 4 & 0 & 1 \end{bmatrix}$$

A **vector** is a one-dimensional array of numbers. The standard form of a vector to be as a **column vector** equivalent to an $n \times 1$ matrix; the corresponding **row vector** is obtained by taking the transpose

The **unit vector** e_i is the vector whose ith element is 1 and all of whose other elements are 0. A **zero matrix** is a matrix that's every entry is 0. A **diagonal matrix** has $a_{ij} = 0$ whenever $i \neq j$. Because all of the off-diagonal elements are zero, listing the elements along the diagonal can specify the matrix. The $n \times n$ **identity matrix** I_n is a diagonal matrix with 1's along the diagonal :

$$I_n = \text{diag}(1,1,\ldots,1)$$

A **tridiagonal matrix** T is one for which $t_{ij} = 0$ if $|i-j| > 1$. Non zero entries appear only on the main diagonal, immediately above the main diagonal ($t_{i,i+1}$ for $i = 1,2,\ldots,n-1$), or immediately below the main diagonal ($t_{i+1,i}$ for $i = 1,2,\ldots,n-1$) :

$$T = \begin{bmatrix} t_{11} & t_{12} & 0 & 0 & \cdots & 0 & 0 & 0 \\ t_{21} & t_{22} & t_{23} & 0 & \cdots & 0 & 0 & 0 \\ 0 & t_{32} & t_{33} & t_{34} & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & t_{n-2,n-2} & t_{n-2,n-1} & 0 \\ 0 & 0 & 0 & 0 & \cdots & t_{n-1,n-2} & t_{n-1,n-1} & t_{n-1,n} \\ 0 & 0 & 0 & 0 & \cdots & 0 & t_{n,n-1} & t_{n,n} \end{bmatrix}$$

An **upper-triangular matrix** U is one for which $u_{ij} = 0$ if $i > j$. All entries below the diagonal are zero :

$$U = \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{bmatrix}$$

An upper-triangular matrix is **unit upper-triangular** if it has all 1's along the diagonal. A **lower-triangular matrix** L is one for which $l_{ij} = 0$ if $i < j$. All entries above the diagonal are zero :

$$L = \begin{bmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{bmatrix}$$

A lower-triangular matrix is **unit lower-triangular** if it has all 1's along the diagonal. A **permutation matrix** P has exactly one 1 in each row or column, and 0's elsewhere. An example of a permutation matrix is

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Such a matrix is called a permutation matrix because multiplying a vector x by a permutation matrix has the effect of permuting (rearranging) the elements of x.

A **symmetric matrix** A satisfies the condition $A = A^T$

33.2 Operations on Matrices

1. Matrix addition & subtraction

We define **matrix addition** as follows. If $A = (a_{ij})$ and $B = (b_{ij})$ are $m \times n$ matrices, then their matrix sum $C = (c_{ij}) = A + B$ is the $m \times n$ matrix defined by

$c_{ij} = a_{ij} + b_{ij}$ for $i = 1,2,\ldots,m$ and $j = 1,2,\ldots,n$. That is, matrix addition is performed component wise.

$$\begin{bmatrix} 0 & 4 \\ 7 & 0 \\ 3 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 2 \\ 2 & 3 \\ 0 & 4 \end{bmatrix} = \begin{bmatrix} 1 & 6 \\ 9 & 3 \\ 3 & 5 \end{bmatrix}$$

A zero matrix is the identity for matrix addition :

$$A + 0 = A = 0 + A$$

Properties of matrix addition

1. **commutative.** $A + B = B + A$
2. **associative.** $(A + B) + C = A + (B + C)$, so we can write as $A + B + C$
3. $A + 0 = 0 + A = A$; $A - A = 0$
4. $(A + B)^T = A^T + B^T$

We can multiply a number (scalar) by a matrix by multiplying every entry of the matrix by the scalar.

$$(-2)\begin{bmatrix} 1 & 6 \\ 9 & 3 \\ 6 & 0 \end{bmatrix} = \begin{bmatrix} -2 & -12 \\ -18 & -6 \\ -12 & 0 \end{bmatrix}$$

$$(\alpha + \beta) A = \alpha A + \beta A \,;\; (\alpha\beta) A = (\alpha)(\beta A)$$

$$\alpha (A + B) = \alpha A + \alpha B$$

$$0 \cdot A = 0 \,;\; 1 \cdot A = 0$$

As a special case, we define the negative of a matrix $A=(a_{ij})$ to be $-1A=-A$, so that the ijth entry of $-A$ is $-a_{ij}$. Thus,

$$A+(-A)=0$$
$$=(-A)+A$$

Given this definition, we can define **matrix subtraction** as the addition of the negative of a matrix :

$$A-B=A+(-B)$$

2. Multiplication of Matrices

Consider two matrices **A** and **B** with the following characteristics : the number of columns in **A** equals the number of rows in **B**. These are conformable with respect to one another, and they can be multiplied together to form a new matrix **Z**.

The expression

$$z_{ij}=a_{i1}*b_{1j}+a_{i2}*b_{2j}+a_{i3}*b_{3j}+\ldots a_{im}*b_{nj}$$

means "add the products obtained by multiplying elements in each i row of matrix **A** by elements in each j column of matrix **B**". Fig illustrates what we mean by this statement.

$$A=\begin{bmatrix}4&1&9\\6&2&8\\7&3&5\\11&10&12\end{bmatrix}\quad B=\begin{bmatrix}2&9\\5&12\\8&10\end{bmatrix}\quad Z=A*B=\begin{bmatrix}85&138\\86&158\\69&149\\168&339\end{bmatrix}$$

$$z_{ij}=a_{i1}*b_{1j}+a_{i2}*b_{2j}+a_{i3}*b_{3j}+\ldots a_{im}*b_{nj}$$

$z_{11}=4*2+1*5+9*8=85$ $\quad$ $z_{12}=4*9+1*12+9*10=138$

$z_{21}=6*2+2*5+8*8=86$ $\quad$ $z_{22}=6*9+2*12+8*10=158$

$z_{31}=7*2+3*5+5*8=69$ $\quad$ $z_{32}=7*9+3*12+5*10=149$

$z_{41}=11*2+10*5+12*8=168$ $\quad$ $z_{42}=11*9+10*12+12*10=339$

The order in which we multiply terms does matter. The reason for this is that we need to multiply row elements by column elements and one by one. Therefore $A*B$ and $B*A$ can produce different results. We say "can produce" because there exist special cases in which the operation is commutative (order does not matter). An example of this is when we deal with diagonal matrices.

3. Multiplication and Division of Matrices by a Scalar

The rules for multiplication and division of a matrix by a scalar are similar. Since multiplying a number x by $1/c$ is the same as dividing x by c, lets consider these operations at once.

If all elements of matrix A are multiplied by a scalar c to construct matrix Z, hence $z_{ij}=c*a_{ij}$. Similarly dividing matrix A by c gives $z_{ij}=(1/c)*a_{ij}$. The expression

$$z_{ij}=c*a_{ij}$$

means "multiply each element in row i column j times c", and the expression

$$z_{ij}=1/c*a_{ij}=a_{ij}/c$$

means "divide each element in row i column j by c". These two operations are shown in below, where $c=2$.

$$A=\begin{bmatrix}3&6\\5&8\\-2&9\end{bmatrix}\quad Z=cA=\begin{bmatrix}6&12\\10&16\\-4&18\end{bmatrix}\quad Z=\frac{A}{c}=\begin{bmatrix}1.5&3\\2.5&4\\-1&4.5\end{bmatrix}$$

$z_{ij}=c*a_{ij}$ $\qquad$ $z_{ij}=\dfrac{a_{ij}}{c}$

If $c=2$ $\qquad$ if $c=2$

$z_{11}=2*3=6$ $\quad$ $z_{12}=2*6=12$ $\quad$ $z_{11}=3/2=1.5$ $\quad$ $z_{12}=6/2=3$

$z_{21}=2*5=10$ $\quad$ $z_{22}=2*8=16$ $\quad$ $z_{21}=5/2=2.5$ $\quad$ $z_{22}=8/2=4$

$z_{31}=2*(-2)=-4$ $\quad$ $z_{32}=2*9=18$ $\quad$ $z_{31}=-2/2=-1$ $\quad$ $z_{32}=9/2=4.5$

Example shows that a scalar matrix is obtained when an identity matrix is multiplied by a scalar.

$$I=\begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix}\quad Z=c*1=\begin{bmatrix}c&0&0\\0&c&0\\0&0&c\end{bmatrix}$$

Identity matrix $\qquad$ Scalar matrix

Multiplying by a zero matrix gives a zero matrix :

$$A0=0$$

Matrix multiplication is associative:

$$A(BC)=(AB)C$$

For compatible matrices A, B, and C. Matrix multiplication distributes over addition:

$$A(B+C)=AB+AC$$
$$(B+C)D=BD+CD$$

4. Matrix-Vector Product

Very important special case of matrix multiplication : $y = Ax$

- A is an $m \times n$ matrix
- x is an n-vector
- y is an m-vector

$$y_i = A_{i1}x_1 + A_{i2}x_2 + \ldots + A_{in}x_n \qquad \text{for } i = 1, \ldots m$$

5. Inner Product

If v is a row n-vector and w is a column n-vector, then vw makes sense, and has size 1×1, *i.e.*, is a scalar :

$$vw = v_1 w_1 + \ldots + v_n w_n$$

if x and y are n-vectors, $x^T y$ is a scalar called inner product or dot product of x, y, and denoted $\langle x, y \rangle$ or $x \cdot y$:

$$\langle x, y \rangle = x^T y = x_1 y_1 + \ldots + x_n y_n$$

6. Matrix inverses, ranks, and determinants

We define the inverse of an $n \times n$ matrix A to be the $n \times n$ matrix, denoted A^{-1} (if it exists), such that $AA^{-1} = I_n = A^{-1}A$. *For example,*

$$\begin{bmatrix} 1 & -1 \\ 1 & 2 \end{bmatrix}^{-1} = \frac{1}{3}\begin{bmatrix} 2 & 1 \\ -1 & 1 \end{bmatrix}$$

Many nonzero $n \times n$ matrices do not have inverses. A matrix without an inverse is is called **noninvertible**, or **singular**. An example of a nonzero singular matrix is

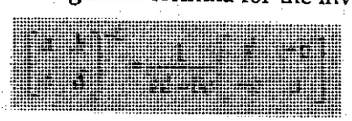
$$\begin{bmatrix} 1 & -1 \\ -2 & 2 \end{bmatrix}$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}\begin{bmatrix} 1 & -1 \\ -2 & 2 \end{bmatrix} = \begin{bmatrix} a-2b & -a+2b \\ c-2d & -c+2d \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

...but we can't have $a - 2b = 1$ and $-a + 2b = 0$.

Inverse of 2 x 2 matrix

It's useful to know the general formula for the inverse of a 2 x 2 matrix:



provided $ad - bc \neq 0$ (if $ad - bc = 0$, the matrix is singular)

If a matrix has an inverse, it is called **invertible**, or **nonsingular**. If A and B are nonsingular $n \times n$ matrices, then

$$(BA)^{-1} = A^{-1}B^{-1}$$

The inverse operation commutes with the transpose operation:

$$(A^{-1})^T = (A^T)^{-1}$$

The vectors $x_1, x_2, \ldots, x_n$ are **linearly dependent** if there exist coefficients $c_1, c_2, \ldots, c_n$ not all of which are zero, such that $c_1x_1 + c_2x_2 + \ldots + c_nx_n = 0$. For example, the vectors $x_1 = (1\ 2\ 3)^T$, $x_2 = (2\ 6\ 4)^T$, and $x_3 = (4\ 11\ 9)^T$ are linearly dependent, since $2x_1 + 3x_2 - 2x_3 = 0$. If vectors are not linearly dependent, they are **linearly independent**. For example, the columns of an identity matrix are linearly independent.

The **column rank** of a nonzero $m \times n$ matrix A is the size of the largest set of linearly independent columns of A. Similarly, the **row rank** of A is the size of the largest set of linearly independent rows of A. A fundamental property of any matrix A is that its row rank always equals its column rank, so that we can simply refer to the **rank of** A. The rank of an $m \times n$ matrix is an integer between 0 and $\min(m, n)$, inclusive. (The rank of a zero matrix is 0, and the rank of an $n \times n$ identity matrix is n.) An alternate, but equivalent and often more useful, definition is that the rank of a nonzero $m \times n$ matrix A is the smallest number r such that there exist matrices B and C of respective sizes $m \times r$ and $r \times n$ such that

$$A = BC.$$

A square $n \times n$ matrix has **full rank** if its rank is n. A null vector for a matrix A is a nonzero vector x such that $Ax = 0$.

For a 2×2 matrix, the **determinant** is $ad - bc$.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

For a 3×3 matrix, the determinant is $aei + bfg + dhc - ceg - ibd - ahf$. We should be able to see a pattern.

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

Really though, for anything bigger than a 2 x 2 matrix we should use the det

Some important properties of determinants

The determinant of a square matrix A has the following properties :

1. If any row or any column of A is zero, then $\det(A) = 0$.
2. The determinant of A is multiplied by λ if the entries of any one row (or any one column) of A are all multiplied by λ.
3. The determinant of A is unchanged if the entries in one row (respectively, column) are added to those in another row (respectively, column).
4. The determinant of A equals the determinant of A^T.
5. The determinant of A is multiplied by -1 if any two rows (respectively, columns) are exchanged.
6. Also, for any square matrices A and B, we have $\det(AB) = \det(A)\det(B)$.
7. An $n \times n$ matrix A is singular if and only if $\det(A) = 0$.

33.3 Strassen's Algorithm for Matrix Multiplication

It is an application of a familiar design technique 'divide and conquer'. Suppose we wish to compute the product $C = AB$ where each A, B and C are $n \times n$ matrices. Assuming that n is an exact power of 2.

We divide each of A, B and C into four $n/2 \times n/2$ matrices.

Rewriting the eqution $C = AB$ as

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & g \\ f & h \end{pmatrix} \quad ...(1)$$

For convenience, the submatrices of A are labelled alphabetical from left to right, whereas those of B are labelled from top to bottom. So that matrix multiplication is performed.

Equation (1) corresponds to the 4 equations

$$r = ae + bf \quad ...(2)$$

$$s = ag + bh \quad ...(3)$$

$$t = ce + df \quad ...(4)$$

$$u = cg + dh \quad ...(5)$$

Each of these 4 equations specifies two multiplications of $n/2 \times n/2$ matrices and the addition of their $n/2 \times n/2$ products. Using these equations to define a stratight-forward divide and conquer strategy. We derive the following recurrence for the time $T(n)$ to multiply two $n \times n$ matrices :

$$T(n) = 8T(n/2) + \theta(n^2)$$

Unfortunately this recurrence has the solution $T(n) = \theta(n^3)$ and thus, this method is no faster than the ordinary one.

Volker Strassen has discovered a different recursive approach that required only 7 recursive multiplication of $n/2 \times n/2$ matrices and $\theta(n^2)$ scalar additions and subtractions yielding the recurrence.

$$T(n) = 7T(n/2) + \theta(n^2)$$

$$= \theta(n^{\log 7}) \approx O(n^{2.81})$$

Strassen's Method has 4 steps :

1. Divide the input matrices A and B into $n/2 \times n/2$ submatrices.
2. Using $\theta(n^2)$ scalar additions and subtractions compute 14 $n/2 \times n/2$ matrices $A_1, B_1, A_2, B_2, \ldots A_7, B_7$.

3. Recursively compute the seven matrix products
   $P_i = A_i B_i$ for $i = 1, 2, \ldots 7$
4. Compute the desired submatrices r, s, t, u of the result matrix C by adding and/or subtracting various combinations of the P_i matrices using only $\theta(n^2)$ scalar additions and subtractions.

Suppose, $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$ $B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$ and $C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

In Strassen method first compute the 7 $n/2 \times n/2$ matrices.

P, Q, R, S, T, U, V as follows :

$$P = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$Q = (A_{21} + A_{22}) B_{11}$$

$$R = A_{11} (B_{12} - B_{22})$$

$$S = A_{22} (B_{21} - B_{11})$$

$$T = (A_{11} + A_{12}) B_{22}$$

$$U = (A_{21} - A_{11})(B_{11} + B_{12})$$

$$V = (A_{12} - A_{22})(B_{21} + B_{22})$$

Then the c_{ij}'s are computed using the formulas

$$c_{11} = P + S - T + V$$

$$c_{12} = R + T$$

$$c_{21} = Q + S$$

$$c_{22} = P + R - Q + U$$

As can be seen P, Q, R, S, T, U and V can be computed using 7 matrix multiplications and 10 matrix additions or substrations and c_{ij}'s require an additional 8 addition or substrations.

The resulting recurrence relation for $T(n)$ is

$$T(n) = \begin{cases} b & n \le 2 \\ 7T(n/2) + an^2 & n > 2 \end{cases}$$

where a and b are constants.

Working with this formula we get

$$T(n) = an^2\left[1+\frac{7}{4}+\left(\frac{7}{4}\right)^2+\ldots+\left(\frac{7}{4}\right)^{k-1}\right]+7^k\,T(1)$$

$$\le cn^2\left(\frac{7}{4}\right)^{\lg n}+7^{\log_2 n}$$

$$= cn^{\log_2 4+\log_2 7+(-\log_2 4)}+n^{\log_2 7}$$

$$= O(n^{\log_2 7}) \approx O(n^{2.81})$$

33.4 Solving Systems of Linear Equations

A linear system can be expressed as a matrix equation in which each matrix or vector element belong to a field. We know a ring is a field if it satisfied two additional properties :

1. The operator is commutative.
2. Every non-zero element in S has a multiplicative inverse.

Let us consider a set of linear equations in n unkwons $x_1, x_2, \ldots, x_n$ are

$$a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1$$
$$a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2$$
$$\text{----------}$$
$$a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n$$

Rewrite these equations as the matrix-vector equation

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Letting $A=(a_{ij})$, $x=(x_j)$ and $b=(b_i)$ as $Ax=b$.

In this case, we only treat exactly n equations in n unknown. So, the rank of A is equal to the number n of unknowns. If the number of equations is less than the number n of unknowns or more generally, if the rank of A is less than n then the system is **under determined**. An under determined system typically has infinitely many solutions although it may have no solutions at all if the equations are inconsistent. If the number of equations exceeds the number n of unknowns, the system is **over determined** and there may not exist any solutions.

To solving the system $Ax=b$ of n equations in n unknowns. One approach is to compute A^{-1} and then multiply both sides yielding $A^{-1}Ax = A^{-1}b$ or $x = A^{-1}b$.

This approach has round-off errors tend to accumulate unduly when floating-point number representations are used instead of ideal real numbers.

There are another approach – *LUP* **decomposition** that is numerically stable and has the further advantage of being about a factor of 3 faster.

The idea behind *LUP* decomposition is to find three $n \times n$ matrices L, U and P such that

$$PA = LU$$

where L is a unit lower-triangular matrix.
U is an upper triangular matrix and
P is a permutation matrix.

Permutation matrix P has exactly one 1 in each row or column and 0's elsewhere.

If A decomposed in such a way *i.e.*, $PA = LU$, then it is called an *LUP* decomposition of the matrix A. Having found an *LUP* decomposition for A, we can solve the equation $Ax = b$ by solving only triangular linear systems.

Multiplying both sides of $Ax = b$ by P, yielding then

$$PAx = Pb$$

i.e.,

$$LUx = Pb$$

Suppose $y = Ux$, where x is the desired solution. First, we solve the lower-triangular system. $Ly = Pb$ for a unknown vector y by a method called **"Forward Substitution"** then solve the upper-triangular system. $Ux = y$ by the **'backward substitution'**. The vector x is our solution to $Ax = b$.

33.5 Computing an *LU* decomposition

When matrix A is an $n \times n$ non-singular matrix and P is absent. In this case, we must find a factorization

$$A = LU.$$

We call the two matrices L and U an *LU* decomposition of A and the process by which we perform *LU* decomposition is called **'Gaussian elimination'**. If $n = 1$ then we can choose $L = I_1$ and $U = A$.

For $n > 1$, we break A into *four* parts :

$$A = \left(\begin{array}{c|ccc} a_{11} & a_{12} & \cdots & a_{1n} \\ \hline a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{array}\right) = \begin{pmatrix} a_{11} & w^T \\ v & A' \end{pmatrix}$$

where v is a size $(n-1)$ column vector.
w^T is a size $(n-1)$ row vector.
A' is an $(n-1)\times(n-1)$ matrix.

$$A = \begin{pmatrix} a_{11} & w^T \\ v & A' \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 0 \\ V/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^T \\ 0 & A' - vw^T / a_{11} \end{pmatrix}$$

The $(n-1)\times(n-1)$ matrix $A' - vw^T / a_{11}$ is called the '**Schur Complement**' of A with respect to a_{11}.

We now recursively find the LU decomposition of the schur complement.

Now, to decompose into LU.

Step 1 Divide all the elements of first column by a_{11} except a_{11}.

Step 2 Now at place of $a_{22} \leftarrow a_{22} - a_{21} \times a_{12}$ and $a_{23} \leftarrow a_{23} - a_{21} \times a_{13}$ and $a_{24} \leftarrow a_{24} - a_{21} \times a_{14}$ etc.

Step 3 Similarly $a_{32} \leftarrow a_{32} - a_{31} \times a_{12}$

$a_{33} \leftarrow a_{33} - a_{31} \times a_{13}$ etc.

Step 4 Similarly $a_{42} \leftarrow a_{42} - a_{41} \times a_{12}$ etc.

