

**BASICS OF IT**  
**(PGDIT01)**  
**(PG - DIPLOMA)**



**ACHARYA NAGARJUNA UNIVERSITY**

**CENTRE FOR DISTANCE EDUCATION**

**NAGARJUNA NAGAR,**

**GUNTUR**

**ANDHRA PRADESH**

## C O N T E N T S

### HISTORY

<b>Chapter –1 INTRODUCTION .....</b>	<b>1.1 - 1.24</b>
1.1 Algorithm definition .....	1.1
1.2 Characteristics of an Algorithm .....	1.1
1.3 How to develop an algorithm .....	1.2
1.4 How to analyze an algorithm .....	1.3
1.4.1 General plan for analyzing efficiency of recursive algorithms .....	1.4
1.5 How to validate of algorithm .....	1.4
1.6 Time & Space Complexities .....	1.4
1.6.1 Space complexity .....	1.4
1.6.2 Time complexity .....	1.4
1.7 Order of Magnitude .....	1.5
1.8 Asymptotic Notations.....	1.8
1.8.1 Optimality .....	1.11
1.8.2 Reduction .....	1.12
1.8.3 Profiling .....	1.12
1.9 Amortized Analysis .....	1.13
1.9.1 Aggregate Analysis .....	1.13
1.9.2 Accounting Method .....	1.14
1.9.3 Potential Method .....	1.15
1.10 Worst-Case Average-Case and Randomized Algorithms .....	1.19
1.10.1 The Basics of Probabilistic Analysis .....	1.21
1.10.2 Linearity of Expectation .....	1.23

<b>Chapter – 2 DIVIDE AND CONQUER .....</b>	<b>2.1 - 2.21</b>
2.1 Introduction .....	2.1
2.2 Binary Search using DAC .....	2.5
2.3 Max–Min problem using DAC .....	2.7
2.4 Merge Sort .....	2.13
2.5 Quick Sort.....	2.17
2.6 Stressens Matrix Multiplicatin .....	2.19
<b>Chapter – 3 GREEDY METHOD .....</b>	<b>3.1 - 3.21</b>
3.1 Introduction .....	3.1
3.2 Optimal Storage on Tapes .....	3.2
3.3 Knapsack Problem .....	3.3
3.4 Job Sequencing .....	3.7
3.5 Optimal Merg Pattern .....	3.9
3.6 Minimum Spanning Tree .....	3.11
3.7 Single Source Shortest path .....	3.17
<b>Chapter–4 DYNAMIC PROGRAMMING .....</b>	<b>4.1 - 4.27</b>
4.1 Introduction .....	4.1
4.2 Principle of Optimality .....	4.1
4.3 Developing a Dynamic Programming algorithm .....	4.2
4.4 Multistage graph problem .....	4.3
4.5 Reliability design .....	4.7
4.6 All pairs shortest path .....	4.8
4.7 Traveling sales person problem .....	4.10
4.8 Optimal Binary Search Tree .....	4.13
4.9 0/1 Knapsack Problem.....	4.19

4.10	Matrix Chain Multiplication .....	4.22
4.10.1	Direct matrix multiplication on of 3 matrices .....	4.22
4.11	Chain Matrix Multiplication .....	4.23

**Chater – 5 BASIC SEARCH AND TRAVERSAL TECHNIQUES ..... 5.1 - 5.15**

5.1	Introduction .....	5.1
5.2	Binary Tree Traversals .....	5.3
5.3	Graph Traversals .....	5.7
5.3.1	Breadth first search .....	5.9
5.3.2	Depth first search .....	5.10
5.4	AND/OR Graphs .....	5.11
5.5	Game Trees .....	5.12
5.6	Biconnected Components .....	5.13

**Chapter – 6 BACK TRACKING ..... 6.1 - 6.14**

6.1	Introduction .....	6.1
6.2	Control Abstract for Back Tracking .....	6.2
6.3	Queens Problem .....	6.3
6.3.1	4- Queens problem .....	6.3
6.3.2	8- Queens problem .....	6.8
6.4	Graph Coloring .....	6.10
6.5	Sum of Subsets .....	6.13

**Chapter – 7 BRANCH AND BOUND..... 7.1 - 7.18**

7.1	Introduction .....	7.1
7.2	Least Cost (LC) Search .....	7.2

7.3	Bounding .....	7.7
7.4	FIFO Branch & Bound .....	7.10
7.5	LC Branch & Bound .....	7.12
7.6	Traveling Sales Men Problem .....	7.13

**Chapter - 8 NP HARD AND NP COMPLETE**

**PROBLEMS ..... 8.1 - 8.6**

8.1	Introduction .....	8.1
8.2	Classes of Problems .....	8.1
8.3	Reducibility .....	8.3
8.4	Non Deterministic Algorithms .....	8.3
8.5	NP Completeness .....	8.3
8.6	Cook's Theorem .....	8.5

**Chapter – 9 SETS AND UNIONS ..... 1 - 9.6**

9.1	Fundamentals of Sets .....	9.1
9.2	Union & Find .....	9.1
9.3	Weighting Rule for union .....	9.4

**Chapter – 10 BALANCED SEARCH TREES ..... 10.1 - 10.16**

10.1	AVL Trees .....	10.1
10.2	2-3 Trees .....	10.5
10.3	Dictionary .....	10.10
10.4	Priority Queue .....	10.10
10.5	Heap .....	10.11
10.6	Heap Sort .....	10.14

**MODEL OBJECTIVE QUESTIONS (QUIZ) ..... Q1-Q16**

QUIZ -1	.....	Q1
QUIZ -2	.....	Q4
QUIZ -3	.....	Q7
QUIZ -4	.....	Q11
QUIZ -5	.....	Q14

**MODEL QUESTION PAPERS..... M1- M14**

SET NO -1	.....	M1
SET NO -2	.....	M3
SET NO -3	.....	M4
SET NO -4	.....	M6
SET NO -5	.....	M7
SET NO -6	.....	M8
SET NO -7	.....	M9
SET NO -8	.....	M10
SET NO -9	.....	M12
SET NO -10	.....	M14

**SOLVED PROBLEMS (from previous papers) ..... S1-S10**

**UNIT WISE IMPORTANT QUESTIONS**

**(from previous papers) ..... U1-U4**

## A WORD ABOUT ALGORITHM IN HISTORY

“That fondness for science, ... that affability and condescension which God shows to the learned, that promptitude with which he protects and supports them in the elucidation of obscurities and in the removal of difficulties, has encouraged me to compose a short work on calculating by al-jabr and al-muqabala , confining it to what is easiest and most useful in arithmetic.”



**Abu Ja'far Muhammad ibn Musa Al-Khwarizmi**  
[Born: about 780 in Baghdad (now in Iraq). Died: about 850]

***[al-jabr means “restoring”, referring to the process of moving a subtracted quantity to the other side of an equation; al-muqabala is “comparing” and refers to subtracting equal quantities from both sides of an equation***

An algorithm, named after the ninth century scholar Abu Jafar Muhammad Ibn Musu Al-Khowarizmi, is defined as follows: [Roughly speaking]:

- 1 *An algorithm is a set of rules for carrying out calculation either by hand or on a machine.*
- 1 *An algorithm is a finite step-by-step procedure to achieve a required result.*
- 1 *An algorithm is a sequence of computational steps that transform the input into the output.*
- 1 *An algorithm is a sequence of operations performed on data that have to be organized in data structures.*
- 1 *An algorithm is an abstraction of a program to be executed on a physical machine (model of Computation).*

The most famous algorithm in history dates well before the time of the ancient Greeks: Euclid's algorithm for calculating the greatest common divisor of two integers.

## **DCS/DIT 314 : DESIGN AND ANALYSIS OF ALGORITHMS**

### **UNIT-I**

Introduction, Divide and Conquer , The Greedy Method - Knapsack Problem, True vertex splitting, Job sequencing, Minimum-cost spanning trees, Kruskal's algorithm, Optimal storage on tapes, Optimal merge pattern, Single source shortest paths.

### **UNIT-II**

Dynamic Programming - General method, Multistage graph, All pairs shortest path, Single-source shortest path, Optimal Binary search trees, String Editing, 0/1 Knapsack, Reliability design, The traveling salesman problem, Flow shop scheduling.

### **UNIT-III**

Basic traversal & search techniques - Techniques for binary trees, techniques for graphs, connected components & spanning trees, Bi-connected components & DFS.

Back tracking - The General Method, The 8-Queens Problem, Sum of subsets, Graph coloring, Hamiltonian cycle, Knapsack problem.

### **UNIT-IV**

Branch and Bound - The method, 0/0 Knapsack problem, Traveling salesperson, Efficiency considerations.

NP hard and NP Complete Problems - Basic concepts, Cook's Theorem, NP-Hard Graph problems, NP-Hard Scheduling problem, Some simplified NP-Hard problems.

### **Textbook:**

1. L Ellis Horwitz, Sartaj Sahni, 'Fundamentals of Computer Algorithms', Galgotia Pubs.

### **Reference Books:**

1. Aho, Hopcroft & Ullman, 'The Design and Analysis of Computer Algorithms', Addison Wesley.
2. Thomas H.Corman et al, 'Introduction to Algorithms', PHI.

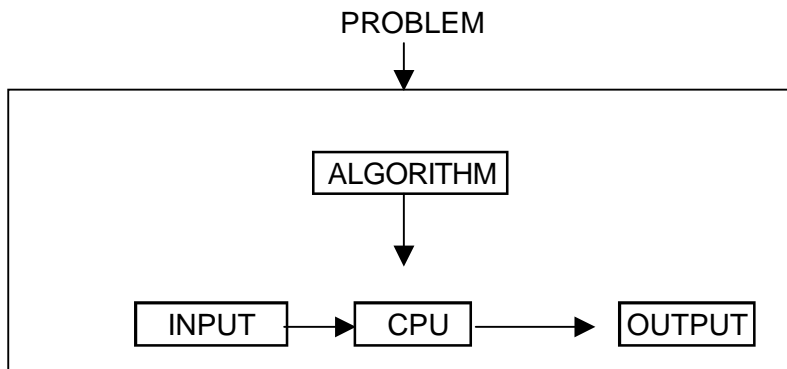


## INTRODUCTION

## NOTES

**1.1 Algorithm, Definition :**

It is a sequence of unambiguous instructions for solving a problem to obtain a required output for any legitimate input in a finite time.



An algorithm is a step by step procedure to represent the solution for a given problem. It consists of a sequence of steps representing a procedure defined in a simple language to solve the problem.

It was first proposed by a Persian mathematician, ABU JAFAR MOHAMMED IBN MUSA AL KHWARIZMI IN 825 A.D.

**1.2 Characteristics of an Algorithm:**

1. An algorithm must have finite number of steps.
2. An algorithm must be simple and must not be ambiguous.
3. It must have zero or more inputs
4. It must produce one or more outputs.
5. Each step must be clearly defined and must perform a specific function.
6. There must be relationship between every two steps.
7. It must terminate after a finite number of steps

Ex: Algorithm for Towers of HANOI

Problem: The objective of the game is to transfer  $n$  disks from leftmost pole to right most pole, without ever placing a larger disk on top of a smaller disk. Only one disk must be moved at a time.

Algorithm TOH ( $n, x, y, z$ )

//Move  $n$  disks from tower  $x$  to tower  $y$ //

```

{   if ( $n \geq 1$ ) then
      { TOH( $n-1, x, z, y$ );
  
```

```
        printf("move top disk from", x "to" y);  
    TOH( n-1, z, y, x);  
    }  
}
```

### **1.3 How to develop an algorithm:**

1. The problem for which an algorithm is being precisely and clearly defined.
2. Develop a mathematical model for the problem. In modeling mathematical structures that are best suited are selected.
3. Data structures and program structures used to develop a solution are planned.
4. The most common method for designing an algorithm is step wise refinement. Step wise refinement breaks the logic into series of steps. The process starts from converting the specifications of the module into an abstract description of an algorithm containing a few abstract statements.
5. Once algorithm is designed, its correctness should be verified. The most common procedure to correct an algorithm is to run the algorithm on various number of test cases. An ideal algorithm is characterized by its run time and space occupied.

### **1.4 How to analyze an algorithm:**

Analysis of algorithm means estimating the efficiency of algorithm in terms of time and space that an algorithm requires.

1. First step is to determine what operations must be done and what are their relative cost. Basic operations such as addition, subtraction, comparison have constant time.
2. Second task is to determine number of data sets which cause algorithm to exhibit all possible patterns.

It requires to understand the best and worst behavior of an algorithm for different data configurations.

#### **1.4.1 General plan for analyzing efficiency of recursive algorithms**

1. Decide a parameter indicating an input size
2. Identify the algorithms basic operation
3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size . If it can the worst case,

average case, and best case efficiencies must be investigated separately

4. Set up a recurrence relation with an appropriate initial condition for the number of times the basic operation is executed
5. Solve the recurrence or at least ascertain the order of growth of its solution

**The analysis of algorithm has two phases:**

1. **Priori analysis:** In this analysis we find some functions which bounds algorithm time and space complexities. By the help of these functions it is possible to compare the efficiency of algorithms.
2. **Posterior Analysis:** Estimating actual time and space when an algorithm is executing is called posterior analysis.

Priori analysis depends only on the number of inputs and operations done, Where as posterior analysis depends on both machine and language used, which are not constant. Hence analysis of most of the algorithms is made through priori analysis.

**1.5 How to validate algorithm:**

- ✦ The process of producing correct answer to given legal inputs is called algorithm validation.
- ✦ The purpose of validation is to prove an algorithm works independent of language.
- ✦ Validation process contains two forms – one is producing respective outputs for given inputs and the other is specification, whether it is producing outputs according to specified requirements.
- ✦ These two forms are shown equivalent for a given input.

The purpose of the validation of an algorithm is to assure that this algorithm will work correctly independently of the issues such as machine, language, platform etc. If an algorithm is according to given specifications and is producing the desired outputs for a given input then we can say it is valid.

**1.6 Time & Space complexities :**

**1.6.1 Space complexity:**

- ✦ It is defined as the amount of memory need to save and execute an algorithm.  
The memory space needed for an algorithm has two parts.
- ✦ One is fixed part, such as memory needed for local variables, constants, instructions etc.
- ✦ Second one is variable part, such as space needed for reference variables, stack space etc. They depend upon the problem instance.
- ✦ The space requirement  $S(P)$  of an algorithm  $P$  can be written as  $S(P) = c + S_p$ , were  $c$  is a constant (representing fixed part). Thus while analyzing an algorithm  $S_p$  (variable part) is estimated.

### 1.6.2 Time complexity:

- ✦ The time taken by a program is the sum of the compile time and the run time.
- ✦ Compile time does not depend on the algorithm.
- ✦ Run time depends on the algorithm
- ✦ The time complexity of an algorithm, is given by the number of steps taken by the algorithm to compute the function it was written for.
- ✦ The number of steps will be computed as a function of the number of inputs and the magnitude of inputs.

The time  $T(P)$  taken by a program is the sum of compile and run time. The compile time does not depend upon the instance characteristics, hence it is fixed. Where as run time depends on the characteristics and it is variable. Thus while analyzing an algorithm run time  $T(P)$  is estimated.

### 1.7 Order of Magnitude

It refers to the frequency of execution of an instruction / statement. The order of magnitude of an algorithm is sum of all frequencies of all statements. The running time of an algorithm increases with size of input and number of operations.

#### Example (1):

In linear search, if the element to be found is in the first position, number of basic operation done is one. If the element is some where in array, basic operation (comparison) is performed  $n$  times.

If in an algorithm, the basic operations are performed same number of times every time then it is called 'Every time complexity analysis'  $T(n)$ .

#### Example (2):

Addition of  $n$  numbers

Input :  $n$

Alg: for 1 to  $n$

Sum = sum +  $n$ ;

$T(n) = n$ ;

#### Example (3):

Matrix multiplication

Input :  $n$

Alg : for  $i = 1$  to  $m$

for  $j = 1$  to  $n$  for  $k = 1$  to  $n$

$c[i][j] = c[i][j] + a[i][k] * b[k][j]$

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

**Worst case w(n):**

It is defined as maximum number of times that an algorithm will ever do.

**Example(1):**

Sequential search / Linear search

Inputs : n

Alg : comparison

If x is the last element, basic operation is done n times.

$$w(n) = n$$

**Best case B(n):**

It is defined as minimum number of times algorithm will do its basic operation.

**Example(1):**

Sequential search

If x is first element B(n) = 1

**Average case A(n):**

It is average number of times algorithm does basic operations for n.

**Example(1):**

Sequential search

Inputs: n

**Case 1:**

If x is in array

Let probability of x to be in kth slot = 1/n

If x is in kth slot, number of times basic operation done is K.

$$A(n) = \sum_{k=1}^n \left( K \times \frac{1}{n} \right) = \frac{1}{n} \left( \frac{n(n+1)}{2} \right) = \frac{n+1}{2}$$

**Case 2:**

If x is not in array

Probability that x is in Kth slot = p/n

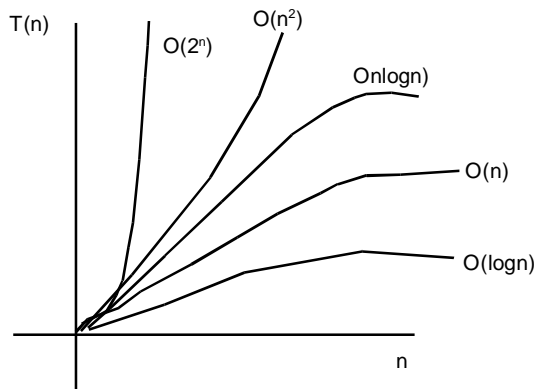
Probability that x is not in array = 1 - P

$$A(n) = \sum_{k=1}^n \left( K \times \frac{P}{n} \right) = n(1-P) = \frac{P}{n} \left( \frac{n(n+1)}{2} \right) + n(1-P) = n \left( 1 - \frac{P}{n} \right) + \frac{P}{2}$$

ORDER	NAME	EXAMPLES
<b>O(1)</b>	<b>Constant</b>	Few algorithms without any loops
<b>O(log n)</b>	<b>Logarithmic</b>	Searching algorithms
<b>O(n)</b>	<b>Linear</b>	Algorithms that scan a list of size n
<b>O(n log n)</b>	<b>n-log-n</b>	Divide and conquer algorithms
<b>O(n<sup>2</sup>)</b>	<b>Quadratic</b>	Operations on n by n matrices
<b>O(n<sup>3</sup>)</b>	<b>Cubic</b>	Nontrivial algorithms from linear algebra
<b>O(2<sup>n</sup>)</b>	<b>Exponential</b>	Algorithms that generate all subsets of a set
<b>O(n!)</b>	<b>Factorial</b>	Algorithms that generate permutations of set

**NOTES**

1. Algorithms with time complexities of  $O(n)$  and  $100n$  are called linear time algorithms since time complexity is linear with input size.
2. Algorithm such as  $O(n^2)$ ,  $0.001 n^2$  are called quadratic time algorithms.
3. Any linear time algorithm is efficient than quadratic time algorithm.



If algorithm is run on same complexity on same type of data, but with higher magnitude of  $n$ , the resulting time is less than some constant time  $f(n)$ .

Thus

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n) \text{ for a given } n.$$

**1.8 Asymptotic Notations:**

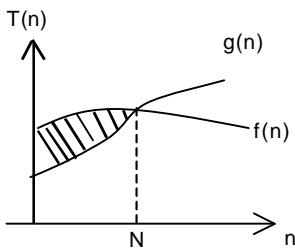
The limiting behaviour of the complexity w.r.t. parameters like size, time etc., is known as Asymptotic complexity. In priori analysis, all factors regarding machine and language are ignored and only number of inputs, size of input and the number of operations are taken into account. The principle indicator of the algorithm's efficiency is the order of growth of an algorithms basic operations. Three notations  $O$  (big oh),  $\Omega$  (big omega) and  $\Theta$  (big theta) are used for this purpose.

**Big O:**

Definition: A function  $f(n)$  is said to be in  $O(g(n))$ , denoted by  $f(n) \leq O(g(n))$ , if  $f(n)$  is bounded above by some constant multiple of  $g(n)$  for all large  $n$ , i.e., iff there exists some positive constant  $C$  and  $N$  such that

$$f(n) \leq C * g(n) \text{ for all } n \geq N$$

It puts an asymptotic upper bound on a function.



**Example (1):**

If  $100n + 5 \leq O(n^2)$ , find  $C$  and  $N$ .

For all  $n \geq 5$

$$100n + 5 \leq 100n + n = 101n \leq 101n^2$$

thus  $C = 101$  and  $N = 5$

**Example (2):**

If  $n^2 + 10n \leq O(n^2)$ , find  $C$  and  $N$

$$f(n) = n^2 + 10n, g(n) = n^2$$

$$f(n) \hat{=} O(g(n))$$

When  $C = 2$  and  $N = 10$

$$\text{If } n \leq 10 \text{ then } c * g(n) \leq f(n)$$

Thus  $N = 10$  keeps an upper bound complexity of above function

**Theorem 1:**

If  $P(n) = a_0 + n a_1 + \dots + n^m a_m$  show that  $P(n) = O(n^m)$

**Proof:**

$$P(n) \leq |a_0| + |a_1|n + \dots + |a_m| n^m$$

$$\leq \left( \frac{|a_0|}{n} + \frac{|a_1|}{n^{m-1}} + |a_m| \right) n^m$$

$$\leq |a_0| + |a_1|n + \dots + |a_m| n^m$$

Where  $n \geq 1$

$$\text{let } C = |a_0| + |a_1| + \dots + |a_m|$$

$$P(n) \leq Cn^m$$

$$P(n) = O(n^m)$$

If  $g(n) = \Theta(f(n))$  then  $f(n)$  is both upper and lower bounds.

**Small O:**  $f(n) = o(g(n))$  iff  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \rightarrow 0$

Example:

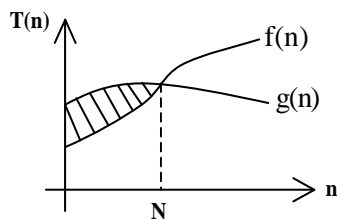
$$\text{If } f(n) = a_k n^k + \dots + a_0$$

then  $f(n) = O(n^k)$  and  $f(n) \sim O(a_k n^k)$ .

**Big  $\Omega$ :**

Definition: A function  $f(n)$  is said to be in  $\Omega(g(n))$ , denoted  $f(n) \geq \Omega(g(n))$ , if  $f(n)$  is bounded below by some positive constant multiple of  $g(n)$  for all large  $n$  i.e., iff there exists positive constants  $C$  and  $N$  such that

$$f(n) \geq C * g(n) \text{ for all } n \geq N$$



**Example (1):**

Show that  $n^3 \geq \Omega(n^2)$

$$n^3 \geq n^2 \text{ for all } n \geq 0$$

\ for  $C=1$  and  $N=0$  we can say  $n^3 \geq \Omega(n^2)$

**Theorem 2:**

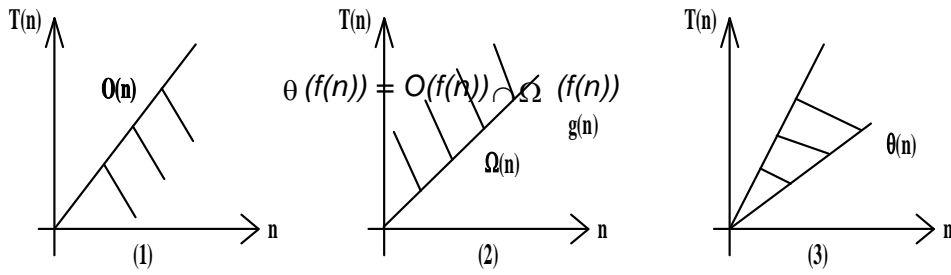
If  $P(n) = a_0 + n a_1 + \dots + n^m a_m$  and if  $a_m > 0$  then show that  $P(n) = \Omega(n^m)$

**Big  $\Theta$ :**

Definition: A function  $f(n)$  is said to be in  $\Theta(g(n))$  denoted by  $f(n) \hat{=} \Theta(g(n))$ , if  $f(n)$  is bound both above and below by some positive constant multiples of  $g(n)$  for all large  $n$  i.e., iff there exists some positive constants  $C_1$  and  $C_2$  and some positive integer  $N$  such that

$$C_1(g(n)) \leq f(n) \leq C_2(g(n))$$





NOTES

**Example (1):**

If we say  $n(n-1)/2 \hat{=} \Theta(n^2)$ , find  $C_1, C_2$  and  $N$

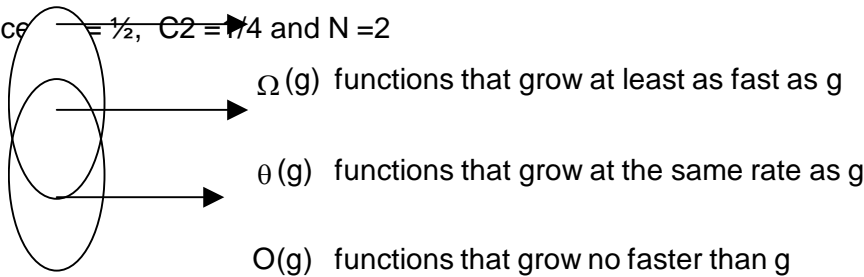
To prove upper bound

$$n(n-1)/2 = n^2/2 - 1/2n \leq n^2/2 \text{ for all } n \geq 0$$

To prove lower bound

$$n(n-1)/2 = n^2/2 - 1/2n \geq n^2/2 - n^2/4 = n^2/4, \text{ for all } n \geq 2$$

hence  $C_1 = 1/2, C_2 = 1/4$  and  $N = 2$



**Theorem 3:**

If  $f_1(n) \in O(g_1(n))$  and  $f_2(n) \in O(g_2(n))$ , then  $f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})$

Proof:

Let  $a_1, a_2, b_1, b_2$  be real numbers and if  $a_1 \leq b_1$  and  $a_2 \leq b_2$

then  $a_1 + a_2 \leq 2 \max(b_1, b_2)$

if  $f_1(n) \in O(g_1(n))$

then  $f_1(n) \leq C_1 * (g_1(n))$  for  $n \geq N_1$

and if  $f_2(n) \in O(g_2(n))$

then  $f_2(n) \leq c_2 * (g_2(n))$  for  $n \geq N_2$

Let  $C_3 = \max\{C_1, C_2\}$  and  $N \geq \max\{N_1, N_2\}$

$$\therefore f_1(n) + f_2(n) \leq C_1 * (g_1(n)) + C_2 * (g_2(n))$$

$$\leq C_3 * (g_1(n)) + C_3 * (g_2(n))$$

$$\leq C_3 \{ (g_1(n)) + (g_2(n)) \} = C_3 * 2 \max\{ (g_1(n)), (g_2(n)) \}$$

}

hence  $f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})$

with the constants C and N required by O definition being  $2C3 = 2 \max\{C1, C2\}$  and  $\max\{N1, N2\}$

### 1.8.1 Optimality

Once the complexity of an algorithm has been estimated, the question arises whether this algorithm is optimal. An algorithm for a given problem is optimal if its complexity reaches the lower bound over all the algorithms solving this problem. For example, any algorithm solving “the intersection of n segments” problem will execute at least  $n^2$  operations in the worst case even if it does nothing but print the output. This is abbreviated by saying that the problem has  $\Omega(n^2)$  complexity. If one finds an  $O(n^2)$  algorithm that solve this problem, it will be optimal and of complexity  $\Theta(n^2)$ .

### 1.8.2 Reduction:

Another technique for estimating the complexity of a problem is the transformation of problems, also called problem reduction. As an example, suppose we know a lower bound for a problem A, and that we would like to estimate a lower bound for a problem B. If we can transform A into B by a transformation step whose cost is less than that for solving A, then B has the same bound as A.

The Convex hull problem nicely illustrates “reduction” technique. A lower bound of Convex-hull problem established by reducing the sorting problem (complexity:  $\Theta(n \log n)$ ) to the Convex hull problem.

### 1.8.3 Profiling:

Profiling is the process of executing a correct program on data sets and measuring the time and space which takes to compute the result

Algorithm of profiling:

Procedure PROFILE

Repeat

{

Read(DATA)

If DATA = end of file then exit

Printf ( A new data set = DATA)

```
CALL STIME(t)
```

```
CALLSOLUTION(DATA,OUTPUT)
```

```
printf (time =s-t)
```

```
}
```

NOTES

### 1.9 Amortized Analysis :

Given a data structure that supports certain operations amortized analysis provides an upper bound on the average cost of each operation for any sequence of a given length  $n$  (i.e., an upper bound for a worst-case sequence). By convention, amortized cost is specified as cost per operation, and not as cost of a sequence of operations of a given length. Although we talk about the 'average cost per operation', no probability involved. While performing amortized analysis, we usually assume that the data structure starts from a canonical start configuration, which is typically that corresponding to the empty set. Sometimes we analyze amortized cost per operation by type of operation.

There are three methods commonly used for amortized analysis.

- 1 Aggregate method •Accounting method •Potential method

All three methods give the same solution (except that the solution returned by the aggregate method does not allow for different amortized costs for different types of operations), but they approach it in different ways.