For example, $\begin{bmatrix} 2 & 3 & 1 & 5 \\ 6 & 13 & 5 & 19 \\ 2 & 19 & 10 & 23 \\ 4 & 10 & 11 & 31 \end{bmatrix}$ here $a_{11} = 2$

$$\begin{bmatrix} 2 & 3 & 1 & 5 \\ 3 & 4 & 2 & 4 \\ 1 & 16 & 9 & 18 \\ 2 & 4 & 9 & 21 \end{bmatrix} \cdot \begin{bmatrix} 2 & 3 & 1 & 5 \\ 3 & 4 & 2 & 4 \\ 1 & 4 & 1 & 2 \\ 2 & 1 & 7 & 3 \end{bmatrix}$$

$$\underset{A}{\begin{bmatrix} 2 & 3 & 1 & 5 \\ 3 & 4 & 2 & 4 \\ 1 & 4 & 1 & 2 \\ 2 & 1 & 7 & 3 \end{bmatrix}} = \underset{L}{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 \\ 1 & 4 & 1 & 0 \\ 2 & 1 & 7 & 1 \end{bmatrix}} \times \underset{U}{\begin{bmatrix} 2 & 3 & 1 & 5 \\ 0 & 4 & 2 & 4 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 3 \end{bmatrix}}$$

i.e., $A = LU$

***LU*-Decomposition (*A*)**

```
n ← rows[A]
for k ← 1 to n
    do u_kk ← a_kk
        for i ← k+1 to n
            do l_ik ← a_ik / u_kk          ▷ l_ik holds v_i
            u_ki ← a_ki                    ▷ u_ki holds w_i^T
        for i ← k+1 to n
            do for j ← k+1 to n
                do a_ij ← a_ij − l_ik u_kj
return L and U
```


33.6 Computing an LUP decomposition

LUP decomposition is similar to that of LU decomposition. We are given an $n \times n$ non-singular matrix A and we wish to find a permutation matrix P, a unit lower-triangular matrix L, and an upper-triangular matrix U such that $PA = LU$. Before we partition the matrix A, as we did for LU decomposition, we move a non-zero element say a_{k1}, from the first column to the (1, 1) position of the matrix that is largest element in the first column.

```
LUP-Decomposition (A)
n ← rows[A]
for i ← 1 to n
    do π[i] ← i
for k ← 1 to n−1
    do p ← 0
        for i ← k to n
            do if |a_ik| > p
                then p ← |a_ik|
                     k' ← i
        if p = 0
            then error "singular matrix"
        exchange π[k] ↔ π[k']
        for i ← 1 to n
            do exchange a_ki ↔ a_k'i
        for i ← k+1 to n
            do a_ik ← a_ik / a_kk
            for j ← k+1 to n
                do a_ij ← a_ij − a_ik a_kj
```

Example : LUP decomposition is as follows

$$A = \begin{bmatrix} 2 & 0 & 2 & 0.6 \\ 3 & 3 & 4 & -2 \\ 5 & 5 & 4 & 2 \\ -1 & -2 & 3.4 & -1 \end{bmatrix}$$

The identity permutation of the rows are on the left of A

$$\begin{array}{c|cccc} 1 & 2 & 0 & 2 & 0.6 \\ 2 & 3 & 3 & 4 & -2 \\ 3 & 5 & 5 & 4 & 2 \\ 4 & -1 & -2 & 3.4 & -1 \end{array}$$

The maximum element in column first is 5 which is in the third row. So interchange third row and first row and update the permutation.

3	5	5	4	2
2	3	3	4	−2
1	2	0	2	0.6
4	−1	−2	3.4	−1

Divide by 5 in first column and update *i.e.*, $a_{22} \leftarrow a_{22} - a_{21} \times a_{12}$. $a_{32} \leftarrow a_{32} - a_{31} \times a_{12}$ etc.

3	5	5	4	2
2	0.6	0	1.6	−3.2
1	0.4	−2	0.4	−0.2
4	−0.2	−1	4.2	−0.6

Swap Row second and Row third.

3	5	5	4	2
1	0.4	−2	0.4	−0.2
2	0.6	0	1.6	−3.2
4	−0.2	−1	4.2	−0.6

3	5	5	4	2
1	0.4	−2	0.4	−0.2
2	0.6	0	1.6	−3.2
4	−0.2	0.5	4	−0.5

Swap Row forth and Row third.

3	5	5	4	2
1	0.4	−2	0.4	−0.2
4	−0.2	0.5	4	−0.5
2	0.6	0	1.6	−3.2

3	5	5	4	2
1	0.4	−2	0.4	−0.2
4	−0.2	0.5	4	−0.5
2	−0.6	0	0.4	−3

Thus, $P = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$ $L = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0.4 & 1 & 0 & 0 \\ -0.2 & 0.5 & 1 & 0 \\ 0.6 & 0 & 0.4 & 1 \end{pmatrix}$ $U = \begin{pmatrix} 5 & 5 & 4 & 2 \\ 0 & -2 & 0.4 & -0.2 \\ 0 & 0 & 4 & -0.5 \\ 0 & 0 & 0 & -3 \end{pmatrix}$

and $PA = LU$

where $A = \begin{bmatrix} 2 & 0 & 2 & 0.6 \\ 3 & 3 & 4 & -2 \\ 5 & 5 & 4 & 2 \\ -1 & -2 & 3.4 & -1 \end{bmatrix}$

Example. *Solve the equation*

$$\begin{pmatrix} 1 & 5 & 4 \\ 2 & 0 & 3 \\ 5 & 8 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 12 \\ 9 \\ 5 \end{pmatrix}$$ *by using an LUP decomposition.*

Solution. Here $A = \begin{pmatrix} 1 & 5 & 4 \\ 2 & 0 & 3 \\ 5 & 8 & 2 \end{pmatrix}$ $x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$ $b = \begin{pmatrix} 12 \\ 9 \\ 5 \end{pmatrix}$ *i.e.*, $Ax = b$.

The LUP decomposition of A is as follows :

1	1	5	4
2	2	0	3
3	5	8	2

3	5	8	2
2	2	0	3
1	1	5	4

3	5	8	2
2	0.4	−3.2	2.2
1	0.2	3.4	3.6

3	5	8	2
2	0.4	−3.2	2.2
1	0.2	3.4	3.6

3	5	8	2
1	0.2	3.4	3.6
2	0.2	−3.2	2.2

3	5	8	2
1	0.2	3.4	3.6
2	0.4	−0.9	5.44

Thus $P = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$ $L = \begin{pmatrix} 1 & 0 & 0 \\ 0.2 & 1 & 0 \\ 0.4 & -0.9 & 1 \end{pmatrix}$ $U = \begin{pmatrix} 5 & 8 & 2 \\ 0 & 3.4 & 3.6 \\ 0 & 0 & 5.44 \end{pmatrix}$

$PA = LU$

Using forward substitution, we solve $Ly = Pb$ for y.

$$\begin{pmatrix} 1 & 0 & 0 \\ 0.2 & 1 & 0 \\ 0.4 & -0.9 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 12 \\ 9 \\ 5 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 0 \\ 0.2 & 1 & 0 \\ 0.4 & -0.9 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 5 \\ 12 \\ 9 \end{pmatrix}$$

Obtaining

$$y_1 = 5$$
$$0.2y_1 + y_2 = 12$$
$$0.4y_1 - 0.9y_2 + y_3 = 9$$

Solving these, we get $y = \begin{pmatrix} 5 \\ 11 \\ 16.9 \end{pmatrix}$ by computing first y_1 then y_2 and finally y_3.

Using back substitution solve $Ux = y$ for x

$$\begin{pmatrix} 5 & 8 & 2 \\ 0 & 3.4 & 3.6 \\ 0 & 0 & 5.44 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 5 \\ 11 \\ 16.9 \end{pmatrix}$$

$$5x_1 + 8x_2 + 2x_3 = 5$$
$$3.4x_2 + 3.6x_3 = 11$$
$$5.44x_3 = 16.9$$

Computing first x_3, then x_2 and finally x_1.

$$x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} -0.17 \\ -0.04 \\ 3.10 \end{pmatrix}$$

Exercise

1. How would you modify strassen's algorithm to multiply $n \times n$ matrices in which n is not an exact power of 2 ? Show that the resulting algorithm runs in time $\theta(n^{\log 7})$.
2. Use strassen's algorithm to compute the matrix product $\begin{pmatrix} 1 & 5 \\ 3 & 8 \end{pmatrix}\begin{pmatrix} 7 & 4 \\ 6 & 2 \end{pmatrix}$. Show your work.
3. Show how to multiply the complex numbers $a + bi$ and $c + di$ using only three real multiplications. The algorithm should take a, b, c and d as input and produce the real component $ac - bd$ and the imaginary component $ad + bc$ seperately.
4. Solve the equation $\begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ -7 & 6 & 1 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ 14 \\ -7 \end{pmatrix}$ by using forward substitution.
5. Solve the equation $\begin{pmatrix} 1 & 5 & 4 \\ 2 & 0 & 3 \\ 5 & 8 & 2 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 12 \\ 9 \\ 5 \end{pmatrix}$ by using an LUP decomposition.
6. Describe the LUP decomposition of a diagonal matrix.
7. Discuss the Strassen's algorithm for matrix multiplication. Show that two $n * n$ matrices with elements from an arbitrary ring can be multiplied in $c(n^{\log 7})$ arithmetic operations.

CHAPTER

34

Number-Theoretic Algorithms

34.1 Some Facts From Elementary Number Theory

We know, set $Z = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ of integers and the set $N = \{0, 1, 2, \ldots\}$ of natural numbers. Given positive numbers a and d, the notation $d \mid a$ (read "d divides a") means that $a = kd$ for some integer k. Every integer divides 0. If $a > 0$ and $d \mid a$, then $|d| \le |a|$. If $d \mid a$, then we also say that a is a multiple of d.

If $d \mid a$ and $d \ge 0$, we say that d is a divisor of a. Note that $d \mid a$ if and only if $-d \mid a$, so that no generality is lost by defining the divisors to be nonnegative, with the understanding that the negative of any divisor of a also divides a. A divisor of an integer a is at least 1 but not greater than $|a|$. For example, the divisors of 24 are 1, 2, 3, 4, 6, 8, 12, and 24.

Let a, b and c are arbitrary integers then

- If $a \mid b$ and $b \mid c$, then $a \mid c$.
- If $a \mid b$ and $b \mid c$, then $a \mid (ib + jc)$, for all integers i and j.
- If $a \mid b$ and $b \mid a$, then $a = b$ or $a = -b$.

Every integer a is divisible by the *trivial divisors* 1 and a. Nontrivial divisors of a are also called factors of a. For example, the factors of 20 are 2, 4, 5, and 10.

Prime and Composite Numbers

An integer $a > 1$ whose only divisors are the trivial divisors 1 and a is said to be a **prime number** (or, more simply, a prime). An integer $a > 1$ that is not prime is said to be a **composite number** (or, more simply, a composite). So, for example, 5, 11, 101 are prime and 25, 39 is composite. The integer 1 is said to be a **unit** and is neither prime nor composite. Similarly, the integer 0 and all negative integers are neither prime nor composite.

The Division Theorem, Remainders, and Modular Equivalence

Given an integer n, the integers can be partitioned into those that are multiples of n and those that are not multiples of n.

Division Theorem

For any integer a and any positive integer n, there are unique integers q and r such that $0 \le r < n$ and $a = qn + r$.

The value $q = \lfloor a/n \rfloor$ is the quotient of the division. The value $r = a \bmod n$ is the remainder (or residue) of the division. We have that $n \mid a$ if and only if $a \bmod n = 0$. It follows that

$$a = \lfloor a/n \rfloor n + (a \bmod n) \text{ or } a \bmod n = a - \lfloor a/n \rfloor n.$$

The integers can be divided into n equivalence classes according to their remainders modulo n. The **equivalence class modulo** n containing an integer a is

$$[a]_n = \{a + kn : k \in Z\}$$

For example, $[3]_7 = \{\ldots, -11, -4, 3, 10, 17, \ldots\}$; other denotations for this set are $[-4]_7$ and $[10]_7$. Writing $a \in [b]_n$ is the same as writing $a \equiv b \pmod n$. The set of all such equivalence classes is

$$Z_n = \{[a]_n : 0 \le a \le n-1\}$$

Or

$$Z_n = \{0, 1, \ldots, n-1\}$$

where 0 represents $[0]_n$, 1 represents $[1]_n$, and so on ; each class is represented by its least nonnegative element.

It is sometimes convenient to talk about **congruence modulo** n. If $a \bmod n = b \bmod n$, we say that a is congruent to b modulo n, which we call the modulus, and we write

$$a \equiv b \pmod n$$

Therefore, if $a \equiv b \pmod n$, then $a - b = kn$ for some integer k.

Common Divisors and Greatest Common Divisors

If d is a divisor of a and also a divisor of b, then d is a **common divisor** of a and b. For example, the divisors of 30 are 1, 2, 3, 5, 6, 10, 15, and 30, and so the common divisors of 24 and 30 are 1, 2, 3, and 6. Note that 1 is a common divisor of any two integers.

An important property of common divisors is that

$$d \mid a \text{ and } d \mid b \text{ implies } d \mid (a+b) \text{ and } d \mid (a-b).$$

More generally, we have that

$$d \mid a \text{ and } d \mid b \text{ implies } d \mid (ax+by) \text{ for any integers } x \text{ and } y$$

The greatest common divisor of two integers a and b, not both zero, is the largest of the common divisors of a and b ; it is denoted $\gcd(a, b)$. Alternatively, we could say that $\gcd(a, b)$ is the number c, such that if $d \mid a$ and $d \mid b$, then $d \mid c$. If $\gcd(a, b) = 1$, we say that a and b are **relatively prime**. For example, $\gcd(24, 30) = 6$, $\gcd(5, 7) = 1$, and $\gcd(0, 9) = 9$. If a and b are not both 0, then $\gcd(a, b)$ is an integer between 1 and $\min(|a|, |b|)$. We define $\gcd(0, 0)$ to be 0 ; The following are elementary properties of the gcd function :

- $\gcd(a, b) = \gcd(b, a)$,
- $\gcd(a, b) = \gcd(-a, b)$,
- $\gcd(a, b) = \gcd(|a|, |b|)$
- $\gcd(a, 0) = \gcd(0, a) = |a|$
- $\gcd(a, ka) = |a|$ for any $k \in Z$.

34.2 Euclid's GCD Algorithm

To compute the greatest common divisor of two numbers, we can use one of the oldest algorithms known, Euclid's algorithm. This algorithm is based on the following property of gcd (a, b)

Theorem

For any nonnegative integer a and any positive integer b, $\gcd(a, b) = \gcd(b, a \bmod b)$

Proof. We shall show that $\gcd(a, b)$ and $\gcd(b, a \bmod b)$ divide each other, so that they must be equal (since they are both nonnegative).

We first show that $\gcd(a, b) \mid \gcd(b, a \bmod b)$. If we let $d = \gcd(a, b)$, then $d \mid a$ and $d \mid b$.

$(a \bmod b) = a - qb$, where $q = \lfloor a/b \rfloor$. Since $(a \bmod b)$ is thus a linear combination of a and b, implies that $d \mid (a \bmod b)$. Therefore, since $d \mid b$ and $d \mid (a \bmod b)$, implies that $d \mid \gcd(a, b)$ or, equivalently, that ...(1)

$$\gcd(a, b) \mid \gcd(b, a \bmod b)$$

Showing that $\gcd(b, a \bmod b) \mid \gcd(a, b)$ is almost the same. If we now let $d = \gcd(b, a \bmod b)$, then $d \mid b$ and $d \mid (a \bmod b)$. Since $a = qb + (a \bmod b)$, where $q = \lfloor a/b \rfloor$, we have that a is a linear combination of b and $(a \bmod b)$. Then, we conclude that $d \mid a$. Since $d \mid b$ and $d \mid a$, we have that $d \mid \gcd(a, b)$ or, equivalently, that ...(2)

$$\gcd(b, a \bmod b) \mid \gcd(a, b)$$

We know $a \mid b = b \mid a$ implies $a = \pm b$

Using this to combine equations (1) and (2) completes the proof.

This theorem leads us easily to an ancient algorithm, known as Euclid's algorithm

Euclid's algorithm

The following gcd algorithm is described in the *Elements* of Euclid (circa 300 B.C.), although it may be of even earlier origin. It is written as a recursive program based directly on above theorem. The inputs a and b are arbitrary nonnegative integers.

```
EUCLID (a, b)
1.  if b = 0
2.      then return a
3.      else return EUCLID (b, a mod b)
```

As an example of the running of EUCLID, consider the computation of gcd (30,21) :

$$\begin{aligned}\text{EUCLID}(30, 21) &= \text{EUCLID}(21, 9)\\ &= \text{EUCLID}(9, 3)\\ &= \text{EUCLID}(3, 0)\\ &= 3\,.\end{aligned}$$

The running time of Euclid's algorithm

Let a and b be two positive integers. Euclid's algorithm computes $\gcd(a,b)$ by executing $O(\log\max(a,b))$ arithmetic operations.

The extended form of Euclid's algorithm

We now rewrite Euclid's algorithm to compute additional useful information. Specifically, we extend the algorithm to compute the integer coefficients x and y such that

$$d = \gcd(a,b) = ax + by$$

Note that x and y may be zero or negative. The procedure EXTENDED-EUCLID takes as input an arbitrary pair of integers and returns a triple of the form (d,x,y).

```
EXTENDED-EUCLID (a, b)
1.  if b = 0
2.      then return (a, 1, 0)
3.  (d', x', y') ← EXTENDED-EUCLID(b, a mod b)
4.  (d, x, y) ← (d', y', x' − ⌊a / b⌋ y')
5.  return (d, x, y)
```

Example. *Compute gcd(99,78) with EXTENDED-EUCLID .*

Solution. Let $a = 99$ and $b = 78$

a	b	$\lfloor a/b \rfloor$	d	x	y
99	78	1	3	−11	14
78	21	3	3	3	−11
21	15	1	3	−2	3
15	6	2	3	1	−2
6	3	2	3	0	1
3	0	—	3	1	0

The call EXTENDED-EUCLID(99, 78) returns (3, − 11, 14), so gcd(99, 78) = 3 and $\gcd(99, 78) = 3 = 99(-11) + 78 \cdot 14$.

Since the number of recursive calls made in EUCLID is equal to the number of recursive calls made in EXTENDED-EUCLID, the running times of EUCLID and EXTENDED-EUCLID are the same, to within a constant factor. That is, for $a > b > 0$, the number of recursive calls is $O(\lg b)$.

34.3 The Chinese Remainder Theorem

Around A.D. 100, the Chinese mathematician sun-tsi solved the problem of finding those integers x that leave remainder 2 when divided by 3, remainder 3 when divided by 5, and remainder 2 when divided by 7. One such solution is $x = 23$; all solutions are of the form $23 + 105k$ for arbitrary integers k.

The "Chinese remainder theorem" provides a correspondence between a system of equations modulo a set of pair wise relatively prime moduli (for example, 3, 5, and 7) and an equation modulo their product (for example, 105).

Theorem

Let $n = n_1 n_2 \ldots n_k$, where the n_i are pairwise relatively prime. Consider the correspondence

$$a \leftrightarrow (a_1, a_2, \ldots, a_k)$$

where $a \in Z_n$, $a_i \in Z_{n_i}$, and $a_i = a \bmod n_i$

for $i = 1, 2, \ldots, k$. Then, mapping of the above equation is a one-to-one correspondence (bijection) between Z_n and the Cartesian product $Z_{n1} \times Z_{n2} \times \ldots \times Z_{nk}$. Operations performed on the elements of Z_n can be equivalently performed on the corresponding k-tuples by performing the operations independently in each coordinate position in the appropriate system. That is, if

$$a \leftrightarrow (a_1, a_2, \ldots, a_k),$$
$$b \leftrightarrow (b_1, b_2, \ldots, b_k),$$

then

$$(a+b) \bmod n \leftrightarrow ((a_1 + b_1) \bmod n_1, \ldots, (a_k + b_k) \bmod n_k),$$
$$(a-b) \bmod n \leftrightarrow ((a_1 - b_1) \bmod n_1, \ldots, (a_k - b_k) \bmod n_k),$$
$$(ab) \bmod n \leftrightarrow ((a_1 b_1) \bmod n_1, \ldots, (a_k b_k) \bmod n_k)$$

34.4 The RSA Public-key Cryptosystem

A public-key cryptosystem can be used to encrypt messages sent between two communicating parties so that an eavesdropper who overhears the encrypted messages will not be able to decode them. A public-key cryptosystem also enables a party to append an unforgeable "**digital signature**" to the end of an electronic message. Such a signature is the electronic version of a handwritten signature on a paper document. It can be easily checked by anyone, forged by no one, yet loses its validity if any bit of the message is altered. It therefore provides authentication of both the identity of the signer and the contents of the signed message. It is the perfect tool for electronically signed business contracts, electronic checks, electronic purchase orders, and other electronic communications that must be authenticated.

The RSA public-key cryptosystem is based on the dramatic difference between the ease of finding large prime numbers and the difficulty of factoring the product of two large prime numbers.

Public-key Cryptosystems

In a public-key cryptosystem, each participant has both a *public key* and a *secret key*. Each key is a piece of information. For example, in the RSA cryptosystem, each key consists of a pair of integers. Suppose the participants are "A" and "B". We denote their public and secret keys as P_A, S_A for A and P_B, S_B for B.

Each participant creates his own public and secret keys. Each keeps his secret key secret, but he can expose his public key to anyone or even publish it. In fact, it is often convenient to assume that everyone's public key is available in a public directory, so that any participant can easily obtain the public key of any other participant.

The function corresponding to A's public key P_A is denoted $P_A(\,)$, and the function corresponding to his secret key S_A is denoted $S_A(\,)$. The public and secret keys for any participant are a "matched pair" in that they specify functions that are inverses of each other. That is,

$$M = S_A(P_A(M))$$

$$M = P_A(S_A(M))$$

for any message M. Transforming M with the two keys P_A and S_A successively, in either order, yields the message M back.

In a public-key cryptosystem, it is essential that no one but A be able to compute the function $S_A(\,)$ in any practical amount of time. The assumption that only A can compute $S_A(\,)$ must hold even though everyone knows P_A and can compute $P_A(\,)$, the inverse function to $S_A(\,)$, efficiently. The major difficulty in designing a workable public-key cryptosystem is in figuring out how to create a system in which we can reveal a transformation $P_A(\,)$ without thereby revealing how to compute the corresponding inverse transformation $S_A(\,)$.

In a public-key cryptosystem, encryption works as follows. Suppose B wishes to send a message M to A and encrypted it, so that it will look like meaningless garbage to an eavesdropper.

The scenario for sending the message goes as follows.

- B obtains A's public key P_A (from a public directory or directly from A).
- B computes the ciphertext $C = P_A(M)$ corresponding to the message M and sends C to A.
- When A receives the ciphertext C, he applies his secret key S_A to retrieve the original message : $M = S_A(C)$.

Because $S_A(\,)$ and $P_A(\,)$ are inverse functions, A can compute M from C. Because only A is able to compute $S_A(\,)$, only A can compute M from C.

Digital signatures are similarly easy to implement in a public-key cryptosystem. Suppose that A wishes to send B a digitally signed response M'. The digital-signature scenario proceeds as follows.

A computes his *digital signature* σ for the message M' using his secret key S_A and the equation $\sigma = S_A(M')$.

A sends the message/signature pair (M', σ) to B.

When B receives (M', σ), he can verify that it originated from A using A's public key by verifying the equation $M' = P_A(\sigma)$. If the equation holds, then B concludes that the message M' was actually signed by A. If the equation doesn't hold, B concludes either that the message M' or the digital signature σ was corrupted by transmission errors or that the pair (M', σ) is an attempted forgery.

An important property of a digital signature is that it is verifiable by anyone who has access to the signer's public key. A signed message can be verified by one party and then passed on to other parties who can also verify the signature. After B verifies A's signature on the check, he can give the check to his bank, who can then also verify the signature and effect the appropriate funds transfer.

The RSA Cryptosystem

In the RSA public-key cryptosystem, a participant creates his public and secret keys with the following procedure.

1. Select at random two large prime numbers p and q.
2. Compute n by the equation $n = pq$.
3. Select a small odd integer e that is relatively prime to $\phi(n)$, which equals to $(p-1)(q-1)$.
4. Compute d as the multiplicative inverse of e, modulo $\phi(n)$.
5. Publish the pair $P = (e, n)$ as his ***RSA public key***.
6. Keep secret the pair $S = (d, n)$ as his ***RSA secret key***.

The transformation of a message M associated with a public key $P = (e, n)$ is

$$P(M) = M^e \pmod{n}$$

The transformation of a ciphertext C associated with a secret key $S = (d, n)$ is

$$S(C) = C^d \pmod{n}$$

These equations apply to both encryption and signatures. To create a signature, the signer applies his secret key to the message to be signed, rather than to a ciphertext. To verify a signature, the public key of the signer is applied to it, rather than to a message to be encrypted.

Exercise

1. Write the gcd binary algorithm for computing gcd of binary numbers.
2. Show that GCD operator is associative.
3. Write an algorithm for chinese remainder theorem using Ex Euclid.
4. Find all solutions to the equations

$$x \equiv 4 \pmod 5 \text{ and } x \equiv 5 \pmod{11}$$

5. Find all integer x that leave remainders 1, 2, 3 when divided by 9, 8, 7 respectively.
6. Consider an RSA key set with $p = 11$, $q = 29$, $n = 319$ and $e = 3$. What value of d should be used in the secret key ? What is the encryption of the message $M = 100$?
7. Discuss RSA cryptosystem.

CHAPTER

35

Polynomials and the FFT

35.1 Polynomial

A Polynomial in the variable x over an algebraic field F is representation of a function $A(x)$ as a formal sum.

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

$a_0, a_1, a_2, \ldots, a_{n-1}$ are called the coefficients of the polynomial. A polynomial $A(x)$ is said to have degree k if its highest non-zero coefficient is a_k. Any integer strictly greater than the degree of a polynomial is a **degree-bound** of that polynomial. Therefore, the degree of a polynomial of degree-bound n may be any integer between 0 and $n-1$.

35.2 Representation of Polynomials

Here, we introduce the *two* representations

1. Coefficient Representation
2. Point-value Representation

Coefficient Representation

A coefficient representation of a polynomial $A(x)=\sum_{j=0}^{n-1} a_j x^j$ of degree bound n is a vector of coefficients $a=(a_0, a_1, a_2, \ldots, a_{n-1})$.

Point-value Representation

A point-value representation of a polynomial $A(x)$ of degree-bound n is a set of n point-value pairs

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$$

such that all of the x_k are distinct and $y_k = A(x_k)$ for $k=0,1,2,\ldots,n-1$.

A polynomial has many different point-value representations since any set of n distinct points $x_0, x_1, \ldots, x_{n-1}$ can be used as a basis for the representation.

Computing a point-value representation for a polynomial given in coefficient form is in principle straight forward. We select n distinct points $x_0, x_1, \ldots, x_{n-1}$ and then evaluate $A(x_k)$ for $k=0,1,2,\ldots,n-1$.

With Horner's Method, this n-point evaluation takes time $\theta(n^2)$.

The inverse of evaluation that is determining the coefficient form of a polynomial from a point-value representation is called **interpolation**.

35.3 Evaluating Polynomial Functions

Consider the polynomial $P(x)=a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$ with real coefficients and $n \geq 1$ then to evaluate $P(x)$ the algorithm is as follows :

Poly (A, n, x)

```
P ← A[0], Power ← 1
for i ← 1 to n
    do power ← power * x
        P ← A[i]* power + p
return P
```

This algorithm does $2n$ multiplications and n additions.

Horner's Method

We have a better way of computing $ab+ac$ with fewer multiplications as $a(b+c)$. So we can write Horner's method for evaluating $P(x)$ using factorization.

$$P(x)=(\ldots((a_n x + a_{n-1})x + a_{n-2})x + \ldots + a_1)x + a_0$$

Horner Polynomial (A, n, x)

```
P ← A[n]
for j ← n − 1 down to 0 steps −1
    P ← P * x + A[j]
```

Thus simply by factoring P we have cut the number of multiplications in half without increasing the additions.

35.4 Complex Roots of Unity

A complex number ω is a complex nth root of unity, for $n \geq 2$, if it satisfies the following properties :

1. $\omega^n = 1$ that is w is an nth root of 1.
2. The numbers $1, \omega, \omega^2, \omega^3, \ldots, \omega^{n-1}$ are distinct.

The notion of complex nth root of unity has several important instances. One important one is complex number.

$$e^{2\pi i/n} = \cos(2\pi/n) + i\sin(2\pi/n)$$

which is complex nth root of unity, when we take our arithmetic over the complex numbers, where $i=\sqrt{-1}$.

The value $\omega_n = e^{2\pi i/n}$ is called the **principal nth root of unity**, and all of the other complex nth roots of unity are powers of ω_n.

Thus, the n complex nth root of unity are :

$\omega_n^0, \omega_n^1, \omega_n^2, \ldots, \omega_n^{n-1}$, which form a group under multiplication. This group has the same structure as the additive group $(Z_n, +)$ modulo n since $\omega_n^n = \omega_n^0 = 1$ implies that $\omega_n^j \omega_n^k = \omega_n^{j+k} = \omega_n^{(j+k) \bmod n}$.

Complex nth root of unity have a number of important properties such as

1. ***For any integer $n \geq 0$ $k \geq 0$ and $d > 0$***

$$\omega_{dn}^{dk} = \omega_n^k$$

Proof. We know $\omega_n = e^{2\pi i/n}$.

$$\omega_{dn}^{dk} = (e^{2\pi i/dn})^{dk}$$

$$= (e^{2\pi i/n})^k = \omega_n^k$$

2. ***For any integer $n \geq 1$ and non-negative integer k not divisible by n.***

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = 0$$

Proof. $$\sum_{j=0}^{n-1} (\omega_n^k)^j = \frac{(\omega_n^k)^n - 1}{\omega_n^k - 1} = \frac{(\omega_n^n)^k - 1}{\omega_n^k - 1}$$

$$= \frac{1^k - 1}{\omega_n^k - 1} = \frac{1-1}{\omega_n^k - 1} = 0$$

Since k is not divisible by n ensures that ω_n^k is not equal to 1 because $\omega_n^k = 1$ only when k is divisible by n. Thus denominator is not 0.

3. ***For any even integer $n > 0$***

$$\omega_n^{n/2} = -1 = \omega_2$$

Proof. $\omega_n^k = e^{2\pi ik/n}$

$$\omega_n^{n/2} = \omega^{2\pi i\, n/2n} = e^{\pi i} = \omega_2$$

$$= \cos(\pi) + i\sin(\pi)$$

$$= -1$$

$\therefore$ $\omega_n^{n/2} = \omega_2 = -1$

4. ***Halving Lemma***

If $n > 0$ is even, then the squares of the n complex nth roots of unity are the $n/2$ complex $(n/2)$th roots of unity.

Proof. We know $\omega_n^{n/2} = -1$

$$\omega_n^{n/2} \cdot \omega_n^k = -1 \cdot w_n^k$$

$$\omega_n^{n/2+k} = -\omega_n^k$$

Thus $(\omega_n^{n/2+k})^2 = (\omega_n^k)^2$

Thus, if we square all of the complex nth roots of unity then each $(n/2)$th root of unity is obtained exactly twice.

$$(\omega_n^k)^2 = (\omega_n^{2k})$$

$$= \omega_{n/2}^k \qquad [\because\ \omega_{dn}^{dk} = \omega_n^k]$$

35.5 Discrete Fourier Transform

We wish to evaluate a polynomial

$$A(x) = \sum_{j=0}^{n-1} a_j x^j \text{ of degree-bound } n \text{ at } \omega_n^k \text{ where } k = 0,1,2,\ldots,n-1.$$

i.e., $\omega_n^0, \omega_n^1, \omega_n^2, \ldots, \omega_n^{n-1}$

We assume that A is given in coefficient form $a = (a_0, a_1, \ldots, a_{n-1})$.

Thus DFT is to evaluate $A(x)$ at the nth roots of unity $\omega_n^0, \omega_n^1, \omega_n^2, \ldots, \omega_n^{n-1}$.

Let us define the results y_k, for $k = 0,1,2,\ldots,n-1$ by

$$y_k = A(\omega_n^k) \qquad i.e.,\ x = \omega_n^k$$

$$= \sum_{j=0}^{n-1} a_j \omega_n^{kj}$$

The vector $y = (y_0, y_1, \ldots, y_{n-1})$ is the Discrete Fourier Transform (DFT) of the coefficient vector $(a_0, a_1, \ldots, a_{n-1})$.

We also write $y = \text{DFT}_n(a)$

35.6 Fast Fourier Transform (FFT)

The Fast Fourier Transform (FFT) algorithm computes a Discrete Fourier Transform (DFT) of an n-length vector in $O(n \lg n)$ time. In the FFT algorithm, we apply the divide-and-conquer approach to polynomial evaluation by observing that if n is even, we can divide a degree-$(n-1)$ polynomial

$$A(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-1} x^{n-1}$$

into two degree $\left(\frac{n}{2} - 1\right)$ polynomials.

$$A^{[0]}(x) = a_0 + a_2 x + a_4 x^2 + \ldots + a_{n-2} x^{n/2-1}$$

$$A^{[1]}(x) = a_1 + a_3 x + a_5 x^2 + \ldots + a_{n-1} x^{n/2-1}$$

Note that $A^{[0]}$ contains all the even index coefficients of A and $A^{[1]}$ contains all the odd index coefficients and noting that we can combine these two polynomials into A, using the equation.

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2) \qquad \ldots(1)$$

So that the problem of evaluating $A(x)$ at w_n^k where $k = 0,1,2,\ldots,n-1$ reduces to

1. evaluating the degree $\left(\frac{n}{2}-1\right)$ polynomial $A^{[0]}(x)$ and $A^{[1]}(x)$ at the point $(\omega_n^k)^2$ *i.e.,*

   $$(\omega_n^0)^2, (\omega_n^1)^2, \ldots, (\omega_n^{n-1})^2$$

   because we know that if ω_n^k is a complex nth root of unity then $(\omega_n^k)^2$ is a complex $n/2$th root of unity. Thus, we can evaluate each $A^{[0]}(x)$ and $A^{[1]}(x)$ at $(\omega_n^k)^2$ values.

2. Combining the results according to the equation (1). This observation is the basis for the following procedure which computes the DFT of an n-element vector $a = (a_0, a_1, \ldots, a_{n-1})$ where for sake of simplicity, we assume that n is a power of 2.

FFT (a, w)

```
1.  n ← length[a]                    ▷ n is a power of 2.
2.  if n = 1
3.      then return a
4.  ω_n ← e^(2πi/n)
5.  x ← ω^0                          ▷ x will store powers of ω, initially x = 1.
6.  a^[0] ← (a_0, a_2, ... a_(n-2))
7.  a^[1] ← (a_1, a_3, ... a_(n-1))
8.  y^[0] ← FFT(a^[0], ω^2)          ▷ Recursive calls with ω^2 as (n / 2)th root of unity.
9.  y^[1] ← FFT(a^[0], ω^2)
10. for k ← 0 to n / 2 - 1
11.     do y_k ← y_k^[0] + x y_k^[1]
12.        y_(k+(n/2)) ← y_k^[0] - x y_k^[1]
13.        x ← x ω_n
14. return y
```


Line 2-3 represents the basis of recursion, the DFT of one element is the element if self. Since in this case

$$y_0 = a_0 \omega_1^0 = a_0 1 = a_0$$

Line 6-7 define the recursive the coefficient vectors for the polynomials $A^{[0]}$ and $A^{[1]}$. $\omega = \omega_n^k$

Line 8-9 perform the recursive DFT$_{n/2}$ computations setting for $k = 0,1,2,\ldots,\frac{n}{2}-1$ *i.e.,*

$$y_k^{[0]} = A^{[0]}(\omega_n^{2k})$$

$$y_k^{[1]} = A^{[1]}(\omega_n^{2k})$$

Lines 11-12 combine the results of the recursive DFT$_{n/2}$ calculations.

For $y_0, y_1, y_2, \ldots y_{(n/2)-1}$ line 11 yields.

$$\begin{aligned} y_k &= y_k^{[0]} + \omega_n^k y_k^{[1]} \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^k A^{[1]}(\omega_n^{2k}) \\ &= A(\omega_n^k) && \triangleright \text{ follows equation (1)} \end{aligned}$$

For $y_{n/2}, y_{(n/2)+1} \ldots y_{n-1}$, line 12 yields.

$$\begin{aligned} y_{k+(n/2)} &= y_k^{[0]} - \omega_n^k y_k^{[1]} \\ &= y_k^{[0]} + \omega_n^{k+(n/2)} y_k^{[1]} && [\because \omega_n^{k+(n/2)} = -\omega_n^k] \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k}) \\ &= A^{[0]}(\omega_n^{2k}\, \omega_n^n) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k}\, \omega_n^n) && [\because \omega_n^n = 1] \\ &= A^{[0]}(\omega_n^{2k+n}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k+n}) \\ &= A(\omega_n^{k+(n/2)}) && \triangleright \text{ follows equation (1)} \end{aligned}$$

for each $k = 0,1,2,\ldots,(n/2)-1$.

Thus, the vector y returned by the FFT algorithm will store the values of $A(x)$ at each of the nth roots of unity.

Analyzing the FFT Algorithm

The FFT algorithm follows the divide-and-conquer paradigm, dividing the original problem of size n into two subproblem of size $n/2$, which are solved recursively. We assume that each

arithmetic operation performed by algorithm takes $O(1)$ time. The divide step as well as the combine step for merging the recursive solution, each take $O(n)$ time. Thus

$$T(n)=2T(n/2)+bn \quad \text{for } b>0$$

By solving, master theorem, we get

$$T(n)=O(n\lg n)$$

Exercise

1. Multiply the polynomial $A(x)=7x^3-x^2+x-10$ and $B(x)=8x^3-16x+3$.
2. Compute the DFT of the vector (0, 1, 2, 3,)
3. Describe the generalization of the FFT procedure to the case in which n is a power of 3. Give a recurrence for the running time and solve the recurrence.

CHAPTER 36

String Matching

36.1 Introduction

String matching consists of finding one or more generally, all of the occurrences of a pattern in a text. Finding a certain pattern in a text is a problem arises in text-editing programs and web "surfing". Given a text array, $T[1..n]$, of n character and a pattern array, $P[1..m]$, of m characters. The problem is to find an integer s, called **valid shift** where $0 \le s < n-m$ and $T[s+1...s+m]=P[1..m]$. In other words, to find whether P in T i.e., where P is a substring of T. The elements of P and T are character drawn from some finite alphabet such as {0, 1} or {A, B,...Z,a,b,.....z}.

Given a string $T[1..n]$, the **substrings** is define as $T[i..j]$ for some $0 \le i \le j \le n-1$, that is, the string formed by the characters in T from index i to index j, inclusive. This means that a string is a substring of itself (simply take $i=0$ and $j=n$).

The **proper substring**, of string $T[1..n]$ is $T[1..j]$ for some $0<i\le j\le n-1$. That is, we must have either $i>0$ or $j<m-1$.

Using these definition, we can say given any string $T[1..n]$, the substrings are

$$T[i..j]=T[i]\,T[i+1]\,T[i+2]...T[j] \text{ for some } 0\le i\le j\le n-1$$

And proper substrings are

$$T[i..j]=T[i]T[i+1]T[i+2]...T[j] \text{ for some } 0<i\le j\le n-1$$

Note that if $i > j$, then $T[i..j]$ is equal to the empty string or null, which has length zero.

We say that a string is a prefix of a string x, denoted $w \sqsubset x$, if $x = wy$ for some string $y \in \Sigma^*$. Note that if $w \sqsubset x$, then $|w| \le |x|$. Similarly, we say that a string w is a suffix of a string x, denoted $w \sqsupset x$, if $x = yw$ for some $y \in \Sigma^*$. It follows from $w \sqsupset x$ that $|w| \le |x|$. The empty string ε is both a suffix and a prefix of every string.

36.2 The Naive String-matching Algorithm

The naive approach simply test all the possible placement of Pattern $P[1..m]$ relative to text $T[1..n]$. Specifically, we try shift $s=0,1,\ldots,n-m$, successively and for each shift, s. Compare $T[s+1..s+m]$ to $P[1..m]$.

The naive algorithm finds all valid shifts using a loop that checks the condition $P[1..m]=T[s+1..s+m]$ for each of the $n-m+1$ possible values of s.

NAIVE-STRING-MATCHER (T, P)

1. $n \leftarrow length[T]$
2. $m \leftarrow length[P]$
3. for $s \leftarrow 0$ to $n-m$
4. do *if* $P[1..m] = T[s+1..s+m]$
5. then print "Pattern occurs with shift" s

The for loop beginning on line 3 considers each possible shift explicitly. The test on line 4 determines whether the current shift is valid or not ; this test involves an implicit loop to check corresponding character positions until all positions match successfully or a mismatch is found. Line 5 prints out each valid shift s.

The naive string-matching procedure can be interpreted graphically as a sliding a pattern $P[1..m]$ over the text $T[1..n]$ and noting for which shift all of the characters in the pattern match the corresponding characters in the text.

In other to analysis the time of naive matching, we would like to implement above algorithm to understand the test involves in line 4.

Note that in this implementation, we use notation P[1 . . j] to denote the substring of P from index i to index j. That is, $P[1..j] = P[i]P[i+1]\ldots P[j]$.

NAIVE-STRING-MATCHER (T, P)

1. $n \leftarrow$ length $[T]$
2. $m \leftarrow$ length $[P]$
3. for $s \leftarrow 0$ to $n-m$ do
4. $j \leftarrow 1$
5. while $j \le m$ and $T[s+j] = p[j]$ do
6. $j \leftarrow j+1$
7. If $j > m$ then
8. return valid shift s
9. return no valid shift exist // *i.e.*, there is no substring of T matching P.

Referring to implementation of naive matcher, we see that the for-loop in line 3 is executed at most $n-m+1$ times, and the while-loop in line 5 is executed at most m times. Therefore, the running time of the algorithm is $O((n-m+1)m)$, which is clearly $O(nm)$. Hence, in the worst case, when the length of the pattern, m are roughly equal, this algorithm runs in the quadratic time.

36.3 The Rabin-Karp Algorithm

The **Rabin-Karp algorithm** is a string searching algorithm created by **Michael O. Rabin** and **Richard M. Karp** that seeks a pattern, *i.e.* a substring, within a text by using **hashing**. This string searching algorithm calculates a hash value for the pattern, and for each M-character subsequence of text to be compared. If the hash values are unequal, the algorithm will calculate the hash value for next M-character sequence. If the hash values are equal, the algorithm will compare the pattern and the M-character sequence. In this way, there is only one comparison per text subsequence, and character matching is only needed when hash values match.

Thus, Rabin-Karp exploits the fact that if two strings are equal, their hash values are also equal. Thus, it would seem all we have to do is compute the hash value of the substring we're searching for, and then look for a substring with the same hash value.

However, there are two problems with this. First, because there are so many different strings, to keep the hash values small we have to assign some strings the same number. This means that if the hash values match, the strings might not match; we have to verify that they do, which can take a long time for long substrings. Luckily, a good hash function promises us that on most reasonable inputs, this won't happen too often, which keeps the average search time good.