Example. Consider a data structure MStack that supports the following 3 operations:

- 1 Push(S,x): add element  $x$  to set  $S$
- 1 Pop(S) : remove the most recently added element from  $S$
- 1 Moltipop (S,k): let  $I = \min(k, |S|)$ ; remove the  $I$  most recently added elements from  $S$  We implement Mstack as a standard stack. Then, Push and Pop take constant time. But the third operation takes  $\theta(1)$  time, since the Moltipop operation takes units of time to remove the  $I$  elements plus a constant time to access the stack and determine the value of .

#### 1.9.1 Aggregate Analysis :

In this type of analysis, we obtain an upper bound  $T(n)$  on the time needed to execute any sequence of  $n$  operations, and hence derive the amortized cost per operation as  $\in T(n)/n$ .

For the MStack we argue as follows :

Although a single Moltipop operation can take as much as  $\mathbb{F}(n)$  time in the worst case, over all Pop and Moltipop operations, no more than  $n-1$  elements can be removed from  $S$  since there are only  $n$  operations in total, and each Push operation can add exactly one element to  $S$ . Hence the total number of additions and removals of elements during these  $n$  operations is  $< 2n$ . Besides these addition/removal of elements we

only spend constant time per operation. Hence for any sequence of  $n$  operations the total time needed to execute it is  $O(n)$ . Hence amortized cost per operation is  $O(n)/n$ , which is a constant.

**1.9.2 Accounting Method:**

In the accounting method, we assign an amortized cost  $C_i$  to each operation ahead of time and we establish that these are valid amortized cost by showing that for all sequences.

$$\sum_{i=1}^n \hat{C}_i \geq \sum_{i=1}^n C_i$$

where  $\hat{C}_i$  is the actual cost of the  $i$ th operation. The difference between the sum of the amortized costs and the actual costs is stored as credit on the elements in the data structure and this credit is used to pay for future costly operations.

For the MStack, the actual costs of Push and Pop are 1 (i.e., a constant), and the cost of a Multipop is  $l$ .

We will use amortized costs of 2 for Push and 1 each for Pop and Multipop.

We now argue that the assigned amortized costs are valid. Each Push operation uses one unit of its amortized cost to execute the operation, and leaves the second unit as credit on the element placed on the stack. Thus each element in  $S$  resides on the stack with one unit of credit on it.

The Pop operation uses its one unit of amortized cost to test if the stack is empty or not.

Similarly, the Multipop operation uses its one unit of amortized cost to compute to test the stack and compute the value of  $l$ . Other than this, each Pop and Multipop operation uses the credit on the element being removed to pay for the cost of removing it. Thus, the sum of the amortized costs is always at least as large as the sum of the actual costs, (and the excess of the sum of the amortized costs over the actual sum is stored as credit on the elements in  $S$ ).

Note that we used the value 1 to represent the constant cost of a Pop or Multipop operation. This is a convention that is common in amortized analysis, where we ignore constant factors and simply use 1 (which represents the largest constant among the constants that may appear for different operations).

**1.9.3 Potential Method:**

In the potential method we assign a potential  $\Phi$  to the evolving data structure. Let  $D_i$  be the data structure after the  $i$ th operation, with

Do the initial data structure.

The potential function we pick should satisfy  $\Phi(D_i) \geq \Phi(D_0)$ , for all  $i > 0$ . Usually, we pick a  $\Phi$  with  $\Phi(D_0) = 0$  and  $\Phi(D_i) \geq 0$ , for all  $i$ .

A potential function - that satisfies the above property induces an amortized cost  $\hat{C}_i$  for the  $i$ th operation, for each  $i > 0$ , by

$$\hat{C}_i = C_i + \Phi(D_i) - \Phi(D_{i-1})$$

where  $C_i$  is the actual cost of the  $i$ th operation.

We observe that the induced amortized costs are valid since

$$\sum_{i=1}^n \hat{C}_i = \sum_{i=1}^n C_i + \Phi(D_n) - \Phi(D_0) \geq \sum_{i=1}^n C_i$$

For the MStack example, we use  $\Phi(D_i) =$  number of elements on the stack. We can then observe the amortized costs of Push and Pop remain constant since both the actual cost and the change in potential are constant for both operations, and the amortized cost of Multipop is also a constant since the drop in potential cancels out the actual cost of deleting multiple elements.

**Example : Incrementing a k-bit binary counter**

Let  $A[0..k-1]$  be an array of bits representing a number  $X$

$A[0]$ -low order bit, so  $X = \sum_{j=0}^{k-1} A[j]2^j$

length[A]=k

Start with  $X = 0$

Increment (A)

J=0

while  $j < \text{length}[A]$  and  $A[j] = 1$  do

$A[j] = 0$

$J = J + 1$

end while

if  $J < \text{length}[A]$  then

$A[J] = 1$

end if

end Increment

Count bits flipped

Worst Case

Increment flips k bits in worst case

Sequence of n Increment operations takes O(nk)

**Aggregate Method**

T(n) = all work done in worst case in sequence of n operations

Amortized cost per operation is T(n)/n

X	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0
1	0	0	0	0	1	1
2	0	0	0	1	0	3
3	0	0	0	1	1	4
4	0	0	1	0	0	7
5	0	0	1	0	1	8
6	0	0	1	1	0	10
7	0	0	1	1	1	11
8	0	1	0	0	0	15
9	0	1	0	0	1	16

A[0] flips each time Increment is called n

A[1] flips every other time  $\lfloor \frac{n}{2} \rfloor$

A[1] flips every fourth time  $\lfloor \frac{n}{2^2} \rfloor$

A[J] flips every 2\*\*J time  $\lfloor \frac{n}{2^J} \rfloor$

Total number of flips is:  $\sum_{j=0}^{\lfloor \lg n \rfloor} \lfloor \frac{n}{2^j} \rfloor < n \sum_{j=0}^{\infty} \frac{1}{2^j} = 2n$

So amortized cost of each operation is 2 = O(1)

**Accounting Method**

Assign an a mortized cost to each operation

Amortized cost may be more or less than the actual cost

If amortized cost is more than the actual cost of the operation assign the difference to part of the data structure as a credit

No negative credit allowed

Total amortized cost is  $\geq$  total worst case cost

**Example : Binary counter**

Amortized cost of setting bit to 1 2 units

1 unit to pay for setting bit to 1

1 unit stored with bit

Amortized cost of setting bit to 0 0 units

Only a 1-bit is set to 0,

All 1-bits have credit of one unit

This pays for setting bit to 0

Increment(A)

    J = 0

    while J < length [A] and A[J] == 1 do

        A(J) = 0

        J = J + 1

    end while

    if J < length [A] then

        A[J] = 1

    end if

end Increment

X	A[4]	A[3]	A[2]	A[1]	A[0]	Amortized cost
0	0	0	0	0	0	0
1	0	0	0	0	1 (1)	2
2	0	0	0	1 (1)	0	4
3	0	0	0	1 (1)	1 (1)	6
4	0	0	1 (1)	0	0	8
5	0	0	1 (1)	0	1 (1)	10
6	0	0	1 (1)	1 (1)	0	12
7	0	0	1 (1)	1 (1)	1 (1)	14
8	0	1 (1)	0	0	0	16
9	0	1 (1)	0	0	1 (1)	18

**Example - Move-to-Front**

Assume we have a list of  $n = 2k$  items:  $a_1, a_2, \dots, a_n$

Will perform  $n$  accesses on the list

Will access item  $a_1$   $n/2$  times

Amortized costs:

First access of a

initial location of a  $\leq n$

All other access of a

1

Accessing a non-a item

actual cost + q  $\leq n+1$   
 assign credit to a

a,	b,	c,	d,	e	Amortized	cost
b,	a(1),	c,	d,	e	accessed	b 2
c,	b,	a(2),	d,	e	accessed	c 4
a,	c,	b,	d,	e	accessed	a 1
e,	a(1),	c,	b,	d	accessed	e 6
a,	c,	b,	d,	e	accessed	a 1
a,	c,	b,	d,	e	accessed	a 1

(Amortized cost of all accessed of a)  $\leq n+n/2-1=3n/2-1$

(Amortized cost of accessing all non-a-items)  $\leq n*(n+1)/2$

(Total Cost)  $\leq (4n+n*n)/2-1$ , (average Cost/access)  $\leq 4+n/2$

**Linked List vs. Dynamic Array Unordered List**

Amortized Costs per operation over n operations

	linked List	Dynamic Array
Insertion	1 call to new	1g(n)/n call to new
	set two links	3 data moves
Deletion after find	two links	3 data moves
	1 delete	1g(n)/n call to new

**Probabilistic Analysis and Randomized Quicksort**

**1.10 Worst-case, average-case, and randomized algorithms:**

The last lecture discussed the notions of  $O$ ,  $\Omega$  and  $\Theta$  bounds, and how to compute them using recurrences. We begin this lecture with a difference .... worst-case versus average case bounds. Note that for comparison based algorithms like ..... and Mergesort, we express running time in terms of the number of comparisons made.



Say  $I$  is some input and  $T(I)$  is running time of ..... algorithm on input  $I$ . We can then define:

$$T_{\text{worstcase}}(n) = \max_{\text{input of size } n} T(I)$$

$$T_{\text{averagecase}}(n) = \text{avg}_{\text{input of size } n} T(I)$$

For instance, Mergesort has both worst-case and average-case time  $O(n \log n)$ . It doesn't really depend on the input at all. On the other hand for some algorithms, the running time depends critically on the input. One example is Quicksort.

**Quicksort** : Given array of some length  $n$ ,

1. Pick an element  $p$  of the array as the pivot (or halt if the array has size 0 or 1)
2. Split the array into sub-arrays LESS, EQUAL, and GREATER by comparing each element to the pivot. (LESS has all elements less than  $p$ , EQUAL has all elements equal to  $p$ , and GREATER has all elements greater than  $p$ ).
3. recursively sort LESS and GREATER

The Quicksort algorithm given above is not yet..... specified because we have not stated how we will pick the pivot element  $p$ . For the first version of the algorithm, let's always choose the leftmost element.

**Basic-Quicksort** : Run the Quicksort alg..... given above, always choosing the leftmost element in the array as the pivot.

What is worst-case running time of Basic-Quicksort? we can see that if the array is already sorted, then in Step 2 all the elements (except  $p_1$ ) will go in to the GREATER bucket. Furthermore since the GREATER array is in sorted order, this process will continue recursively, resulting in time  $\Omega(n^2)$ . We can also see that the running time is  $\omega(n^2)$  on any array of  $n$  elements because Step 1 can be executed at most a times, and step 2 take most  $n$  steps to perform. Thus the worst-case running time is  $\Theta(n^2)$ .

On the other hand it turns out (and we will ..... that the average-case running time for Basic Quicksort (averaging over all different initial orderings of the  $n$  elements in the array) is  $O(n \log n)$ . So, Basic-Quicksort has good average case performance but not good worst-case performance.

The fact that algorithm works well on most inputs may be small consolation if the inputs we are faced with are the bad ones (e.g., if our lists .....sorted already). One way we can try to get around this problem is to add randomization to the algorithm itself:

**Randomized-Quicksort: Run the Quicksort algorithm as given above, each time picking a random element in the array as the pivot.**

We will prove that for any given array input ..... of  $n$  elements, the expected time of this algorithm  $E(T(I))$  is  $O(n \log n)$ . This is called a Worst-case Expected-Time bound. Notice that this is better than an average-case bound because we are no longer assuming any special properties of the input. E.g. it could be that in our decisive application the input arrays tend to be mostly sorted or in some special order, and this does not affect our bound because it is a worst-case bound with respect to the input. It is a little peculiar..... making the algorithm probabilistic gives us more control over the running time.

To prove these bounds, we first detour into the basics of probabilistic analysis.

**1.10.1 The Basics of Probabilistic Analysis**

Consider rolling two dice and observing the results. There are 36 possible outcomes it could be that the first die comes up 1 and the second comes up 2, or that the first comes up 2 and the second comes up 1, and so on. Each of these outcomes has probability  $1/36$  (assuming these are fair dice). Suppose we care about some quantity such as ‘what is the probability the sum of the dice equals 7?’ We can compute that by adding up the probabilities of all the outcomes satisfying this condition (there are six of them, for a total probability of  $1/6$ ).

In the language of probability theory, any probabilistic setting is defined by sample space  $S$  and a probability measure  $p$ . The points of the sample space are called elementary events. E.g., in our case, the elementary events are the 36 possible outcomes for the pair of dice. In a discrete probability distribution (as opposed to a ..... one), the probability ..... is a function  $p(e)$  over elementary events  $e$  such that  $p(e) > 0$  for all  $e \in S$ , and  $\sum_{e \in S} p(e) = 1$ . We will also use  $\Pr(e)$  interchangeably with  $p(e)$ .

An event is a subset of the sample space. For instance, one event we might care about is the event that the first die comes up 1. Another is the event that the two dice sum to 7. The probability of an event is just the sum of the probabilities of the elementary events contained inside it (again, this is just for discrete distributions<sup>2</sup>).

A random variable is a function from elementary events to integers or reals. For instance, another way we can talk formally about these dice is to define the random variable  $X_1$  representing the result of

the first die  $X_1$  representing the result of the first die,  $X_2$  representing the result of the second die, and  $X=X_1+X_2$  representing the sum of the two. We could then ask what is the probability that  $X=7$ ?

One property of a random variable we often care about is its expectation. For a discrete random variable  $X$  over sample space  $S$ , the expected value of  $X$  is:

$$E[X] = \sum_{e \in S} \Pr(e)X(e)$$

In other words, the expectation of a random variable  $X$  is just its average value over  $S$ , where each elementary event  $e$  is weighted according to its probability. For instance if we roll a single die and look at the outcome, the expected value is 3.5, because all six elementary events have equal probability. Often one groups together the elementary events according to the different values of the random variable and rewrites the definition like this:

$$E[X] = \sum_a \Pr(X = a)a.$$

More generally for any partition of the probability space into disjoint events  $A_1, A_2, \dots$  we can rewrite the expectation of random variable  $X$  as:

$$E[X] = \sum_i \Pr(A_i)E(X|A_i).$$

where  $E(X|A_i)$  is the expected value of  $X$  given  $A_i$ , defined to be

$$\frac{1}{\Pr(A_i)} \sum_{e \in A_i} \Pr(e)X(e).$$

The formula (3.3) will be useful when we analyze Quicksort. In particular, note that the running time of Randomized Quicksort is a random variable, and our goal is to analyze its expectation.

### 1.10.2 Linearity of Expectation:

An important fact about expected values is Linearity of Expectation: for any two random variables  $X$  and  $Y$ ,  $E(X+Y)=E(X)+E(Y)$ . This fact is incredibly important for analysis of algorithms because it allows us to analyze a complicated random variable by writing it as a sum of simple random variables and then separately analyzing these simple RVs. Let's first prove this fact and then see how it can be used.

#### Example 1: Card shuffling

Suppose we unwrap a fresh deck of cards and shuffle it until the cards are completely random. How many cards do we expect to be in the same position as they were at the start? To solve this, let's think formally about what we are asking. We are looking for the expected value of a random variable  $X$  denoting the number of cards that end in the same position as they started. We can write  $X$  as a sum of random variables  $X_i$  one for each card, where  $X_i=1$  if the  $i$ th card ends in position  $i$  and  $X_i=0$  otherwise. These  $X_i$  are easy to analyze:  $\Pr(X_i=1) = 1/n$  where  $n$  is the number of cards.  $\Pr(X_i=1)$  is also  $E(X_i)$ . Now we use linearity of expectation

$$E[X] = E[X_1 + \dots + X_n] = E[X_1] + \dots + E[X_n] = 1$$

So, this is interesting: no matter how large a deck we are considering the expected number of cards that end in the same position as they started is 1.

**DIVIDE AND CONQUER**

**2.1 Introduction:**

The strategy of divide and conquer which was successfully employed by British rulers in India may also be applied to develop efficient algorithms for complicated problems.

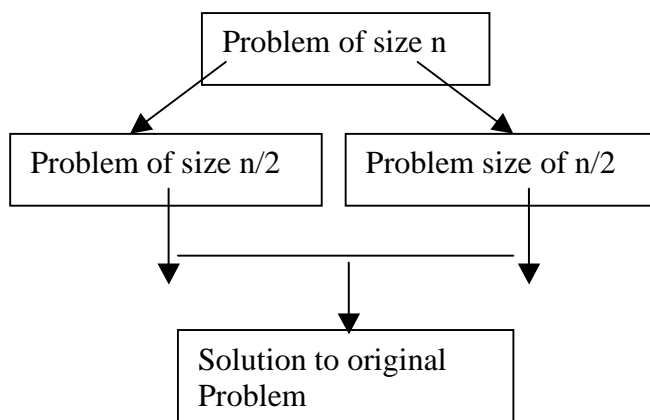
Divide-and-conquer is a top-down technique for designing algorithms that consists of dividing the problem into smaller subproblems hoping that the solutions of the subproblems are easier to find and then composing the partial solutions into the solution of the original problem.

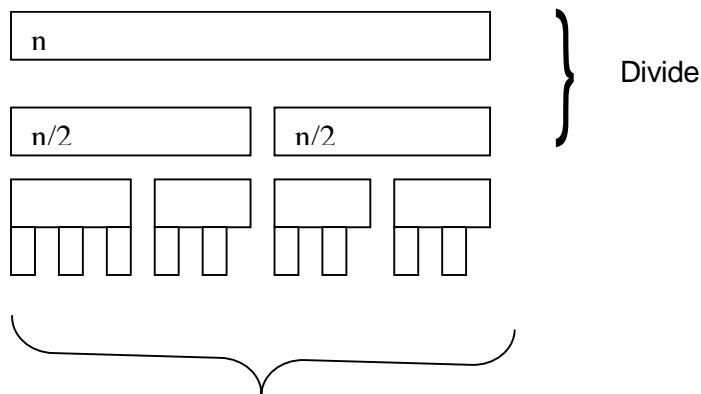
Little more formally, divide-and-conquer paradigm consists of following major phases:

- ✦ Breaking the problem into several sub-problems that are similar to the original problem but smaller in size,
- ✦ Solve the sub-problem recursively (successively and independently), and then
- ✦ Combine these solutions to subproblems to create a solution to the original problem.

It is similar to top down approach but, TD approach is a design methodology (procedure) where as DAC is a plan or policy to solve the given problem. TD approach deals with only dividing the program into modules where as DAC divides the size of input..

In DAC approach the problem is divided into sublevels, where it is abstract. This process continues until we reach concrete level, where we can conquer(get solution). Then combine to get solution for the original problem. The problem containing 'n' inputs is divided into into K distinct sets, yielding k sub problems. These sub problems must be solved and sub solutions are combined to form solution.





Conquer for the solution Fig : 2.1

**Control Abstraction:**

Procedure DAC(p, q)

```

{
  if RANGE(p, q) is small
    Then return G(p, q)
  Else
    {
      m = DIVIDE(p, q)
      return (COMBINE (DAC(p, m), DAC(m + 1, q)))
    }
}

```

- Step 1:** If input size (range)  $(q - p + 1)$  is small, find solution  $G(p, q)$
- Step 2:** If input size is large split range into two data sets  $(p, m)$  and  $(m + 1, q)$  and repeat.
- Step 3:** Repeat the process until sub problem is small enough to solve.

Time complexity of DAC:

$$T(n) = \begin{cases} g(n) & n \text{ is small} \\ 2T(n/2) + f(n) & \text{otherwise} \end{cases}$$

where  $f(n)$  is time taken to divide and combine

$g(n)$  is time taken for conquer

let  $n = 2^m$

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

$$= 2T\left(\frac{2^m}{2}\right) + f(2^m)$$

$$\frac{T(2^m)}{2^m} = 2T\frac{(2^{m-1})}{2^m} + \frac{f(2^m)}{2^m}$$

$$= \frac{T(2^{m-1})}{2^{m-1}} + \frac{k \cdot 2^m}{2^m} = \frac{T(2^{m-1})}{2^{m-1}} + 1$$

$$= \frac{(2^{m-2})}{2^{m-2}} + 1 + 1$$

$$= \frac{T(2^{m-m})}{2^{m-m}} + m$$

$$= T(1) + m$$

$$= m + 1$$

$$T(2^m) = 2^m(m + 1)$$

$$T(n) = n(\log n + 1)$$

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

**Example (1):**

Find time complexity for DAC

a) If  $g(n) = O(1)$  and  $f(n) = O(n)$ ,

b) If  $g(n) = O(1)$  and  $f(n) = O(1)$

a) It is the same as above case

$$\text{i.e., } O(n) = k \cdot 2^m = f(n)$$

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

b)  $T(n) = 2T(n/2) + f(n)$

$$= 2^2 T(n/2) + O(1)$$

$$= 2(2T(n/4) + 1) + 1$$

$$= 2T(n/4) + 2 + 1$$

$$= 2^k T(n/2^k) + 2^{k-1} + \dots + 2 + 1$$

$$\begin{aligned}
 \text{let } n &= 2^k \\
 &= nT(1) + \sum_{i=0}^{k-1} 2^i \\
 &= n + \sum_{i=0}^k 2^i - 2^{-1} \\
 &= n + (n - 1 - 1/2) \\
 &= 2n - 3/2 \\
 T(n) &= O(n)
 \end{aligned}$$

## 2.2 Binary search using DAC:

### Problem:

To search whether an element  $x$  is present in the given list. If  $x$  is present, the value  $j$  must be displayed such that  $a_j = x$

### Algorithm:

**Step 1:** DAC suggests to break the input 'n' elements  $I = (a_1, a_2, \dots, a_n)$  into

$$\begin{aligned}
 I_1 &= (a_1, a_2, \dots, a_{k-1}), I_2 = a_k \text{ and} \\
 I_3 &= (a_{k+1}, a_{k+2}, \dots, a_n) \text{ into data sets.}
 \end{aligned}$$

**Step 2:** If  $x = a_k$  no need of searching  $I_1, I_3$ ,

**Step 3:** If  $x < a_k$  only  $I_1$  is searched

**Step 4:** If  $x > a_k$  only  $I_3$  is searched

**Step 5:** Repeat the steps 2 to 4.

Every time  $K$  is chosen such that  $a_k$  is middle of 'n' elements  $k = n + 1/2$ .

### Control abstraction:

Procedure BINSRECH (1, n)

```

{
mid = n+1/2
if A(mid) = x then
{
loc = mid;
return;
}
}
    
```



```

}
  Else
    if A(mid) > x then
      BINSRCH (1, mid - 1)
    Else
      BINSRCH(mid + 1, n)
}
    
```

Time complexity:

The given data set is divided into two half and search is done in only one half wither left half or right half depending on the value of x

$$T(n) = \begin{cases} g(n) & n \text{ is } \leq 1 \\ T(n/2) + g(n) & n > 1 \end{cases}$$

$$\begin{aligned}
 T(n) &= T(n/2) + g(n) \\
 &= T\left(\frac{n/2}{2}\right) + 2g(n) \\
 &= T(n/2^2) + 2g(n) \\
 &= T(n/2^k) + k g(n)
 \end{aligned}$$

If  $n = 2^k$  and  $g(n) = 1$

$$\begin{aligned}
 T(n) &= T(1) + k \\
 &= k + 1 = O(\log n)
 \end{aligned}$$

It is valid if  $n$  is  $2^{k-1} \leq n \leq 2^k$

**Example (1):**

Take instances 12      14      18      22      24      38  
                   1      2      3      4      5      6

The element to be found is the key for searching and let Key = 24

Initially mid = (1+6)/2 = 3

low	high	mid	
1	6	3	Key > 18
4		6	5    Key = 24

∴

Thus number of comparisons needed are two.

**Theorem 1:**

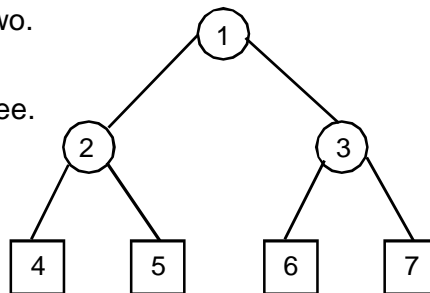
Prove that  $E = I + 2n$ , for a binary search tree.

E = sum of all lengths of external nodes

I = sum of all lengths of internal nodes

n = internal nodes

for level 2 for n = 3



$$E = 2^2 \cdot 2 = 8 \quad E = I + 2n$$

$$I = 2^1 > 1 + 2 \quad 8 = 2 + 6 = 8$$

By induction if we add these branches to form a tree it satisfied  $E = I + 2n$

**Theorem 2:**

Prove average successful search time  $S$  is equal to average unsuccessful search time  $U$ , if  $n \rightarrow \infty$

$$\lim_{n \rightarrow \infty} S_{avg} = \lim_{n \rightarrow \infty} U_{avg}$$

$$U_{avg} = \frac{E}{n+1} S_{avg} = \frac{1}{n} + 1$$

$$U_{avg} = \frac{1+2n}{n+1} = \frac{1+2n}{n}$$

$$= \frac{1}{n} + 2 = \frac{1+2n}{n} = E_{avg}$$

$$U = \frac{1+2n}{n+1}$$

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

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

$$\lim_{n \rightarrow \infty} S = U - 2 + 1 = U - 1 \cong U$$

$$S_{avg} = \frac{\text{Total internal pathlengths}}{n} + 1$$

$$U_{avg} = \frac{\text{Total external pathlengths}}{n+1}$$

**2.3 Max-Min Problem using DAC :**

**Problem:**

To find out maximum and minimum elements in a given list of elements.

Analysis:

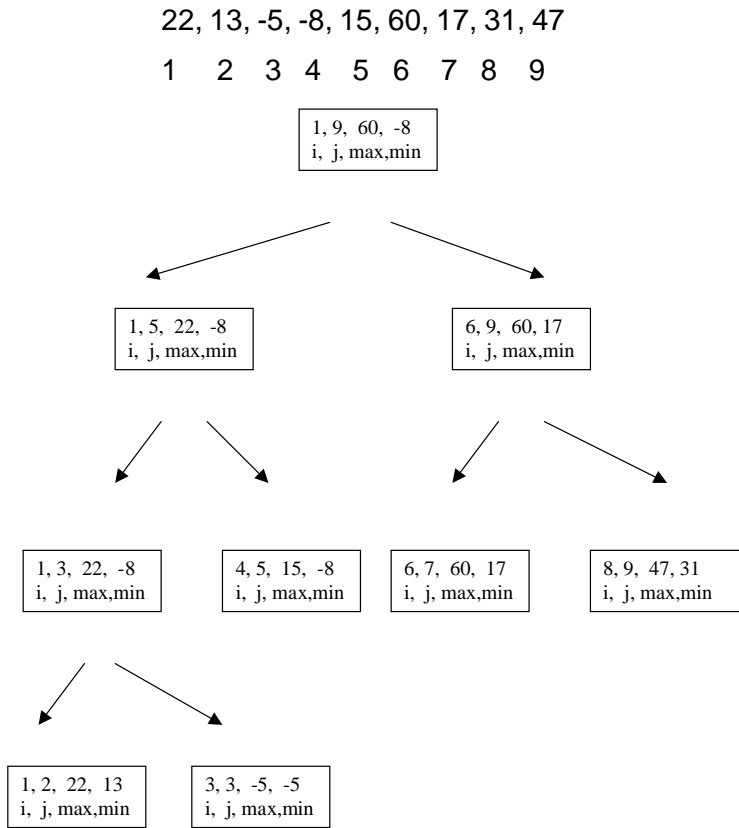
According to divide and conquer algorithm the given instance is divided into smaller instances.

Step 1: For example an instance  $I=(n, A(1), A(2), \dots, A(n))$  is divided into  $I1=(n/2, A(1), A(2), \dots, A(n/2))$  and  $I2=(n-n/2, A(n/2+1), \dots, A(n))$ .

Step 2: If  $\text{Max}(I)$  and  $\text{Min}(I)$  are the maximum and minimum of the elements in  $I$  the  $\text{Max}(I) = \text{the larger of } \text{Max}(I1) \text{ and } \text{Max}(I2)$  and  $\text{Min}(I) = \text{the smaller of } \text{Min}(I1) \text{ and } \text{Min}(I2)$ .

Step 3: The above procedure is repeated until  $I$  contains only one element and then the answer is computed.

Example: Let us consider an instance 22,13,-5,-8,15,60,17,31,47 from which maximum and minimum elements to be found.



**Algorithm 1:**

**Procedure:**

Max – Min (A[1].....A[n])

```

{ for i = 2 to n do
    if A[i] > max
    then min = A[i]
    endif
    if A[i] < min
    then min = A[i]
    endif
endfor
}
    
```

This algorithm requires  $(n-1)$  comparisons for finding largest number and  $(n-1)$  comparisons for smallest number.