This algorithm makes use of elementary number-theoretic notions such as the equivalence of two numbers modulo a third number. Given a pattern $P[1..m]$, we let p denote its corresponding decimal value. In a similar manner, given a text $T[1..n]$, we let t_s denote the decimal value of the length-m substring $T[s+1..s+m]$, for $s=0,1,\ldots,n-m$. Certainly, $t_s = p$ if and only if $T[s+1..s+m] = P[1..m]$; thus, s is a valid shift if and only if $t_s = p$. If we could compute p in time $O(m)$ and all of the t_i values in a total of $O(n)$ time, then we could determine all valid shifts s in time $O(n)$ by comparing p with each of the t_s's. We can compute p in time $O(m)$ using **Horner's rule.**

$$p = P[m]+10(P[m-1]+10(P[m-2]+\ldots+10(P[2]+10P[1])\ldots))$$

The value t_0 can be similarly computed from $T[1..m]$ in time $O(m)$.

To compute the remaining values $t_1, t_2, \ldots, t_{n-m}$ in time $O(n-m)$, it suffices to observe that t_{s+1} can be computed from t_s in constant time, since

$$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1]$$

Subtracting $10^{m-1}T[s+1]$ removes the high-order digit from t_s, multiplying the result by 10 shifts the number left one position, and adding $T[s+m+1]$ brings in the appropriate low-order digit.

In general, with a d-ary alphabet $\{0,1,\ldots,d-1\}$, we choose q so that $d\ q$ fits within a computer word and adjust the above recurrence equation to work modulo q, so that it becomes

$$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q,$$

where $h \equiv d^{m-1} \pmod q$ is the value of the digit "1" in the high-order position of an m-digit text window. Since $t_s \equiv p \pmod q$ does not imply that $t_s = p$. On the other hand, if $t_s \neq p \pmod q$, then we definitely have that $t_s \neq p$, so that shift s is invalid. We can thus use the test $t_s \equiv p \pmod q$ as a fast heuristic test to rule out invalid shifts s. Any shift s for which $t_s \equiv p \pmod q$ must be tested further to see if s is really valid or we just have a spurious hit. This testing can be done by explicitly checking the condition $P[1..m] = T[s+1..s+m]$. If q is large enough, then we can hope that spurious hits occur infrequently enough that the cost of the extra checking is low.

The following procedure makes these ideas precise. The inputs to the procedure are the text T, the pattern P, the radix d and the prime q to use.

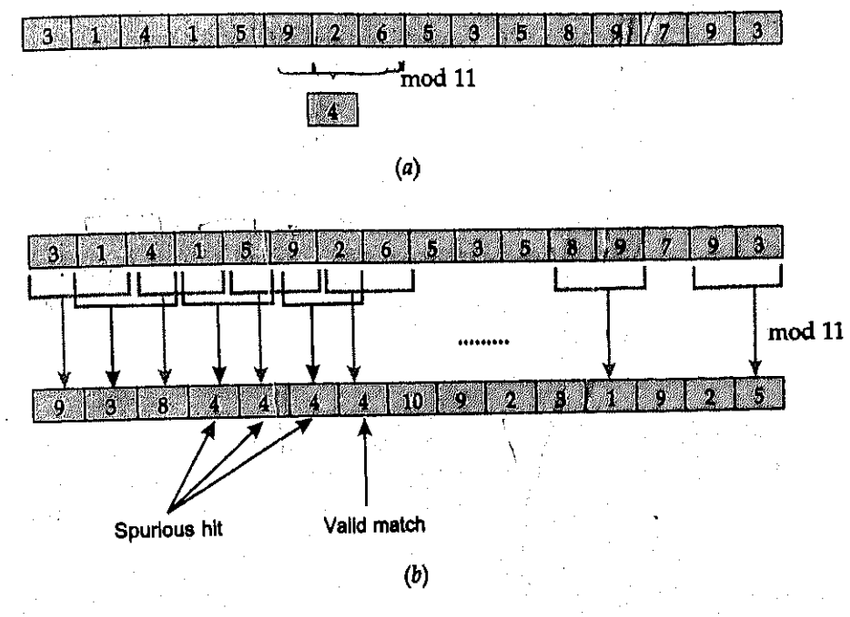
```
RABIN-KARP-MATCHER (T, P, d, q)
1.  n ← length[T]
2.  m ← length[P]
3.  h ← d^(m-1) mod q
4.  p ← 0
5.  t_0 ← 0
6.  for i ← 1 to m
7.      do p ← (dp + P[i]) mod q
8.      t_0 ← (dt_0 + T[i]) mod q
9.  for s ← 0 to n - m
10.     do if p = t_s
11.         then if P[1..m] = T[s + 1..s + m]
12.             then "Pattern occurs with shift" s
13.     if s < n - m
14. then t_(s+1) ← (d(t_s - T[s + 1]h) + T[s + m + 1]) mod q
```

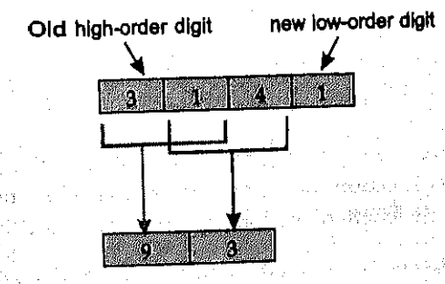
The procedure RABIN-KARP-MATCHER works as follows. Line 3 initializes h to the value of the high-order digit position of an m-digit window. Lines 4-8 compute p as the value of $P[1..m] \bmod q$ and t_0 as the value of $T[1..m] \bmod q$. The for loop beginning on line 9 iterates through all possible shifts s. The loop has the following invariant: whenever line 10 is executed, $t_s = T[s+1..s+m] \bmod q$. If $p = t_s$ in line 10 (a "hit"), then we check to see if $P[1..m] = T[s+1..s+m]$ in line 11 to rule out the possibility of a spurious hit. Any valid shifts found are printed out on line 12. If $s < n-m$ (checked in line 13), then the for loop is to be executed at least one more time, and so line 14 is first executed to ensure that the loop invariant holds when line 10 is again reached. Line 14 computes the value of $t_{s+1} \bmod q$ from the value of $t_s \bmod q$ in constant time.

The running time of RABIN-KARP-MATCHER is $\Theta((n-m+1)m)$ in the worst case, since (like the naive string-matching algorithm) the Rabin-Karp algorithm explicitly verifies every valid shift. If $P = a^m$ and $T = a^n$, then the verifications take time $\Theta((n-m+1)m)$, since each of the $n-m+1$ possible shifts is valid.

Example. *For string matching working module $q=11$, how many spurious hits does the Rabin-Karp matcher encounter in the text $T = 3141592653589793$, when looking for the pattern $P = 26$?*

Solution.





$t_0 = 9$

$t_1 = (10(t_0 - 3 \times 10) + T(3)) \bmod 11$

$(10(9-30)+4) \bmod 11$

Figure 36.1

Here $m=2$ so

$$\begin{aligned} t_{s+1} &= 10(31 - 10^{2-1} \cdot 3) + 4 \\ &= 10(31-30)+4 \\ &= 14 \end{aligned}$$

Fig. 36.1(a) shows a text string and a window of length 2. The numerical value of the window is computed modulo 11 yielding the value 4.

The same text string with values computed modulo 11 for each possible position of a length two window.

Four such window are found. The first three windows are spurious hit because in these modulo is same but the pattern is not same.

Thus, the total number of spurious hits are 3.

36.4 String Matching with Finite Automata

String-matching automata are very efficient because they examine each text character exactly once, taking constant time per text character. The time used-after the automaton is built - is therefore $\theta(n)$. The time to build the automaton, however, can be large if Σ is large. A finite automaton M is a 5-tuple $(Q, q_0, A, \Sigma, \delta)$, where

- Q is a finite set of states,
- $q_0 \in Q$ is the start state,
- $A \subseteq Q$ is a distinguished set of accepting states,
- Σ is a finite input alphabet,
- δ is a function from $Q \times \Sigma$ into Q called the transition function of M

The finite automaton begins in state q_0 and reads the characters of its input string one at a time. If the automaton is in state q and reads input character a, it moves from state q to state $\delta(q, a)$. Whenever its current state q is a member of A, the machine M is said to have **accepted** the string read so far. An input that is not accepted is said to be **rejected**. A finite automaton M induces a function ϕ called the **final-state function**, from Σ^* to Q such that $\phi(w)$ is the state M ends up in after scanning the string w. Thus, M accepts a string w if and only if $\phi(w) \in A$. The function f is defined as

$$\phi(\varepsilon) = q_0$$

$$\phi(wa) = \delta(\phi(w), a) \text{ for } w \in \Sigma^*,\ a \in \Sigma$$

In order to specify the string-matching automaton corresponding to a given pattern $P[1..m]$, we first define an auxiliary function σ, called the **suffix function** corresponding to P. The function σ is a mapping from Σ^* to $\{0,1,\ldots,m\}$ such that $\sigma(x)$ is the length of the longest prefix of P that is a suffix of x:

$$\sigma(x) = \max\{k : P_k \sqsupset x\}$$

We define the string-matching automaton corresponding to a given pattern $P[1..m]$ as follows.

The state set Q is $\{0,1,\ldots,m\}$. The start state q_0 is state 0, and state m is the only accepting state.

The transition function δ is defined by the following equation, for any state q and character a:

$$\delta(q, a) = \sigma(P_q a)$$

As for any string-matching automaton for a pattern of length m, the state set Q is $\{0,1,\ldots,m\}$, the start state is 0, and the only accepting state is state m.

FINITE-AUTOMATON-MATCHER (T, δ, m)

```
n ← length[T]
q ← 0
for i ← 1 to n
    do q ← δ(q, T[i])
        if q = m
            then s ← i − m
                print "Pattern occurs with shift" s
```

The simple loop structure of FINITE-AUTOMATON-MATCHER implies that its running time on a text string of length n is $O(n)$.

Computing the Transition Function

The following procedure computes the transition function δ from a given pattern $P[1..m]$

COMPUTE-TRANSITION-FUNCTION (P, Σ)

```
m ← length[P]
for q ← 0 to m
    do for each character a ∈ Σ*
    do k ← min(m + 1, q + 2)
repeat k ← k − 1
until
    δ(q, a) ← k
return δ
```

36.5 The Knuth-Morris-Pratt (KMP) Algorithm

Knuth, Morris and Pratt discovered first linear time string-matching algorithm by following a tight analysis of the naive algorithm. Knuth-Morris-Pratt algorithm keeps the information that naive approach wasted gathered during the scan of the text. By avoiding this waste of information, it achieves a running time of $O(n+m)$, which is achieved by using the auxiliary function π. That is, in the worst-case Knuth-Morris-Pratt algorithm we have to examine all the characters in the text and pattern at least once.

The function π computes the shifting of pattern matches within itself. It can be observed that if we know that how the pattern matches shifts against itself, then we can slide the pattern more characters towards right than just one character.

The basic idea is to slide the pattern towards the right along the string so that the longest prefix of P that we have matched matches the longest suffix of T that we have already matched. If the longest prefix of P that matches a suffix of T is nothing, then we slide the whole pattern towards right. The algorithm computing prefix function is as follows.

```
COMPUTE-PREFIX-FUNCTION (P)
m ← length [P]
π[1] ← 0
k ← 0
for q ← 2 to m
    do while k > 0 and P[k + 1] ≠ P[q]
        do k ← π[k]
    if P[k + 1] = P[q]
        then k ← k + 1
    π[q] ← k
return π
```

The Knuth-Morris-Pratt matching algorithm is given in pseudocode below as the procedure KMP-MATCHER. KMP-MATCHER calls the auxiliary procedure COMPUTE-PREFIX-FUNCTION to compute π

```
KMP-MATCHER (T,P)
n ← length [T]
m ← length [P]
π ← COMPUTE-PREFIX-FUNCTION (P)
q ← 0
for i ← 1 to n
    do while q > 0 and P[q + 1] ≠ T[i]
        do q ← π[q]
    if P[q + 1] = T[i]
        then q ← q + 1
if q = m
then print "Pattern occurs with shift" i − m
q ← π[q]
```

Example. *Compute the prefix function π for the pattern ababbabbabbababbabb when the alphabet is $\Sigma = \{a, b\}$.*

Solution. Here pattern $P = ababbabbabbababbabb$

length $[P] = 19$ so $m = 19$

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$P[i]$	a	b	a	b	b	a	b	b	a	b	b	a	b	a	b	b	a	b	b
$\pi(i)$	0	0	1	2	0	1	2	0	1	2	0	1	2	3	4	5	6	7	8

Initially $\pi(1) = 0$ and $k = 0$ and $q = 2$ and $P[k+1] \neq P[q]$. So $\pi[2] = 0$

Now $P[k+1] = P[q]$ for $q = 3$

then $k = k+1$ *i.e.*, $k = 1$ and $\pi[3] = 1$

then $k > 0$ and $P[k+1] = P[q]$ for $q = 4$

then $k = k+1$ *i.e.*, $k = 2$

and $\pi[4] = 2$

Now $q = 5$ $k = 2$

i.e., $k > 0$ and $P[k+1] \neq P[q]$

$a \neq b$

$k = \pi[2]$

$k = 0$

Now, $P[1] \neq P[q]$ so $\pi[5] = 0$

Similarly, we compute other prefix functions.

36.6 The Boyer-Moore Algorithm

The Boyer-Moore algorithm is consider the most efficient string-matching algorithm in usual applications, for example, in text editors and commands substitutions. The reason is that it woks the fastest when the alphabet is moderately sized and the pattern is relatively long. It was developed by **Bob Boyer** and **J Strother Moore** in 1977

The algorithm scans the characters of the pattern from right to left beginning with the rightmost character, which is also referred to as "**looking glass heuristic**". If the text symbol that is compared with the rightmost pattern symbol does not occur in the pattern at all, then the pattern can be shifted by m positions behind this text symbol.

Example

0	1	2	3	4	5	6	7	8	9	...
a	b	b	a	d	a	b	a	c	b	a
b	a	b	a	c						
					b	a	b	a	c	

The first comparison $d - c$ at position 4 produces a mismatch. The text symbol d does not occur in the pattern. Therefore, the pattern cannot match at any of the positions 0, ..., 4, since all corresponding windows contain a d. The pattern can be shifted to position 5. The best case for the Boyer-Moore algorithm is attained if at each attempt the first compared text symbol does not occur in the pattern. Then the algorithm requires only $O(n/m)$ comparisons.

For determining the smallest possible shifts, Boyer-Moore works with two different preprocessing strategies. Each time when a mismatch occurs the algorithm computes both and chooses the largest possible shift, thus making use of the most efficient strategy for each individual case.

Bad character Heuristics

This method is called bad character heuristics. It can also be applied if **the bad character**, i.e. the text symbol that causes a mismatch, occurs somewhere else in the pattern. Then the pattern can be shifted so that it is aligned to this text symbol. The next example illustrates this situation.

Example

Comparison $b-c$ causes a mismatch. Text symbol b occurs in the pattern at positions 0 and 2. The pattern can be shifted so that the rightmost b in the pattern is aligned to text symbol b.

0	1	2	3	4	5	6	7	8	9	...
A	*b*	*b*	*a*	*b*	*a*	*b*	*a*	*c*	*b*	*a*
B	*a*	*b*	*a*	*c*						
		b	*a*	*b*	*a*	*c*				

Good Suffix Heuristics

Sometimes the bad character heuristics fails. In the following situation the comparison $a-b$ causes a mismatch. An alignment of the rightmost occurence of the pattern symbol a with the text symbol a would produce a **negative shift**. Instead, a shift by 1 would be possible. However, in this case it is better to derive the maximum possible shift distance from the structure of the pattern. This method is called **good suffix heuristics**.

Example

0	1	2	3	4	5	6	7	8	9	...
A	*b*	*a*	*a*	*b*	*a*	*b*	*a*	*c*	*b*	*a*
c	*a*	*b*	*a*	*b*						
		c	*a*	*b*	*a*	*b*				

The suffix *ab* has matched. The pattern can be shifted until the next occurence of *ab* in the pattern is aligned to the text symbols *ab, i.e.* to position 2.

In the following situation the suffix *ab* has matched. There is no other occurence of *ab* in the pattern.Therefore, the pattern can be shifted behind *ab, i.e.* to position 5.

BOYER-MOORE-MATCHER (T, P, Σ)

```
n ← length[T]
m ← length[P]
λ ← COMPUTE-LAST-OCCURRENCE-FUNCTION (P, m, Σ)
γ ← COMPUTE-GOOD-SUFFIX-FUNCTION (P, m)
s ← 0
while s ≤ n − m
    do j ← m
        while j > 0 and P[j] = T[s + j]
            do j ← j − 1
        if j = 0
            then print "Pattern occurs at shift" s
                s ← s + γ[0]
            else s ← s + max(γ[j], j − λ[T[s + j]])
```


COMPUTE-LAST-OCCURRENCE-FUNCTION (P, m, Σ)

```
for each character a ∈ Σ
    do λ[a] = 0
for j ← 1 to m
    do λ[P[j]] ← j
return λ
```

COMPUTE-GOOD-SUFFIX-FUNCTION (P, m)

```
π ← COMPUTE-PREFIX-FUNCTION (P)
P' ← reverse(P)
π' ← COMPUTE-PREFIX-FUNCTION (P')
for j ← 0 to m
    do γ[j] ← m − π[m]
for l ← 1 to m
    do j ← m − π'[l]
        if γ[j] > l − π'[l]
            then γ[j] ← 1 − π'[l]
return γ
```

The worst-case running time of the Boyer-Moore algorithm is clearly $O((n-m+1)m+|\Sigma|)$, since COMPUTE-LAST-OCCURRENCE-FUNCTION takes time $O(m+|\Sigma|)$, COMPUTE-GOOD-SUFFIX-FUNCTION takes time $O(m)$, and the Boyer-Moore algorithm (like the Rabin-Karp algorithm) spends $O(m)$ time validating each valid shift s. In practice, however, the Boyer-Moore algorithm is often the algorithm of choice.

Example. *Suppose that all characters in the pattern P are different. Show how to accelerate NAIVE-STRING-MATCHER to run in time O(n) on an n-character text T.*

Solution. Assume all the characters of P are different. A mismatch with T a position i of P in line 4 of NAIVE-STRING-MATCHER then implies that we can continue our search from position $s+i$ in T. Thus a linear search of T is a sufficient.

Example. *Suppose we allow the pattern P to contain occurrences of a gap character ◊ that can match an arbitrary string of characters. (even one of zero length).*

For example, the pattern ab◊ba◊c occurs in the text cabccbacbacab as

$$c\ \underbrace{ab}_{ab}\ \underbrace{cc}_{\Diamond}\ \underbrace{ba}_{ba}\ \underbrace{cba}_{\Diamond}\ \underbrace{c}_{c}\ ab$$

Note that the gap character may occur an arbitrary number of times in the pattern but is assumed not to occur at all in the text. Give a polynomial time algorithm to determine if such a pattern P occurs in a given text T, and analyze the running time of your algorithm.

Solution. To determine if a pattern P with gap characters exists in T. We partition P into substrings $P_1, P_2, P_3, \ldots, P_k$ determine by the gap characters.

Search for P_1 and if found continue searching for P_2 and so on. This clearly find a apttern if one exists.

Example. *Given a pattern P containing gap characters. Show how to build a finite automaton that can find an occurrence of P in a text T in O(n) time, where* $n = |T|$.

Solution. We can construct the finite automaton corresponding to a pattern. P using the same idea as in above question. Partition P into substrings $P_1, P_2, \ldots P_k$ determined by the gap characters. Construct finite automatons for each P_i and combine sequentially *i.e.*, the accepting state of $P_i, 1 \geq i < k$ is no longer accepting but has a single transition to P_{i+1}.

Example. *Give a linear-time algorithm to determine if a text T is a cycle rotation of another string T'. For example arc and car are cyclic rotations of each other.*

Solution. We can determine if T is a cycle rotation of T' matching T' against TT.

Exercise

1. Show that the worst-case time for the naive string matcher to find the first occurrence of a pattern in a text is $\theta((n-m+1)(m-1))$.
2. Show the comparisons the naive string matcher makes for the pattern $P = 10001$ in the text $T =$ 0000100010010
3. How would you extend the Rabin-karp method to the problem of searching a text string for an occurrence of any once of a given set of k patterns ?
4. Draw a state-transition diagram for a string-matching automaton for the pattern $ababbabbababbababbabb$ over the alphabet $\sum = \{a,b\}$.
5. Compute the prefix function π for the pattern $ababbabbababbababbabb$ when the alphabet is $\sum = \{a,b\}$.
6. Explain how to determine the occurrences of pattern P in the text T by examining the π function for the string PT. (the string of length $m+n$ that is the concatenation of P and T).
7. Compute the λ and γ functions for the pattern $P = 0101101201$ and the alphabet $\sum = \{0,1,2\}$.

CHAPTER

37

Computational Geometry

37.1 Introduction

Computational geometry is a term claimed by a number of different groups. The term was coined perhaps first by **Marvin Minsky** in his book **"Perceptrons"**, which was about pattern recognition, and it has also been used often to describe algorithms for manipulating curves and surfaces in solid modeling.

In computer science, computational geometry is the study of algorithms to solve problems stated in terms of geometry. Some purely geometrical problems arise out of the study of computational geometry. In modern engineering and mathematics, computational geometry has applications in, among other fields, computer graphics, robotics, VLSI design, computer-aided design, and statistics. The input to a computational-geometry problem is typically a description of a set of geometric objects, such as a set of points, a set of line segments, or the vertices of a polygon in counterclockwise order. The output is often a response to a query about the objects, such as whether any of the lines intersect, or perhaps a new geometric object, such as the **convex hull** (smallest enclosing convex polygon) of the set of points.

There are many fields of computer science that deal with solving problems of a geometric nature. These include computer graphics, computer vision and image processing, robotics, computer-aided design and manufacturing, computational fluid dynamics, and geographic information systems, to name a few. One of the goals of computational geometry is to provide the basic geometric tools needed from which application areas can then build their programs. There has been significant progress made towards this goal, but it is still far from being realized.

37.2 Strengths and Limitations of Computational Geometry

Strengths of Computational Geometry

1. Development of Geometric Tools
2. Emphasis on Provable Efficiency
3. Emphasis on Correctness/Robustness
4. Linkage to Discrete Combinatorial Geometry

Limitations of Computational Geometry

1. Emphasis on discrete geometry
2. Emphasis on flat objects
3. Emphasis on low-dimensional spaces

37.3 Polygons

A polygon is just a collection of line segments, forming a cycle, and not crossing each other. We can represent it as a sequence of points, each of which is just a pair of coordinates. For instance the points (0, 0), (0, 1), (1, 1), (1, 0) form a square. The line segments of the polygon connect adjacent points in the list, together with one additional segment connecting the first and last point.

Not all sequences of points form a polygon ; for instance the points (0,−1), (0,1), (1,0), (0,−1) would result in two segments that cross each other. Sometimes we use the phrase simple polygon to emphasize the requirement that no two segments cross.

Testing if a point in a polygon

It is a famous theorem (the **Jordan curve theorem**) that any polygon cuts the plane into exactly two connected pieces : the *inside* and the *outside*. (The inside always has some finite size, while the outside contains points arbitrarily far from the polygon.) Actually, the Jordan curve theorem is more generally true of certain curves in the plane, not just shapes formed by straight line segments ; the more general fact is often proved by approximating these curves by polygons.

For uncomplicated enough polygons, it's easy to see by eyes, which parts of the place are inside and which are outside. But this is not always easy. For instance is the marked point inside the following polygon ?

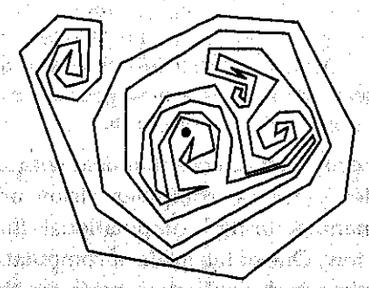


Figure 37.1

Seemingly even more difficult, we'd like to answer questions like this on a computer which doesn't have build into it the powerful visual processing system that we have, and can only deal with this sort of problem by translating it to a collection of numbers. The general problem we'd like to solve is, given a point (x, y) (represented by those two numbers) and a polygon P (represented by its sequence of vertices), is (x, y) in P, on the boundary, or outside ?

This is a commonly occurring problem. For instance if we want to display a polygon on a computer screen, we need to be able to test whether each pixel of the screen corresponds to a point that's inside or outside the polygon, so we can tell what color to draw it.

Fortunately the problem has a simple and elegant answer. Just draw a ray (portion of a line extending infinitely in one direction) down (or in any other direction) from (x, y) :

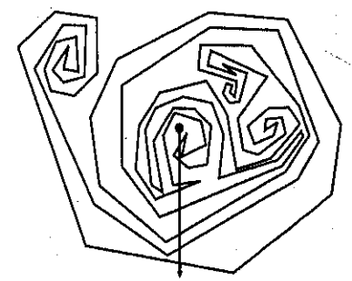


Figure 37.2

Every time the polygon crosses this ray, it separates a part of the ray that's inside the polygon from a part that's outside. Eventually the ray will get far from the polygon, at which point we know that part is outside the polygon. We can then work backwards from there, declaring different parts of the ray to be inside or outside alternately at each crossing. Actually, we don't even need to look at what order these crossings occur in ; all we really need to know is whether there's an even or odd number of them.

In the example shown above, there are eight crossings, so the point is outside the polygon. We can now write a rough outline of pseudo-code for this problem :

```
int crossings = 0
for (each line segment of the polygon)
    if (ray down from (x, y) crosses segment)
        crossings++ ;
if (crossings is odd)
    return (inside) ;
else return (outside) ;
```

odd inside
even outside

37.4 Convex Polygon

A figure is convex if every line segments drawn between any two points inside the figure lies entirely inside the figure. A figure that is not convex is called a concave figure.

Example :

The following figures are convex.

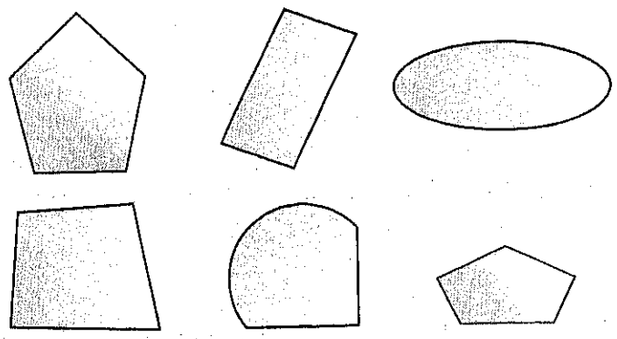


Figure 37.3

The following figures are concave.

Figure 37.4

A **convex polygon** can also be defined formally as having the property that any two points inside the polygon can be connected by a line segment that doesn't cross the polygon. A polygon vertex is *convex* if its interior angle $\leq \pi(180°)$. It is *reflex* if its interior angle $> \pi$. In a convex polygon, all the vertices are convex. In other words, any polygon with a reflex vertex is not convex.

37.5 Convex Hulls

A convex hull is an important structure in geometry that can be used in the construction of many other geometric structures. The convex hull of a set S of points in the plane is defined to be the smallest convex polygon containing all the points of S. The vertices of the convex hull of a set S of points form a (not necessarily proper) subset of S.

There are two variants of the convex hull problem :

1. Obtain the vertices of the convex hull (these vertices are also called extreme points) and
2. Obtain the vertices of the convex hull in same order (clockwise or anticlockwise).

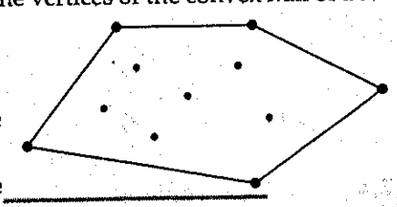


Figure 37.4A Convex hull

Here is a simple algorithm for obtaining the extreme points of a given set S of points in the plane. To check whether a particular point $p \in S$ is extreme, look at each possible triplet of points and see whether p lies in the triangle formed by these three points. If p lies in any such triangle, it is

not extreme, otherwise it is. Testing whether p lies in a given triangle can be done in $\theta(1)$ time. Since there are $\theta(n^3)$ possible triangles, it takes $\theta(n^3)$ time to determine whether a given input is an extreme point or not. Since there are 'n' points, this algorithm runs in a total of $\theta(n^4)$ time.

Using divide and conquer, we can solve both versions of the convex hull problem in $\theta(n \log n)$ time. Given a set $S = \{p_1, \ldots\ldots p_n\}$ ..... .

Given a set $S = \{p_1, p_2, \ldots, p_n\}$ of points in the plane, the convex hull of a set S of points is the smallest convex polygon P for which each point in S is either on the boundary of P or in its interior. We denote the convex hull of S by $CH(S)$. The intersection of convex sets is convex, so the convex hull is a convex set. It is also a polytope, a bounded intersection of half spaces, which in two dimensions means that it is a polygon. Another useful characterization of the convex hull is as a convex closure : take all possible convex combinations of points in P, and the result is the convex hull.

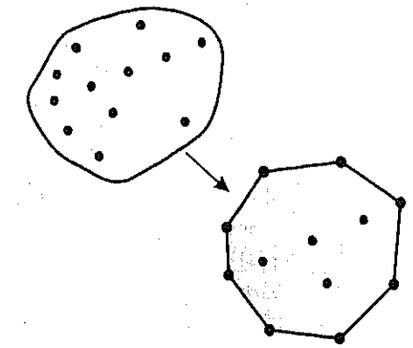


Figure 37.5

Here, we shall present two algorithms that compute the convex hull of a set of n points. Both algorithms output the vertices of the convex hull in counterclockwise order. The first, known as **Graham's scan**, runs in $O(n \lg n)$ time. The second, called **Jarvis's march**, runs in $O(nh)$ time, where h is the number of vertices of the convex hull.

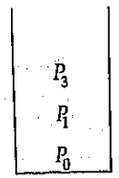
37.6 Graham's Scan Algorithm

In 1972, **R. L. Graham** proposed an efficient algorithm for planar convex hull. Historically, this is the first publication that showed convex hull computation in $O(n \log n)$ time in the worst case. In 1981, **A. C. Yao** proved that it is optimal in the worst-case sense. The problem with Graham's algorithm is that it has no obvious extension to three dimensions. The reason is that the Graham's scan depends on angular sorting, which has no direct counterpart in three dimensions. In this, first we select a base point p_0, normally this is the point with minimum y-coordinate. We select leftmost point in the set in case of the tie. Next, sort the points of Q lexicographically by polar angle, measured in radians. Interior points on the ray cannot be a convex hull points and remove these points during sort. Sort remaining points in counterclockwise order with respect to p_0. Push each point of set Q onto the stack once and the points that are not of $CH(Q)$ are eventually popped from the stack.

Actually, the algorithm works in *three* phases :

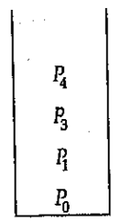
1. Find an extreme point. This point will be the pivot, is guaranteed to be on the hull, and is chosen to be the point with smallest y-coordinate.
2. Sort the points in order of increasing angle about the pivot. We end up with a star-shaped polygon (one in which one special point, in this case the pivot, can "see" the whole polygon).
3. Build the hull, by marching around the star-shaped poly, adding edges when we make a left turn, and backtracking when we make a right turn.

Now $\angle P_0, P_1, P_3$ is left turn, then push P_3 in the stack. Now the elements in the stack are



Stack S

$\angle P_1, P_3, P_4$ is counter-clock wise *i.e.*, left turn. So push p_4 in the stack and the elements in the stack are



Stack S

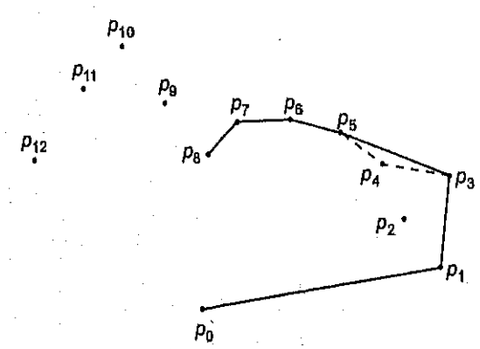
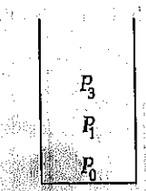


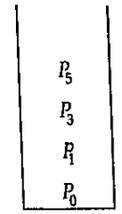
Figure 37.6 (c)

$\angle P_3\ P_4\ P_5$ clockwise turn *i.e.*, non left turn. So pop the stack. Now the elements in the stack are



Stack S

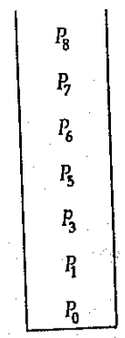
$\angle P_1\ P_3\ P_5$ is left turn. So push P_5 in the stack, now stack S contains



Stack S

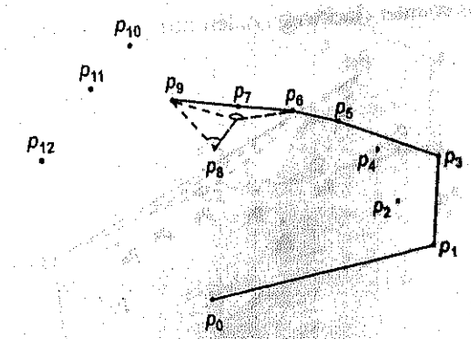
$\angle P_3\ P_5\ P_6$ is also left turn so push P_6 in the stack.

$\angle P_5\ P_6\ P_7$ is also left turn push P_7 in the stack and $\angle P_6\ P_7\ P_8$ is also left turn push P_8 in the stack and now, the stack S contains



Stack S

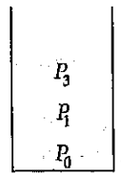
$\angle P_7\ P_8\ P_9$ is clockwise turn *i.e.*, non left turn. So pop the stack then $\angle P_6\ P_7\ P_9$ is also non left turn, so again pop the stack. Now, the stack S contains.



(d)

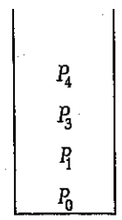
Figure 37.6

Now $\angle P_0, P_1, P_3$ is left turn, then push P_3 in the stack. Now the elements in the stack are



Stack S

$\angle P_1, P_3, P_4$ is counter-clock wise *i.e.*, left turn. So push p_4 in the stack and the elements in the stack are



Stack S

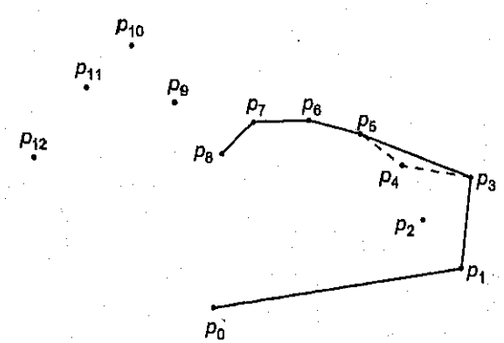
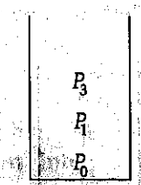


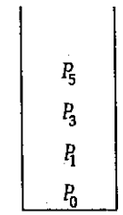
Figure 37.6 (c)

$\angle P_3\ P_4\ P_5$ clockwise turn *i.e.*, non left turn. So pop the stack. Now the elements in the stack are



Stack S

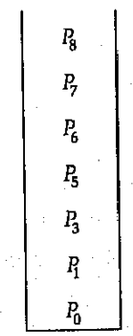
$\angle P_1\ P_3\ P_5$ is left turn. So push P_5 in the stack, now stack S. contains



Stack S

$\angle P_3\ P_5\ P_6$ is also left turn so push P_6 in the stack.

$\angle P_5\ P_6\ P_7$ is also left turn push P_7 in the stack and $\angle P_6\ P_7\ P_8$ is also left turn push P_8 in the stack and now, the stack S contains



Stack S

$\angle P_7\ P_8\ P_9$ is clockwise turn *i.e.*, non left turn. So pop the stack then $\angle P_6\ P_7\ P_9$ is also non left turn, so again pop the stack. Now, the stack S contains.

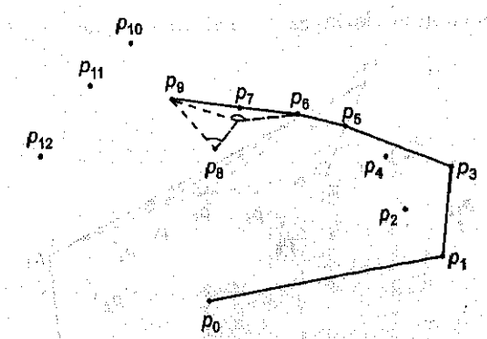
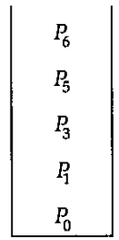
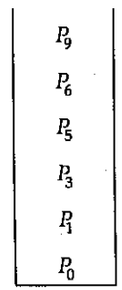


Figure 37.6 (d)



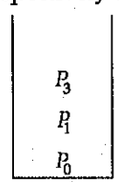
Stack S

$\angle P_5\ P_6\ P_9$ is left turn so push P_9 in the stack and now the elements of stack S are



Stack S

Now $\angle P_6\ P_9\ P_{10}$ is non left turn so pop P_9 and $\angle P_5\ P_6\ P_{10}$ is also non left turn and $\angle P_3\ P_5\ P_{10}$ is non left turn also, thus pop P_6 and P_5 respectively and the stack S contains.



Stack S

Now, $\angle P_1, P_3, P_{10}$ is counter clockwise *i.e.*, left turn. So push P_{10} into the stack

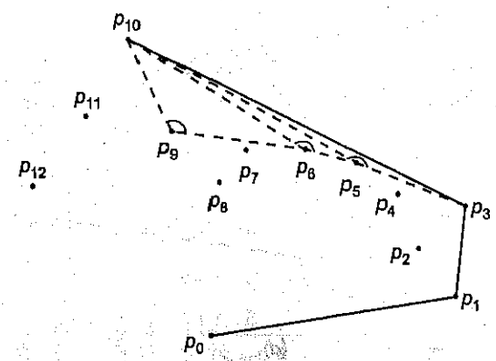
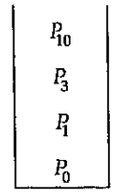


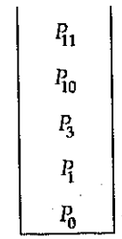
Figure 37.6 (e)

Now the stack contains.



Stack S

$\angle P_3\ P_{10}\ P_{11}$ is counter clock wise that is left turn so push P_{11} into the stack and stack contains.



Stack S

$\angle P_{10}\ P_{11}\ P_{12}$ is clock wise that is not left turn so pop the element P_{11} and $\angle P_3\ P_{10}\ P_{12}$ is anticlockwise.

Hence push this into stack S and the final convex hull returned by the produce is show in the figure.

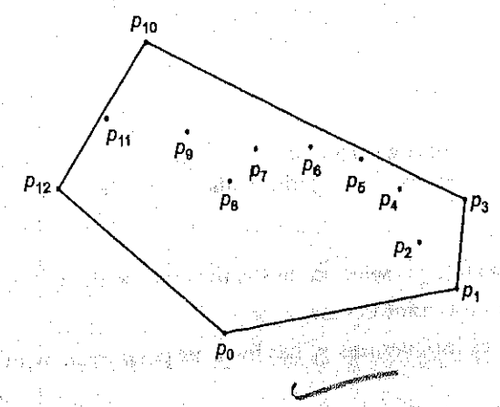


Figure 37.6 (f)

37.7 Jarvis's March Algorithm

Jarvis's march computes the convex hull of a set Q of points by a technique known as package wrapping (or gift wrapping). The algorithm runs in time $O(nh)$, where h is the number of vertices of $CH(Q)$. The algorithm operates by considering any one point that is on the hull, say, the lowest point. We then find the "next" edge on the hull in counterclockwise order. Assuming that $p(k)$ and $p(k-1)$ were the last two points added to the hull, compute the point q that maximizes the angle $[p(k-1)\, p(k)\, q]$. Thus, we can find the point q in $O(n)$ time. After repeating this h times, we will return back to the starting point and we are done. Thus, the overall running time is $O(nh)$. Note that if h is $o(\log n)$ (asymptotically smaller than $\log n$) then this is a better method than Graham's algorithm Jarvis's march is asymptotically faster than Graham's scan.

The basic idea is as follows :

- Locating the point p_0 with minimum y-coordinate (or Start at some extreme point, which is guaranteed to be on the hull).
- Let L be the horizontal line through p_0, L is clearly tangent to $CH(S)$ at p_0. We orient L from left to right. Then we perform "wrapping" step to locate point p_1 that forms the smallest counterclockwise angle with L Point p_1 must also be on $CH(S)$ (See Fig).
- Repeat the wrapping step at p_1 with line pp_1 until we return to the point p_0

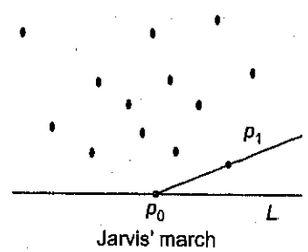


Figure 37.7

Because this process marches around the hull in counter-clockwise order, like a ribbon wrapping itself around the points, this algorithm also called the "gift-wrapping" algorithm.

Algorithm Jarvis March

- First, a base point p_0 is selected, this is the point with the minimum y-coordinate.
  - Select leftmost point in case of tie.
- The next convex hull vertices p_1 has the least polar angle w.r.t. the positive horizontal ray from p_0.
  - Measure in counterclockwise direction.
  - If tie, choose the farthest such point.
- Vertices $p_2, p_3, \ldots, p_k$ are picked similarly until $y_k = y_{max}$

- p_{i+1} has least polar angle w.r.t. positive ray from p_0.
- If tie, choose the farthest such point.

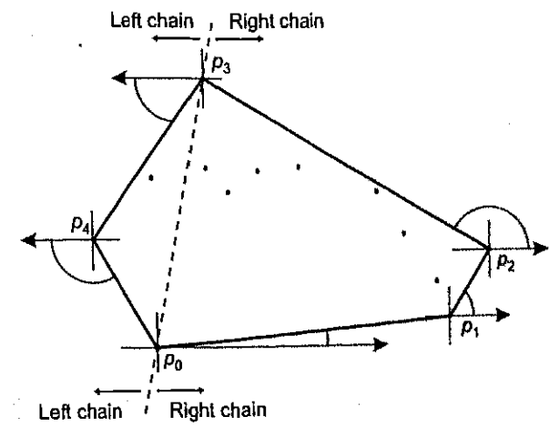


Figure 37.9

- The sequence $p_0, p_1, \ldots, p_k$ is right chain of $CH(Q)$
- To choose the left chain of $CH(Q)$ start with p_k.
  - Choose p_{k+1} as the point with least polar angle w.r.t. the negative ray from p_k.
  - Again measure counterclockwise direction.
  - If tie occurs, choose the farthest such point.
  - Continue picking $p_{k+1}, p_{k+2}, \ldots, p_t$ in same fashion until obtain $P_t = p_0$.

Complexity of Jarvis March

For each vertex p belongs to $CH(Q)$, it takes

- $O(1)$ time to compare polar angles.
- $O(n)$ time to find minimum polar angel.
- $O(n)$ total time.

If $CH(Q)$ has h vertices, then running time $O(nh)$. If $h = o(\lg n)$, then this algorithm is asymptotically faster than the Graham's scan. If points in set Q are generated by random generator, then we expect $h = c \lg n$ for $c \approx 1$.