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

**Algorithm 2:**

If the above algorithm is modified as

```

If A[i] > max
    then max = A[i]
Else
    A[i] < min
    then min = A[i]
    
```

Best case occurs when elements are in increasing order. Number of comparisons for this case is  $(n - 1)$ . Worst case occurs when elements are in decreasing order. Number of comparisons for this case is  $2(n-1)$ .

$$\therefore \text{Average number of comparisons} = \frac{(n-1) + 2(n-1)}{2} = \frac{3n}{2} - 1$$

**ALGORITHM BASED ON DAC**

Step 1:  $I = (a[1], a[2], \dots, a[n])$  is divided into

$$I_1 = (a[1], a[2], \dots, a[n/2])$$

$$I_2 = (a[n/2 + 1], \dots, a[n])$$

Step 2:  $\text{Max}(I) = \text{large of } (\text{Max}(I_1); \text{Max}(I_2))$

$\text{Min}(I) = \text{Small of } (\text{Min}(I_1); \text{Min}(I_2))$

*Procedure MAX – MIN(i, j, Max, Min)*

```

{
    If i = j then
        Max = Min = A[I]
    else
        If j – i = 1
            If A[i] < A[j]
                max = A[j]
                min = A[j]
    
```

```

else
    max = A[j]
    min = A[i]
else
{
    p = (i+j)/2
    MAX - MIN ((i, p, h_max, h_min)
    MAX - MIN ((p + 1, j, L_max, L_min)

                Max = large (h_max, L_max)

                Min = small (h_min, L_min)

}
}

```

**Time complexity:**

$$T(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ 2T(n/2) + 2 & n > 2 \end{cases}$$

f(n) = 2 since there are two operations 1. Merging for largest 2. Merging for smallest

$$\begin{aligned}
 \text{Thus } T(n) &= 2T\left(\frac{n}{2}\right) + 2 \\
 &= 2\{2T\left(\frac{n}{2}\right) + 2\} + 2 \\
 &= 2^2 T(n/2^2) + 2^2 + 2 \\
 &= 2^{k-1}T(n/2^{k-1}) + 2^{k-1} + \dots + \dots + 2
 \end{aligned}$$

let n = 2<sup>k</sup>

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

but  $T(2) = 1$

$$\begin{aligned}
 &= 2^{k-1} + \sum_{(n)}^{k-1} 2^i - 1 \\
 &= 2^{k-1} + 2^k - 1 - 1 \\
 &= 2^k/2 + 2^k - 2 \\
 &= 3/2 \cdot 2^k - 2 = 3/2 \cdot n - 2 \\
 T(n) &= O(3/2 \cdot n)
 \end{aligned}$$

**Theorem 3:**

Prove that  $T(n) = O(n^{\log m})$  if  $T(n) = m(T(n/2)) + an^2$

Ans:  $T(n) = nT(n/2) + an^2$

$$\begin{aligned}
 &= m(mT(n/2/2) + an^2)an^2 \\
 &= m^2T(n/2^k) + (m+1)an^2 \\
 &= m^k T(n/2^k) + an^2 \sum_{(n)}^k m^{i-1}
 \end{aligned}$$

$n = 2^k$

$$\begin{aligned}
 &= m^k + an^2 \left( \sum_{i=1}^k m^{i-1} = llm \right) \\
 &= m^k + an^2 \left( \frac{m^{k+1}-1}{m-1} - \frac{1}{m} \right) \\
 &= m^k + an^2 \left( \frac{m^k-1}{m-1} - \frac{1}{m} \right) \\
 &\cong m^k(1+an^2/m)
 \end{aligned}$$

If  $m \gg n$

$$= m^k = m^{\log n}$$

but  $O(m^{\log n}) = O(n^{\log m})$

since  $(x^{\log y} = y^{\log x})$

**2.4 Merge Sort:**

**Problem:**

Sorting the given list of elements in ascending or decreasing order.

**Analysis:**

Step 1: Given a sequence of n elements, A(1),A(2),....A(n). Split them into two sets A(1).....A(n/2) and A(n/2+1).....A(n).

Step 2: Splitting is carried over until each set contain only one element.

Step 3: Each set is individually sorted and merged to produce a single sorted sequence.

Step 4: Step 3 is carried over until a single sequence of n elements is produced

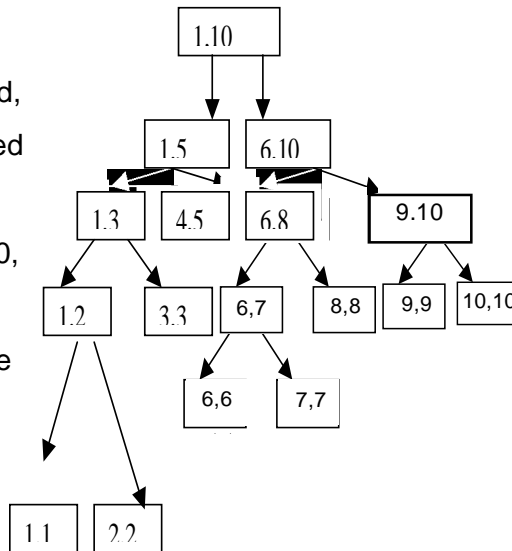
Example: Let us consider the instances 10, 285, 179, 652, 351, 423, 861, 254, 450, 520

310, 285, 179, 652, 351, 423, 861, 254, 450, 520

1 2 3 4 5 6 7 8 9 10

For first recursive call left half will be sorted (310) and (285) will be merged, then (285, 310) and 179 will be merged and at last (179, 285, 310) and (652, 351) are merged to give 179, 285, 310, 351, 652 Similarly, for second recursive call right half will be sorted to give 254, 423, 520, 681

A partial view of merge sorting is shown in fig



Finally merging of these two lists produces 179, 254, 285, 310, 351, 423, 450, 520, 652, 820

**Algorithm:**

**Step 1:** Given list of elements

$I = \{A(1), A(2) \dots\dots\dots A(n)\}$  is divided into two sets

$I_1 = \{A(1), A(2) \dots\dots\dots A(n/2)\}$

$I_2 = \{A(n/2 + 1 \dots\dots\dots A(n)\}$

**Step 2:** Sort  $I_1$  and sort  $I_2$  separately

**Step 3:** Merge them

**Step 4:** Repeat the process

Procedure merge sort (p, q)

```
{
    if p < q
        mid = p + q/2;
        merge sort (p, mid);
        merge sort (mid + 1, q);
        merge (p, mid, q);
    endif
}
```

Procedure Merge (p, mid, q)

```
{
    let h = p; l = p; j = mid + 1
    while (h < mid and j < q)
        do
            {
                if A(h) < A(j)
                    B(l) = A(h);
                    h = h + 1;
                else
                    B(i) = A(j);
```

NOTES



```

                j = j + 1;
            end if
            l = l + 1;
        }

    if h > mid
        for k = j to q
            B(i) = A(k);
            l = l + 1;
        repeat
    else
        if j > q
            for k = h to mid
                B(i) = A(k);
                l = l + 1;
            repeat
        endif

    for k = p to q
        A(k) = B(k);

    Repeat
}

```

**Time Complexity:**

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

$$= 2(2T(n/4) + 2cn)$$

$$= 4T(n/4) + 2cn$$

$$= 2^k T(n/2^k) + kcn$$

$$\text{Let } n = 2^k$$

$$= n.a + kcn$$

$$= na + cn \log n$$

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

NOTES

**2.5 QUICK SORT :**

Problem: Arranging all the elements in a list in either ascending or descending order.

Analysis:

File  $A(1, \dots, n)$  was divided into subfiles, so that sorted subfiles does not need merging. This can be achieved by rearranging the elements in  $A(1 \dots n)$  such that  $A(i) \leq A(j)$  for all  $1 < i < m$  and all  $m+1 < j < n$  for some  $m$ ,  $1 \leq m \leq n$ . Thus the elements in  $A(1 \dots m)$  and  $A(m+1, \dots, n)$  may be independently sorted.

**Step 1:** Select first element as pivot in the given sequence  $A(1 \dots n)$

**Step 2:** Two variables are initialized

$$\text{Lower} = a + 1 \qquad \text{upper} = b$$

Lower scans from left to right

Upper scans from right to left

**Step 3:** If  $\text{lower} < \text{right}$

Corresponding elements are swapped. When lower and upper cross each other, process is stopped.

**Step 4:** Pivot is swapped with  $A(\text{upper})$  which becomes new pivot and repeat the process.

**Algorithm:**

Procedure Quick sort (a, b)

```
{
    if a < b then
        k = partition (a, b);

        quick sort (a, k - 1);

        quick sort (k + 1, b);
}
```

Procedure partition (m, p)

```
{
    let x = A[m], l = m;

    while (l < p)
    {
        do

            l = l + 1;

        Until (A[l] >= x)

        Do

            p = p - 1;

        until (A[p] <= x)
        If l < p

            Swap(A[l], A[p])

    }

    A[m] = A[p];
    A[p] = x
}
```

**Time complexity:**

Let us assume file size n is  $2^m$ ,  $m = \log_2 n$

Also assume position of pivot is exactly in the middle of array.

In first pass  $(n - 1)$  comparisons are made.

In second pass  $(n/2 - 1)$  comparisons are made.

In average case, the total number of comparisons are given by  $(n - 1) + 2^*(n/2 - 1) + 4(n/4 - 1) + \dots n^*(n/2^m - 1) \propto mn$

NOTES

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

In the worst case, if the original file is already sorted, the file is split into sub files of size into 0 and  $n = 1$ .

$$\text{Total comparisons} = (n - 1) + (n - 2) + (n - 3) \dots \dots \dots (n - n) \propto n = O(n^2)$$

**2.6 Strassen's Matrix Multiplication:**

Let A and B be two  $n \times n$  matrix whose  $i, j$ th element is formed by taking the elements in the  $i$ th row of A and the  $j$ th column of B and multiplying them to get

$$C(i, j) = \sum A(i, k) B(k, j)$$

For all  $i$  and  $j$  between 1 to  $n$ . To compute  $C(i, j)$  using this formula we need  $n$  multiplications. As the matrix has  $n^2$  elements, the time for the resulting matrix multiplication algorithm has a time complexity of  $O(n^3)$ .

According to divided and conquer the given matrices A and B are divided into four square sub matrices each of dimension  $n/2$  and  $n/2$ . The product AB can be computed using the product of  $2 \times 2$  matrices.

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \rightarrow \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$$C_{11} = A_{11}B_{11} + A_{12}B_{12}$$

$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$

$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

To find AB we need 8 multiplications and 4 additions, Since two matrices can be added in time  $cn^2$  for a constant  $c$ .

**Strassen's equations are:**

$P = (A_{11} + A_{12})(B_{11} + B_{22})$	$Q = (A_{21} + A_{22})B_{11}$
$R = A_{11}(B_{12} - B_{22})$	$S = A_{22}(B_{21} - B_{11})$
$T = (A_{11} + A_{12})B_{22}$	$U = (A_{21} - A_{11})(B_{11} + B_{12})$
$V = (A_{12} - A_{22})(B_{21} + B_{22})$	

$$\begin{aligned} C_{11} &= P + S - T + V & C_{12} &= R + T \\ C_{21} &= Q + S & C_{22} &= P + R - Q + U \end{aligned}$$

The recurrence relation is  $T(n) = 7T(n/2) + 18n^2$

$$T(n) = \begin{cases} b & n \leq 2 \\ 8T(n/2) + cn^2 & n > 2 \end{cases}$$

Where b and c are constants

This recurrence can be solved in the same way as earlier recurrences to obtain  $T(n) + O(n^3)$ . Hence no improvement over the conventional method has been made. Since matrix multiplications are more expensive than matrix addition  $O(n^3)$  versus  $O(n^2)$ , we can attempt to reformulate the equations for  $C_{ij}$  so as to have fewer multiplications and possibly more additions. Volker Strassen has discovered a way to compute the  $C_{ij}$ 's of using only 7 multiplications and 18 additions or subtractions. This method involves first computing the seven  $n/2 \times n/2$  matrices P, Q, R, S, T, U and V. Then the  $C_{ij}$ 's are computed using the formulas, as can be seen, P, Q, R, S, T, U and V can be computed using 7 matrix multiplications and 10 matrix additions or subtractions. The  $C_{ij}$ 's require an additional 8 additions or subtractions.

$$\begin{aligned} P &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ Q &= (A_{21} + A_{22})B_{11} \\ R &= A_{11}(B_{12} - B_{22}) \\ S &= A_{22}(B_{21} - B_{11}) \\ T &= (A_{11} + A_{12})B_{22} \\ U &= (A_{21} - A_{11})(B_{11} + B_{22}) \\ V &= (A_{12} - A_{22})(B_{21} + B_{22}) \\ C_{11} &= P + S - T + V \\ C_{12} &= R + T \\ C_{21} &= Q + S \\ C_{22} &= P + R - Q + U \end{aligned}$$

The resulting recurrence relation for T(n) is

$$T(n) = \begin{cases} b & n \leq 2 \\ 7T(n/2) + an^2 & n > 2 \end{cases}$$

Where a and b are constants. Working with this formula, we get

$$\begin{aligned} T(n) &= an^2[1 + 7/4 + (7/4)^2 + \dots + (7/4)^{k-1}] + 7^k T(1) \\ &\leq cn^2 (7/4)^{\log n} + 7^{\log n}, c \text{ a constant} \\ &= O(n^{\log 7}) \approx O(n^{2.81}). \end{aligned}$$

\*\*\*\*\*

***Design and Analysis of Algorithms***

Design and Analysis of Algorithms

NOTES  
2.19

## GREEDY METHOD

### 3.1 Introduction:

Greedy method is a general design technique despite the fact that it is applicable to optimization problems only. It suggests constructing a solution through a sequence of steps each expanding a partially constructed solution obtained so far until a complete solution is reached.

According to greedy method an algorithm works in stages. At each stage a decision is made regarding whether an input is optimal or not.

To find an optimal solution two parameters must be considered. They are

1. Feasibility: Any subset that satisfies some specified constraints, then it is said to be feasible.
2. Objective function: This feasible solution must either maximize or minimize a given function to achieve an objective.
3. Irrevocable: Once made, it cannot be changed on subsequent stages.

**Step 1:** A decision is made regarding whether or not a particular input is in an optimal solution. Select an input

Step 2: If inclusion of input results infeasible solution, then delete from solution.

Step 3: Repeat until optimization is achieved.

Control abstract:

Procedure GREEDY(A[], n)

```
{
    Solution = null;
    for i = 1 to n do
        {
            x = select (A(i))
            if FEASIBLE (solution, x)
                Solution = UNION(solution, x)
        }
}
```



```

return(solution);
}

```

**3.2 Optimal storage On Tapes:**

**Problem 1:**

To store n programs on a tape of length L such that mean retrieval time is minimum.

Solution:

Let each program is of length  $L_i$ , ( $1 \leq i \leq n$ ), such that the length of all tapes must be  $\leq L$ . If the programs are in the order  $l = i_1, i_2, \dots, i_n$ ,

then the time  $t_j$  to retrieve a program is proportional to  $\sum_{i \leq k \leq j} L_{i_k}$

If all the programs are retrieved with equal probabilities, mean retrieval time  $MRT = \frac{1}{n} \sum_{i \leq k \leq n} t_j$ . To minimize MRT, we must store the programs such that their lengths are in non decreasing order  $l_1 < l_2 < l_n$ .

$$MRT = \frac{1}{n} \sum_{i \leq j \leq n} \sum_{i \leq k \leq j} L_{i_k}$$

**Example 1:**

There are 3 programs of length  $l = (5, 10, 3)$

1, 2, 3  $5 + 15 + 18 = 38$

1, 3, 2  $5 + 5 + 18 = 31$

3, 1, 2  $3 + 8 + 18 = 29$  is the best case where retrieval time is minimized, if the next program of least length is added.

Time complexity:

If programs are assumed to be in order, then there is only one loop in algorithm, thus

Time complexity =  $O(n)$ . If we consider sorting also, sorting requires  $O(n \log n)$

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

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

**Problem 2:**

To store different programs of length  $l_1, l_2, l_3, \dots, l_n$  with different frequencies  $f_1, f_2, \dots, f_n$  on a single tape so that MRT is less.

Solution:

If frequencies are taken into account then MRT,  $t_j = \sum_{i \leq k \leq j} \frac{f_{ik}}{l_{ik}}$  The programs must be arranged such that ratio of frequency and length must be in non increasing order.

$$l_1 \leq l_2 \leq l_3 \dots \dots \dots l_n$$

$$f_1 \geq f_2 \geq f_3 \dots \dots \dots f_n$$

$$\frac{f_1}{l_1} \geq \frac{f_2}{l_2} \dots \dots \dots \frac{f_n}{l_n}$$

**3.3 Knapsack problem:**

Given n objects and a knapsack, object i has a weight  $w_i$  and knapsack has capacity of M. If a fraction  $x_i$ ,  $0 \leq x_i \leq 1$  of object i is placed into knapsack then a profit  $p_i x_i$  is earned.

Problem: To fill the knapsack so that total profit is maximum and satisfies two conditions.

1) Objective  $\sum_{i=1}^n p_i x_i = \text{maximum}$       2)  $\sum_{i=1}^n w_i x_i \leq M$

**Types of knapsack:**

1. Normal knapsack: in normal knapsack, a fraction of the last object is included to maximize the profit.

i.e.,  $0 \leq x \leq 1$  so that  $\sum w_i x_i = M$

2. O/1 knapsack: In 0/1 knapsack, if the last object is able to insert completely, then only it is selected otherwise it not selected.

i.e.,  $x = 1/0$  so that  $\sum w_i x_i \leq M$

**Control Abstract for Normal Knapsack:**

Let  $P(l : n)$ ,  $w(l : n)$  contains profits & weights of n objects are ordered so

that  $\frac{P_i}{w_i} \geq \frac{P_{i+1}}{w_{i+1}}$  and  $(l : n)$  is the solution vector.

Normal knapsack (p, w, m, x, n)

{

$x = 0; c = M;$

for i = 1 to n do

```
        if w[l] ≤ c then
            {
                x[i] = 1;
                c = c - w[i];
            }
    else
        {
            x[i] = c/w[i]
            return;
        }
    }
```

**Algorithm for 0/1 knapsack:**

0/1 knapsack (p, w, m, x, n)

```
{
    x = 0; c = M;
    for l = 1 to n do
    {
        if w[i] > c then return
        else
            x[i] = 1
            c = c - w[l]
        }
    }
```

**Example:** Find an optimal solution to the knapsack instance  $n = 7$  objects and the capacity of knapsack  $m = 15$ . The profits and weights of the objects are given below.

$P_1, \dots, P_7 =$	10	5	15	7	6	18	3
$W_1, \dots, W_7 =$	2	3	5	7	1	4	1

**Solution:**

$P_1/W_1, \dots, P_7/W_7 =$	5	1.6	3	1	6	4.5	3
-----------------------------	---	-----	---	---	---	-----	---

Arrange in non increasing order of  $P_i/W_i$

Order =	5	1	6	3	7	2	4
P/w =	6	3	4.5	3	3	1.6	1

$C = 15 \quad x = 0$

- |   |                                     |
|---|-------------------------------------|
| 1) $x[1] = 1 \quad x = 0$                   | 2) $x[2] = 1 \quad c = 15 - 1 = 14$ |
| 3) $x[3] = 1 \quad c = 14 - 2 = 12$         | 4) $x[4] = 1 \quad c = 12 - 4 = 8$  |
| 5) $x[5] = 1 \quad c = 8 - 5 = 3$           | 6) $x[6] = 2/3 \quad c = 3 - 1 = 2$ |
| 7) $x[7] = 0 \quad c = 2 - 2/3 \cdot 3 = 0$ |                                     |

$X =$	1	1	1	1	1	2/3	0
-------	---	---	---	---	---	-----	---

Rearranging into original form

$X =$	1	2/3	1	0	1	1	1
-------	---	-----	---	---	---	---	---

Solution vector for O/I knapsack is  $X = 1010111$

$$\text{Total profit } \sum p_i x_i = 10 + 15 + 0 + 6 + 18 + 3 = 52$$

**Theorem:**

If  $p_1/w_1 > p_2/w_2 > \dots > p_n/w_n$  then greedy knapsack generate optimal solution.

**Proof:** Let  $X = (x_1, \dots, x_n)$  be solution vector, let  $j$  be the least index such that  $x_j \neq 1$

$$X_i = 1 \text{ for } 1 \leq i \leq j$$

$$X_i = 0 \text{ for } j \leq i \leq n$$

$$X_i = 0 \text{ for } 0 \leq x_j \leq 1$$

Let  $y = (y_1, \dots, y_n)$  be optimal solution

$$\sum w_i x_i = M$$

let  $k$  be least index such that  $y_k \neq x_k$  and  $y_k < x_k$

- 1) if  $k < j$  then  $x_k = 1$ , but  $y_k < x_k$

2) if  $k = j$  then  $\sum w_i x_i = M$   $y_i = x_i$  for  $1 < i < j$ ,  $y_k < x_k$

if  $k > \sum w_i x_i = M$  which is not possible.

If we increase  $y_k$  to  $x_k$  and decrease as many of  $(y_{k+1}, \dots, y_n)$ , this results a new solution.

$Z = (z_1, \dots, z_n)$  with  $z_1 = x_1, 1 \leq i \leq k$  and  $\sum w_i (y_i - z_i) = w_k (z_k - y_k)$

$$\begin{aligned} \sum p_i z_i &= \sum_{1 < i < n} p_i y_i + (z_k - y_k) w_k p_k / w_k - \sum_{k < i < n} (y_i - z_i) w_i p_i / w_i \\ &= \sum_{1 < i < n} p_i y_i + [(z_k - y_k) w_k - \sum_{k < i < n} (y_i - z_i) w_i] p_k / w_k \\ &= \sum_{1 < i < n} p_i y_i \end{aligned}$$

### 3.4 Job Sequencing:

Let there are  $n$  jobs, for any job  $i$ , profit  $P_i$  is earned if it is completed in dead line  $d_i$ .

**Object:** Obtain a feasible solution  $J = (j_1, j_2, \dots)$  so that the profits i.e.,  $\sum_{i \in J} p_i$  is maximum.

The set of jobs  $J$  is such that each job completes within deadline.

**Problem:** There are  $n$  jobs, for any job  $I$  a profit  $P_i$  is earned iff it is completed in a dead line  $d_i$ .

**Feasibility:** Each job must be completed in the given dead line.

**Objective:** The sum of the profits of doing all jobs must be minimum, i.e.  $\sum P_i$  maximum.

**Solution:** Try all possible permutations and check if jobs in  $J$ , can be processed in any of these permutations without violating dead line. If  $J$  is a set of  $K$  jobs and  $s = i_1, i_2, \dots, i_k$  a permutation of jobs in  $J$  such that  $d_{i_1} \leq d_{i_2} \leq \dots \leq d_{i_k}$ . Then  $J$  is a feasible solution.

**Control abstract:**

Procedure JS(D, J, N, K)

{

$D(0) = J(0) = 0;$

$K = 1, J(1) = 1$

For  $l = 2$  to  $n$  do

{

```

r = k

while D(J(r)) > max (D(i), r)

    {
        r = r - 1;
    }

    if D(i) > r then
        for (l = k; i = r + 1, i = i - 1)
            J(i + i) = J(i)
    }

    J(r + 1) = i;

k = k + 1;
}

}

}
    
```

Example: Given 5 jobs, such that they have the profits and should be complete within dead lines as given below. Find and optimal sequence to achieve maximum profit.

$P_1 \dots P_5 =$	20	15	10	5	1
$D_1 \dots d_5 =$	2	2	1	3	3

	Feasible solution	Processing sequence	Value
1)	1, 2	2, 1 or 1, 2	35
2)	1, 3	3, 1	30
3)	1, 4	1, 4	25
4)	1, 2, 3	-	-
5)	1, 2, 4	1, 2, 4	40

Optimal solution is 1, 2, 4

Try all possible permutations and if J can be produced in any of these permutations without violating dead line.

If J is a set of k jobs and  $s = i_1, i_2, \dots, i_k$ , such that  $d_{i_1} \leq d_{i_2} \leq \dots \leq d_{i_k}$  then j is a feasible solution.

**3.5 Optimal Merge Pattern:**

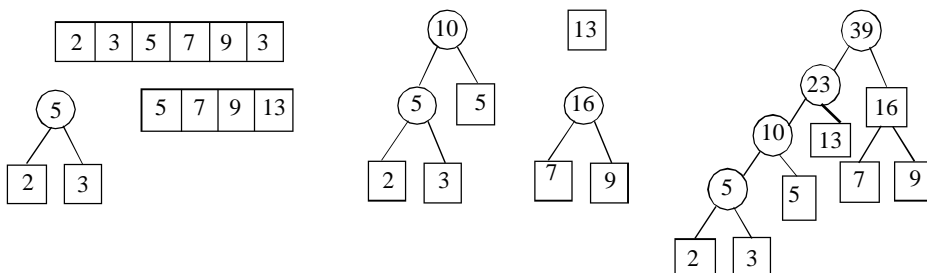
Merging of an 'n' record file and 'm' record file requires n + m records to move. At each step merge smallest size files. Two way merge pattern is represented by a merge tree. Leaf nodes (squares) represent the given files. File obtained by merging the files is a parent file(circles). Number in each node is length of file or number of records.

**Problem:** To merge n sorted files, at the cost of minimum comparisons.

**Objective:** To merge given field.

If  $d_i$  is the distance from root to external node for file  $f_i$ ,  $q_i$  is the length of  $f_i$ , then total number of record moves in a tree =  $\sum_{i=1}^n d_i q_i$  known as weighted external path length

Example:



NOTES

**Algorithm:**

Procedure Tree(l, n)

For l = l to n

{

    Call GETNODE(T)

    L CHILD(T) = LEAST(L);

    R CHILD(T) = LEAST (L);

        WEIGHT(T)

    WEIGHT(LCHILD(T)) + WEIGHT(RECHILD(T));

        Call INSERT(L, T);

    }

        return(LEAST(L))

    }

**ANALYSIS:**

Main loop is executing  $(n - 1)$  times. If L is in non decreasing order

LEAST(L) requires only  $O(l)$  and INSERT(L, T) can be done in  $O(n)$  times.

Total time taken =  $O(n^2)$

**3.6 Minimum Spanning Tree:**

Let  $G = (V, E)$  is an undirected connected graph. A sub graph  $T = (V, E)$  of  $G$  is a spanning tree iff  $T$  is a tree.

Any connected graph with  $n$  vertices must have at least  $n - 1$  edges and all connected graphs with  $n - 1$  edge are trees. Each edge in the graph is associated with a weight. Such a weighted graph is used for construction of a set of communication links at a minimum cost. Removal of any one of the links in the graph if it has cycles will result a spanning tree with minimum cost. The cost of minimum spanning tree is sum of costs of edges in that tree.

A greedy method is to obtain a minimum cost spanning tree. The next edge to include is chosen according to optimization criteria in the sum of costs of edges so far included.

- 1) If A is the set of edges in a spanning tree.

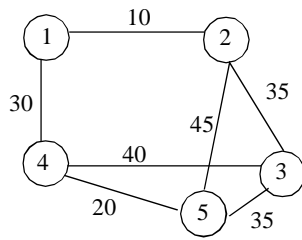
NOTES



2) The next edge  $(u, v)$  to be included in  $A$  is a minimum cost edge and  $A \cup (u, v)$  is also a tree.

**PRIMS algorithm:**

The edge  $(i, j)$  to be is such that  $i$  is a vertex already included in the tree,  $j$  is a vertex not included and cost of  $(i, j)$  must be minimum among all edges  $(k, l)$  such that  $k$  is in the tree and  $l$  is not in the tree.



Example.

Edge	Cost	ST
(1, 2)	10	
(1, 4)	30	
(4, 5)	20	
(5, 3)	35	

Algorithm:

Procedure PRIM( $E, COST, n, \text{mincost}, \text{int NEAR}(n) < [n] [n], i, j, k, l$ )

{  $(k, l) = \text{edge with mini cost}$

$\text{min cost} = \text{cost}(k, l)$

```

(T(1, 1), T(1, 2)) = (k, 1)
    for i = 1 to n
        {   if COST (i, l) < cost (i, k)
            then NEAR(i) = 1
        else
            NEAR(i) = k
        }
    }

    NEAR(k) = NEAR(l) = 0
    For j = 2 to n - 1
        {
If Near(j) = 0 find cost(j, NEAR(j)) which is mini
    T(i, l), T(i, 2) = (j, NEAR(j))
    NEAR(j) = 0
    For k = 1 to n
        {   If NEAR(k) != 0 and cost (k, NEAR(k)) >COST(k, j)
Then NEAR(k) = j;
        }
    }

```

Time complexity:

The total time required for the algorithm is  $O(n^2)$ .

**Krushkal's algorithm:**

If E is the set of all edges of G, determine an edge with minimum.

Cost(v, w) and delete from E. If the edge(v, w) does not create any cycle in the tree T, add(v, w) to T. Repeat the process until all edges are covered.

NOTES

**Algorithm:**

Procedure KRUSKAL(E, COST, n, T)

```
{
    construct a heap with edge costs
        I = 0
        Min cost = 0
        Parent (l, n) = -1
        While I < n - 1 and heap not empty
            Delete minimum cost edge (u, v)
            From heap;
            ADJUST heap;
            J = FIND(u)
            K = FIND(v)
            If j != k
        {
            I = I + 1;
            T(l, 1) = u;
            T(1, 2) = v;
            Mincost = mincost + cost(u, v)
            UNION (j, k)
        }
    }
}
```

Procedure FIND(i)

{

J = i

While PARENT(j) > 0

{

J = PARENT(j)

}

K = i;

While K != j

{

T = PARENT(k);

PARENT(k) = j;

K = T

}

Return(j)

}

Procedure UNION(i, j)

{

NOTES

```
x = PARENT(i) + PARENT(j)
  if PARENT(i) > PARENT(j)
    {
      PARENT(i) = j
      PARENT(j) = x
    }
```

Else

```
PARENT(j) = i
  PARENT(i) = x
}
```

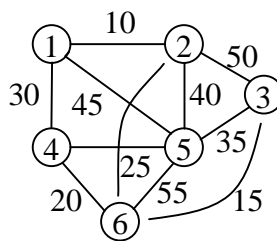
Time Complexity:

To develop a heap the time taken is  $O(n)$ . To adjust the heap it takes a time complexity of  $O(\log n)$ . The set of edges to be included in minimum cost spanning tree is  $n-1$ , hence it is  $O(n)$ .

Total time complexity is in the order of  $O(n \log n)$

Kruskals Alg

Example:



Edge	Cost	SF
1, 2	10	
3, 4	15	
4, 6	20	
2, 6	25	
1, 4	30	
3, 5	35	

NOTES

### 3.7 Single Source Shortest Paths:

Graphs may be used to represent highway structure in which vertices represent cities and edges represent sections of highway. The edges are assigned with weight which might be distance or time.

Now the problem of single source shortest path is to find shortest path between given source and destination. In the problem a directed graph  $G = (V, E)$ , a weighting function  $C(e)$  for the edges of  $G$  and source vertex  $V_0$  are given. So we must find the shortest paths from  $v_0$  to all remaining vertices of  $G$ .

Greedy algorithm for single source shortest path

**Step 1** : shortest paths are build one by one so that the problem becomes multisage solution problem

**Step 2** : For optimization, at every stage sum of the lengths all paths so far generated is minimum.

The greedy way to generate shortest paths from  $v_0$  to remaining vertices would be to generate these paths in non-decreasing order of path length.

The algorithm for the single source shortest paths using greedy method leads to a simple algorithm called Dijkstras algorithm.

Procedure SHORTEST – PATHS ( $v$ , COST, DIST,  $n$ )

```

{
  for i = 1 to n
    {
      S[i] = 0
      DIST [i] = COST[v] [i]
      {
        S[v] = 1
        DIST[v] = 0
        For num = 2 to n - 1
          {

```

Choose a vertex u such that

$$\text{DIST}(u) = \min\{\text{DIST}(w)\}.$$

$$S[w] = 0$$

$$S[u] = 1$$

For all w with s[w] = 0

$$S[u] = 1$$

For all w with S[w] = 0

{

$$\text{DIST}(w) = \min(\text{DIST}[w], \text{DIST}[u] + \text{COST}[u] [w])$$

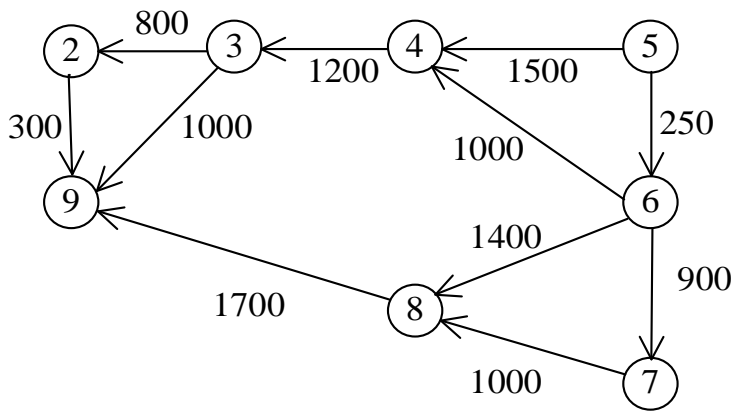
}

}

}

Example: Let us consider 8 vertex digraph shown below  
 For this graph the cost adjacency matrix is given below

*Design and Analysis of Algorithms*



NOTES

	1	2	3	4	5	6	7	8
1	0							
2	300	0						
3	1000	800	0					
4			1200	0				
5				1500	0	250		
6				1000		0	900	1400
7							0	1000
8	1700							0



S	Vertex Selected	1	2	3	4	5	6	7	8
5	6	$\infty$	$\infty$	$\infty$	1250	0	250	$\infty$	$\infty$
5, 6	7	$\infty$	$\infty$	$\infty$	1250	0	250	1150	1650
5, 6, 7	4	$\infty$	$\infty$	$\infty$	1250	0	250	1150	1650
5, 6, 7, 4	8	$\infty$	$\infty$	2450	1250	0	250	1150	1650
5, 6, 7, 4, 8	3	3350	$\infty$	2450	1250	0	250	1150	1650
5, 6, 7, 4, 8, 3	2	3350	3250	2450	1250	0	250	1150	1650

**Analysis of SHORTEST PATH Algorithm**

The edges on the shortest paths from a vertex  $v$  to all remaining vertices in a graph  $G$  form a spanning tree. This spanning tree is called shortest path spanning tree

**Time Complexity:**

The algorithm must examine each edge in the graph, since any of the edges could be in a shortest path. Therefore the minimum possible time is  $O(n)$ , if  $n$  edges are there. The cost of adjacent matrix takes a time of  $O(n^2)$  to determine which edges are in  $G$ . So the algorithm takes  $O(n^2)$ .

\*\*\*\*\*

## DYNAMIC PROGRAMMING

### 4.1 Introduction :

Dynamic programming is a stage-wise search method suitable for optimization problems whose solutions may be viewed as the result of a sequence of decisions. The most attractive property of this strategy is that during the search for a solution it avoids full enumeration by pruning early partial decision solutions that cannot possibly lead to optimal solution. In many practical situations, this strategy hits the optimal solution in a polynomial number of decision steps. However, in the worst case, such a strategy may end up performing full enumeration. It was first introduced by Richard Bellman in 1957.

Dynamic programming takes advantage of the duplication and arrange to solve each subproblem only once, saving the solution (in table or something) for later use. The underlying idea of dynamic programming is: avoid calculating the same stuff twice, usually by keeping a table of known results of subproblems. Unlike divide-and-conquer, which solves the subproblems top-down, a dynamic programming is a bottom-up technique.

Bottom-up means

- i. Start with the smallest subproblems.
- ii. Combining their solutions obtain the solutions to subproblems of increasing size.
- iii. Until we arrive at the solution of the original problem.

### 4.2 The Principle of Optimality

The dynamic programming relies on a principle of optimality. This principle states that in an optimal sequence of decisions or choices, each subsequence must also be optimal. For example, in matrix chain multiplication problem, not only the value we are interested in is optimal but all the other entries in the table are also represent optimal.

The principle can be related as follows: the optimal solution to a problem is a combination of optimal solutions to some of its subproblems.

The difficulty in turning the principle of optimality into an algorithm is that it is not usually obvious which subproblems are relevant to the problem under consideration.

The major difference between greedy method and dynamic programming is that in the greedy method only one decision sequence is ever generated. Where

## Design and Analysis of Algorithms

## NOTES

as in dynamic programming many decision sequences may be generated. Another important feature of dynamic programming is that optimal solutions to sub problems are retained so as to avoid recomputing their values.

The main advantage of this method was that it replaced an exponential time computation by a polynomial time computation. It is well suited to problems where a recursive algorithm would solve many of the sub problems repeatedly.

### 4.3 Developing a Dynamic Programming Algorithm

1. It is use full to tackle the problem by top down approach as if we were going to develop a recursive algorithm. Large problem is reduced to small problems'
2. A dictionary is defined for saving results to avoid repeated computation.
3. Based on number of sub problems and the number of edges in the sub problem graph , the complexity of the dynamic programming procedure can be analyzed by its relationship to depth first search on the sub problem graph.
4. Decide an appropriate data structure for the dictionary.
5. Find a simple order in which the dictionary entries can be computed.
6. Determine how to get the solution to the problem from the data in the dictionary.
7. Since the dictionary has data for all sub problems in the sub problem graph , usually only a small subset of the data is related to the final optimum solution.

Dynamic program model may be written as

$$F_n(R) = \max \{P_n(R) + F_{n-1}(R - R_n)\}$$

Where  $n=2,3,4,\dots$

$$F_0(R) = 0, F_1(R) = P_1(R)$$

### 4.4 Multistage Graph Problem:

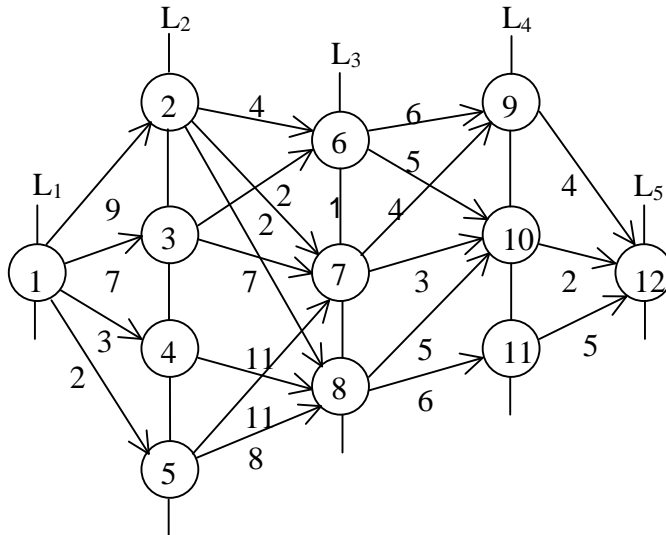
A multistage graph  $G = (V, E)$  is a directed graph in which the vertices are partitioned into two or more disjoint sets as shown in figure.

The multistage graph problem is to find the minimum cost path from a source to target. Let  $c(i, j)$  be the cost of edge  $\langle i, j \rangle$ . The cost of a path from source to target is the sum of the costs of edges on the path.

## Design and Analysis of Algorithms

### FORWARD APPROACH:

Dynamic programming formulation for a K-stage graph problem is obtained by first noticing that every source to target path is the result of a sequence of k-2 decisions. The  $i$ th decision involves determining which vertex in level  $i+1$ ,  $1 \leq i \leq k-2$  is to be on the path.



Let  $p(i,j)$  be a minimum cost path from vertex  $j$  in stage  $i$  to vertex target. Let  $\text{cost}(i,j)$  be the cost of its path, then  $\text{Cost}(i,j) = \min\{c(j,l) + \text{cost}(i+1,l)\}$  where  $\langle j,l \rangle$  is an intermediate edge, where  $c(j,l)$  is cost of edge with vertices  $(j,l)$  and  $\text{Cost}(i+1,l)$  is the cost to reach a node  $l$  in level  $i+1$ , the

- 1)  $\text{COST}(1,1) = \min\{9 + \text{COST}(2,2), 7 + \text{COST}(2,3), 3 + \text{COST}(2,4), 2 + \text{COST}(2,5)\} = 16$
- 2)  $\text{COST}(2,2) = \min\{4 + \text{COST}(3,6), 2 + \text{COST}(3,7), 1 + \text{COST}(3,8)\} = 7$   
 $\text{COST}(2,3) = \min\{2 + \text{COST}(3,6), 2 + \text{COST}(3,7)\} = 12$   
 $\text{COST}(2,4) = 11 + \text{COST}(3,8) = 18$   
 $\text{COST}(2,5) = \min\{11 + \text{COST}(3,7), 8 + \text{COST}(3,8)\} = 16$
- 3)  $\text{COST}(3,6) = \min\{6 + \text{COST}(4,9), 5 + \text{COST}(4,10)\} = 7$   
 $\text{COST}(3,7) = \min\{4 + \text{COST}(4,9), 3 + \text{COST}(4,10)\} = 5$   
 $\text{COST}(3,8) = \min\{5 + \text{COST}(4,10), 6 + \text{COST}(4,11)\} = 7$
- 4)  $\text{COST}(4,9) = 4 + \text{COST}(5,12) = 4$   
 $\text{COST}(4,10) = 2 + \text{COST}(5,12) = 2$   
 $\text{COST}(3,8) = 5 + \text{COST}(5,12) = 5$

Minimum of 4<sup>th</sup> stage,  $\text{Min}(4) = \text{cost}(4,10) = 2$

Minimum of 3<sup>rd</sup> stage,  $\text{Min}(3) = \text{cost}(3,7) = 5$

### NOTES

**Design and Analysis of Algorithms****NOTES**

Minimum of 2<sup>nd</sup> stage,  $\text{Min}(2) = \text{cost}(2, 2) = 7$

Minimum of 1<sup>st</sup> stage,  $\text{Min}(1) = \text{cost}(1, 1) = 16$

Therefore minimum path cost = 16 and the path is  $1 \text{ ® } 2 \text{ ® } 7 \text{ ® } 10 \text{ ® } 12$

Procedure FGRAPH(E, k, n, p)

```
{
  float COST(n), integer D(n - 1), p(k), r, j, k, n;
  Cost(n) = 0.0;
  For j = n - 1 down to 1
    {
      let r be a vertex and E be set of edges
      if  $\langle j, r \rangle \in E$  and  $c(j, r) + \text{COST}(r)$  is minimum
        COST(j) =  $c(j, r) + \text{COST}(r)$ 
        D(j) = r
    }
  P(1) = 1; p(k) = n
  For j = to 2 k - 1
    {
      P(j) = D(p(j - 1))
    }
}
```

**Time Complexity :**

The complexity of above algorithm is  $O(n)$ . If the graph G has E edges and V vertices, then the time taken for the algorithm is  $O(V + E)$ .

$$T(n) = O(n)$$

**BACKWARD APPROACH :**

The multistage graph problem can also be solved using the backward approach. Let  $B_p(i, j)$  be a minimum cost path from vertex source to a vertex j in stage i. Let  $B_{\text{cost}}(i, j)$  be the cost of  $B_p(i, j)$  from the backward approach we obtain

$$B_{\text{cost}}(i, j) = \min_{\langle l, j \rangle \in E} \{B_{\text{cost}}(i-1, l) + c(l, j)\}$$

Procedure BGRAPH(E, k, n, p)

**Design and Analysis of Algorithms**

```

{
  BCOST(1) = 0
  For j = 2 to n
    {
      Let r be a vertex and E set of edges
      If  $\langle r, j \rangle \in E$  and  $BCOST(r) + c[r, j]$  is minimum.
       $BCOST(j) = BCOST(r) + c[r, j]$ 
       $D(j) = r$ 
    }
  P(1) = 1; P(k) = n
  For j = k - 1 to 2
    {
       $P(j) = D(p(j + 1))$ 
    }
}

```

**NOTES****Backward approach**

For the previous example shown in forward approach,

- $BCOST(5, 12) = \min \{BCOST(4, 9) + 4, BACOST(4, 10) + 2, BACOST(4, 11) + 5\}$
- $BCOST(4, 9) = \min \{BCOST(3, 6) + 6, BACOST(3, 7) + 4\}$   
 $BCOST(4, 10) = \min \{BCOST(3, 7) + 3, BACOST(3, 8) + 5, BACOST(3, 6) + 5\}$   
 $BCOST(4, 11) = \min \{BCOST(3, 8) + 6\}$
- $BCOST(3, 6) = \min \{BCOST(2, 2) + 4, BACOST(2, 3) + 2, BCOST(3, 7)\}$   
 $= \min \{BCOST(2, 2) + 2, BACOST(2, 3) + 7, BACOST(2, 5) + 11\}$   
 $BCOST(3, 8) = \min \{BCOST(2, 2) + 1, BACOST(2, 4) + 11, BACOST(2, 5) + 8\}$
- $BCOST(2, 2) = \min \{BCOST(1, 1) + 9\} = 9$   
 Since  $BCOST(1, 1) = 0$   
 $BCOST(2, 3) = 7$   
 $BCOST(2, 4) = 3$   
 $BCOST(2, 5) = 2$

Substituting these values in the above equations and going back

**Design and Analysis of Algorithms**

**NOTES**

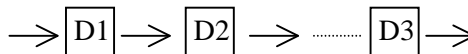
- BCOST(3, 8) = 10
- BCOST(3, 7) = 11
- BCOST(3, 6) = 9
- BCOST(4, 9) = 15
- BCOST(4, 10) = 14
- BCOST(4, 11) = 16
- BCOST(5, 12) = 16

This algorithm also has same time complexity as that of forward approach

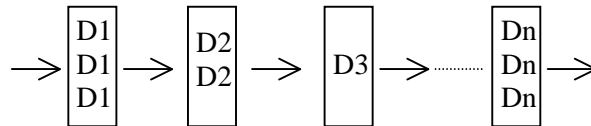
**4.5 Reliability Design:**

Reliability design problem is concerned with the design of a system using several devices connected in series with high reliability. Let  $r_i$  be the reliability of device  $D_i$ . Then the reliability of entire system is  $\prod r_i$ . Even the reliability of individual device is good the reliability of entire system may not be good.

Example: If  $n = 10, r_i = 0.99$   
 $\prod r_i = 0.904$



Hence it is desirable to use multiple copies of devices in parallel in each stage to improve the reliability.



If stage  $i$  contains  $m_i$  copies of device  $D_i$   
 then the probability that all  $m_i$  have a malfunction is  $(1 - r_i)^{m_i}$   
 Hence reliability of stage  $i$  is  $1 - (1 - r_i)^{m_i}$

Device duplication is to maximize reliability. This maximization is carried under a cost constraint.

If  $c_i$  is the cost of device  $i$  and  $C$  is maximum allowable cost., then

$$\text{Maximize } \prod \phi_i(m_i) \text{ subjected to } \sum c_i m_i < C$$

Where  $\phi_i$  is reliability  $1 \leq i \leq \text{nof stage } i$ . Once a value  $m_n$  has chosen the remaining decisions must be such that to use remaining funds in  $C - C_n m_n$  in an optimal way. We can generalize that for any  $F_i(x) = \max (1 \leq m_i \leq u_i)$   
 $\{ \phi_i(m_i) f_{i-1}(c - c_i m_i) \}$

## Design and Analysis of Algorithms

### Example:

Design a 3 stage system with devices types  $D_1, D_2, D_3$ . The costs and reliabilities of the devices are 30, 15, 20 and 0.9, 0.8, 0.5 respectively. Maximum number of devices available are  $D_1 = 2, D_2 = 3$  and  $D_3 = 3$ . The cost of system must not exceed

### Solution:

If stage  $i$  has  $m_i$  devices of type  $i$  in parallel then  $\epsilon_i(m_i) = 1 - (1 - r_i)^{m_i}$

Let  $S_j^i$  represent all combinations obtainable from  $S^{i-1}$  by choosing  $m_i = j$

$$S_1^1 = \{0.9, 30\}$$

$$S_1^2 = \{(0.9, 30); (0.99, 0.60)\}$$

$$S_1^3 = \{(0.72, 45); (0.792, 75)\}$$

$$S_1^4 = \{(0.864, 60); (0.9504, 90)\}$$

The triple  $(0.9504, 90)$  is eliminated since we cannot insert  $D_3$

$$S_3^2 = \{(0.8928, 75)\}$$

$$\setminus S^2 = \{(0.72, 45), (0.864, 60), (0.8928, 75)\}$$

$$S_1^3 = \{(0.36, 65), (0.432, 80), (0.4464, 95)\}$$

$$S_2^3 = \{(0.54, 85), (0.648, 100)\}$$

$$S_3^3 = \{(0.63, 105)\}$$

$$\setminus S^3 = \{(0.36, 65), (0.432, 80), (0.54, 85), (0.648, 100)\}$$

The best design has a reliability of 0.648 and cost of 100

$$\setminus m_1 = 1, m_2 = 2 \text{ and } m_3 = 2$$

### 4.6 All pairs shortest path:

Let  $G = (V, E)$  be a directed graph with  $n$  vertices. Let  $c$  be a cost adjacency matrix for  $G$  such that  $c(i, i) = 0$  and  $c(i, j)$ , is the length of edge  $(i, j)$  if  $\langle i, j \rangle \in E$  and  $C(i, j) = \infty$  if  $\langle i, j \rangle \notin E$ . The all pairs shortest path problem is to determine a matrix  $a$  such that  $A(i, j)$  is the length of shortest path from  $i$  to  $j$ .

The solution for the problem can be obtained from

$$A(i, j) = \min\{ \min(1 \leq k \leq n) \{A^{k-1}(i, k) + A^{k-1}(k, j)\} + \text{cost}(i, j) \}$$

Where  $A^k(i, j)$  is length of shortest path from  $i$  to  $j$  through no vertex has index greater than  $k$ .

### Algorithm:

For  $l = 1$  to  $n$

{

## NOTES



**Design and Analysis of Algorithms**

```

for j ← 1 to n
    {
        A(i, j) ← cost(i, j)
    }
}

For k ← 1 to n
    {
        For l ← 1 to n
            {
                For J ← 1 to n
                    A(i, j) = min{A(i, j), (A(i, k) +
A(k, j))}
            }
        }
    }

```

NOTES

Example: Let us consider the graph whose cost matrix is shown.

		1	2	3
A(0)	1	-	4	11
	2	6	-	2
	3	3	∞	1

		1	2	3
A(1)	1	-	4	11
	2	6	-	2
	3	3	7	-

$$(3, 2) = (3, 1) + (1, 2)$$

**Design and Analysis of Algorithms****NOTES**

		1	2	3
A(2)	1	-	4	6
	2	6	-	2
	3	3	7	0

$$(1, 1) = (2, 3) + (3, 1)$$

The time complexity of APSP is  $O(n^3)$ .

**4.7 Traveling sales person problem:**

Let  $G = (V, E)$  be a directed graph with edge costs  $c_{ij}$ .

$c_{ij} > 0$  for all  $i$  and  $j$  if  $(i, j) \in E$ ,  $c_{ij} = \infty$  if  $(i, j) \notin E$ . A tour of  $G$  is a directed cycle that includes every vertex in  $V$ . The cost of tour is the sum of the cost of edges on the tour and must be minimum.

Let us assume the tour starts from vertex 1 and ends at vertex 1. Every tour consists of an edge  $(1, k)$  for some  $k \in V - \{1\}$ . The path from  $k$  to 1 goes through each vertex  $v - \{1, k\}$ .

Let  $g(i, S)$  be the length of shortest path starting from  $i$  going through vertices in  $S$  and terminating at vertex 1.

$$g(1, V - \{1\}) = \min\{c_{1k} + g(k, V - \{1, k\})\}$$

Where  $v$  is total number of vertices.

In general.

$$g(i, S) = \min\{c_{ij} + g(j, S - \{j\})\}$$

$$\text{where } S = V - \{i\}$$

Example: Let us consider a graph with five nodes. Cost matrix of the graph is given below.

		1	2	3	4	5
1		-	10	15	20	15
2		5	-	9	10	15
3		6	13	-	12	7
4		8	8	9	-	4
5		2	5	7	10	-

**Solution:**

When  $|S| = \emptyset$

$$g(2, \emptyset) = c_{21} = 5$$

**Design and Analysis of Algorithms****NOTES**

$$g(3, \phi) = c_{31} = 6$$

$$g(4, \phi) = c_{41} = 8$$

$$g(5, \phi) = c_{51} = 2$$

When  $|s| = 1$

$$g(2, \{3\}) = c_{23} + g(3, \phi) = 9 + 6 = 15$$

$$g(2, \{4\}) = c_{24} + g(4, \phi) = 10 + 8 = 18$$

$$g(2, \{5\}) = c_{25} + g(5, \phi) = 15 + 2 = 17$$

$$g(3, \{2\}) = 18, g(4\{2\}) = 13$$

$$g(3, \{4\}) = 20, g(4\{3\}) = 15$$

$$g(3, \{5\}) = 9, g(4\{5\}) = 6$$

$$g(5, \{2\}) = 10 = c_{52} + g(2, \phi) = 5 + 5$$

$$g(5, \{3\}) = 13 = c_{53} + g(3, \phi) = 7 + 6$$

$$g(5, \{4\}) = 18 = c_{54} + g(4, \phi) = 10 + 8$$

When  $|s| = 2$

$$g(2, \{3, 4\}) = \min(c_{23} + g(3, \{4\}) = 29 \text{ or } c_{24} + g(4, \{3\}) = 25) = 25$$

$$g(2, \{3, 5\}) = \min(c_{23} + g(3, \{5\}) = 18 \text{ or } c_{25} + g(5, \{3\}) = 28) = 18$$

$$g(2, \{4, 5\}) = 16 \quad g(3, \{2, 5\}) = 17$$

$$g(3, \{2, 4\}) = 25 \quad g(3, \{4, 5\}) = 23$$

$$g(5, \{2, 4\}) = 23 \quad g(4, \{2, 3\}) = 23$$

$$g(4, \{3, 5\}) = 17 \quad g(5, \{3, 4\}) = 25$$

$$g(4, \{2, 5\}) = 14 \quad g(5, \{2, 3\}) = 20$$

When  $|s| = 3$

$$g(2, \{3, 4, 5\}) = \min \quad 9 + 23, 10 + 17, 15 + 25 = 27$$

$$g(3, \{2, 4, 5\}) = \min \quad 13 + 16, 12 + 14, 7 + 23 = 26$$

$$g(5, \{2, 3, 4\}) = \min \quad 5 + 25, 7 + 25, 10 + 23 = 30$$

$$g(4, \{2, 3, 5\}) = \min \quad 8 + 18, 9 + 17, 4 + 20 = 24$$

When  $|s| = 4$

$$g(1, \{2, 3, 4, 5\}) = \min \{10 + 27, 15 + 26, 20 + 24, 15 + 30\} = 37$$

minimum cost path

**Design and Analysis of Algorithms**

vertices	1	2	4	5	3
cost	10	10	4	7	6

**NOTES****Time complexity**

N be the number of  $g(i, s)$ 's to be computed. For each of S there are  $(n - 1)$  choices for i.

Number of sets s of size k (excluding 1 and i) =  $\binom{n-2}{k}$

$$N = \sum_{k=0}^{n-2} (n-1) \binom{n-2}{k} = (n-1)2^{n-2}$$

Therefore time complexity of the algorithm is  $\theta(n^2 2^n)$  and space complexity  $O(n 2^n)$

**4.8 Optimal Binary search Tree:**

A BST is a binary tree, either empty or each node is an identifier and

- i) All identifiers in LST are less than identifier at root node.
- ii) All identifiers in RST are greater than identifier at root node.

To find whether an identifier X is present or not.

- 1) X is compared with root
- 2) If X is less than identifier at root, then search continues in left sub tree.
- 3) If X is greater than id at root, then search continues in right sub tree.

Procedure SEARCH(T, X, I)

```
{
    i ← T
    While i ≠ 0
        {
case
        : X < IDENT(i) : i ← LCHILD(i)
        : X < IDENT(i) : return
        : X > IDENT(i) : i ← RCHILD(i)
endcase
        }
}
```

**Design and Analysis of Algorithms**

**NOTES**

}

Let us assume that given set of identifiers is  $\{a_1, a_2, \dots, a_n\}$  and let  $P(i)$  be the probability of successful search of an identifier  $a_i$ , and let  $Q(i)$  be the probability unsuccessful search of  $a_i$ .

$$\text{Therefore } \sum p(i) + \sum Q(i) = 1$$

To obtain a cost function for binary tree, it is useful to add a fictions node in place of every empty node.

If  $n$  identifiers are there, there will be  $n$  internal nodes and  $n + 1$  external nodes. Every external node represents an unsuccessful search.

The identifiers not in binary tree are partitioned into  $n + 1$  equivalence classes  $E_i$ .

- $E_0$  contains identifier  $X < a_1$
- $E_1$  contains identifier  $a_1 < X < a_2$
- $E_2$  contains identifier  $a_2 < X < a_{i+1}$

If a failure node for  $E_i$  is at level  $l$  then only  $l - 1$  iterations are made.