In practice, Jarvis march is normally faster than Graham's scan on most application. Worst case occurs when $O(n)$ points lie on the convex hull *i.e.*, all points lie on the circle.

Exercise

1. Show how to determine in $O(n^2 \lg n)$ time whether any three points in a set of n points are collinear.
2. Show how to compute the area of an n-vertex simple but not necessarily convex, polygon in $\theta(n)$ time.
3. Write an algorithm to find whether two polygons intersect.
4. Give an algorithm for finding Convex Hull of a star shaped polygon.
5. Give a divide-and conquer method for finding Convex Hull ?
6. Describe the method for finding Convex Hull using Graham's scan and using Jarvis's march.
7. Explain Graham Scan's method ?

CHAPTER

38

NP-Completeness

The field of complexity theory deals with how fast can one solve a certain type of problem. Or more generally how much resource does it take : time, memory-space, number of processors etc. The most common resource is time : number of steps. This is generally computed in terms of n, the length of the input string. We will use an informal model of a computer and an algorithm. All the definitions can be made precise by using a model of a computer such as a **Turing machine**.

While we are interested in the difficulty of a computation, we will focus our hardness results on the difficulty of yes/no questions. These results immediately generalize to questions about general computations. It is also possible to state definitions in terms of languages, where a language is defined as a set of strings : the language associated with a question is the set of all strings representing questions for which the answer is Yes.

38.1 Classes of Problems

We can categorize the problems into the following broad classes :

1. Problems which cannot even be defined formally.
2. Problems which can be formally defined but cannot be solved by computational means.
3. Problems which, though theoretically can be solved by computational means, yet are infeasible, *i.e.*, these problems require so large amount of computational resources that practically is not feasible to solve these problems by computational means. These problems are called **intractable** or **infeasible** problems.

4. Problems that are called feasible or theoretically not difficult to solve by computational means. The distinguishing feature of the problems is that for each instance of any of these problems, there exists a **Deterministic Turing Machine** that solves the problem having time-complexity as a polynomial function of the size of the problem. The class of problem is denoted by *P*.
5. Last, but probably most interesting class includes large number of problems for each of which it is not known whether it is in *P* or not in *P*.

These problems fall somewhere between class (3) and class (4) given above. However, for each of the problems in the class, it is known that it is in *NP*. *i.e.*, each can be solved by at least one-Non-Deterministic Turing Machine, the time complexity of which is a polynomial function of the size of the problem. Now, we can go still further and categorize the problems as :

Decision Problems

Decision problems are the computational problems for which the intended output is either "yes" or "no". In other words, a decision problem is a problem with yes/no answers. Hence in a decision problem, we can equivalently talk of the language associated with the decision problem, namely, the set of inputs for which the answer is yes.

Typically, we assume that the input is coded in binary, so the set of all possible inputs is $\{0,1\}^*$ and the language associated with a decision problem Q is

$$L(Q) = \{x \in \{0,1\}^* \mid \text{the answer is yes for problem } Q \text{ on input } x\}$$

38.1.1 Optimization Problems vs. Decision Problems

Decision Problems have Yes/No answers. *Optimization Problems* require answers that are optimal configurations. Decision problems are "easier" than optimization problems; if we can show that a decision problem is hard that will imply that its corresponding optimization problem is also hard.

38.1.2 The Classes *P* and *NP*

We define *P* (Polynomial) as the class of decision problems that are solvable by algorithms that run in time polynomial in the length of the input. That is, a decision question is in *P* if there exists an exponent k and an algorithm for the question that runs in time $O(n^k)$ where n is the length of the input.

P roughly captures the class of practically solvable problems. Or at least that is the conventional wisdom. Something that runs in time 2^n requires double the time if one adds one character to the input. Something that runs in polynomial time does not endure from this problem.

Class *P* : The set of all polynomially solvable problems.

- *P* is *closed* under addition, multiplication and composition.
- *P* is *independent* of particular formal models of computation or implementations.
- Any problem not in *P* is *hard*.
- A problem in *P* does not necessarily have an efficient algorithm.

Examples :

(a) **True Boolean Formulas.** A Boolean formula consists of variables and negated variables (known collectively as literals), and the operations "and" and "or". We use $\vee$ for "or", $\wedge$ for "and", and $\bar{x}$ for "not x". For example : $x \wedge (x \vee y \vee \bar{z}) \wedge (\bar{x} \vee \bar{y})$

An assignment is a setting of each of the variables to either true or false. For example if x and y are true and z is false, then the above boolean formula is false.

```
TRUEBF
Input : ϕ a boolean formula, asignment Ψ
Question : Does ψ make ϕ TRUE ?
```

Given a boolean function and an assignment of the variables, we can determine whether the formula is true or not in linear time. (A formula can be evaluated like a normal arithmetic expression using a single stack.) So it follows that the TRUEBF problem is in *P*.

(b) **Paths.** Recall that a path is a sequence of edges leading from one node to another.

```
PATH
INPUT : graph G, nodes a and b
Question : Is there a path from a to b in G ?
```

This problem is in *P*. To see if there is a path from node a to node b, one might determine all the nodes reachable from a by doing for instance a breadth-first search or Dijkstra's algorithm. Note that the actual running time depends on the representation of the graph.

We will next define the class NP, or Nondeterministic Polynomial Time. Before we can define this class, we need some definitions.

38.1.3 Verification Algorithm.

A verification algorithm is an algorithm A, that takes two inputs : an ordinary input x (coded in binary), and a **certificate** y, and outputs a 1 on certain combinations of x and y.

Verification algorithm A verifies an input string x if there exists a certificate y such that

$$A(x,y) = 1$$

The language verified by verification algorithm A is

$$L = \{\text{input strings } x \mid \text{there exists certificate string } y \text{ such that } A(x,y) = 1\}$$

Note that for an x that is not in L, for every certificate y, $A(x,y) \neq 1$.

38.1.4 Polynomial-time Verification Algorithm

A verification algorithm A for a language L is a **polynomial-time verification algorithm** for L if

- For each $x \in L$, there is a certificate y of size polynomial in the size of x such that $A(x,y) = 1$, and $A(x,y)$ returns 1 in time polynomial in x.
- Since A is a verification algorithm for L, for every x not in L there is no certificate y for which $A(x,y) = 1$.

38.2 The Class NP (Non Deterministic Polynomial)

The set of all problems that can be solved if we always **guess correctly** what computation path we should follow. Roughly speaking, it includes problems with exponential algorithms but

has not proved that they cannot have polynomial time algorithms. A language $L \in NP$ if and only if there exists a polynomial-time verification algorithm A for L.

Note that $P \subseteq NP$. At this time we do not know if $P = NP$ or $P \subset NP$.

Examples of Languages in NP

Let us consider the following examples of decision problems

- HAM-CYCLE = $\{\langle G \rangle \mid G$ is a Hamiltonian graph$\}$
- CIRCUIT-SAT = $\{\langle C \rangle \mid C$ is a satisfiable boolean circuit$\}$
- SAT = $\{\langle \phi \rangle \mid \phi$ is satisfiable bollean formula$\}$
- CNF – SAT = $\{\langle \phi \rangle\ \phi$ is a satisfiable bollean formula in CNF$\}$
- 3-CNF-SAT = $\{\langle \phi \rangle \mid \phi$ is a satisfiable boolean formula in CNF$\}$
- CLIQUE = $\{\langle G, k \rangle \mid G$ is an undirected graph with a clique of size $k\}$
- IS = $\{\langle G, k \rangle \mid G$ is an undirected graph with an independent set of size $k\}$
- VERTEX-COVER = $\{\langle G, k \rangle \mid$ undirected graph G has a vertex cover of size $k\}$
- TSP = $\{\langle G, c, k \rangle \mid G = (V, E)$ is a complete graph, $c : V \times V \to Z$ is a cost function, $k \in Z$ and G has a traveling salesman tour with cost at most $k\}$
- SUBSET-SUM = $\{\langle S, t \rangle \mid$ there is a subset $S' \subseteq S$ such that $t = \sum_{s \in S'} s\}$

It is quite straightforward to come up with a polynomial-time verification algorithm for each of these problems, *i.e.*, it is quite easy to show that each of these problems is in *NP*.

38.3 *NP*-Completeness

38.3.1 What is Reduction ?

Let L_1 and L_2 be two decision problems. Suppose algorithm A_2 solves L_2. That is, if y is an input for L_2 then algorithm A_2 will answer Yes or No depending upon whether $y \in L_2$ or not.

The idea is to find a transformation f from L_1 to L_2 so that the algorithm A_2 can be part of an algorithm A_1 to solve L_1.

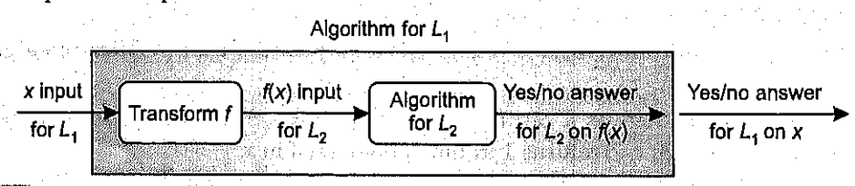


Figure 38.1

38.3.2 Polynomial-time Reductions

Let L_1 and L_2 be languages that are subsets of $\{0,1\}^*$.

We say that L_1 is polynomial-time reducible to L_2 if there exists a function f

$$f : \{0,1\}^* \to \{0,1\}^*$$

with the following properties :

- f transforms an input x for L_1 into an input $f(x)$ for L_2 such that $f(x)$ is a yes-input for L_2 if and only if x is a yes input for L_1. We require a yes-input of L_1 maps to a yes-input of L_2, and a no-input of L_1 maps to a no-input of L_2.

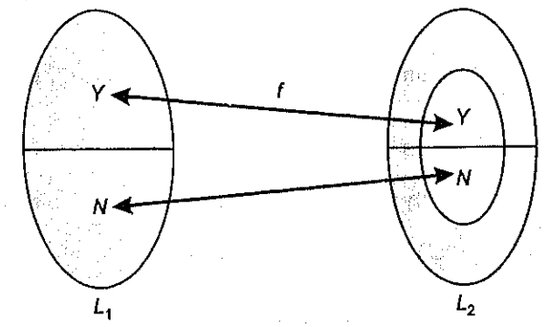


Figure 38.2

- $f(x)$ is computable in polynomial time (in size(x)). If such an f exists, we say that L_1 is **polynomial-time reducible** to L_2, and write $L_1 \le_p L_2$.

Example. *What can we do with a polynomial time reduction $f : L_1 \to L_2$.*

Solution. Given an algorithm A_2 for the decision problem L_2, we can develop an algorithm A_1 to solve L_1.

If A_2 is a polynomial time algorithm for L_2 and $L_1 \le_p L_2$ then we can construct a polynomial time algorithm for L_1.

Theorem

If $L_1 \le_p L_2$ and $L_2 \in P$, then $L_1 \in P$.

Proof. $L_2 \in P$ means that we have a polynomial time algorithm A_2 for L_2. Since $L_1 \le_p L_2$, we have a polynomial-time transformation f mapping input x for L_1 to an input for L_2. Combining these, we get the following polynomial-time algorithm for solving L_1 :

1. take input x for L_1 and compute $f(x)$;
2. run A_2 on input $f(x)$, and return the answer found (for L_2 on $f(x)$) as the answer for L_1 on x.

Each of Steps (1) and (2) takes polynomial time. So the combined algorithm takes polynomial time. Hence $L_1 \in P$.

Note *This does not imply that if $L_1 \le_p L_2$ and $L_1 \in P$, then $L_2 \in P$. This statement is not true.*

38.3.3 NP-Completeness

Polynomial-time reductions provide a formal means for showing that one problem is at least as hard as another, to within polynomial-time factor. That is, if $L_1 \le_p L_2$, then L_1 is not more than a polynomial factor harder than L_2, which is why the "less than or equal to " notation for reduction is mnemonic. We can now define the set of NP-Complete languages, which are the hardest problems in NP.

A language $L \subseteq \{0,1\}^*$ is **NP-complete** if it satisfies the following *two* properties :

1. $L \in NP$; and
2. For every $L' \in NP$, $L' \le_p L$

If a language L satisfies property 2, but not necessarily property 1, we say that L is **NP-hard.**

We use the notation $L \in NPC$ to denote that L is NP-complete.

Theorem

If any NP-complete problem is polynomial time solvable, then P = NP. If any problem in NP is not polynomial time solvable; then all NP complete problems are not polynomial time solvable.

Proof. Suppose that $L \in P$ and also that $L \in NPC$. For any $L' \in NP$, we have $L' \le_p L$ by property 2 of the definition of NP-completeness. We know if $L' \le_p L$ then $L \in P$ implies $L' \in P$, which proves the first statement.

To prove the second statement, suppose that there exists an $L \in NP$ such that $L \notin P$. Let $L' \in NPC$ be any NP-complete language, and for the purpose of contradiction, assume that $L' \in P$. But then we have $L \le_p L'$ and thus $L \in P$.

38.3.4 P = NP ?

One of the most important problems in computer science is whether $P = NP$ or $P \ne NP$? Observe that $P \subseteq NP$. Given a problem $A \in P$, and a certificate, to verify the validity of a yes-input (an instance of A), we can simply solve A in polynomial time (since $A \in P$). It implies $A \in NP$.

Intuitively, $NP \subseteq P$ is doubtful. After all, just able to verify a certificate (corresponds to a yes-input) in polynomial time does not necessary mean we can able to tell whether an input is an yes-input of no-input in polynomial time.

However, 30 years after the $P = NP$? problem was first proposed, we are still no closer to solving it and do not know the answer. The search for a solution, though, has provided us with deep insights into what distinguishes an "easy" problem from a "hard" one.

38.4 The Class co-NP

Note that if $L \in NP$, there is no guarantee that $\overline{L} \in NP$ (since having certificates for yes-inputs, does not mean that we have certificates for the no-inputs).

The class of decision problems L such that $\overline{L} \in NP$ is called co-NP (observe it is does not require $L \in NP$).

Example. COMPOSITE $\in NP$ and so PRIMES = $\overline{\text{COMPOSITE}} \in$ co-NP :

Remark. in contrast, to the fact that $L \in NP$ does not necessarily imply $\overline{L} \in NP$ we do have that

$L \in P$, if and only if $\overline{L} \in P$

This is because a polynomial time algorithm for L is also a polynomial time algorithm for (the NO-answers for L become Yes-answers for $\overline{L}$ and vice-versa).

38.5 Satisfiability (SAT)

A given Boolean formula is **satisfiable** if there is a way to assign truth values (0 or 1) to the variables such that the final result is 1.

Example. $f(x,y,z) = (x \wedge (y \vee \overline{z})) \vee (\overline{y} \wedge z \wedge \overline{x})$

x	y	z	$(x \wedge (y \vee \overline{z}))$	$(\overline{y} \wedge z \wedge \overline{x})$	$f(x,y,z)$
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	1	0	1
1	0	1	0	0	0
1	1	0	1	0	1
1	1	1	1	0	1

The assignment $x = 1$, $y = 1$, $z = 0$ makes $f(x,y,z)$ true, and hence it is satisfiable.

SAT problem. Determine whether an input Boolean formula is satisfiable. If a Boolean formula is satisfiable, it is a yes-input ; otherwise, it is a no-input.

Lemma

SAT $\in NP$.

Proof. The certificate consists of a particular 0 or 1 assignment to the variables. Given this assignment, we can evaluate the formula of length n (counting variables, operations, and parentheses), it requires at most n evaluations, each taking constant time. Hence, to check a certificate takes time $O(n)$. So we have SAT $\in$ NP.

38.6 Circuit Satisfiability

A combinational circuit is called **circuit satisfiable** (CIRCUIT-SAT) if for set of inputs applied, output of this circuit should always be one.

The circuit-satisfiability problem is, **"Given a Boolean combinational circuit composed of AND, OR, and NOT gates, is it satisfiable ?"** We can give a formal language for circuit satisfiability as

CIRCUIT-SAT = $\{\langle C \rangle$: C is a satisfiable Boolean combinational circuit$\}$

Given a circuit C, we might attempt to determine whether it is satisfiable by simple checking all possible assignments to the inputs. If there are k inputs, there are 2^k possible assignments. When the size of C is polynomial in k, each one leads to a super polynomial time algorithm. In fact there is strong indication that no polynomial-time algorithm exists that solves the *circuit-satisfiability problem is NP-Complete (Cook-Levin Theorem).* We break the proof of this into two parts. In the first part we prove this problem belongs to NP class and in second part we show that this problem is NP-hard.

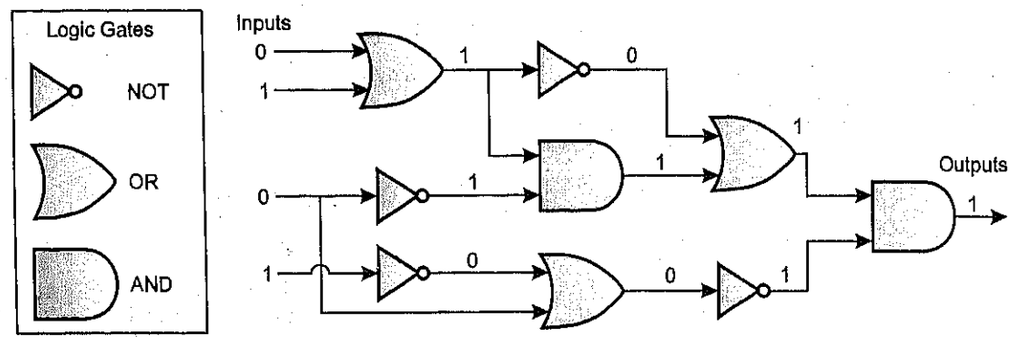


Figure 38.3 An example Boolean circuit.

Lemma

CIRCUIT-SAT is in NP.

Proof. We first use the choose method to "guess" the values of the input nodes as well as the output value of each logic gate. Then, we simply visit each logic gate g in C, *i.e.* each vertex with at least one incoming edge. We then check that the "guessed" value for the output of g is in fact the correct vale for g's Boolean function based on the given values for the inputs of g. This evaluation process can easily be performed in polynomial time. If any check for a gate fails, or if the "guessed" value for the output is 0, then we output "no". If, on the other, the check for every gate succeeds and the output is "1", the algorithm outputs "yes". Thus, if there is indeed a satisfying assignment of input values for C, then there is a possible collection of outcomes to the choose statements so that the algorithm will output "yes" in polynomial time. Likewise, if there is a collection of outcomes to the choose statements so that the algorithm outputs "yes" in polynomial time algorithm, there must be a satisfying assignment of input values for C. Therefore CIRCUIT-SAT is in NP.

Lemma

CIRCUIT-SAT is NP-hard.

Proof. We must show that every language in NP is polynomial-time reducible to CIRCUIT-SAT. Let L be any language in NP. We shall describe a polynomial time algorithm D computing a reduction function f that maps every binary string x to a circuit $C = f(x)$ such that $x \in L$ if and only if $C \in$ CIRCUIT-SAT.

Recall that any deterministic algorithm, such as D, can be implemented on a simple computational model (called RAM) that consists of a CPU and a bank M of addressable memory cells. In our case, the memory M contains the input x, the certificate y, the working storage W, that D needs to perform its computations, and the code for algorithm D itself. The working storage W for D includes all the registers used for temporary calculations and the stack frames for the procedures that D calls during its execution. The topmost such stack frame in W contains the program counter (PC) that identifies where D currently is in its program execution. Thus, there are no memory cells in the CPU itself. In performing each step of D, the CPU reads the next instruction i and performs the calculation indicated by i, and then updates the PC to point to the next instruction to be performed. Thus, the current state of D is completely characterized by the contents of its memory cells. Since D accepts an x in L in a polynomial $p(n)$ number of steps, where n is the size of x, then the entire effective collection of its memory cells can be assumed to consist of just $p(n)$ bits. For in $p(n)$ steps, D can access at most $p(n)$ memory cells. We refer to the $p(n)$ sized collection M of memory cells for an execution of D as the configuration of the algorithm D.

The heart of the reduction of L to CIRCUIT-SAT depends on our constructing a Boolean circuit that simulates the workings of the CPU in our computational model. CPU can be designed as a Boolean circuit consisting of AND, OR, and NOT gates. This circuit can be designed so as to take a configuration of D as input and provide as output the configuration resulting from processing the next computational step. In addition, this simulation circuit, which we will call S, can be constructed so as to consist of at most $cp(n)^2$ AND,OR, and NOT gates, for some constant $c > 0$.

To then simulate the entire $p(n)$ steps of D, we make $p(n)$ copies of S, with the output from one copy serving as input for the next. Part of the input for the first copy of S consists of "hard wired" values for the program for D, the value of x, the initial stack frame and the remaining working storage (initialized to all 0's). The only unspecified true inputs to the first copy of S are cells of D's configuration for the certificate y. These are the true inputs to our circuit. Similarly, we ignore all the outputs from the final copy of S, except the single output that indicates the answer from D, with "1" for "yes" and "0" for "no". The total size of the circuit C is $O(p(n)3)$, which is still polynomial in the size of x.

Consider an input x that D accepts for some certificate y after $p(n)$ steps. Then there is an assignment of values to the input to C corresponding to y, such that, by having C simulate D on this input and the hard-wired values for x, we will ultimately have C output a "1". Thus, C is satisfiable in this case. Conversely, consider a case when C is satisfiable. Then there is a set of inputs, which correspond to the certificate y, such that C outputs a "1". But, since C exactly simulates the algorithm D, this implies that there is an assignment of values to the certificate y, such that D outputs "yes". Thus, D will verify x in this case. Therefore, D accepts x with certificate y if and only if C is satisfiable.

38.7 Proving NP-Completeness

To prove that a problem P is NP-complete, we have following methods:

Method 1 (direct proof) :

(*a*) P is in NP.

(*b*) All problems in **NP-Complete** can be reduced to P.