Cost of failure node is

$$Q(i) * (\text{level}(E_i) - 1)$$

$$\text{Cost of binary search tree} = \sum p(i) * \text{level}(a_i) + \sum Q(i) * (\text{level}(E_i) - 1)$$

Where  $P(i)$  is probability of successful search

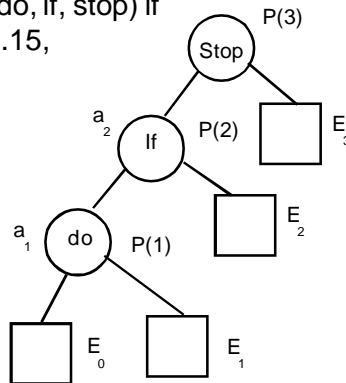
$Q(i)$  is probability of unsuccessful search

Example: Find OBST for a set  $(a_1, a_2, a_3) = (\text{do}, \text{if}, \text{stop})$  if  $p(1) = 0.5, P(2) = 0.1, P(3) = 0.05$  and  $q(0) = 0.15, q(1) = 0.1, q(2) = 0.05, q(3) = 0.05$

The possible binary search trees are

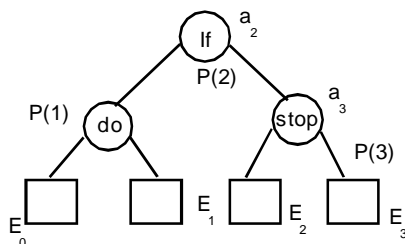
**Cost (tree a) :**

$$\begin{aligned} \text{Cost} &= 0.05 + 0.1 * 2 + 0.5 * 3 + 0.05 * 2 + 0.1 * 3 + 0.15 * 3 \\ &= 0.05 + 0.2 + 1.5 + 0.05 + 0.1 + 0.3 + 0.45 \\ &= 1.75 + 0.9 = 2.65 \end{aligned}$$



**Cost(tree b) :**

$$\begin{aligned} \text{Cost} &= 0.1 + 0.05 * 2 + 0.5 * 2 \\ &+ (0.15 + 0.1 + 0.05 + 0.05) * 2 \\ &= 1.2 + 0.7 = 1.9 \end{aligned}$$



**Design and Analysis of Algorithms**

**Cost (tree c) :**

$$\begin{aligned} \text{Cost} &:= 0.5 + 0.1 \cdot 2 + 0.05 \cdot 3 \\ &\quad + 0.15 + 0.1 \cdot 2 + 0.05 \cdot 3 + \\ &\quad 0.05 \cdot 3 \\ &= 0.5 + 0.2 + 0.15 + 0.15 + \\ &\quad 0.2 + 0.15 + 0.15 = 1.5 \end{aligned}$$

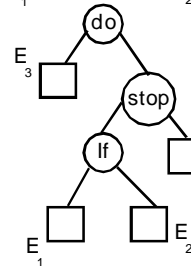
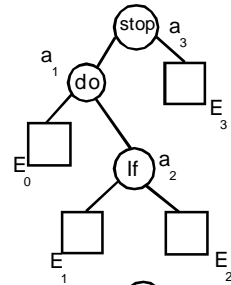
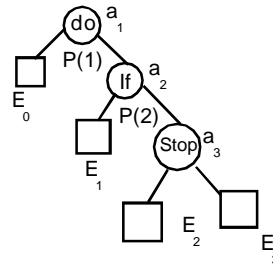
**Cost(tree d) :**

$$\begin{aligned} \text{Cost} &:= 0.05 + 0.5 \cdot 2 + 0.1 \cdot 3 + 0.05 + 0.15 \cdot \\ &\quad 2 + 0.1 \cdot 3 + 0.05 \cdot 3 = 1.35 + 0.8 = 2.15 \end{aligned}$$

**Cost(tree e):**

$$\begin{aligned} \text{Cost} &:= 0.5 + 0.05 \cdot 2 + 0.15 + 0.05 \cdot 2 + 0.1 \cdot 3 \\ &\quad + 0.05 \cdot 3 + 0.9 \\ &= 0.9 + 0.25 + 0.45 = 1.6 \end{aligned}$$

Hence OBST is the tree in figure C.



**NOTES**

**Analysis :**

If  $a_k$  is the root node, all the nodes  $a_1, a_3, \dots, a_{k-1}$  and  $E_0, E_1, \dots, E_{k-1}$  will lie in left sub tree L and remaining in right sub tree R.

$$\therefore \text{Cost}(L) = \sum_{1 \leq i \leq k} p(i) \cdot \text{level}(a_i) + \sum_{0 \leq i \leq k-1} Q(i) \cdot (\text{level}(E_i) - 1)$$

$$\text{Cost}(R) = \sum_{k \leq i \leq n} p(i) \cdot \text{level}(a_i) + \sum_{k \leq i \leq n} Q(i) \cdot (\text{level}(E_i) - 1)$$

$$\text{Cost of tree} = p(k) + \text{cost}(L) + \text{cost}(R) + w(0, k - 1) + w(k, n)$$

$$\text{Where } w(i, j) = Q(i) + \frac{i}{\sum_{l=i+1}^j (Q(l) + p(l))}$$

$$\text{Where } w(0, k - 1) = Q(0) + \sum_{l=1}^{k-1} (Q(l) + P(l))$$

If T is optimal then cost must be minimum. K must be chosen such that  $P(k) + c(0, k - 1) + c(k, n) + w(0, k - 1) + w(k, n)$  is minimum

Hence for  $c(0, n) = \min_{1 \leq k \leq n} \{c(0, k - 1) + c(k, n) + p(k) + w(0, k - 1) + w(k, n)\}$

In general

$$C(i, j) = \min_{1 \leq k \leq j} \{c(i, k - 1) + c(k, j) + p(k) + w(i, k - 1) + w(k, j)\}$$

$$= w(i, j) + \min_{1 \leq k \leq j} \{c(i, k - 1) + c(k, j)\}$$

**Design and Analysis of Algorithms**

Example: Let there are 4 identifiers, so  $n = 4$  and their probabilities are

$$P(1 : 4) = (3, 3, 1, 1)$$

$$Q(0 : 4) = (2, 3, 1, 1, 1)$$

$$\text{Initially } w(\dot{i}, \dot{i}) = Q(i), c(\dot{i}, \dot{i}) = 0, R(\dot{i}, \dot{i}) = 0$$

$$\text{We have } W(\dot{i}, j) = P(j) + Q(j) + w(\dot{i}, j - 1)$$

$$\backslash W(0, 1) = P(1) + Q(1) + w(0, 0) = 8$$

$$C(0, 1) = W(0, 1) + \min\{c(0, 0) + c(1, 1)\} = 8$$

$$R(0, 1) = 1$$

$$W(1, 2) = P(2) + Q(2) + w(1, 1) = 7$$

$$C(1, 2) = W(1, 2) + \min\{c(1, 1) + c(2, 2)\} = 7$$

$$R(0, 2) = 2$$

$$W(i, j) = P(j) + Q(j) + w(lj - 1)$$

$$W(2, 3) = P(3) + Q(3) + w(2, 2) = 3$$

$$C(2, 3) = W(2, 3) + \min\{c(2, 2) + c(3, 3)\} = 3$$

$$R(2, 3) = 3$$

$$W(l, j) = P(j) + Q(j) + w(lj - 1)$$

$$W(3, 4) = P(4) + Q(4) + w(3, 3) = 3$$

$$C(3, 4) = W(3, 4) + \min\{c(3, 3) + c(4, 4)\} = 3$$

$$R(3, 4) = 4$$

$$W(0, 2) = P(2) + Q(2) + W(0, 1) = 3 + 13 + 8 = 12$$

$$C(0, 2) = w(0, 2) + \min\{C(0, 1) + C(2, 2) \text{ or } c(0, 0) + c(1, 2)\}$$

$$= 14 + \text{MIN}\{8 + 0\} \text{ or } \{0 + 7\} = 12 + 7 = 19$$

$$w(2, 4) = w(2, 4) + \min\{c(2, 3) + c(3, 4) = 6 \text{ or } c(2, 2) + (3, 4) = 3\} = 8$$

$$w(1, 3) = P(3) + P(3) + W(1, 2) = 2 + 7 = 9$$

$$C(1, 3) = W(1, 3) + \min\{C(1, 1) + C(2, 3) = 3 \text{ or } C(1, 2) + C(3, 3) = 7\} = 12$$

$$R(1, 3) = 2$$

$$W(0, 3) = P(3) + Q(3) + w(0, 2) = 14$$

$$C(0, 3) = w(0, 3) + \min\{c(0, 1) + c(2, 3) \text{ C}(0, 2) + c(3, 3)\}$$

$$= 14 + \min \left\{ \begin{array}{l} 11 \\ 19 \end{array} \right\} = 25$$

NOTES

**Design and Analysis of Algorithms**

First solve all  $c(i, j)$  such that  $j - i = 1$  next compute  $c(i, j)$  such that  $j - i = 2$  and so on during the computation record  $R(i, j)$  root of tree  $T_{ij}$ . By observation we have  $w(i, j) = P(j) + Q(j) + W(i, j - 1)$

Row Col	0	1	2	3	4
0	2, 0, 0	3, 0, 0	1, 0, 0	1, 0, 0	1, 0, 0
$j-i=1$	8, 8, 1	7, 7, 2	3, 3, 3	3, 3, 4	
2	12, 19, 1	9, 12, 2	5, 8, 3		
3	16, 25, 2	11, 19, 2			
4	16, 32, 2				

$C(0,4) = 32$  is minimum

The tree with root as 2 node is OBST.

**Algorithm:**

Procedure OBST(P, Q, n)

{Real P(1 : n), Q(0 : n), C(0 : n, 0 : n),

W(0 : n, 0 : n)

Int R (0 : n, 0 : n)

For I = 0 to n - 1

{

{w(i, i), R(i, i), c(i, i) ← (Q(i), 0, 0)

(W(i, i + 1), R(i, i + 1), c(i, i + 1) ← (Q(i) + Q(i + 1) + P(i + 1), i + 1, Q(i) + Q(i + 1) + P(i + 1))

}

(w(n, n), R(n, n), C(n, n) ← Q(n), 0, 0)

For m : 2 to n

{

For i : 0 to n - m

{

j = i + m

w(i, j) = W(i, j - 1) + P(j) + Q(i)

k = i

**NOTES**



**Design and Analysis of Algorithms**

**NOTES**

where  $R(i, j - 1) \in 1 \in R(i + 1, j)$  that minimize  $c(i, j)$

$$C(i, j) = w(i, j) + c(i, k - 1) + c(k, j)$$

$$R(i, j) = k$$

}

}

**Time Complexity :**

Each  $C(i, j)$  can be computed in the time  $O(m)$

There are  $(n - m + 1) C(i, j)$ s

$$\text{Total time} = O(m) \cdot O(n - m + 1)$$

For evaluating

All  $c(i, j) = O(nm - m^2 + m)$  with  $(j - i) = m$

$$\begin{aligned} \text{Total Time} &= \sum_{1 < i < n} O(nm - m^2) = (n^3) \\ &= O(n^3) \end{aligned}$$

**4.9 0/1 Knapsack Problem**

Given  $n$  objects and a knapsack, object  $i$  has a weight  $w_i$  and knapsack has capacity of  $M$ . If a fraction  $x_i, 0 \leq x_i \leq 1$  of object  $i$  is placed into knapsack then a profit  $p_i x_i$  is earned.

Problem: To fill the knapsack so that total profit is maximum and satisfies two conditions.

- 1) Objective  $\sum_{1 \leq i \leq n} p_i x_i = \text{maximum}$       2) Feasibility  $\sum_{1 \leq i \leq n} w_i x_i \leq m$

**Types of knapsack :**

1. Normal knapsack: in normal knapsack, a fraction of the last object is included to maximize the profit.

$$\text{i.e., } 0 \leq x \leq 1 \text{ so that } \sum_i w_i x_i = M$$

2. 0/1 knapsack: In 0/1 knapsack, if the last object is able to insert completely, then only it is selected otherwise it is not selected.

$$\text{i.e., } x = 1/0 \text{ so that } \sum_i w_i x_i = M$$

According to dynamic programming, solution for 0/1 knapsack is  $F_n(M) = \max \{F_{n-1}(M) + F_{n-1}(M - w_n) + P_n\}$

Where  $M$  is the size of the knapsack

When  $x_n = 1$  then the size of the bag is reduced by  $w_n$  which is the weight

**Design and Analysis of Algorithms****NOTES**

of the  $n$ th item. As we are placing the  $n$ th item we should add the profit for the last item. Similarly we find for  $F_{n-1}(M)$  and so on upto  $F_1(M)$

$$S_1^i = \{ (P_i, w_i) / (P - P_i, w - w_i) \hat{=} S^{i-1} \}$$

If we try to remove the  $i$ th item then the profit of  $i$ th item is reduced from the total profit and weight of  $i$ th item is removed from the total weight which belongs to the profit and weight of  $(i-1)$ th item.

$$S_0^0 = \{0,0\}$$

$$S_1^i = S^{i-1} + (P_i, w_i) \quad \text{addition}$$

$$S^i = S_1^i + S^{i-1}$$

Ex: Let us consider a knapsack problem in which  $n=3$ ,  $M=6$ ,  $(W_1, W_2, W_3) = 2, 3, 4$  and  $(P_1, P_2, P_3) = 1, 2, 5$

$$S_0^0 = \{0,0\}$$

$$S_1^i = S^{i-1} + (P_i, w_i)$$

$$\begin{aligned} S_1^1 &= S^0 + (P_1, w_1) \\ &= \{(0,0)\} + \{(1,2)\} = \{(1,2)\} \end{aligned}$$

$$S^1 = \{(0,0), (1,2)\}$$

$$\begin{aligned} S_1^2 &= S^1 + (P_2, w_2) \\ &= \{(0,0), (1,2)\} + \{(2,3)\} \\ &= \{(2,3), (3,5)\} \end{aligned}$$

$$\begin{aligned} S^2 &= S^1 + S_1^2 \\ &= \{(0,0), (1,2), (2,3), (3,5)\} + \{(5,4)\} \\ &= \{(5,4), (6,6), (7,7), (8,9)\} \end{aligned}$$

$$\begin{aligned} S^3 &= S^2 + S_1^3 \\ &= \{(0,0), (1,2), (2,3), (3,5), (5,4), (6,6), (7,7), (8,9)\} \end{aligned}$$

Purging Rule(Dominance rule):

If one of  $S^{i-1}$  and  $S_1^i$  has a pair of  $(P_j, W_j)$  and other has a pair  $(P_k, W_k)$  and  $P_j \leq P_k$  while  $W_j \geq W_k$  then the pair  $(P_j, W_j)$  is discarded.

$$S^3 = \{(0,0), (1,2), (2,3), (5,4), (6,6), (7,7), (8,9)\}$$

**Design and Analysis of Algorithms****NOTES**

After applying purge rule, we will check following condition in order to find solution. If  $(P_i, W_i) \in S^n$  and  $(P_i, W_i) \in S^{n-1}$

Then  $x_n = 1$

Otherwise  $x_n = 0$

$(6,6) \in S^3$

and  $(6,6) \notin S^2$

$x_3 = 1$

$(6,6) - (5,4) = (1,2) \in S^2$

$(6,6) \notin S^1$  which is false

$x_2 = 0$

$(1,2) \in S^1$

$(1,2) \in S^0$  which is true

$x_1 = 1$

Maximum profit is  $\sum P_i X_i = P_1 X_1 + P_2 X_2 + P_3 X_3 = 1 \times 1 + 2 \times 0 + 5 \times 1 = 6$

0/1 Knapsack Problem Algorithm:

Procedure DKP(p, w, n, M)

{

$S^0 = (0,0)$

For(I=1, I++; I=n-1)

{

$S^i = \{(p_i, w_i) \mid (P - p_i, W - w_i) \in S^{i-1}\}$

$S^i = \text{MERGE PURGE}(S^{i-1}, S^i_1)$

}

$(P_x, W_x) = \text{last tuple in } S^{n-1}$

$(P_y, W_y) = (P_i + P_n, W_i + W_n)$  where  $W_1$  is the largest  $W$  in any tuple in  $S^{n-1}$

such that  $W + W_n \leq M$

If  $P_x > P_y$  then  $X_n = 0$

Else

$X_n = 1$

}

**Design and Analysis of Algorithms****NOTES**

Time complexity of knapsack problem is  $O(n)$ . Time complexity of 0/1 knapsack problem is  $O(2^{n/2})$ .

**4.10. Matrix Chain Multiplication:**

An  $n \times m$  matrix  $A = [a(i,j)]$  is a 2D array

$$A = \begin{bmatrix} a(1,1) & \dots & a(1,m) \\ a(n,1) & \dots & a(n,m) \end{bmatrix}$$

and  $m \times r$  matrix  $B = [b(i,j)]$

$$B = \begin{bmatrix} b(1,1) & \dots & b(1,r) \\ b(m,1) & \dots & b(m,r) \end{bmatrix}$$

$C = AB$  of  $n \times r$  size

$$\text{where } C[i, j] = \sum_{k=1}^m a[i, k] b[k, j]$$

for  $1 \leq i \leq n$  and  $1 \leq j \leq r$

**Draw back of above algorithm**

- ✱ If  $AB$  is defined,  $BA$  may not be defined
- ✱  $AB \neq BA$
- ✱ Multiplication is recursively defined

$$A_1 A_2 A_3 \dots A_{5-1} A_5$$

$$= A_1 (A_2 (A_3 \dots (A_{5-1} A_5)))$$

- ✱ Matrix multiplication is associative

$$A_1 A_2 A_3 = (A_1 A_2) A_3 = A_1 (A_2 A_3) \text{ so parenthe nization does not change result.}$$

**Complexity of direct Matrix Multiplication**

The  $C$  has  $n \times r$  entries and each entry takes  $O(m)$  time to compute so the total procedure takes  $O(nmr)$  time.

**4.10.1 Direct matrix multiplication on of 3 matrices:**

Given a  $p \times q$  matrix  $A$ , a  $q \times r$  matrix  $B$  and a  $r \times s$  matrix  $C$  then  $ABC$  can be computed in two ways  $(AB)C$  and  $A(BC)$ .

The number of multiplications needed are:

$$\text{mult}+[(AB)C] = pqr + prs$$

$$\text{mult}+[A(BC)] = qrs + pqs$$

$$\text{mult}+[(AB)C] \neq \text{mult}+[A(BC)]$$

→ While multiplication, sequence is important

## Design and Analysis of Algorithms

### 4.11 Chain Matrix Multiplication:

Given dimensions  $P_0, P_1, \dots, P_n$  corresponding to matrix sequence  $A_1, A_2, \dots, A_n$  where  $A_i$  has dimension  $P_{i-1} \times P_i$  determine the multiplication sequence that minimizes the number of Scalar multiplications in computing  $A_1 A_2 \dots A_n$ .

$$\begin{aligned} A_1 A_2 A_3 A_4 &= (A_1 A_2) (A_3 A_4) \\ &= A_1 (A_2 (A_3 A_4)) = \\ &= A_1 ((A_2 A_3) A_4) = \\ &= ((A_1 A_2) A_3) (A_4) = \\ &= (A_1 (A_2 A_3)) (A_4) \end{aligned}$$

Time taken for above algorithm is  $\Omega(4^n / n^{3/2})$

To reduce the time complexity we can use dynamic programming strategy

#### Algorithm for MCM using DP

**Step 1 :** Find the structure of an optimal solution

Decompose the problem into subproblems. For each pair  $1 \leq i \leq j \leq n$ , determine the multiplication sequence for  $A_i, \dots, A_j = A_i A_{i+1} \dots A_j$  that minimizes the number of multiplications

Clearly,  $A_{i..j}$  is a  $P_{i-1} \times P_j$  matrix.

For any optimal multiplication sequence at the last step you are multiplying two matrices  $A_{i..k}$  and  $A_{k+1..j}$  for some  $k$ .

$$\begin{aligned} A_{i..j} &= (A_i \dots A_k) (A_{k+1} \dots A_j) \\ &= A_{i..k} A_{k+1..j} \end{aligned}$$

$$\begin{aligned} \text{Ex : } A_{3..6} &= (A_3 (A_4 A_5)) (A_6) \\ &= A_{3..5} A_{6..6} \end{aligned}$$

Where  $k = 5$  for optimal sequence of multiplication  $k$  must be decided and parenthesizing the subchains must be decided

**Optimal Substructure Property :** If final optimal solution of  $A_{i..j}$  involves splitting into  $A_{i..k}$  and  $A_{k+1..j}$  at final step then parenthesization of  $A_{i..u}$  and  $A_{x+i..j}$  in final optimal solution must also be optimal for the subproblems standing above

If parenthesization of  $A_{i..n}$  was not optimal we could replace it by a better parenthesization and get a cheaper final solution leading to contradiction.

Similarly, if parenthesization of  $A_{k+1..j}$  was not optimal we could replace it by a better parenthesization and get a cheaper solution, which is also a contradiction

NOTES

### Design and Analysis of Algorithms

**Step 2:** Recursively define the value of an optimal solution.

for  $1 \leq i \leq j \leq n$ , let  $m[i,j]$  denote the minimum number of multiplications needed to compute  $A_{i \dots j}$

Optimum cost can be described by for  $c = j$   
for  $i = j$

$$m[i, j] = \begin{cases} 0 \\ \min_{i \leq k < j} (m[i, k] + m[k+1, j] + P_{i-1} P_k P_j) \text{ for } i < j \end{cases}$$

**Proof :** Any optimal sequence of multiplication for  $A_{i \dots j}$  is equivalent to some choice of splitting

$$A_{i \dots j} = A_{i \dots k} A_{k+1 \dots j}$$

for some  $k$ , where the sequences of multiplications for  $A_{i \dots k}$  and  $A_{k+1 \dots j}$  also are optimal. Hence

$$m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$$

we know that for some  $k$

$$m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$$

For  $k$ , there  $j-i$  possible values and find one which returns a smallest cost.

for  $i = j$

$$\therefore m[i, j] = \begin{cases} 0 \\ \min_{i \leq k < j} (m[i, k] + m[k+1, j] + P_{i-1} P_k P_j) \text{ for } i < j \end{cases}$$

**Step 3 :** Compute the value of an optimal solution in a bottom up fashion.

To calculate  $m[i, j]$  we can use  $m[i, j] = \min_{i \leq x < j} (m[i, x] + m[x+1, j] + P_{i-1} P_x P_j)$

and we have already evaluated  $m[i, k]$  and  $m[x+1, j]$

For both the cases, length of the matrix chain are both less than  $j-i+1$ . Thus we must calculate in the increasing order of length of matrix chain

$$m[1,2], m[2,3], m[3,4], \dots, m[n,n]$$

$$m[1,3], m[2,4], m[3,5], \dots, m[n-2,n]$$

$$m[1,4], m[2,5], \dots$$

$$m[1,n-1], m[2,n]$$

$$m[1,n]$$

When designing a dynamic program algorithm there are two parts

1. Finding an appropriate optimal substructure property and corresponding recurrence relation
2. Filling in the table properly. This requires finding an ordering of the table elements so that when a table item is calculated using the recurrence relations all the table values needed by the recurrence relation have already been calculated

NOTES

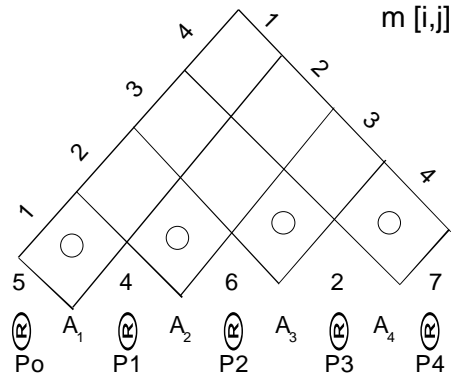
**Design and Analysis of Algorithms**

**Example for bottom up computation**

Example : Given a chain of four matrices  $A_1, A_2, A_3, A_4$  with  $P_0 = 5, P_1 = 4, P_2 = 6, P_3 = 2$  and  $P_4 = ?$

Find  $m[1,4]$

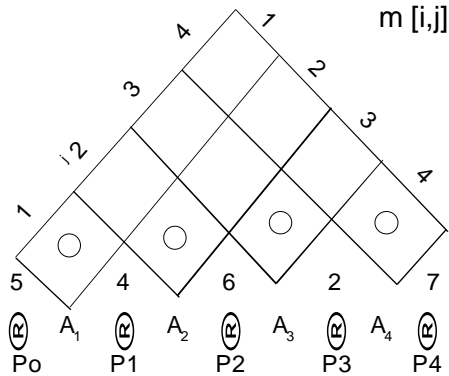
So: Initializations



**Step 1 : Compute  $m[1,2]$**

$$m[1,2] = \min (m[1,x] + m[k+1,2] + P_0 \times P_x \times P_2)$$

$$= m[1,1] + m[2,2] + P_0 P_1 P_2 = 120$$



**Step 2: Compute  $m[2,3]$**

$$m[2,3] = \min (m(2, 1) + m[k+1,3] + 2 \forall k < 3 P_1 P_2 P_3)$$

$$= m[2,2] + m[3,3] + P_1 P_2 P_3 = 106$$

**Step 3: Compute  $m[3,4]$**

$$m[3,4] = \min (m(3, 4) + m[k+1,4] + P_2 P_3 P_4)$$

$$3 \forall k < 4$$

$$= m[3,3] + m[4,4] + P_2 P_3 P_4 = 84$$

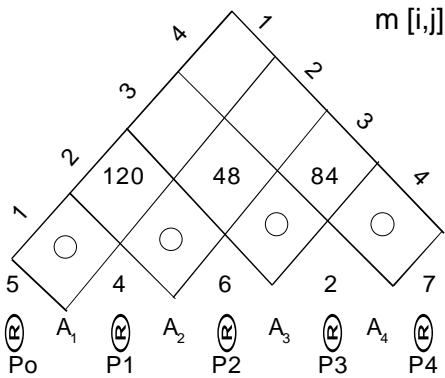
**Step 4: Compute  $m[1,3]$**

$$m[1,3] = \min (m[1, k] + m[k+1,3] + P_0 P_k P_3)$$

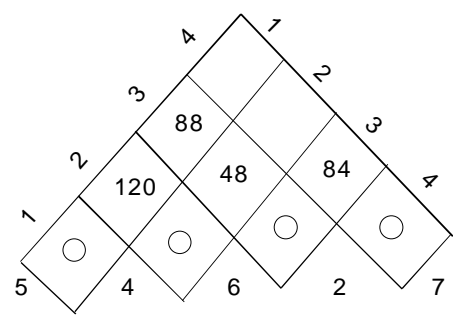
**NOTES**

**Design and Analysis of Algorithms**

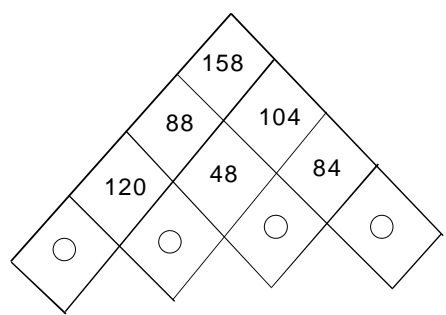
**NOTES**



$$\begin{aligned}
 & 1 \leq k < 3 \\
 = \min & \left\{ \begin{aligned} & = m[1,1] + m[2,3] + P_0 P_1 P_3 \\ & m[1,2] + m[3,3] + P_0 P_1 P_3 = 88 \end{aligned} \right\} \\
 & = 88
 \end{aligned}$$



**Step 5:** Compute  $m[2,4] = 104$   
 $m[1,4] = 158$



Construct an optimal solution from computed information extract the actual sequence

Maintain an array  $S[l...n, i...o]$  where  $S[i,j]$  denotes  $k$  for the optimal splitting in computing

$$A_{i...j} = A_{i...k} A_{k+1...j}$$

The array  $S[l...n, l...n]$  can be used recursively to recover the sequence.

\*\*\*\*\*



# BASIC SEARCH AND TRAVERSAL TECHNIQUES

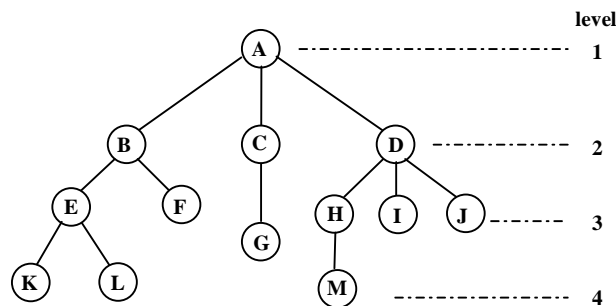
## NOTES

### 5.1 Introduction:

The solution of many problems involves the manipulation of trees and graphs. The manipulation requires to search an object in a tree or graph by systematically examining their vertices according to a given property. When searching involves examining every vertex then it is called traversal.

#### Tree traversals:

Tree: A tree is a finite set of one or more nodes such that there is a special node called root and remaining are partitioned into 'disjoint sets  $T_1, \dots, T_n$  where each of these sets are called as sub trees.



Degree of a Node: Number of sub trees of a node is called degree of node.

The degree of node  
 A is 3,  
 B is 2,  
 C is 1

Leaf Nodes: Nodes that have degree 0 are called leaf nodes or terminal nodes.

The nodes K, L, F, G, M, I, J are leaf nodes

Siblings : Children of same parent are said to be siblings.

Degree of a tree : It is the maximum degree of the nodes in a tree.

Level of a node : The root is assumed at level 1 and its children level are incremented by one if a node is at level P, then its children are  $a + p + 1$

Height or depth of a tree: It is defined to be the maximum level of any node in the tree.

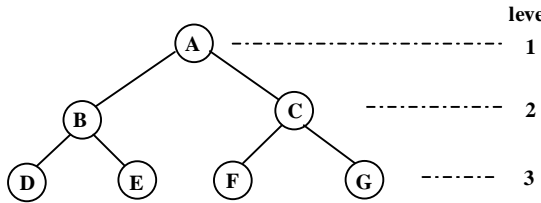
Forest : It is the set of n disjoint trees.

Binary tree : It is a finite set of nodes which is either empty or consists of a root and two disjoint binary trees called left and right sub trees.

**Example:**

The maximum number of nodes on level  $i$  of a binary tree is  $2^{i-1}$ .

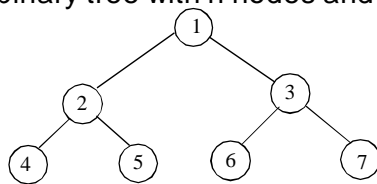
The maximum number of nodes in a binary tree of depth  $k$  is  $2^k - 1$ .



NOTES

Full binary tree: The binary tree of depth  $k$  which has exactly  $2^k - 1$  nodes is called full binary tree.