Example : Conjunctive Normal Form (CNF) Satisfiability problem

Method 2 (equally general but potentially easier) :

(*a*) P is in NP.

(*b*) Find a problem P' that has already been proven to be in NP-Complete.

(*c*) Show that $P' \leq_p P$

Example. The k-clique problem.

(*a*) K-clique is in NP.

(*b*) CNF- Satisfiability is in NP-Complete.

(*c*) CNF- Satisfiability $\leq_p$ k-clique.

Method 3 (restriction; simple but not always available to all problems)

(*a*) P is in NP.

(*b*) Find a special case of P is in NP-Complete.

Example. Sub graph isomorphism : Given two graphs $G(V_1, E_1)$ and $H(V_2, E_2)$, does G contains a sub graph isomorphic to H ? That is, can we find a subset V of V_1 and E of E_1 such that $|V| = |V_2|$ and $|E| = |E_2|$ and there exists a one to one function $f : V_2 \rightarrow V$ such that $\{u, v\}$ in E_2 if and only if $\{f(u), f(v)\}$ is in E ?

(*a*) Sub graph isomorphism is in NP.

(*b*) k-clique is a special case of sub graph isomorphism when H is a clique with k nodes.

Thus, a formal method for proving that a language L is NP-Complete is :

1. Prove $L \in$ NP.
2. Select a known NP-Complete language L'.
3. Describe an algorithm that computes a function f mapping every instance of L' to an instance of L.
4. Prove that the function f satisfies $x \in L'$ if and only if $f(x) \in L$ for all x.
5. Prove that the algorithm computing f runs in polynomial time.

38.8 Techniques for NP-complete Problem

Techniques for dealing with NP-complete problems in practice are :

1. Backtracking 2. Branch and Bound 3. Approximation Algorithms

Examples of **NP-complete** problems :

1. Circuit satisfiability
2. Formula satisfiability
3. 3-CNF satisfiability
4. Clique
5. Vertex cover
6. Subset-sum
7. Hamiltonian cycle
8. Traveling salesman
9. Sub graph isomorphism
10. Integer programming
11. Set-partition
12. Graph coloring

A problem is NP-complete if it is both NP-hard and an element of NP. NP-complete problems are the hardest problems in NP. If anyone finds a polynomial-time algorithm for even one NP-complete problem, then that would imply a polynomial-time algorithm for every NP-complete problem. Literally thousands of problems have been shown to be NP-complete, so a polynomial-time algorithm for one (*i.e.*, all) of them seems incredibly unlikely.

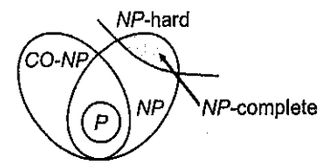


Figure 38.4

To prove that a problem is NP-hard, we use a reduction argument, exactly like we're trying to prove a lower bound.

To prove that problem A is NP-hard, reduce a known NP-hard problem to A.

38.9 (Formula) Satisfiability

An instance of the formula satisfiability problem (SAT) is a Boolean formula ϕ composed of :

1. Boolean variables variables : $x_1, x_2, \ldots$
2. Boolean connectives connectives : any Boolean function with one or two inputs and one output (such as $\wedge$ (AND), $\vee$ (OR), $\neg$ (NOT), $\rightarrow$ (IMPLICATION), $\leftrightarrow$ (if and only if)).
3. Parenthesis

The satisfiability problem asks whether a given Boolean formula is satisfiable.

$$\text{SAT} = \{\langle\phi\rangle : \phi \text{ is a satisfiable boolean formula}\}$$

SAT is NP-complete

SAT $\in$ NP

SAT is **NP-Hard**

SAT belongs to NP, since a certificate consisting of a satisfying assignment for an input formula ϕ can be easily verified in polynomial time. Replace each variable in the formula with its value, and then evaluate the expression.

To prove that SAT is NP-Hard, we show that CIRCUIT-SAT $\leq_p$ SAT

Circuit satisfiability can be reduced in polynomial time to formula satisfiability :

The formula produced by the reduction algorithm has one variable for each wire in the circuit.

Polynomial reduction : for each gate, generate a formula expressing the proper operation of the gate in terms of a formula involving its incident wires ...

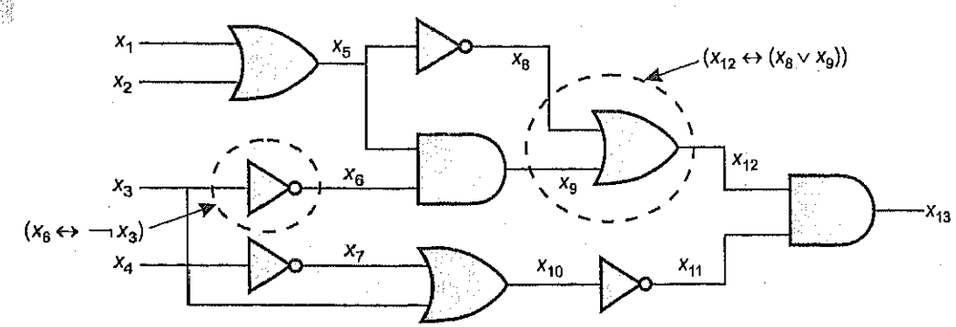


Figure 38.5

The formula produced by the algorithm is the AND of the circuit output with the conjuction of clauses describing the operation of each gate.

By construction, the Boolean combinational circuit is satisfiable if and only if formula ϕ is satisfiable.

$$\phi = x_{13} \wedge (x_5 \leftrightarrow (x_1 \vee x_2)) \wedge (x_6 \leftrightarrow \neg x_3) \wedge (x_7 \leftrightarrow \neg x_4) \wedge (x_8 \leftrightarrow \neg x_5) \wedge (x_9 \leftrightarrow (x_6 \wedge x_5))$$
$$\wedge (x_{10} \leftrightarrow (x_7 \vee x_3)) \wedge (x_{12} \leftrightarrow (x_8 \vee x_9)) \wedge (x_{11} \leftrightarrow \neg x_{10}) \wedge (x_{13} \leftrightarrow (x_{12} \wedge x_{11}))$$

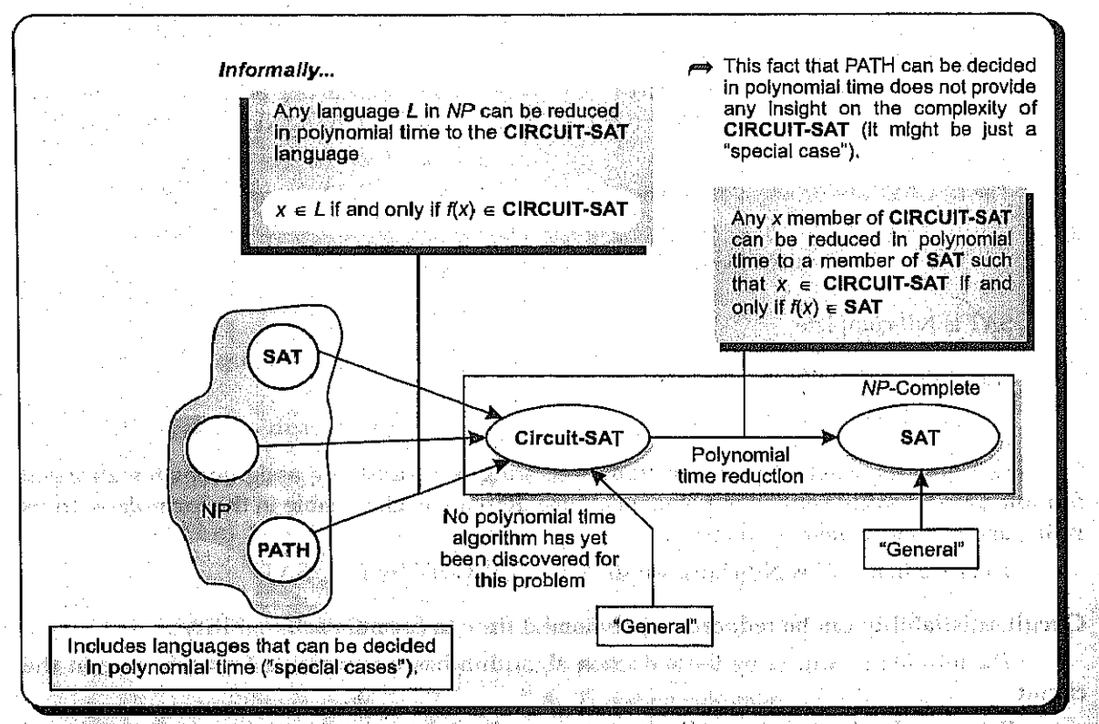


Figure 38.6

38.10 3-CNF Satisfiability

A literal in a Boolean formula is an occurrence of a variable or its negation. A Boolean formula is in **conjunctive normal form** (or CNF) if it is expressed as an AND of clauses, each of which is the OR of one or more literals.

A Boolean formula is in 3-conjunctive normal form (or 3-CNF) if each clause has ecaxtly three distinct literals

The 3-CNF-SAT problem asks whether a given Boolean formula ϕ in 3-CNF is satisfiable.

3-CNF-SAT = {⟨ϕ⟩ : ϕ is a satisfiable 3-CNF Boolean formula}

3-CNF-SAT is NP-complete :

- 3-CNF-SAT $\in$ NP
- 3-CNF-SAT is **NP-Hard** that is $L' \leq_p$ 3-CNF-SAT for every $L' \in$ NP

3-CNF-SAT belongs to NP, since a certificate consisting of a satisfying assignment for an input formula ϕ can be easily verified in polynomial time. Replace each variable in the formula with its value, and then evaluate the expression.

To prove that 3-CNF-SAT is NP-Hard, we show that SAT $\leq_p$ 3-CNF-SAT

Formula satisfiability can be reduced in polynomial time to 3-CNF satisfiability :

- Construct a binary parse tree for the input formula, with literals as leaves and connectives as internal nodes (expression tree) (Associatively can be used to parenthesize the expression fully so that every internal node in the resulting tree has 1 or 2 children).

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

- Introduce a variable y_i variable for the output of each internal node (temporary)

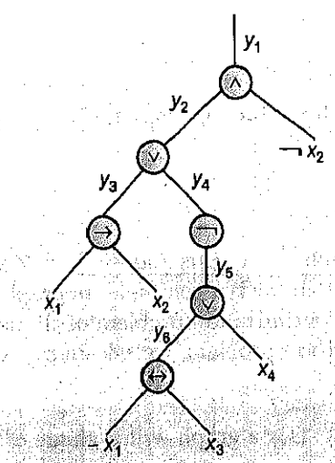


Figure 38.7

We must show that this transformation of ϕ into G is a reduction. First, suppose that ϕ has a satisfying assignment. Then, each clause C_r contains at least one literal l_i^r that is assigned 1. Picking one such literal from each clause yields a set of V' of k vertices. We claim that V' is a clique. For any two vertices $v_i^r, v_j^s \in V'$ where $r \neq s$, both corresponding literals l_i^r and l_j^s are mapped to 1 by the given satisfying assignment and thus the literals cannot be complements. Thus, by the construction of G, the edge (v_i^r, v_j^s) belongs to E.

Conversely, suppose that G has a clique V' of size k. No edges in G connect vertices in the same triple and so v' contains exactly one vertex per triple.

We can assign 1 to each literal l_i^r such that $v_i^r \in V'$ without fear of assigning 1 to both a literal and its complement, since G contains no edges between inconsistent literals. Now, each clause is satisfied and so ϕ is satisfied.

38.12 The Vertex-cover Problem

A vertex-cover of an undirected graph $G=(V, E)$ is a subset $V' \subseteq V$ such that if $(u, v) \in E$ then $u \in V'$ or $v \in V'$ (or both). Vertex-cover for G is a set of vertices that covers all the edges in E. The size of a vertex-cover is the number of vertices in it. In vertex-cover problem we find a vertex-cover of minimum size in a given graph. As a decision problem, we wish to determine whether a graph has a vertex cover of a given size k.

VERTEX-COVER = $\{\langle G, k\rangle$: graph G has vertex cover of size $k\}$

To show that VERTEX-COVER is NP-complete, we first show that VERTEX-COVER $\in$ NP and then we prove that the vertex-cover problem is NP-hard by showing that CLIQUE $\leq_p$ VERTEX-COVER.

38.13 The Subset-Sum Problem

In the subset-sum problem, we are given a finite set $S \subset N$ and a target $t \in N$. We ask whether there is a subset $S' \subseteq S$ whose elements sum to t.

We define the problem as a language

[illegible]

38.14 The Hamiltonian-Cycle Problem

A hamiltonian cycle of an undirected graph $G=(V, E)$ is a simple [illegible] ontai[illegible] vertex in V. A graph that contains a hamiltonian cycle is said to be ham[illegible] nerw[illegible] non-hamiltonian. We can define the hamiltonian-cycle problem as "D[illegible] h G[illegible] hamiltonian cycle ?" in formal language.

HAM-CYCLE = $\{\langle G\rangle$: G is a hamiltonian graph$\}$

38.15 The traveling-Salesman Problem

In the traveling salesman problem, a salesman must visit n cities. We can say that salesman wishes to make a tour or hamiltonian cycle, visiting each city exactly once and finishing at the city he starts from. There is an integer cost $C(i, j)$ to travel from city i to city j and the salesman wishes to make the tour whose total cost is minimum, where the total cost is the sum of individual costs along the edges of the tour.

In formal language,

$$\text{TSP} = \{\langle G, C, k\rangle : G=(V, E) \text{ is a complete graph } C \text{ is the function from } V \times V \to Z \; k \in Z \text{ and } G \text{ has a traveling salesman tour with cost at most } k\}$$

To show that TSP is NP-complete, we first show that TSP $\in$ NP.

Given an instance of the problem, we use as a certificate the sequence of n vertices in the tour. The verification algorithm checks that this sequence contains each vertex exactly once, sums up the edge costs and checks whether the sum is at most k. This process can certainly be done in polynomial time.

To show NP-hard, we show that HAM-CYCLE $\leq_p$ TSP. Let $G=(V, E)$ be an instance of HAM-CYCLE. We construct an instance of TSP as follows. We form the complete graph $G' = (V, E')$ where $E' = \{(i, j) : i, j \in V\}$ and we define the cost function c by

$$c(i, j) = \begin{cases} 0 & \text{if } (i, j) \in E \\ 1 & \text{if } (i, j) \notin E \end{cases}$$

The instance of TSP is then $(G', C, 0)$ which is easily formed in polynomial time.

We now show that graph G has a hamiltonian cycle if and only if graph G' has a tour of cost at most 0. Suppose that graph G has a hamiltonian cycle h. Each edge in h belongs to E and thus has cost 0 in G'.

Thus h is a tour in G' with cost 0. Conversely, suppose that graph G' has a tour h' of cost at most 0. Since the costs of the edges in E' are 0 and 1 the cost of tour h' is exactly 0. Therefore, h' contains only edges in E. We conclude that h is a hamiltonian cycle in graph G.

Exercise

1. Give a formal definition for the problem of finding the longest simple cycle in an undirected graph. Give a related decision problem. Give the language corresponding to the decision problem.
2. Prove that if G is an undirected bipartite graph with an odd number of vertices, then G is non-hamiltonian.
3. Prove that the class NP of languages is closed under UNION, INTERSECTION, CONCATENATION, and KLEENE STAR. Discuss the closure of NP under complement.
4. Prove that if NP $\neq$ Co-NP, then $P \neq$ NP.

We must show that this transformation of ϕ into G is a reduction. First, suppose that ϕ has a satisfying assignment. Then, each clause C_r contains at least one literal l_i^r that is assigned 1. Picking one such literal from each clause yields a set of V' of k vertices. We claim that V' is a clique. For any two vertices $v_i^r, v_j^s \in V'$ where $r \neq s$, both corresponding literals l_i^r and l_j^s are mapped to 1 by the given satisfying assignment and thus the literals cannot be complements. Thus, by the construction of G, the edge (v_i^r, v_j^s) belongs to E.

Conversely, suppose that G has a clique V' of size k. No edges in G connect vertices in the same triple and so v' contains exactly one vertex per triple.

We can assign 1 to each literal l_i^r such that $v_i^r \in V'$ without fear of assigning 1 to both a literal and its complement, since G contains no edges between inconsistent literals. Now, each clause is satisfied and so ϕ is satisfied.

38.12 The Vertex-cover Problem

A vertex-cover of an undirected graph $G=(V, E)$ is a subset $V' \subseteq V$ such that if $(u,v) \in E$ then $u \in V'$ or $v \in V'$ (or both). Vertex-cover for G is a set of vertices that covers all the edges in E. The size of a vertex-cover is the number of vertices in it. In vertex-cover problem we find a vertex-cover of minimum size in a given graph. As a decision problem, we wish to determine whether a graph has a vertex cover of a given size k.

VERTEX-COVER = $\{\langle G,k\rangle$: graph G has vertex cover of size $k\}$

To show that VERTEX-COVER is NP-complete, we first show that VERTEX-COVER $\in$ NP and then we prove that the vertex-cover problem is NP-hard by showing that CLIQUE $\leq_p$ VERTEX-COVER.

38.13 The Subset-Sum Problem

In the subset-sum problem, we are given a finite set $S \subset N$ and a target $t \in N$. We ask whether there is a subset $S' \subseteq S$ whose elements sum to t.

We define the problem as a language

SUBSET-SUM = $\{\langle S,t\rangle$: there exists a subset $S' \subseteq S$ such that $t = \sum_{s \in S'} s\}$

38.14 The Hamiltonian-Cycle Problem

A hamiltonian cycle of an undirected graph $G=(V, E)$ is a simple cycle that contains each vertex in V. A graph that contains a hamiltonian cycle is said to be hamiltonian otherwise, it is non-hamiltonian. We can define the hamiltonian-cycle problem as "Does a graph G have a hamiltonian cycle ?" in formal language.

HAM-CYCLE = $\{\langle G\rangle$: G is a hamiltonian graph$\}$

38.15 The traveling-Salesman Problem

In the traveling salesman problem, a salesman must visit n cities. We can say that salesman wishes to make a tour or hamiltonian cycle, visiting each city exactly once and finishing at the city he starts from. There is an integer cost $C(i,j)$ to travel from city i to city j and the salesman wishes to make the tour whose total cost is minimum, where the total cost is the sum of individual costs along the edges of the tour.

In formal language,

$$\text{TSP} = \{\langle G,C,k\rangle : G=(V,E) \text{ is a complete graph } C \text{ is the function from } V \times V \to Z,\ k \in Z \text{ and } G \text{ has a traveling salesman tour with cost at most } k\}$$

To show that TSP is NP-complete, we first show that TSP $\in$ NP.

Given an instance of the problem, we use as a certificate the sequence of n vertices in the tour. The verification algorithm checks that this sequence contains each vertex exactly once, sums up the edge costs and checks whether the sum is at most k. This process can certainly be done in polynomial time.

To show NP-hard, we show that HAM-CYCLE $\leq_p$ TSP. Let $G=(V, E)$ be an instance of HAM-CYCLE. We construct an instance of TSP as follows. We form the complete graph $G'=(V, E')$ where $E' = \{(i,j) : i,j \in V\}$ and we define the cost function c by

$$c(i,j) = \begin{cases} 0 & \text{if } (i,j) \in E \\ 1 & \text{if } (i,j) \notin E \end{cases}$$

The instance of TSP is then $(G', C, 0)$ which is easily formed in polynomial time.

We now show that graph G has a hamiltonian cycle if and only if graph G' has a tour of cost at most 0. Suppose that graph G has a hamiltonian cycle h. Each edge in h belongs to E and thus has cost 0 in G'.

Thus h is a tour in G' with cost 0. Conversely, suppose that graph G' has a tour h' of cost at most 0. Since the costs of the edges in E' are 0 and 1 the cost of tour h' is exactly 0. Therefore, h' contains only edges in E. We conclude that h is a hamiltonian cycle in graph G.

Exercise

1. Give a formal definition for the problem of finding the longest simple cycle in an undirected graph. Give a related decision problem. Give the language corresponding to the decision problem.
2. Prove that if G is an undirected bipartite graph with an odd number of vertices, then G is non-hamiltonian.
3. Prove that the class NP of languages is closed under UNION, INTERSECTION, CONCATENATION, and KLEENE STAR. Discuss the closure of NP under complement.
4. Prove that if NP $\neq$ Co-NP, then $P \neq$ NP.

5. Prove that $P \subseteq$ Co-NP.
6. Show that the hamiltonian-path problem can be solved in polynomial time on directed acyclic graphs. Give an efficient algorithm for the problem.
7. Show that ciruit satisfiability problem is NP-complete.
8. Show that the $\leq_p$ relation is a transitive relation on languages. That is, show that if $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$ then $L_1 \leq_p L_3$.
9. Show that the hamiltonian-path problem is NP-complete.
10. Write short notes on the following :
    (*i*) NP-hard problem
    (*ii*) NP-complete problem.
11. Prove that satisfiability of boolean formula in 3-conjunctive normal form (3-CNF) is NP-complete.
12. What do you mean by formula. Satisfiability. Check this for the formula $\phi = ((x_1 \rightarrow x_2)v)$

CHAPTER

39

Approximate Algorithms

39.1 Introduction

Many important problems are NP-Complete (NPC), which are likely to be quite hard to solve exactly. We cannot just forget those problems, as they are so important. There are several things we can do :

- Try *exponential time algorithm*. An optimal solution is found. Not feasible if problem size is large.
- Try *general optimization methods, e.g.* branch and-bound, genetic algorithms, neural nets. Some is hard to show how good they are compared with the optimal solution.
- Try *approximate algorithms*. Generally fast, but may not get an optimal solution. But they can be proved to be close to the optimal solutions.

So, among many approaches, one approach to solving NP-Complete optimization problems is the use of fast (*i.e.* polynomial bounded) algorithms that are not guaranteed to give best solution but will give one that is close to the optimal. Such algorithms are called *approximation algorithms* or *heuristic algorithms*.

The use of heuristics in an existing algorithm may enable it to quickly solve a large instance of a problem provided the heuristic works on that instance. A heuristic, however, does not work

equally effectively on all problem instances. Exponential time algorithms, even coupled with heuristics, still show exponential behaviour on some set of inputs.

In approximation algorithm, we remove the requirement that the algorithm that solves the optimization problem P must always generate an optimal solution. This requirement is replaced by the requirement that the algorithm for P must always generate a feasible solution with value close to the value of an optimal solution. A feasible solution with value close to the value of an optimal solution is called an **approximate solution**. An approximation algorithm for P is an algorithm that generates approximate solutions for P.

In many applications an approximate solution is good enough, especially when the time required finding an optimal solution is considered. In the case of *NP*-hard problems, approximate solutions have added importance as exact solutions (*i.e.*, optimal solutions) may not be obtainable in a feasible amount of computing time. An approximate solution may be all one can get using a reasonable amount of computing time.

The goal of an approximation is to come as close to the optimum value as possible in reasonable amount of time *i.e.*

- For the traveling salesperson problem, the optimization problem is to find the shortest cycle, and the approximation problem is to find the short cycle.
- For the vertex cover problem, the optimization problem is to find the vertex cover with fewest vertices and the approximation problem is to find the vertex cover with few vertices.

39.2 Performance Ratios

Suppose we work on an optimization problem where each solution carries a cost. An approximate algorithm returns a legal solution, but the cost of that legal solution may not be optimal.

For example, suppose we are looking for a **minimum size vertex-cover** (VC). An approximate algorithm returns a VC for us, but the size (cost) may not be minimum.

Another example is we are looking for a **maximum size independent set** (IS). An approximate algorithm returns an IS for us, but the size (cost) may not be Maximum. Let C be the cost of the solution returned by an approximate algorithm, and C^* is the cost of the optimal solution.

We say the approximate algorithm has an *approximation ratio* $P(n)$ for an input size n, where

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \le P(n)$$

Intuitively, the *approximation ratio* measures how bad the approximate solution is compared with the optimal solution. A large (small) *approximation ratio* means the solution is much worse than (more or less the same as) an optimal solution.

Observe that $P(n)$ is always ≥ 1; if the ratio does not depend on n, we may just write P. Therefore, a 1-approximation algorithm gives an optimal solution. Some problems have polynomial-time approximation algorithms with small constant approximate ratios, while others have best-known polynomial-time approximation algorithms whose approximate ratios grow with n.

39.3 Examples of Approximate Algorithms

Some examples of approximate algorithms are as follows :

39.3.1 Vertex-cover

Vertex Cover. A vertex cover of a graph G is a set of vertices such that every edge in G is incident to at least one of these vertices.

The decision vertex-cover problem was proven NPC. Now, we want to solve the optimal version of vertex cover problem, *i.e.*, we want to find a minimum size vertex cover of a given graph. We call such vertex cover an optimal vertex cover C^*.

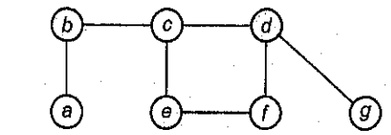


Figure 39.1

An approximate algorithm for Vertex Cover :

```
Approx-Vertex-Cover (G = (V, E) )
 {
     C = empty-set ;
     E' = E ;
     while E' is not empty do
     {
        let (u,v) be any edge in E' ; (*)
        add u and v to C ;
        remove from E' all edges incident to
        u or v ;
     }
     return C ;
 }
```

The idea is to take an edge (u, v) one by one, put both vertices to C, and remove all the edges incident to u or v. We carry on until all edges have been removed. Obviously, C is a *VC*. But how good is C ?

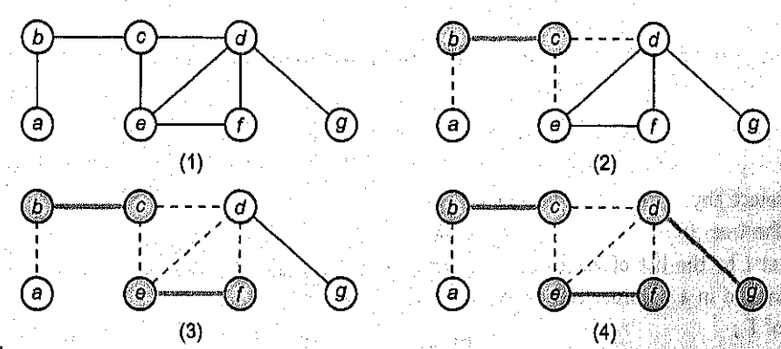


Figure 39.2

$VC = \{b, c, d, e, f, g\}$

Theorem

Approx-Vertex-Cover is a 2-approximation algorithm, i.e.,

$$\frac{|C|}{|C^*|} \le 2$$

It means the number of vertices in C returned by Approx-Vertex-Cover guarantees to be at most twice of the optimal value.

Proof.

1. Let A be the edge set selected by line (*). Observe that $|C| = 2|A|$.
2. Observe that the edge in A does not have any common vertex between them. It means for $e = (x, y) \in A$, either x or y must be selected to the optimal vertex cover C^*. It follows $|C^*| \ge |A|$.
3. Now we have

$$\frac{|C|}{2} = |A| \le |C^*| \quad \Rightarrow \quad \frac{|C|}{|C^*|} \le 2.$$

39.3.2 Traveling-salesman problem

Imagine we are a salesman, and we need to visit n cities. We want to start a *tour* at a city and visit every city *exactly one time*, and finish the tour at the city from where we start. There is a non-negative cost $c(i, j)$ to travel from city i to city j. The goal is to find a tour (which is a Hamiltonian cycle) of minimum cost. We assume every two cities are connected. Such problem is called *Traveling-salesman problem* (TSP).

We can model the cities as a complete graph of n vertices, where each vertex represents a city.

It can be shown that TSP is NPC.

If we assume the cost function c satisfies the *triangle inequality*, then we can use the following approximate algorithm.

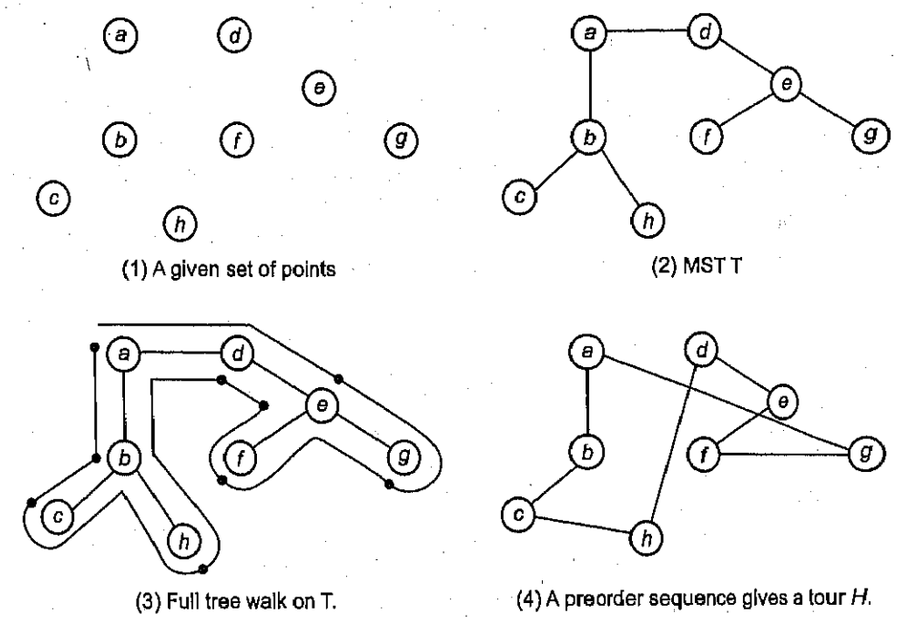
Triangle inequality. Let u, v, w be any three vertices, we have

$$c(u, w) \le c(u, v) + c(v, w)$$

One important observation to develop an approximate solution is if we remove an edge from H^*, the tour becomes a *spanning tree*.

```
Approx-TSP (G = (V, E)) {
1. compute a MST T of G ;
2. select any vertex r be the root of
   the tree ;
3. let L be the list of vertices
   visited in a preorder tree walk
   of T ;
4. return the hamiltonian cycle H that
   visits the vertices in the order L ;
}
```


Traveling-salesman Problem



(1) A given set of points

(2) MST T

(3) Full tree walk on T.

(4) A preorder sequence gives a tour H.

Figure 39.3

Intuitively, Approx-TSP first makes a full walk of MST T, which visits every edge exactly two times. To create a Hamiltonian cycle from the full walk, it bypasses some vertices (which corresponds to making a shortcut).

Theorem

Approx-TSP is a 2-approximation algorithm i.e.,

$$\frac{c(H)}{c(H^*)} \le 2$$

Proof.

1. Observe that if we remove any edge from H^*, then it becomes to a spanning tree, hence we have $c(T) \le c(H^*)$:
2. The cost of a full walk on T is $2c(T)$; since H makes a short-cut on the full walk, by *triangle inequality*, we have $c(H) \le 2c(T)$;
3. Combining, we have

$$\frac{c(H)}{2} \le c(T) \le c(H^*) \quad \Rightarrow \quad \frac{c(H)}{c(H^*)} \le 2$$

Exercise

1. Suppose that a complete undirected graph $G = (V, E)$ with at least 3 vertices has a cost function c that satisfies the triangle inequality. Prove that $c(u,v) \geq 0$ for all $u,v \in V$.
2. Show that the decision version of the set-covering problem is NP-complete by reduction from the vertex-cover problem.
3. Write the short note on approximation algorithm.
4. Explain approximation algorithm with an appropriate example.

CHAPTER

40

Randomized Algorithm

40.1 Introduction

Historically, the study of randomized algorithms was spurred by the discovery by **Miller** and **Robin** in 1976 that the problem of determining the primality of a number can be solved efficiently by a randomized algorithm. At that time, no practical deterministic algorithm for primality was known. Given an integer 'n', the problem of deciding whether 'n' is a prime is known as primality testing.

A randomized algorithm is defined as an algorithm that is allowed to access a source of independent, unbiased bits, and it is then allowed to use these random bits to influence its computation. An algorithm is randomized if its output is determined by the input as well as the values produced by a random-number generator.

A randomized algorithm is one that makes use of a randomizer such as a random number generator. Some of the decisions made in the algorithm depend on the output of the randomizer. Since the output of any randomizer might differ in an unpredictable way from run to run, the output of a randomized algorithm could also differ from run to run for the same input. So, the execution time of a randomized algorithm could also vary from run to run for the same input.

In common practice, this means that the machine implementing the algorithm has access to a pseudo-random number generator. The algorithm typically uses the random bits as an auxiliary input to guide its behaviour, in the hope of achieving good performance in the "average case". Formally, the algorithm's performance will be a random variable determined by the random bits, with (hopefully) good expected value ; this expected value is called the *expected runtime*. The "worst case" is typically so unlikely to occur that it can be ignored.

Randomized algorithms are particularly useful when faced with a malicious "adversary" or attacker who deliberately tries to feed a bad input to the algorithm. It is for this reason that randomness is ubiquitous in cryptography. In cryptographic applications, pseudo-random numbers cannot be used, since the adversary can predict them, making the algorithm effectively deterministic. Therefore either a source of truly random numbers or a cryptographically secure pseudo-rando number generator is required. Another area in which randomness is inherent is quantum computing.

Randomized algorithms can be categorized into *two* classes :

1. Las Vegas algorithms
2. Monte Carlo algorithms

Las Vegas algorithm always produces the same (correct) output for the same input. The execution time of a Las Vegas algorithm depends on the output of the randomizer. If we are lucky, the algorithm might terminate fast, and if not, it might run for a longer period of time. In general, its runtime for each input is a random variable whose expectation is bounded.

The second is algorithms whose output might differ from run to run for the same input. These are called **Monte Carlo algorithms.** Consider any problem for which there are only two possible answers, say yes and no. If a Monte Carlo algorithm is used to solve such a problem, then the algorithm might give incorrect answers depending on the output of the randomizer. We require that the probability of an incorrect answer from a Monte Carlo algorithm be low.

Typically, for a fixed input, a Monte Carlo algorithm does not display much variation in execution time between runs, whereas in the case of a Las Vegas algorithm this variation is significant.

40.2 Applications

40.2.1 Randomized Quick Sort Algorithm

- In randomized quick sort, we will pick randomly an element as the pivot for partitioning.
- The expected runtime of any input is $O(n \log n)$.

Analysis of Randomized Quick Sort

- Let $s(i)$ be the ith smallest in the input list S.
- X_{ij} is a random variable such that $X_{ij} = 1$ if $s(i)$ is compared with $s(j)$; $X_{ij} = 0$ otherwise.

- Expected runtime t of randomized QS is :

$$t = E\left[\sum_{i=1}^{n} \sum_{j \geq 1} X_{ij}\right] = \sum_{i=1}^{n} \sum_{j \geq 1} E[X_{ij}]$$

- $E[X_{ij}]$ is the expected value of X_{ij} over the set of all random choices of the pivots, which is equal to the probability p_{ij} that $s(i)$ will be compared with $s(j)$.

The randomized quick sort algorithm uses $i \leftarrow$ Random (p, r) to be the new pivot element.

Randomized-Partition (A, p, r)

```
i ← Random (p, r)
exchange A[r] ↔ A[i]
return Partition (A, p, r)
```

Randomized-Quicksort (A, p, r)

```
If p < r
    then q ← Randomized-Partition (A, p, r)
         Randomized-Quicksort (A, p, q − 1)
         Randomized-Quicksort (A, q + 1, r)
```

40.2.2 Randomized Minimum-cut Algorithm

A more complex example, representative of the use of randomized algorithms to solve graph theoretic problems, is the following **randomized minimum cut algorithm :**

```
Find-min-cut(undirected graph G)
{
  while there are more than 2 nodes in G do
  {
      pick an edge (u,v) at random in G
      contract the edge, while preserving multi-edges
      remove all loops
  }
  output the remaining edges
}
```

Here, contracting an edge (u, v) means adding a new vertex w, replacing any edge (u, x) or (v, x) with (w, x), and then deleting u and v from G.

- Let k be the min-cut of the given $G(E, V)$ where $|V| = n$.
- Then $|E| \geq kn/2$.

- The probability q_1 of picking one of those k edges in the first merging step $\leq 2/n$.
- The probability p_1 of not picking any of those k edges in the first merging step $\geq (1-(2/n))$
- Repeat the same argument for the first $n-2$ merging steps.
- Probability p of not picking any of those k edges in all the merging steps $\geq (1-2/n)(1-2/(n-1)(1-2/(n-2))\ldots(1-2/3)$.
- Therefore, the probability of finding the min-cut :

$$p \geq \left(1-\frac{2}{n}\right)\left(1-\frac{2}{n-1}\right)\ldots\left(1-\frac{2}{3}\right)$$

$$= \frac{(n-2)(n-3)\ldots(1)}{n(n-1)\ldots 3}$$

$$= \frac{2}{n(n-1)}$$

- If we repeat the whole procedure $n^2/2$ times, the probability of not finding the min-cut is at most

$$(1-2/n^2)^{n^2/2} \cong 1/e$$

Randomized Min-cut is a **Monte Carlo Algorithm.**

40.2.3 There are some interesting complexity classes involving randomized algorithms

- Randomized Polynomial time (*RP*)
- Zero-error Probabilistic Polynomial time (*ZPP*)
- Probabilistic Polynomial time (*PP*)
- Bounded-error Probabilistic Polynomial time (*BPP*)

1. *RP Definition.* The class *RP* consists of all languages L that have a randomized algorithm A running in worst-case polynomial time such that for any input x in Σ^*.

$$x \in L \quad \Rightarrow \quad \Pr[A(x) \text{ accepts}] \geq \frac{1}{2} \qquad \text{Pr means probability.}$$

$$x \notin L \quad \Rightarrow \quad \Pr[A(x) \text{ accepts}] = 0$$

Independent repetitions of the algorithms can be used to reduce the probability of error to exponentially small.

Notice that the success probability can be changed to an inverse polynomial function of the input size without affecting the definition of *RP*.

2. *ZPP Definition.* The class ZPP is the class of languages, which have Las Vegas algorithms running in expected polynomial time.

$$\text{ZPP} = \text{RP} \cap \text{co-RP}$$

(Note that a language L is in co-X where X is a comlexity class if and only if it's complement $\Sigma^* - L$ is in X.)

3. PP Definition. The class *PP* consists of all languages L that have a randomized algorithm A running in worst-case polynomial time such that for any input x in Σ^*.

$$x \in L \quad \Rightarrow \quad \Pr[A(x) \text{ accepts}] \geq \frac{1}{2}$$

$$x \notin L \quad \Rightarrow \quad \Pr[A(x) \text{ accepts}] < \frac{1}{2}$$

To reduce the error probability, we can repeat the algorithm several times on the same input and produce the output which occurs in the majority of those trials.

However, the definition of *PP* is quite weak since we have no bound on how far from 1/2 the probabilities are. It may not be possible to use a small number (*e.g.*, polynomial number) of repetitions to obtain a significantly small error proability.

4. *BPP Definition.* The class BPP consists of all languages L that have a randomized algorithm A running in worst-case polynomial time such that for any input x in Σ^*.

$$x \in L \quad \Rightarrow \quad \Pr[A(x) \text{ accepts}] \geq \frac{3}{4}$$

$$x \notin L \quad \Rightarrow \quad \Pr[A(x) \text{ accepts}] \leq \frac{1}{4}$$

For this class of algorithms, the error probability can be reduced to $1/2n$ with only a polynomial number of iterations.

In fact, the probability bounds 3/4 and 1/4 can be changed to $\frac{1}{2}+\frac{1}{p(n)}$ and $\frac{1}{2}-\frac{1}{p(n)}$ respectively where $p(n)$ is a polynomial function of the input size n without affecting the definition of BPP.

40.3 Advantages and Disadvantages

Two of the most important advantages of using randomized algorithms are their **simplicity** and **efficiency.** A majority of the randomized algorithms found in the literature are simpler than the best deterministic algorithms for the same problems. Randomized algorithms have also been shown to yield better complexity bounds. Not only do randomized algorithms yield superior asymptotic run-time bounds, but they also have demonstrated to be competitive in practice.

The randomized algorithm performs badly only if the random-number generator produces an unlucky permutation to be sorted.

A randomized strategy is typically useful when there are many ways in which an algorithm can proceed but it is difficult to determine a way that guaranteed to be good. If many of the alternatives are good, simply choosing one randomly can yield a good strategy.

Often, an algorithm must make many choices during its execution. If the benefits of good choice outweigh the costs of bad choices, a random selection of good and bad choices can yield an efficient algorithm.

Main Disadvantages of Randomized Algorithm

1. Computer architecture tries to predict the future behaviour of algorithms to optimize its execution.
2. Randomization is useful only for very hard problems.

Randomized algorithms can be used to solve a wide variety of real-world problems like approximation algorithms, they can be used to more quickly solve tough *NP*-complete problem. An advantage over the approximation algorithms, however, is that a randomized algorithm will eventually yield an exact answer if executed enough times.

BIBLIOGRAPHY

Cormen, Thomas H. *"Introduction to algorithms"*, Cambridge, Mass : MIT Press ; New York : McGraw-Hill, c1990

Aho, Alfred V., *"Data structures and algorithms"*, Reading, Mass : Addison-Wesley, c1983.

Aho, Alfred V., *"The design and analysis of computer algorithm"*. edited by Attallah, Mikhail J. *"Algorithms and theory of computation handbook"*, CRC Press, 1998 (QA76.9.A43 1999).

Baase, Sara., *"Computer algorithms : introduction to design and analysis"*, Reading, Mass : Addison-Wesley Pub. Co., c1978.

Even, Shimon., *"Graph Algorithms"*,Computer Science Press, 1979.

Garey, Michael R.,*"Computers and intractability : a guide to the theory of NP"*, San Francisco : W. H. Freeman, c1979.

Gibbons, Alan M., *"Efficient Parallel Algorithms"*, Cambridge University Press, 1988.

Greene, Daniel H., *"Mathematics for the analysis of algorithms"*, Boston : Birkhauser, c1981.

Goodrich T. Michael, *"Algorithm Design - Foundations, Analysis & Internet Examples "*

Gonnet, Gaston H., *"Handbook of algorithms and data structures : in Pascal and C"*, Wokingham, England ; Reading, Mass : Addison-Wesley Pub. Co.,

Gonnet, Gaston H. and Baeza-Yates, Ricardo editors, *"Handbook of Algorithms and Data Structures in Pascal and C"*, Addison-Wesley, 1991.

Harel, David., *"Algorithmics : the spirit of computing"*, second edition Addison-Wesley, 1992.

Horowitz, Ellis., *"Fundamentals of computer algorithms"*, Potomac, Md : Computer Science Press, c1978.

Smith, Jeffrey D., *"Design and Analysis of Algorithms"*, PWS-Kent, 1989.

Kingston, Jeffrey Howard., *"Algorithms and data structures : design, correctness, analysis"*, Sydney : Addison-Wesley, 1990.

Koren, Israel., *"Computer Arithmetic Algorithms"*, Prentice-Hall, 1993.

Kozen, Dexter C. *"The design and analysis of algorithms"*, 1992.(QA76.9.A43 K69 1992).

Kreher, Donald L. and Stinson, Douglas Combinatorial Algorithms : Generation, Enumeration and Search, (CRC Press, 1998)

Kruse, Robert L., *"Data Structures and Program Design"*, second edition Prentice-Hall, 1996.