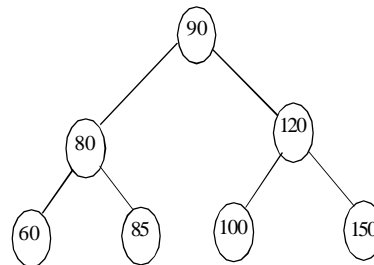
Complete Binary tree: A binary tree with  $n$  nodes and of depth  $k$  is complete iff its nodes



Fully binary tree of depth 3

corresponds to the nodes which are numbered 1 to  $n$  in the full binary tree of depth  $k$ .

Binary search tree: If the value present at left child is less than that of root and value present at right child is greater than that of root, at every stage in a tree, then it is called BST



Heap: A heap is a complete binary tree with the property that the value at each node is as large as the values at its children

**5.2 Binary Tree Traversals:**

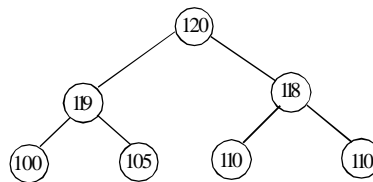
If L, D, R stand for moving left, printing data and moving right, there are six possible combinations of traversals

LDR, LRD, DLR, DRL, RDL and RLD.

If we adopt a convention that we traverse left before right, then we have only 3 traversals.

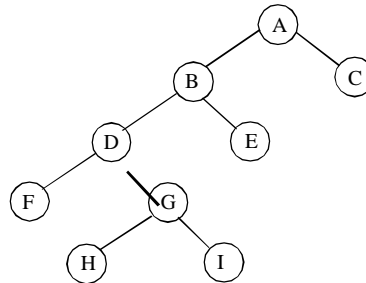
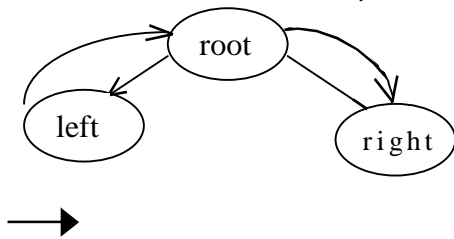
LDR, LRD and DLR

Inorder Traversal (LDR):



Moving down the tree towards left until you cannot go further, then print the data

at that node and move on to right and continue.  
If you cannot move to right and continue  
go back one more node and repeat.



NOTES

Example:

In order Traversal for the tree is  
F D H G I B E A C

Recursive Algorithm for INORDER TRAVERSAL

Procedure INORDER(T)

```
{  
  If T != 0  
  {  
    Call INORDER(LCHILD(T))  
    — Call PRINT(T)  
    — Call INORDER(RCHILD(T))  
  }  
}
```

For the example shown in previous slide algorithm works as shown here

## Design and Analysis of Algorithms

Call of INORDER	Value in root	Action	NOTES
Main	A		
1	B		
2	D		
3	F		
4	-	Print F	
4	-	Print D	
3	G		
4	H		
5	-	Print H	
5	-	Print G	
4	1		
5	-	Print I	
5	-	Print B	
2	E		
3	-	Print E	
3	-	Print A	
1	C		
2	-	Print C	
2	-		

Non recursive Algorithm for INORDER

Procedure INORDER (T)

```

{
    integer stack (m), i, m
    if T = 0 then return
    P = T, i = 0;
    Loop
        {
            While (LCHILD(P) != 0)
                {
                    i = i + 1;
                    if (i > m) then print "Stack overflow";
                    Stack(i) = P;
                    P = LCHILD(P)
                }
        }
}
    
```

```

                                loop
{
    Call print (P)
    P = RCHILD(P)
    If (Pj = 0) exist;
        If (i = 0) return;
        P = stack(i);
        i = i - 1;
}
}

```

**PREORDER TRAVERSAL (DLR)**

First visit the root node(print the data at root node), then visit the left subtree and then right subtree. After printing the data at a root node move down to left subtree, if we cannot continue further then move to right sub tree.

```

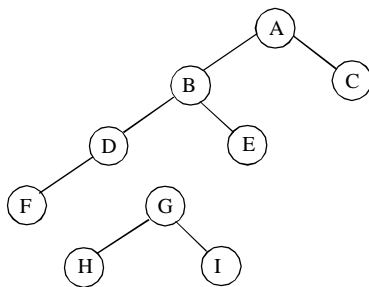
Procedure P REORDER(T)
{
    if{T! = 0)
    {
        Call PRINT(T)
        Call PREORDER(LCHILD(T))
        Call PREORDER(RCHILD(T))
    }
}

```

Example:

Preorder of above tree

A B D F G H I E C



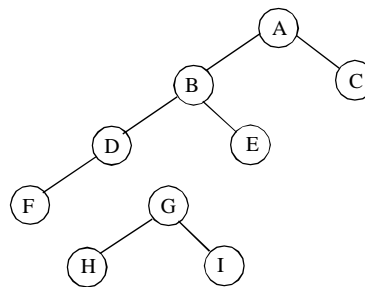
**POSTORDER TRAVERSAL(LRD):**

Procedure POSTORDER(T)

```

{
  If (T1 = 0)
    {
      then call POSTORDER(LCHILD(T))
      call POSTORDER(RCHILD(T))
      call PRINT(T)
    }
}

```



Example:

The post order traversal of the tree shown is

F H I G D E B C A

Time and space complexities of tree traversals:

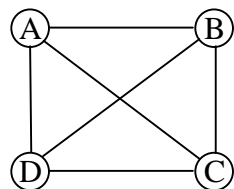
If the time and space needed to visit a node is  $\theta(n)$  then  $t(n) = \theta(n)$  and  $S(n) = O(n)$

**TRAVERSAL TECHNIQUES**

**5.3 Graph traversals**

**Graph:**

A graph G is defined as a set containing of two sets called the verices V and edges E.



$G = \{V, E\}$

NOTES

**Directed graph:**

If the pairs are ordered i.e.,  $\langle i, j \rangle, \langle j, i \rangle$  it is called directed graph.

**Adjacent vertices:**

If a vertex  $i$  is connected to  $j$  i.e., there exists an edge  $\langle i, j \rangle$  then we call  $i$  and  $j$  are adjacent.

In the figure  $(A, B), (B, C), (A, C)$  are adjacent



**Path:**

A path from vertex  $v_p$  to  $v_q$  is a sequence of vertices  $v_p, v_{i_1}, v_{i_2}, \dots, v_{i_n}, v_q$  such that  $(v_p, v_{i_1}), (v_{i_1}, v_{i_2}), \dots, (v_{i_n}, v_q)$  are edges.

In the figure ABCDA is a path

**Simple path:**

Is a path in which all vertices except first and last are distinct

In the figure ABC is a simple path

**Cycle:**

It is a path in which first and last vertices are same

In the figure ABCA is a cycle

**Connected graph:**

If for every pair of vertices there exists a path between them, then it is connected.

The graph shown in above figure is connected

**Strongly connected graph:**

If for every pair of vertices,  $i, j$  there exists a path from  $i$  to  $j$  and a path from  $j$  to  $i$  then we say it is strongly connected.

**Adjacency Matrix:**

An undirected graph is represented by a 2D matrix,  $G[n] [n]$  whose number of rows and columns are equal to vertices of graph and the elements of matrices are 0 and 1. If there is an edge between  $\langle i, j \rangle = 1$  otherwise  $G[i] [j] = 0$ .

**Weighted graph:**

For a weighted graph each edge has a weight and adjacency matrix is initialized with these weights.

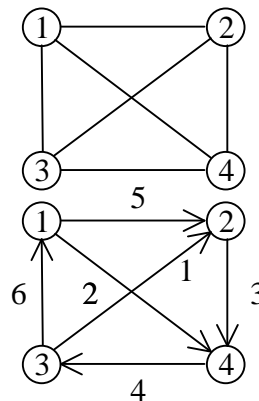
**Example 1:**

$G[4] [4] =$

Example 2:

$G[4][4] =$

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ - & 1 & 1 & 0 \\ 2 & 1 & - & 1 \\ 3 & 1 & 1 & - \\ 4 & - & 1 & 1 \end{pmatrix}$$



NOTES

There are two types of traversal techniques for a graph. They are

1. Breadth first search
2. Depth first search

**5.3.1 Breadth first search:**

- Step 1 : Start at vertex v and mark it as visited
- Step 2 : The v is unexplored (its children or its adjacent vertices are not visited). Now visit all adjacent vertices of V, then v is explored.
- Step 3 : Now the adjacent vertices of v are unexplored vertices. All these vertices are entered into a queue
- Step 4 : Now explore the first vertex in the queue.
- Step 5 : Repeat the process until no unexplored vertex is left

Example

Algorithm for BFS

Procedure BFS(v)

```

{
    VISITED(v) = 1;
    u = v
    Q is made empty
Loop
{
    for all vertices adjacent fromle do
        if (VISITED(w) = 0)
            {
                Call INSERT (w, Q)
                VISITED(w) = 1
            }
    }
}
    
```



```

    }
  }
  If Q is empty then return
    Call DELETE(Q)
  }
}

```

Analysis:

Let  $t(n, e)$  and  $S(n, e)$  be maximum time and space needed by algorithm BFS on any graph with  $n$  vertices and  $e$  edges

If  $G$  is represented by adjacency matrix

$$t(n, e) = q(n^2)$$

$$S(n, e) = q(n)$$

If  $G$  is represented by adjacency list

$$t(n, e) = q(n + e)$$

$$s(n, e) = q(n)$$

### 5.3.2 Depth first search:

Step 1 : Start at a vertex  $v$  and mark it as visited.

Step 2 : Try to explore  $v$  and exploration of  $v$  is suspended as soon as new vertex is reached.

Step 3 : Now try to explore  $u$ . the new vertex

Step 4 : If  $u$  is explored, continue to explore  $v$ .

Step 5 : Repeat the process until all the vertices are explored.

Example:

Algorithm:

Procedure DFS( $v$ )

```

{
  VISITED( $v$ ) = 1
  For(each vertex  $w$  adjacent from  $v$ )
    {
      if(VISITED( $w$ ) = 0)
        {
          Call DFS( $w$ )
        }
    }
}

```

}  
}  
}

Analysis:

It also takes a time complexity

$t(n, e) = q(n + e)$  when adjacency lists are used and

$t(n, e) = q(n^2)$  when adjacency matrix is used.

Space complexity  $S(n, e) = q(n)$

#### **5.4 AND/OR Graphs:**

Many problems are broken into reach such problems until they are solvable. This breakdown can be represented by directed graphs, in which nodes represent problems and descendents of a node represent sub problems.

Example:

Square nodes represent terminal node which does not have descendents  
circular nodes has descendents

AND nodes:

Groups of sub problems that must be solved to produce a solution to parent node are called AND nodes.

OR nodes:

If either all descendents or only one descendent is sufficient to solve, then they are called OR nodes.

Example: A and A<sup>2</sup> are OR nodes

A' is AND node

When problem is reduced two different problems may generate a common sub problem. Therefore graph is no longer a tree.

Example: A solution graph is a sub graph of solvable nodes that shows that problem is solved.

#### **5.5 Game Trees:**

There trees are used to represent games like chess, checkers etc.

Every game has some legal movements, board configuration at a given time.

Finite game:

It is one in which there are no valid sequences of infinite length.

Instance of game:

A valid sequence of board configuration with a terminal configuration is an instance of game.

Now we can define a game tree as possible instances of a finite game.

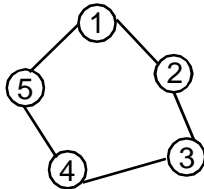
Example: The game tree for nim with  $n = 6$  is shown below. Each node of the tree represents a board configuration Terminal configurations are leaf nodes, leaf nodes have been labeled by name of the player who wins.

**5.6 Biconnected Components:**

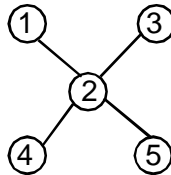
Articulation Point: If a vertex  $v$  in a connected graph  $G$  is deleted with all edges incident on it, so that the graph is divided into two or more non empty components, then the vertex  $v$  is called as articulation point.

Biconnected Graph : If a graph doesn't contain articulation points then it is said to be biconnected.

Ex :



Biconnected graph  
Node 2 is articulation point



Non biconnected graph

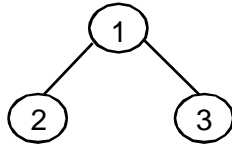
We can assume graph is a simulation of communication network such as LAN. The articulation point may be assumed as centralized server. If it fails then the entire network will fail. On the other hand if there is no centralized server (articulation point) and all nodes are of same configuration even if one fails there is no problem for communication.

To identify articulation points and biconnected components, depth first spanning trees can be used. If  $(u,v)$  is any edge in  $G$  then the relative to the depth first spanning tree  $T$  either  $u$  is an ancestor of  $v$  or  $v$  is an ancestor of  $u$ . So there are not cross edges relative to a depth first spanning tree,  $(u,v)$  is a cross edge relative to  $T$  iff neither  $u$  is an ancestor of  $v$  nor  $v$  an ancestor of  $u$ . So  $(u,v)$  must be a back edge.

Depth first number is the order in which a depth first search visits these vertices.  $DFN(1) = 1, DFN(2) = 2$  in the below graph.

We can assume  $DFN(u) < DFN(v)$  in the above discussion. For each vertex  $u$  define  $L(u)$  lowest depth first number that can be reached from  $u$ .

NOTES



$$L(u) = \min \{DFN(u), \min \{L(w)/w \text{ is child of } u\}, \min \{DFN(w)/(u,w) \text{ is back edge}\}$$

Example:

Algorithm:

Procedure DFS(v)

```

{
    Proc_articulation (u,v)
{
    global DFN(n),L(n),num,n;
    DFN(u)=num; L(u) = num; num=num+1;
    for each vertex w adjacent from u
{
        if (DFN(w) = n)
            articulation (u,w);
        L(u) = min (L(u),L(w));
    Else
    If w! = v
    L(u) = min(L(u),DFN(w))
    }
    }
}
    
```

The above procedure performs depth first search of  $G$ . Each newly visited vertex gets assigned its depth first number.  $L(i)$  is computed for each vertex in the tree. Once  $L(n)$  is computed articulation point can be identified in  $O(e)$  of time, where  $e$  is the number of edges. Time taken to find articulation point is  $O(n+e)$ .

To find the biconnected components the algorithm is

Procedure \_ biconnected (u,v)

```

{
    
```

```
global S, DFN(n),L(n), num,n;
DFN(u)=num; L(u)=num;num=num+1;

for each vertex w adjacent from u
{
if v!=w and DFN(w)<DFN(u)
add (u,w) to stack S;
    if (DFN(w) = 0)
    articulation (u,w);
    if L(w)>=DFN(u) then print ('new biconnected component')
    do
    {delete an edge (x,y) from the top of the stack S
    print(x,y)
    } while ((x,y) = (u,w) || (w,u))
    }
    L(u) = min (L(u),L(w));
    Else
    If w! = v
    L(u) = min (L(u), DFN(w))
}
}
```

The time complexity of the above algorithm is  $O(n+e)$

## BACK TRACKING

### 6.1 Introduction

- Problems dealing with larger search space use back tracking to find out optimal solution in a set of solutions.
- It was first proposed by D.H. Lehmer in 1950. Algorithmic approach was given by R.J. Walker. It was clearly described by Baumet
- If the solution is expressible as an n-tuple( $x_1, x_2, \dots, x_n$ ) where the  $x_i$  are chosen from some finite set  $S_i$ , then we can apply backtracking
- All the solutions using backtrack must satisfy two constraints, they are explicit and implicit constraints.
- Explicit Constraints: These are the rules which restrict each  $x_i$  to take on values only from a given set.
- All tuples which satisfy those constraints define solution space.
- Implicit Constraints; These determine the tuples in the solution space that actually satisfy the objective function.
- The search process using back track is a systematic approach and forms a tree structure.
- Each node in this tree defines a problem state.

- All paths from the root to other nodes defines state space.
- Solution states are the problems states  $S$  for which the path from the root to  $s$  defines a tuple in the solution space.
- Answer states are those solution states  $S$  for which the path from the root to  $S$  defines a tuple which is a member of the set of solutions of the problem.
- The tree organization of the solution space will be referred to as the state space tree.
- Once a state space tree is generated, the given problem is solved by generating the problem states, determining which of these are solution state and finally answer state.
- We begin with 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 E node.
- A dead node is a node whose children have been generated or not to be expanded further.
- Bounding functions will be used to kill live nodes without generating all their children.
- Depth first node generation with bounding function is called BACKTRACKING.

In many applications, the desired solution is expressible as  $n$  tuple  $(x_1, \dots, x_n)$  where  $x_i$  are chosen from some finite set  $S_i$ . If  $m_i$  is the size of set  $S_i$ . Then there are  $m$  tuples which are possible solutions for satisfying criteria function,  $P$ . If it is realized that a partial vector can no way possible, then test vectors may be ignored completely.

BT algorithm searches solution space systematically in a tree organization. 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 that problem states  $s$  for which the path from the root to  $s$  defines a tuple in the solution space. In the tree all nodes are solution states whereas leaf nodes are solution states. Answer states are that solution states  $s$  for which the path from the root to  $s$  defines a tuple that is a member of the set of solutions of the problem. The tree organization of the solution space is referred as the state space tree.

- 1) Time to generate  $X(k)$
- 2) Number of  $X(k)$  satisfying constraints
- 3) Time for calculating bounding function  $B_i$ .
- 4) Number of  $X(k)$  satisfying  $B_i$  for all  $i$ .

The first three factors are independent of the problem instance; fourth one varies from problem to problem.

If the number of nodes in a solution space is  $n!$  or  $2^n$ . Therefore worst case time complexity will be in the order of  $(p(n)2^n)$  or  $(q(n)n!)$ .

**6.2 CONTROL ABSTRACT FOR BACKTRACKING:**

```

Proc_backtrack(n)
{
    integer k, n, local X[n];
    while (k > 0)
        { if there remains an untried x(k)
        if  $x(k) \hat{=} T(x(1), x(2), \dots, x(k-1))$  and
 $(x(1), x(2), \dots, x(k)) = \text{true}$ 
            do if  $(x(1), \dots, x(k))$  is path to an answer node then print  $(x(1), x(2), \dots, x(k))$ ,
            endif.
            k = k + 1
        else k = k - 1, endif.
    }
}

```

NOTES

**6.3 QUEENS PROBLEM :**

Let us assume that there are 8 columns and 8 rows on a chess board. Eight queens are given and each queen must be placed on a different row so that no two queens will lie on same row, same column and same diagonal. Hence we can assume queen  $i$  is to be 8 queens problem is represented by 8 tuples  $(x_1, x_2, \dots, x_8)$  where  $x_i$  is column on which queen  $i$  is placed.

The solution space consists of  $8^8$ , 8-tuples. The constraint for the problem is no two  $x_i$  s, must be on same row, on same column and no two queens must be on same diagonal.

According to these two constraints all solutions are permutations of 8 tuple (1, 2, 3, 4, 5, 6, 8). This reduces the solution space from  $8^8$  to  $8!$  Tuples.

Before going to 8 queens let us analyze 4-queens problem for better understanding.

**6.3.1 4- Queens problem :**

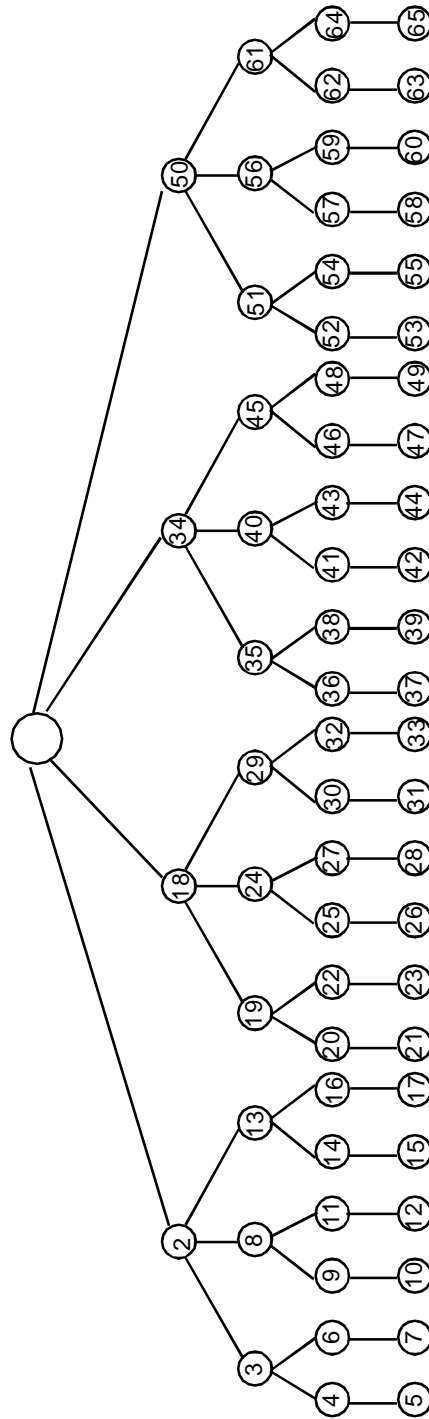
**Problem :** A  $4 \times 4$  chess is given and we must place 4 queens on the chess board so that no two queens must be placed on same row, same column and same diagonal.

**Solution :**

Let 4-queens be represented as  $(x_1, x_2, x_3, x_4)$ . Thus a queen  $x_i$  means, the queen placed in the row. The value of  $x_i$  gives the column number in  $i$ th row. Since row number of all  $x_i$ 's are different there is no need to verify whether the queens are on same row. Therefore if we verify column and diagonals we can obtain solution.

If we observe the total search space of the problem, it is a tree having 65 nodes shown in the figure.

Search space tree of 4-queens problem :

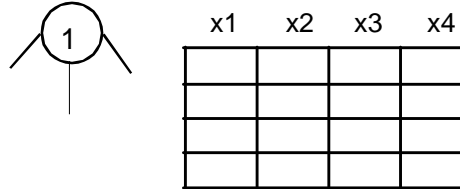


NOTES

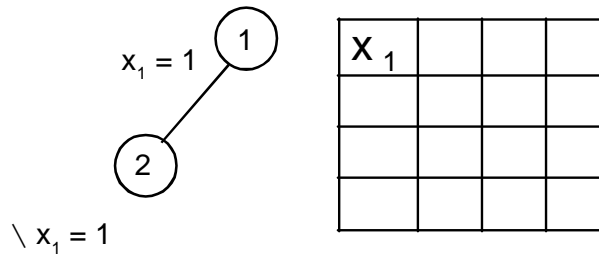


**Analysis :**

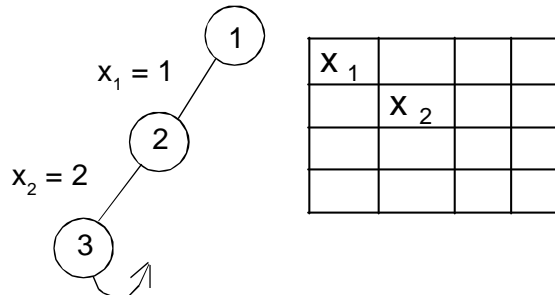
**Step 1 :** We start at root node as live node. This node becomes E node and path is ()



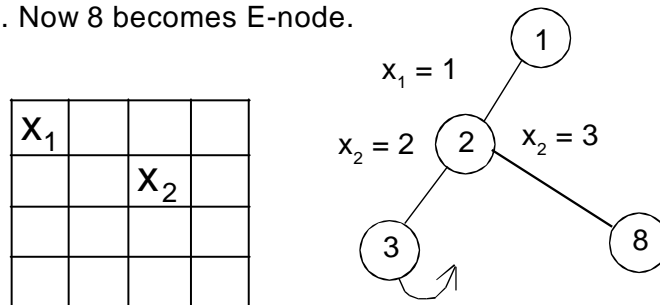
**Step 2 :** Generate one child node 2, now the path is (1). This corresponds to place queen x1 in column 1.



**Step 3 :** Now node 2 becomes Enode. Node 3 is generated and killed, since it doesnot satisfy conditions hence back track to node 2.



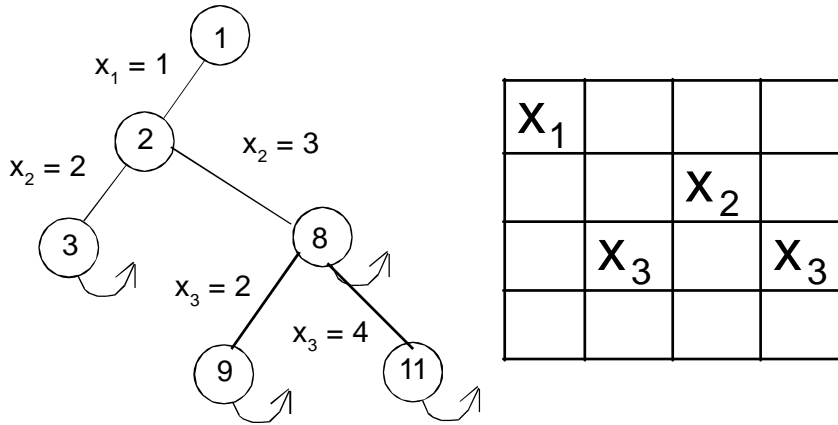
**Step 4 :** Still node 2 is Enode, generate node 8 and now path is (1, 3). Now 8 becomes E-node.



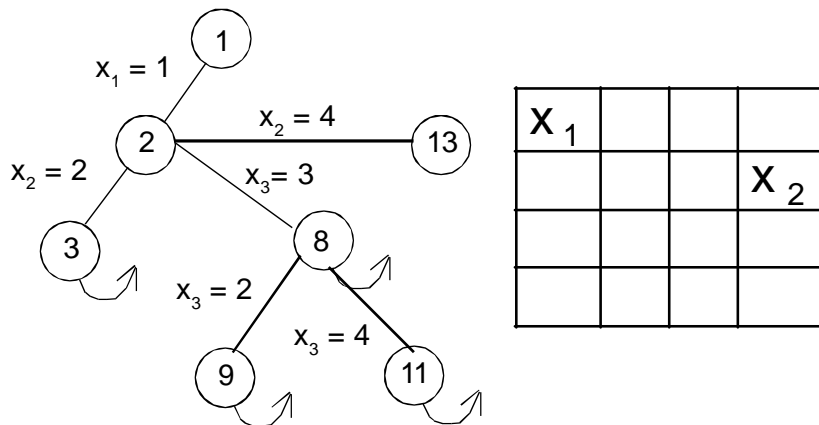
## Design and Analysis of Algorithms

**Step 5 :** The children of node 8 cannot satisfy the conditions hence they are killed, since they are killed node 8 is back tracked to node 2

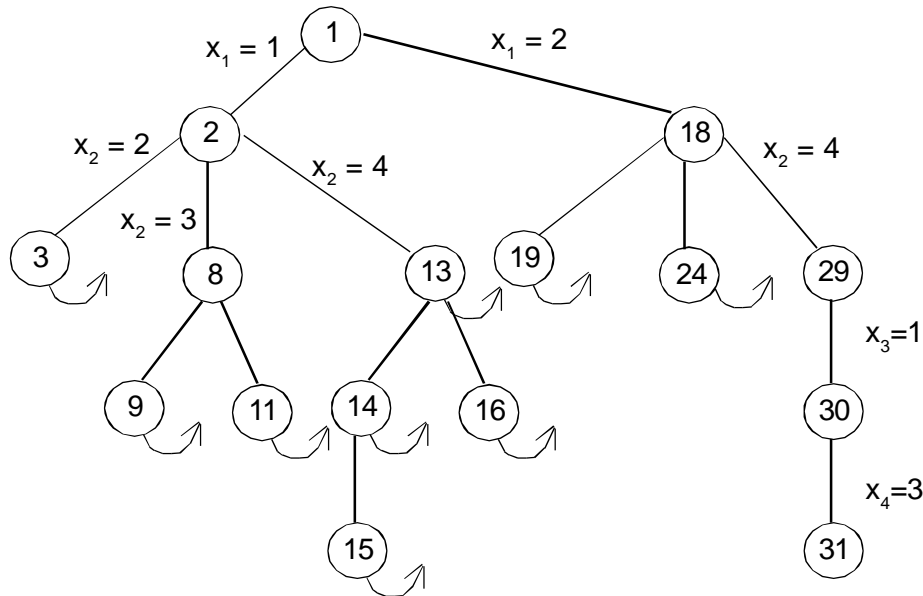
NOTES



**Step 6 :** Still node 2 is E node and next node generated is 13. Now the pa.... is (1, 4)



**Step 7 :** In this way back tracking algorithm goes to find a solution given below.



	X <sub>1</sub>		

	X <sub>1</sub>		
			X <sub>2</sub>

	X <sub>1</sub>		
			X <sub>2</sub>
X <sub>3</sub>			

	X <sub>1</sub>		
			X <sub>2</sub>
X <sub>3</sub>			
		X <sub>4</sub>	

**6.3.2 8- Queens problem :**

If the squares of chess board are represented as 2D array A(1 : n, 1 :n).

- 1) For every element on same diagonal which runs form UL to LR has same 'row-col' value.
- 2) Every element on same diagonal which runs from UR to LL has same 'row+col' value.

Suppose two queens are placed at (i, j) and (k, l), then they are on same diagonal, if  $i - j = k - l$

(or)

$$i + j = k + l$$

Therefore  $j - l = k - i$  and  $j - l = k - i$

Therefore two queens lie on the same diagonal if and only if

$$|j - l| = |i - k|$$

Procedure placed (k)

{

NOTES

```

global X(1 : k), int i, k
  for i = 1 to k
    {
      if X(i) = X(k) or abs (x(i)-x(x))=abs(i-k)
        then return (false)
      end if
    }
  return (true)
}

```

**Procedure N queens**

```

{
  integer k, n, X(1 : n)
  X(1) ← 0; k ← 1
  while K > 0
    {
      X(k) = X(k) + 1
      While X(k) ∈ n and not place (k)
        {
          X(k) ← x (k +1)
        }
      If x(k) ∈ n
        Then if k = n
          then print (X)
        else k ← k + 1; x(k) ← 0
        End if
      Else k ← k - 1
        End if
    }
}
}

```

			X <sub>1</sub>				
					X <sub>2</sub>		
							X <sub>3</sub>
	X <sub>4</sub>						
X <sub>6</sub>							
		X <sub>7</sub>					
				X <sub>8</sub>			

NOTES

**Analysis of 8 queens**

The analysis of 8-queens is also same as that of 4- queens. The search space tree is of 8-queens also similar to that of 4-queens.

Solution vector

$$X = (4, 6, 8, 2, 7, 1, 3, 5)$$

The total number of nodes in 8 queens state space tree is

$$1 + \sum_{j=0}^7 \prod (8-i) = 69, 281$$

**6.4 Graph Coloring:**

If G is a graph, nodes of G must be colored such that no two adjacent nodes have same color (among m colors). This is known as m-colorability decision. D smallest m for the problem.

A graph is said to be planar if not two edges cross each other. Each can be converted into a graph. Each region of map becomes a node and if two regions are adjacent they are joined by an edge.

A graph can be represented as adjacent matrix G (1 : n, 1 : n)

G (i : j) = true if (i , j) is an edge of G

Else

G (i,j) = false

Procedure M Coloring(k)

```

{
    Global int m,n X(1 : n );
    Boolean GRAPH (1 : n, 1 : n)
    Integer k
    {
        Call NEXT VALUE (k)
        If X(k) = 0 then exit
        If k = n
            Then print (X)
            Else call M Coloring(K+1)
        Endif
    }
}

```

Procedure NEXTVALUE produces the possible colors for X(k) after X(1) through X(k-1). Main loop M Coloring picks an element from set of

NOTES

possibilities, assigns to X(k).

**Procedure :**

NEXTVALUE(k)

```

{
    Global integer m, n , X( 1:n)
        Boolean ( 1 : n, 1 : n )
        Integer j, k
            {
                X(k) ← (X(k) + 1) mod ( m + 1)
                If X(k) = 0 then return
                For j = 1 to n
            }
        {
            If GRAPH (K,J) and
                X(k) = X(j)
                Then exit
        }
        If j = n+1 then return
    }
}

```

**Time Complexity :**

The number of internal nodes in state space tree is  $\sum_{i=0}^{h-1} m^i$  At each internal node O(MN) time is spent by NEXTVALUE.

$$\begin{aligned}
 \text{Total time } \sum_{i=1}^n m^i n &= n( m^{n+1})/( m - 1) \\
 &= O(nm^n)
 \end{aligned}$$

**6.5 Sum of Subsets :**

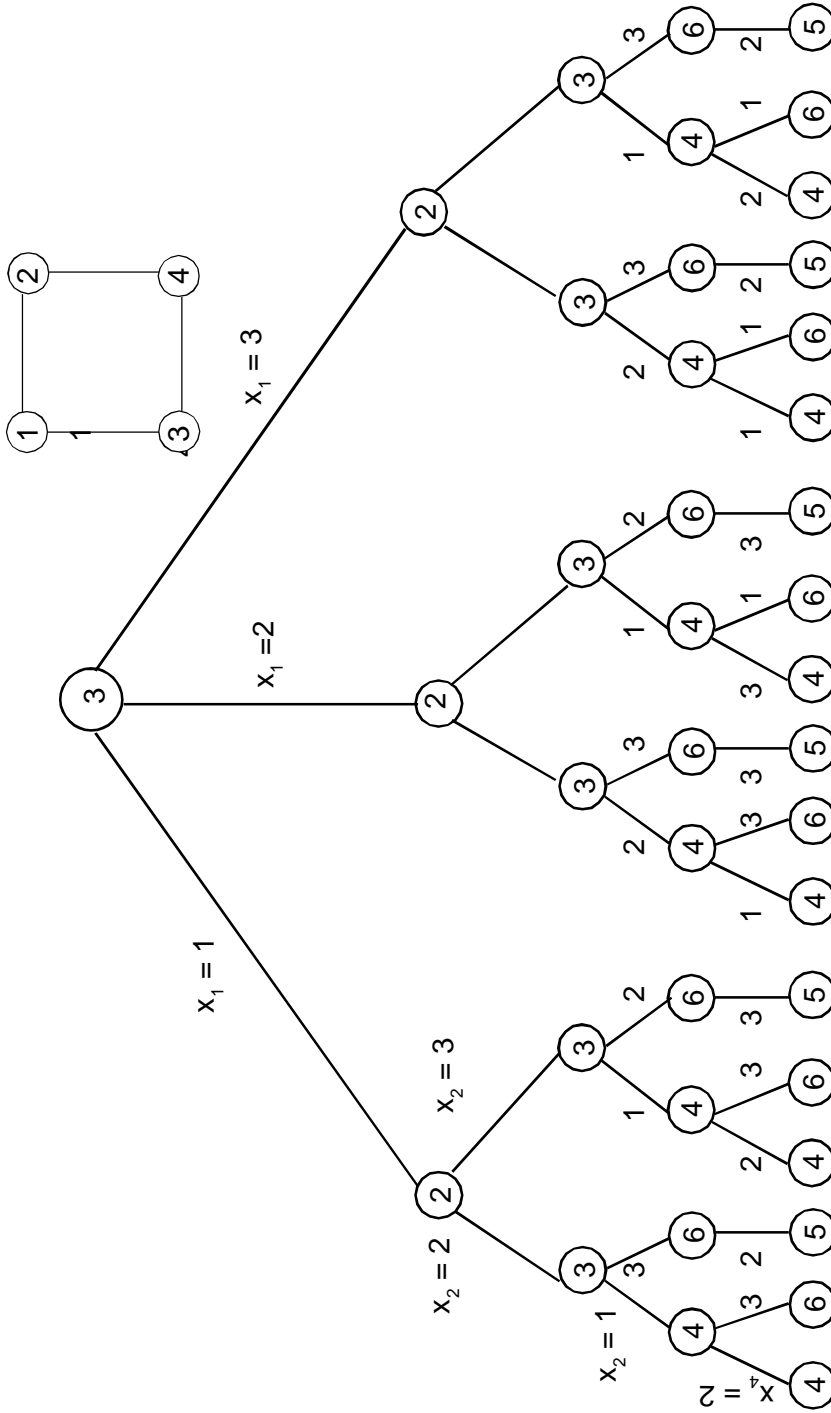
Given n distinct positive numbers and must find all combinations of these numbers whose sum is M

If n = 4, (w<sub>1</sub>, w<sub>2</sub>, w<sub>3</sub>, w<sub>4</sub>) = (11, 13, 24, 7) and M = 31 then the desired subsets are (11, 13, 7) and (24, 7). The two solutions are described by (1,2,4) and (3,4).

Each solution is a n tuple (x<sub>1</sub>, x<sub>2</sub>,.....x<sub>n</sub>) such that x<sub>i</sub> ∈ (0, 1), 1 ≤ i ≤ n. x<sub>i</sub> = 0

NOTES

Example : Let us consider a 4-node graph coloring with 3 colors.



NOTES

if  $w_i$  is chosen.

For a node at level, the left child correspond to  $X(i) = 1$  and right child corresponds to  $X(i) = 0$

$B(X(i), \dots, X(k))$  is true if

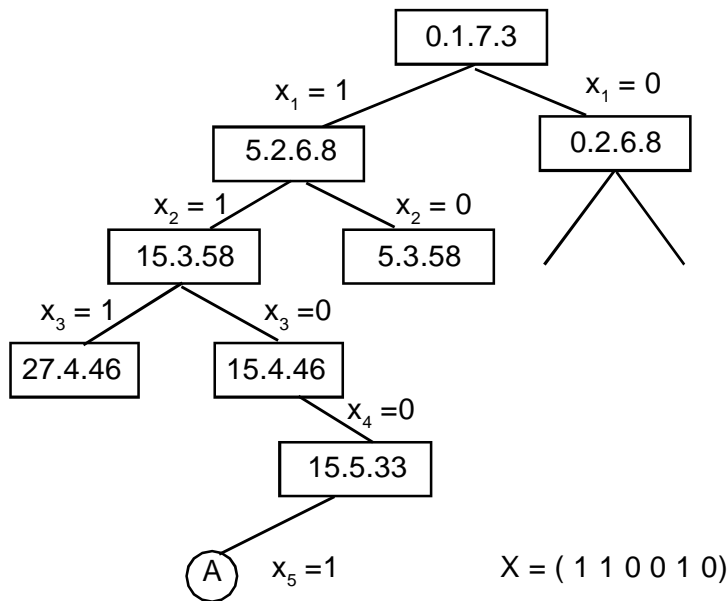
$$\sum_{i=1}^k w(i) \times (i) + \sum_{i=1}^k w(i) \geq M$$

if  $w(i)$  are initially in non decreasing order, we can rewrite

$B_k(X(1), \dots, X(k)) = \text{true if}$

$$\sum_{i=1}^k w(i) \times (i) + \sum_{i=1}^k w(i) \geq M \text{ and } \sum_{i=1}^k w(i) \times (i) + \sum_{i=1}^k w(k+1) \leq M$$

Example : Let us take  $n = 6$ ,  $M = 30$  and  $w(1:6) = (5, 10, 12, 13, 15, 18)$  The state space tree for  $n=6$  contains  $2^6 - 1 = 63$  nodes. A part of the space tree is given below.



Procedure SUMOFSUB (s, k, r)

{

Global integer M, n;

Global real w (1 : n);

Boolean k,j; real r, s

X(k) = 1

If s + w(k) = M

Then print (X(i), j = 1 to k)

NOTES



Else

If  $S = w(k) + w(k+1) \in M$  then

    Call SUMOFSUB ( $S + W(k), k + 1, r - w(k)$ )

    Endif

    Endif

If  $s+r - w(k) \geq M$  and

$S + w(k + 1) \in M$

    Then  $x(k) = 0$

    Call SUMOFSUB ( $s, k, + 1, v - w(k)$ )

    Endif}

NOTES

## BRANCH AND BOUND

### 7.1 Introduction:

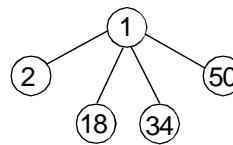
The concept behind B & B is, all the children of E-node are generated before any other live node becomes E-node. This type of exploration can be seen in BFS and D search (is same as BFS but, the next node to explore is the most recently reached unexplored node).

In B & B algorithm a number is computed to determine whether the node is promising or not. The number is a bound on the value of the solution that could be expanding beyond the node. If the bound is no better than the value of the best solution found so far then the node is non-promising.

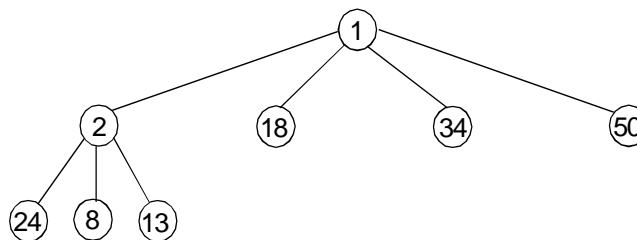
Instead of using a bound, we can compare the bounds of promising nodes and visit the children of the one with the best bound. This approach is called best first search with B & B pruning and faster optimal solution. BFS is implementation of breadth first search with B & B or FIFO B & B. Where as D-search B & B is called LIFO B & B.

Example: Let us take 4-queens problem using FIFO B & B algorithm.

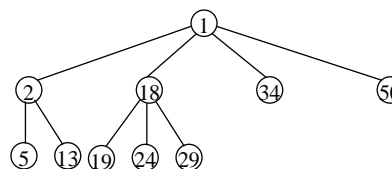
Initially there is only one live node i.e. node 1. This will becomes E-node. It is explored and its children, node 2, 18, 34 and 50 are generated.



Now the E-node is node 2. It is expanded and node 3, 8, 13 are generated.



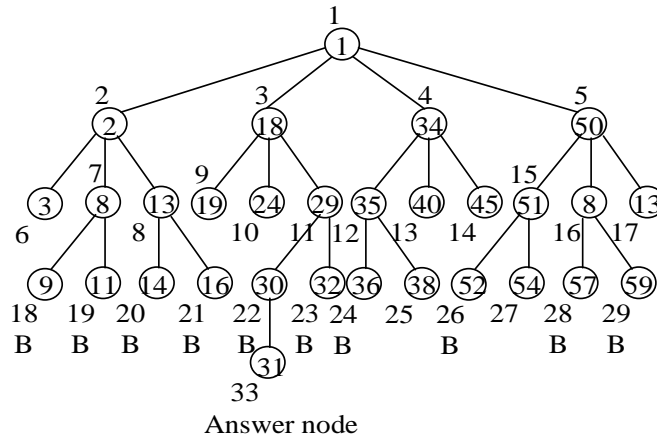
Node 3 is immediately killed by using bounding function. Nodes 8 and 13 are added to queue. Now 18 will become E-node. Nodes 19, 24, 29 are generated..



Now node 19 and 24 are killed since they do not satisfy bounding function. Node 29 is added to queue.

Now 34 is the next E-node and it is explored. This process continues and at the time of answer node 31 is reached, the only live nodes are 38 and 54. Entire search space tree is shown in figure.

NOTES



Particularly for this problem Backtracking is superior than Branch and Bound.

**7.2 Least cost(LC) search :**

In FIFO & LIFO search B & B, the selection rule for next E-node is blind. The rigid FIFO rule requires expansion of all live nodes before answer node was expanded. Thus an intelligent ranking function  $C'(\cdot)$  is needed to select the next E-node.

Let  $g'(x)$  be the additional effort needed to reach answer node from  $x$ .  $x$  is assigned a rank using a function  $c'(\cdot)$  such that  $c'(x) = f(h(x)) + g'(x)$

Where  $h(x)$  is cost for reaching  $x$  from root. A live node with least  $C'(\cdot)$  is selected as next E-node. Hence it is called LC search.

- 1) If  $g(x) = 0$  and  $f(h(x)) = \text{level of node } x$  then it is called BFS.
- 2) If  $f(h(x)) = 0$  and  $g(x) > g(y)$  where  $y$  is the child of  $x$  then it is called as D-search.

Example: Let us consider the problem of 15 puzzle.

A square frame with 16 tiles in which 15 numbered tiles are there and is empty. An initial arrangement is given. By using legal moves arrange them into goal arrangement (a legal move is moving a tile adjacent to the empty spot ES is moved to ES).

Initial arrangement

From this 4 moves are possible. We can move 2, 3, 5, 6 to ES. Each move creates a new arrangement. These arrangements are called states. There are 16! Such arrangements.

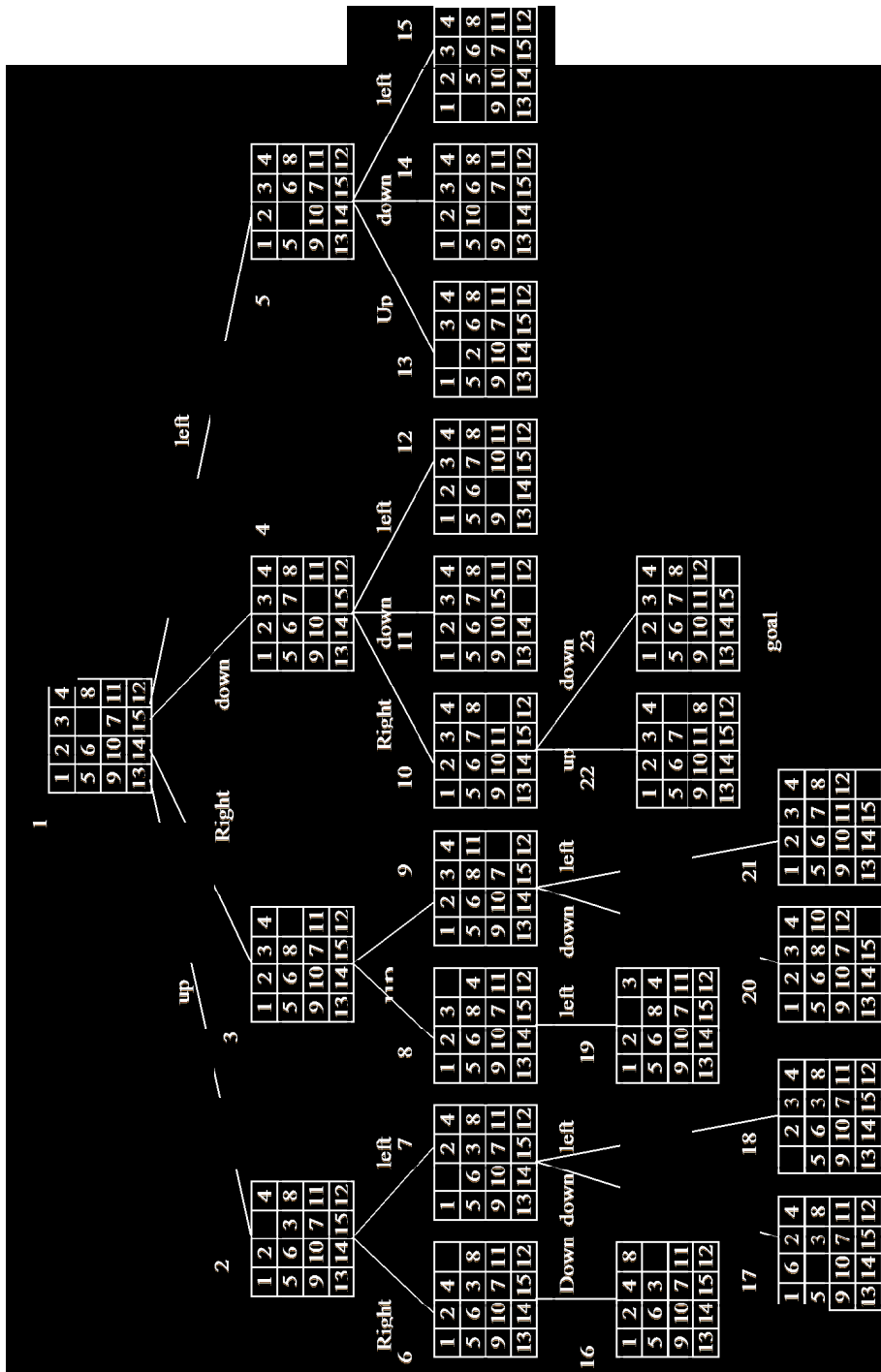
1	3	4	5
2		5	12
7	6	11	14
8	9	10	13

Let position (l) be the number in the initial state of the title numbered i. Therefore position (16) is zero in goal arrangement.

For any state let less (l) be number of tiles  $j$  such that  $j < i$  and position (j) > position (i).

Example: if  $i = 12, j = 6$

Therefore  $\text{less}(12) = 6$  since position (6) > position(12)



Each node  $x$  in the space tree, a cost function  $C(x)$  is associated.  $C(x)$  is the length of a path from the root to a nearest goal node.

Let us say  $C(1) = C(4) = C(10) = C(23) = 3$  such a cost function is available, efficient searching is performed. From root node, nodes 2, 3, 5 are eliminated and only one node 4 becomes a live node. Node 4 becomes live node.

Its first child node 10 has  $C(10) = C(4) = 3$ . The remaining node 10 is greater than 3 except node 23. Next E-node is 23.

But it is difficult to estimate such a cost function. Thus we can compute an estimate  $C'(x)$  of  $C(x)$ . We can write  $C'(x) = f(x) + g'(x)$  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 the shortest path from  $x$  to a goal node in the subtree with root  $x$ .

Therefore we can assume  $g'(x) =$  number of non blank tiles not in their goal position.

Thus at least  $g'(x)$  moves will have to be made to transform state  $x$  to goal state.

For example in the figure shown  $g'(x) = 1$  since only tile 7 is not in its position.

1	2	3	4
5	6		8
9	10	11	12
13	14	15	7

But number of moves to reach goal state is more than 1.

According to LC search we begin at node 1 and its children are generated.

Now live nodes are 2, 3, 4 and 5.

$C'(x)$  for 2, 3, 4 and 5 nodes is

$$C'(2) = 1 + 4$$

$$C'(3) = 1 + 4$$

$$C'(4) = 1 + 2$$

$$C'(5) = 1 + 4$$

Therefore node 4 becomes E-node

Now the children of node 4 are generated. The live nodes at this time are 10, 11 and 12.

$$C'(10) = 2 + 1$$

$$C'(11) = 2 + 3$$

$$C'(12) = 2 + 3$$

Therefore the live node is 10 and its children are 22 and 23 are generated and 23 is goal node

NOTES

Algorithm LC search(t)

```
{
    E node = t
    Repeat
        {
            for each child x of E do
                {
                    if x is answer node then output the path from x to t return
                    add(x),
                    x @ parent = E
                }
            If there are no more live nodes
                {
                    write "no answer", return
                }
            E = least();
        }
    until (false);
}
```

The algorithm uses two functions Least(x) and add(x). Least(x) finds the live node with least C'(x) and Add(x) to delete and add a live node with least C(x). When the answer node G is found then the function parent(x) is used to follow sequence from current E-node G to the root node.

- 1) Root node is first E-node.
- 2) The children of E-node are examined. If one of the children x is answer node, then the path from x to target is printed.
- 3) If a child is not an answer node it becomes live node. Using add(x) it is entered into list of live nodes.
- 4) The present field of x is set to E. When all the children of E have been generated E becomes a dead node.
- 5) If there is no live node left search was completed.
- 6) Otherwise by using least(), next Enode is selected and search continues.

### 7.3 Bounding :

Let us assume each answer node  $x$  has a cost  $C(x)$ , and minimum cost answer node is to be found.

Every node  $x$  has a cost function  $C'(x)$  associated with it, such that  $C'(x) \leq C(x)$  is used to provide lower bound on solution. If  $U$  is upper bound on the cost of minimum cost solution then all live nodes  $x$  with  $C'(x) > U$  are killed as all answer nodes reachable from  $x$  have cost  $C(x) \geq C'(x) > U$ .

If an answer node with cost  $U$  has already reached then all live nodes with  $C'(x) \geq U$  may be killed.

The starting value of  $U$  may be set to  $a$ . Each time a new answer node is found, the value of  $U$  may be updated.

Thus only minimum cost answer nodes will correspond to optimal solution.

Example: Job sequence problem.

Each job  $I$  is associated with a 3 tuple  $P_i, D_i, T_i$ . Job  $I$  requires  $T_i$  processing time if not completed in a dead line  $D_i$  a penalty  $P_i$  is to be payed.

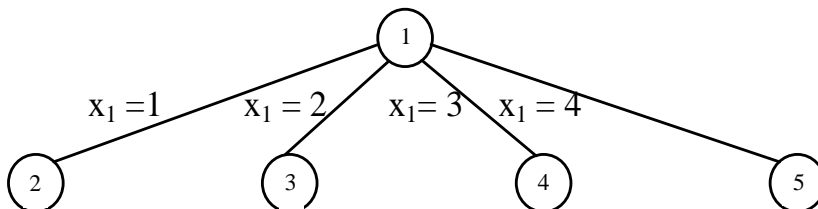
Given  $n = 4$  jobs

$P_i$ (penalty)	5	10	6	3
$D_i$ (deadline)	1	3	2	1
Process time)	1	2	1	1

Let  $S_x$  be the subset of jobs selected for  $j$  at node  $x$

$$C(x) = \sum_{\substack{i < m \\ i \notin S_x}} P_i \quad u(x) = \sum_{i \in S_x} P_i$$

Where  $m = \max \{i \in S_x\}$



NOTES

- 1)  $U = \mu$
- 2) Node 1 as Enode.
- 3) Next live nodes are 2, 3, 4, 5

$$U(2) = 19 (10 + 6 + 3)$$

$$U(3) = 14 (6 + 3 + 5)$$

$$U(4) = 18 (3 + 5 + 10)$$

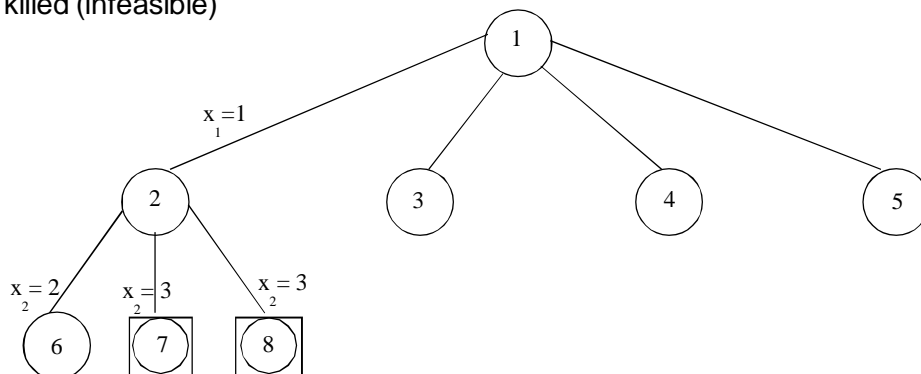
$$U(5) = 21 (5 + 10 + 6)$$

When node 3 is generated  $U$  is updated to 14. Thus all other nodes except 2 and 3 are killed and  $u$  is updated to 14.

- 4) Next Enode is 2 its children are 6, 7, 8  $U(6) = 9$  and  $U$  is updated to  $U = 9$

$$C'(7) > U$$

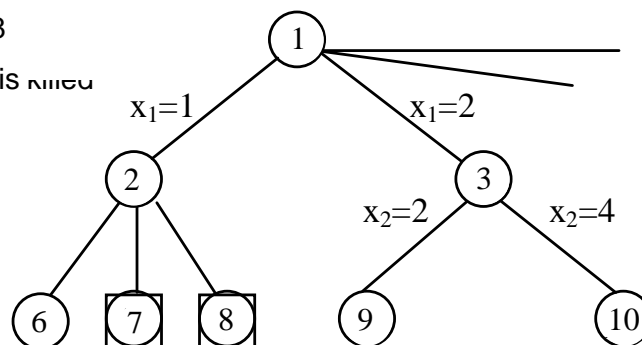
$C'(7) = 10 > (U = 9)$  it is killed,  
 parallelly  $c(8) > U$  is also  
 killed (infeasible)



- 5) Next Enode is 3 its children are 9, 10

$$U(9) = 8 \text{ hence } u = 8$$

$$C'(10) = 11 \text{ hence it is pruned}$$



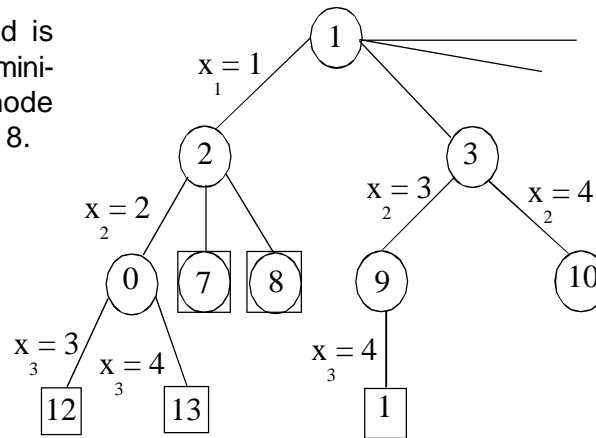
- 6) Next Enode is 6 its children are 12, 13

$$C(12) = 0$$

$$C(13) = 6 \text{ both are infeasible}$$



Node 9's only child is also infeasible. The minimum cost answer node is 9. It has a cost of 8.



NOTES

**7.4 FIFO Branch & Bound:**

All live nodes are in the queue in the order in which they were generated. Hence nodes with  $C'(x) > U$  are distributed randomly. Live nodes with  $C'(x) > U$  are killed when they are about to become E-nodes.

So a small positive constant  $\epsilon$  is used so that if for any two feasible nodes  $x$  and  $y$ ,  $U(x) < U(y)$  then  $U(x) < U(y) + \epsilon < U(y)$ . This  $\epsilon$  is needed to distinguish between the case when a solution has with cost  $U(x)$  has been found and the case when such a solution has not been found.

If the later is the case  $U$  is updated to  $\min\{U, U(x) + \epsilon\}$ . When  $U$  is updated in this way, live nodes  $y$  with  $C'(y) \geq U$  may be killed. This does not kill the node that promised to lead to a solution with value  $\leq U$ .

Procedure FIFOBB( $T, C', U, \epsilon, \text{cost}$ )

```

{
    E = T; Parent (E) = 0;
    If t is a solution node then
        U = min (cost(T), U(T) +  $\epsilon$ ); ans = T
    Else
        U = U(T) +  $\epsilon$ ; ans = T
    Endif
    Initialize queue to be empty
    Repeat for each child x of E
        {
            If  $C'(x) < U$  then call ADD(x);
            Parent(x) = E
        }
}
    
```

Case

:  $x$  is a solution node and  $\text{cost}(x) < U$ ;

$U = \min(\text{cost}(x), U(x) + e)$

Ans =  $x$ ;

:  $U(x) + e < U$ ;

$U = U(x) + e$ ;

Endcase

Endif

}

loop

{

if queue is empty then print "least cost =  $U$ ";

while ans! = 0 do

{ print ans;

ans = Parent(ans);

}

endif

}

Call DELETE Q(E)

if  $C'(E) < U$  then exit

}

}

}

### 7.5 LC Branch & Bound:

It also operates on the same assumptions as FIFOBB. Here we use two functions ADD and LEAST to add a node to a min heap and delete a node from a min heap. An LC branch and bound algorithm will terminate when the next E node E has  $c'(E) \geq U$ .

Initially here also  $U = a$  and node 1 is first E node. Node 1 is expanded and its children 2, 3, 4 and 5 are generated and so on

Procedure LCBT(T, C', U, e, cost)

{

E ← T: Parent(E) + ):

NOTES

```

        If t is a solution node then
            U = min(cost(T), U(T) + e): ans = T
Else
    U = U(T) + e; ans = 0;
    Endif
        Initialize the list of live nodes to be empty
        Repeat for each child x of E
    {
        If C'(x) < U then call ADD(x):
            Parent(x) = E
    Case
        : x is a solution node and cost(x) < U;
            U = min(cost(x), U(x) + e)
        : U(x) + e < U;
            U = U(x) + e;
            Endif
    }
loop
{
    if list is empty or the next E node has C'>= U, then print "least cost = U"
        While ans! = 0 do
            {
                print ans;
                ans = Parent(ans);
            }
            endif
        }
        Call LEAST(E)
    }
}
}

```

NOTES

**7.6 TRAVELLING SALES MEN PROBLEM:**

The time complexity of travelling sales men problem is  $O(n^2 2^n)$  can be reduced by using good bounding function in B & B algorithms.

NOTES

**LCBB approach for Travelling sales men Problem:**

Let us define cost function  $C(x)$  such that  $C'(x) \leq C(x) \leq U(x)$  for all nodes  $x$ .

$C(x)$  = length of tour defined by the path from root to A, if A is leaf/cost of minimum cost leaf in the subtree A, if A is not a leaf.

A simple  $C'()$  such that  $C'(A) < C(A)$  for all A is obtained by defining  $C'(A)$  to be the length of the path defined at node A. A better  $C'()$  may be obtained by using the reduced cost matrix of a given graph G.

Subtracting a constant  $t$  from every entry in one column or one row of the cost matrix reduces the length of every tour by exactly  $t$ . If  $t$  is chosen to be minimum entry in row  $i$ , then subtracting it from all entries in row  $i$  will introduce a 0 into row  $i$ .

The total amount subtracted from all the columns and rows is a lower bound on the length of a minimum cost tour and maybe used as the  $C'$  value for rows 1, 2, 3, 4, 5 and columns 1 and 3 of matrix shown here

$$\begin{bmatrix} \alpha & 20 & 30 & 10 & 11 \\ 15 & \alpha & 16 & 4 & 2 \\ 3 & 5 & \alpha & 2 & 4 \\ 19 & 6 & 18 & \alpha & 3 \\ 16 & 4 & 7 & 16 & \alpha \end{bmatrix}$$

Cost Matrix

$$\begin{bmatrix} \alpha & 10 & 17 & 0 & 1 \\ 12 & \alpha & 11 & 2 & 0 \\ 0 & 3 & \alpha & 0 & 2 \\ 15 & 3 & 12 & \alpha & 0 \\ 11 & 0 & 0 & 12 & \alpha \end{bmatrix}$$

Reduced cost matrix

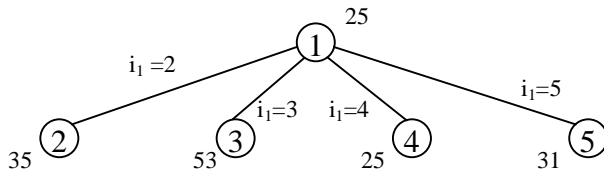
Let A be the reduced cost matrix for node R. Let S be the child of R such that the tree edge (R, S) corresponds to including edge  $\langle i, j \rangle$  in the tour. If S is not a leaf then reduced cost matrix for S may be obtained as

1. Change all entries in row  $i$  and column  $j$  of A to  $a$ . This prevents the use of any more edge leaving vertex  $i$  or entering  $j$ .
- 2) Set  $A(j, 1)$  to  $a$ . This prevents the use of edge  $\langle j, 1 \rangle$
- 3) Reduce all rows and columns in the resulting matrix except for rows and columns containing only  $a$ .

If  $r$  is the total amount subtracted then  $C'(s) = C'(R) + A(i, j) + r$

The initial reduced matrix is 
$$\begin{bmatrix} \alpha & 10 & 17 & 0 & 1 \\ 12 & \alpha & 11 & 2 & 0 \\ 0 & 3 & \alpha & 0 & 2 \\ 15 & 3 & 12 & \alpha & 0 \\ 11 & 0 & 0 & 12 & \alpha \end{bmatrix}$$

Initially  $U = a$ , the state space tree starts at root node and its children 2, 3, 4 and 5 are generated.



By setting all entries in row 1 and column 2 to a we get

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$$

a) path 1, 2; node 2

By setting all entries in row 1 and column 3 to a we get

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{bmatrix}$$

b) path 1, 3; node 3

By setting all entries in row 1 and column 4 to a we get

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

c) path 1, 4; node 4

By setting all entries in row 1 and column 5 to a we get

NOTES

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

$$\hat{C}(4) = 25$$

c) path 1, 4; node 4

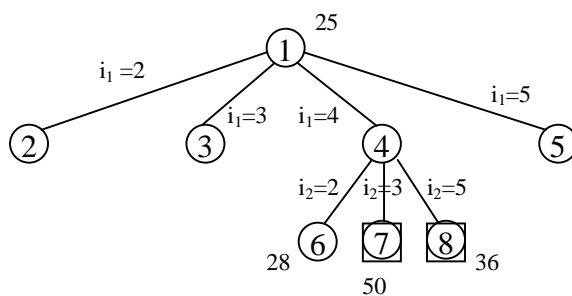
By setting all entries in row 1 and column 5 to a we get

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 12 & 0 & 9 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$$

$$\hat{C}(5) = 31$$

d) path 1, 5; node 5

Now the next E node is node 4.



By setting all entries in row 4 and column 2 to a we get

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

e) path 1, 4, 2; node 6

By setting all entries in row 4 and column 3 to a we get

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & \infty \end{bmatrix} \quad \hat{C}(7) = 50$$

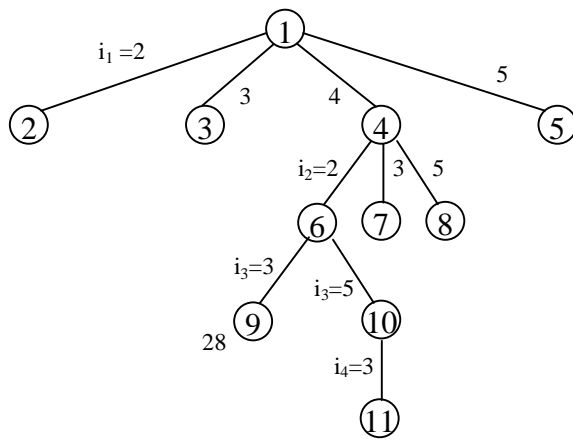
f) path 1, 4, 3; node 7

By setting all entries in row 4 and column 5 to a we get

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix} \quad \hat{C}(8) = 36$$

g) path 1, 4, 5; node 8

Now the next E node is node 6.



By setting all entries in row 2 and column 3 to a we get

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix} \quad \hat{C}(9) = 52$$

h) path 1, 4, 2, 3; node 9

By setting all entries in row 2 and column 5 to a we get

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix} \quad \hat{C}(10) = 28$$

i) path 1, 4, 2, 5; node 10

Thus the minimum cost path is 1, 4, 2, 5, 3, 1.



## NP HARD AND NP COMPLETE PROBLEMS

## NOTES

### 8.1 Introduction :

An algorithm is analyzed by studying the frequency of execution of its statements. If  $T(n)$  is time for an algorithm on inputs, and if  $T(n) = O(f(n))$  mean that time is bounded by a function  $f(n)$ . If  $T(n) = \Omega(g(n))$  then the time is bounded below the function  $g(n)$ .

If  $f(n)$  is a polynomial, some problems are bounded by polynomial time algorithm and some for which no polynomial time known  $g(n)$  is larger than any polynomial.

Thus we can group algorithms into two groups :-

1. Problems whose solution is bounded by a polynomial of small degree.  
Ex : ordered search  $O(\log n)$  sorting  $O(n \log n)$  etc.
2. Problems whose solution is not bounded by a polynomial  
Ex : Traveling sales person  $O(n^2 2^n)$ . knapsack problem  $O(2^{n/2})$ .

### 8.2 Classes of Problems:

(1) **P** Problems (**P**: Polynomial): the set of all polynomially solvable problems

- 1 **P**. Problems that can be solved in polynomial time. ("P" stands for polynomial)
- 1 **P** is closed under addition, multiplication and composition.
- 1 **P** is independent of particular formal models of computation or implementations.
- 1 Any problem not in **P** is hard.
- 1 A problem in **P** does not necessarily have an efficient algorithm.

(2) **NP** Problems (**NP**: Nondeterministic Polynomial); the set of all problems that can be solved if we always guess correctly what computation path we should follow.

- + Roughly speaking: include problems with exponential algorithms but have not proved that they cannot have polynomial time algorithms.
- + Alternate definition (e.g. Corman): the set of all problems whose answers can be verified in polynomial time.
- + NP problems thus deal with decision problems as opposed to optimization problems.
- + **P** is a subset of **NP**, but we don't know whether **P = NP**.

(2") **NP-hard** problems the set of all problems such that any NP problems

## Design and Analysis of Algorithms

can be reduced to L in polynomial times.

(2") NP-Complete problems; the set of problems that are both NP and NP-Hard.

- 1 **NP-complete** problems are equivalent in the sense that if one problem has an efficient algorithm (i.e. in P), then all **NP-complete** problems have efficient algorithms.

(3) Intractable problems: the set of problems that have been proven not to have polynomial algorithms.

A problem is in NP if you can quickly (in polynomial time) test whether a solution is correct (without worrying about how hard it might be to find the solution). Problems in NP are still relatively easy: if only we could then quickly test it.

NP does not stand for "non-polynomial". There are many complexity classes that are much harder than NP.

- ✦ **PSPACE**. Problems that can be solved using a reasonable amount of memory (again defined formally as a polynomial in the input size) without regard to how much time the solution takes.
- ✦ **EXPTIME**. Problems that can be solved in exponential time. This class contains most problems you are likely to run into, including everything in the previous three classes. It may be surprising that this class is not all-inclusive; there are problems for which the best algorithms take even more than exponential time.
- ✦ **Undecidable**. For some problems, we can prove that there is no algorithm that always solves them, no matter how much time or space is allowed. One very uniformative proof of this is based on the fact that there are as many problems as there real numbers and only as many programs as there are inegers, so there are not enough programs to solve all the problems. But we can also define explicit and useful problems which can't be solved.

### 8.3 Reducibility:

A problem P1 is polynomial time reducible to P2 (P1 > P2), if

- ✦ There exists a one to one mapping from of input I1 of P1 to input I2 of P2.
- ✦ P1(I1) if and only if P2(I2)

Note that if P1 ->P2, then

- 1 If P2 is in **P**, then P1 is in P, but
- 1 If P1 is in **P**, it does not follow that P2 is in P.

NOTES

**Design and Analysis of Algorithms****NOTES****8.4 Non deterministic algorithms:**

If the result of every operation is uniquely defined, then the algorithm is called as deterministic algorithm. Some algorithms contain operations whose outcome are not uniquely defined but are limited to set off possibilities. Subjected to termination condition any one of these outcomes is selected.

A non deterministic algorithm terminates unsuccessfully if and only if there exists not set of choices leading to a successful outcome.

Ex: Searching an element  $x$  in a given set. If we are required to search an element  $x$  which is not in set with an index  $l = -1$ , non deterministic algorithm for this is

```
For(j=-l;j<n;j++)
```

```
: if A(j) = x then print(j)
```

```
else print ('failure');
```

```
:
```

The above algorithm is of nondeterministic complexity of  $O(1)$ .

**8.5 NP-Completeness:**

To prove that a problem  $P$  is NP-complete:

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

Variable: true or false.

Literal: positive or negative literals.

Clause literals or'ed together

CNF : clauses and'ed together.

CNF-Satisfiability problem: given a CNF are there assignments (of truth values) to variable in the CNF to make the CNF true.

**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  $P' \rightarrow P$ .

**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: Subgraph isomorphism: Given two graphs  $G(V_1, E_1)$  and  $H(V_2, E_2)$ , does  $G$  contain a subgraph isomorphic to  $H$ ? That is, can we find subset  $V$  of  $V_1$  and  $E$

**Design and Analysis of Algorithms****NOTES**

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) Subgraph isomorphism is in NP

(b)  $k$ -clique is a special case of subgraph isomorphism when  $H$  is a clique with  $k$  nodes.

**Solving NP-Complete Problems**

Given a NP-Complete Problems, what should you do?

- + Use brute force: may be the algorithm performance is acceptable for small input sizes.
- + Use time limit: terminates the algorithm after time limit.
- + Use approximate algorithms for optimization problems: find a good solution, but not necessarily the best solution.

How to measure the performance of an approximate algorithms?

P: Problem

I: Input

FS(I): the set of all feasible solution of P for input I.

V(I): FS(I)  $\rightarrow \mathbf{N}$ ; measure how good a feasible solution is.

opt(I): V(I) for the optimal solution for the input I.

For a feasible solution (provided by an approximate algorithm).  $A(I)$ . we can measure:

$$r(A, I) = \frac{V(A(I))}{\text{opt}(I)}$$

Note that  $r(A, I) = 1$ .

Thus.  $A(I)$  is optimal for I if  $r(A, I) = 1$ .

How good an approximate algorithm can be measure by worst case analysis with respect to  $\text{opt}(I)$  or input size.

$$R(A, m) = \max \{r(A, I) \mid \text{opt}(I) = m\}$$

$$S(A, n) = \max \{r(A, I) \mid I \text{ is of size } n\}$$

Example: the knapsack problem.

**8.6 Cook's Theorem:**

S.A. Cook raised a fundamental question: Is there a problem in NP that is as hard (up to polynomial factors) as every other problem in NP? In other words, is there a problem C in NP such that if A is any problem in NP, then  $A \leq C$ ? If such a problem exists then it is called NP-complete (Note that all NP-complete problems are automatically polynomially equivalent to one another.) In one of the most celebrated results in theoretical computer science. Cook showed in 1971 that the satisfiability problem for Boolean expressions (as defined presently) is

**Design and Analysis of Algorithms****NOTES**

NP-complete. About the same time, Levin in showed that a certain tiling problem was NP-complete. The following proposition follows easily from the transitivity of the relation  $\leq$ . Suppose A is in NP, and B is NP-complete. If  $B \leq A$ , then A is NP-complete. If any NP-complete problem A also belongs to P, then  $P = NP$

A problem A (not necessarily a decision problem) is NP-hard if it is as hard to solve as any problem in NP. More precisely, A is NP-hard if a polynomial time solution for A would imply  $P = NP$ .

For example, if the decision version of an optimization problem is NP-complete, then the optimization problem itself is NP-hard. Note that the NP-complete problems are precisely those problems in NP that are NP-hard.

To describe Cook's result, we need to recall some terms from order Logic. As with out pseudocode conventions, a Boolean (logical) variable is a variable that can only take on two values, true (T) or false (F). Given a Boolean variable  $x$  we denote by  $\bar{x}$  the variable has that the value T if and only if  $x$  has the value F. Boolean variables can be combined using logical operators and (denoted by  $\wedge$  and called conjunction), or (denoted by  $\vee$  and called disjunction), not (denoted by  $\neg$  and called negation), and arthesization, to form Boolean expressions (formulas). A literal in a Boolean formula is of the form  $x$  or  $\bar{x}$ , where  $x$  is a Boolean variable. Note  $\neg \bar{x}$  has the same truth value as  $x$ . A Boolean formula is said to be in Conjunctive Normal Form (CNF) if it has the form  $C_1 \wedge C_2 \wedge \dots \wedge C_m$ , where each clause  $C_i$  is a disjunction of  $n(i)$  literals.  $i=1 \dots m$ .

A Boolean formula is called satisfiable if there is an assignment of truth values to the literals occurring in the formula that makes the formula evaluate to true. For example the CNF formula  $(x_2 \vee x_3) \wedge (x_1 \wedge x_3) \wedge (x_1 \vee x_3)$  is satisfiable ( $x_1 = x_2 = x_3 = T$ ), whereas the CNF formula  $(x_2 \vee x_3) \wedge (x_1 \vee x_2) \wedge (x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$  is not satisfiable.

The CNF-satisfiability problem (CNF SAT) is the problem of determining whether a given CNF formula is satisfiable. CNF SAT is a fundamental problem in mathematical logic and computer science, with numerous applications. The input size of CNF formula can be defined as the total number of literals (counting repetitions) occurring in the formula. Clearly  $CNF SAT \in NP$ , since a linear scan of the formula determines whether a candidate assignment to the literals occurring in the formula results in the formula evaluation to true. CNF SAT was the first example shown to be complete.

## Sets and Unions

### 9.1 Fundamentals of Sets:

MEMBER(a,s): Determines whether a is a member of S , if so print yes otherwise no.

INSERT(a,S); It is used to insert the given element a into the set S

DELETE(a,S): It is used to delete an element a from set S.

UNION(S1,S2); It is used to merge two disjoint sets S1 and S2.

FIND(a) It is used to find the set for which a belongs

### 9.2 UNION & FIND:

Makeset(x) : creates a one element set (x) . Is assumed that this operation can be applied to each of the elements of set S only once

#### UNION

It constructs the union of the disjoint subsets. If there are two sets I and j then the UNION (I,J) is the set containing both the set elements of I and j. In the case of a tree UNION is represented as UNION (I,J) is means I is the parent and j is the child.

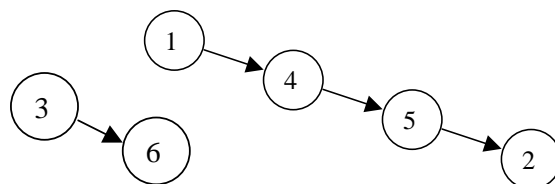
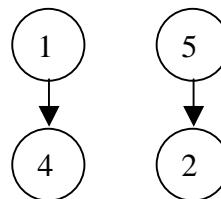
Ex: Let  $S = \{1,2,3,4,5,6\}$  Then make (i) creates the set {I} and applying this operation six times initializes the structure to the collection of six singleton sets.

$\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}$

Performing union (1,4) and union of (5,2) yields

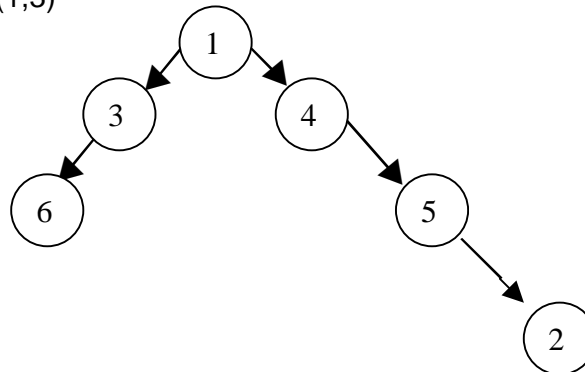
$\{1,4\}, \{5,2\}, \{3\}, \{6\}$

And if followed by union (4,5) and union (3,6)



NOTES

Again if we perform union (1,3)



Algorithm for UNION

UNION (I,j)

```
{
integer I,j;
```

```
  PARENT(j) = I;
}
```

Quick union:

Union (x,y) appends the y's list to the end of the x's list , update information about their representative for all the elements in the y's list and then delete the y's list from the collection. The sequence of union operations

Union(2,1),union(3,2)...union(l+1,l)...union(n,n-1) runs in  $q(n^2)$

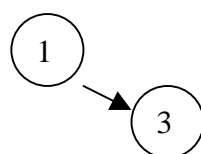
To improve the efficiency of a sequence of union operations is to always append the shorter of the two lists to the longer one, with ties broken arbitrarily. This is called as Union by size. The worst case time of the union by size is  $O(n \log n)$ .

Since union operation attaches a smaller tree to the root of a larger one, the size of the tree can be measured either by the number of nodes or by its height of the tree. Since height of the tree is  $O(\log n)$  each find requires  $O(\log n)$  and thus union requires at most  $n-1$  unions and  $m$  finds , to lead a time complexity of  $O(n \log n)$ .

FIND

Find(i) returns a subset containing i. It finds the root node of ith node.

Ex: UNION of (1,3)



FIND (3) =1

Algorithm for FIND(i)

```

FIND(i)
{
integer l,j;
j=l;
while (PARENT (j)>0 )
    {
        j=PARENT(j);
    }
return j;
}
    
```

Ex: FIND (5), l=5, j=l=5

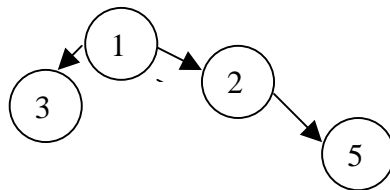
While P(j)>0

J=P(j)=2

Again P(2) >0

J=P(2)=1

P(1) >0 is false return j i.e., 1. Thus the root node of node 5 is 1.



Time complexity

Each FIND requires following a chain of PAREBT links from node 1 to the root. The time required to process FIND for an element at level l of the tree is O(l). Hence the total time needed to process n-2 finds is O(n<sup>2</sup>).

### 9.3 Weighting Rule for UNION:

If the number of nodes on tree l is less than the number in tree j, then make j the parent of l, otherwise make l the parent of j.

UNION(l,j)

```

{
integer l,j,x;
x =PARENT(i) + PARENT(j);
if (PARENT(i)>PARENT(j) )
    {
    
```

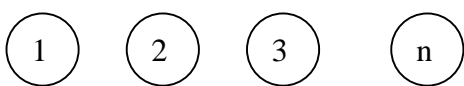
NOTES



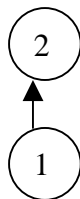
```

        PARENT(i)=j;
        PARENT (j)=x;
    Else
        PARENT(j)= i;
        PARENT(i) =x;
    }

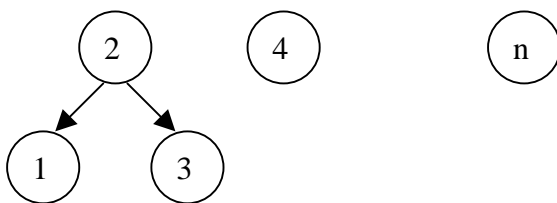
}
    
```

Ex: Let there are n sets 

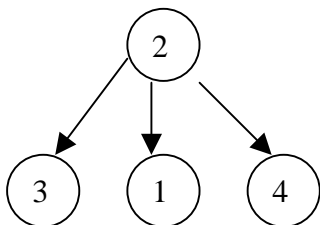
Union (1,2) the number of nodes in tree 1 equal to number of nodes in tree 2



Union(1,2) number of nodes in tree2 is 2, number of nodes in tree 3 is 1 , so 2>1 make parent as 2 and 3 as child



UNION(FIND(3),4)



COLLAPSING RULE

If  $j$  is a node on the path from  $l$  to its root then set  $PARENT(j) = root(i)$ .

FIND( $i$ )

```
{
    j=l;
    while PARENT(j) >0
    {
        j == PARENT(j);
    }

    k=l;
    while k!= j
    {
        t=PARENT(k);
        PARENT(k) =j;
        K=1
    }
    return j;
}
```

NOTES

## **BALANCED SEARCH TREES**

Binary trees which are used to implement dictionaries have a time efficiency of searching, insertion and deletion in the average case  $\theta(\log n)$ . In the worst case it is  $\theta(n)$ , because the tree can degenerate into a severely unbalanced one with height equal to  $n-1$ .

Thus a structure is needed to preserve the properties of binary search tree and also to avoid its worst case degeneracy.

1. The first approach is, transforming an unbalanced tree into a balanced tree. Examples are AVL trees and red-black trees.

2. The second approach is allowing more than one element in a node of a search tree. Examples are B-trees and 2-3-4 trees.

### **10.1 AVL Trees:**

These were invented in 1962 by G.M. Adelson-Velsky and E.M. Landis.

An AVL tree is a binary search tree in which the balance factor of every node which is defined as the difference between the heights of the nodes left and right subtrees is either 0 or +1 or -1.

A tree  $T$  is said to be height balanced if and only if

1.  $T_L$  and  $T_R$  are height balanced
2.  $h_L - h_R \leq 1$  where  $h_L$  and  $h_R$  are heights of left and right subtrees.

Rotations:

If an insertion of a new node makes an AVL tree unbalanced we transform the tree by a rotation. A rotation in an AVL tree is a local transformation of its subtree rooted at a node whose balance has become either +2 or -2.

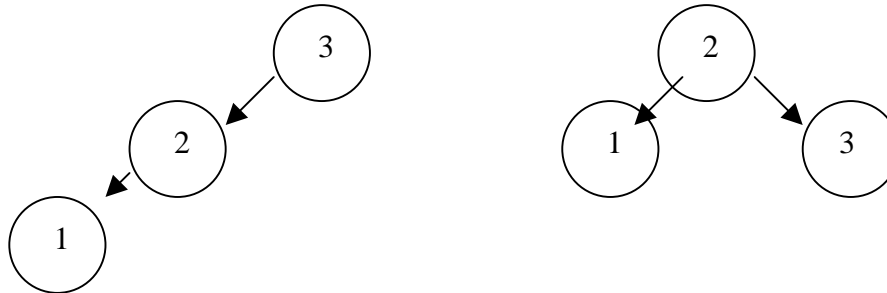
There are only 4 types of rotations. Two of them are mirror images of the other two. They are

1. Single R rotation
2. Single L rotation
3. Double LR rotation
4. Double RL rotation

## NOTES

**Single R Rotation:**

This is performed after a new key is inserted into the left subtree of the left child of a tree whose root had the balance of +1 before the insertion .



**NOTES**

**Single L rotation:**

It is performed after a new key is inserted into the right subtree of the right child of a tree whose root had the balance of -1 before the insertion.



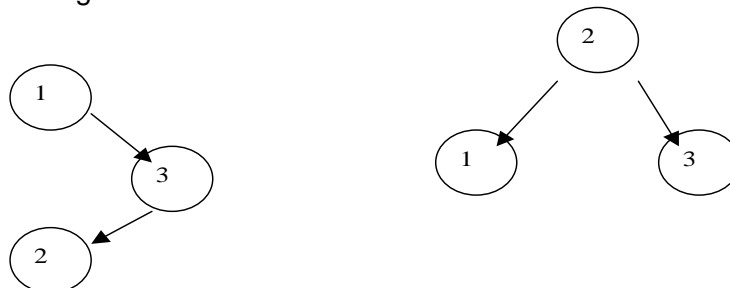
**Double left right rotation or LR rotation:**

We perform L rotation of the left subtree of root r followed by the R rotation of the new tree rooted at r. It is performed after a new key is inserted into the right subtree of the left child of a tree whose root had the balance of +1 before the insertion.



**Double right left rotation or RL rotation:**

It is the mirror image of double LR rotation. It is performed after a new key is inserted into the left subtree of the right child of a tree whose root had the balance of +1 before the insertion.



Time complexity:

The height  $h$  of any AVL tree with  $n$  nodes satisfies

$$\log n \leq h \leq 1.4405 \log(n+2) - 1.3277$$

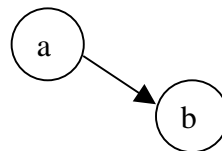
The operations of searching and insertion are  $\theta(\log n)$  in worst case. On an average AVL tree requires same number of comparisons as searching in a sorted array by binary search.

Algorithm to create an AVL TREE:

1. Find the place to insert a new element from root. Keep track the most recently seen node with balance factor  $-1$  or  $+1$ . Let it be node  $x$ .
2. If there is no node  $x$ , then make another pass from the root, updating balance factors.
3. If  $BF(x)=1$  and the new node was inserted in the right subtree of  $x$  or if  $BF(x)=-1$  and the insertion took place in the left subtree, then the new balance factor of  $x$  is  $0$ . In this case update balance factors on the path from  $x$  to the new node and terminate.
4. Classify the imbalance at  $x$  and perform the appropriate rotation. Change balance factors as required by the rotation as well as those of nodes on the path from the new subtree root to the newly inserted node.

Ex: Let us construct an AVL tree with following elements a,b,c,d,e,f,g,h.

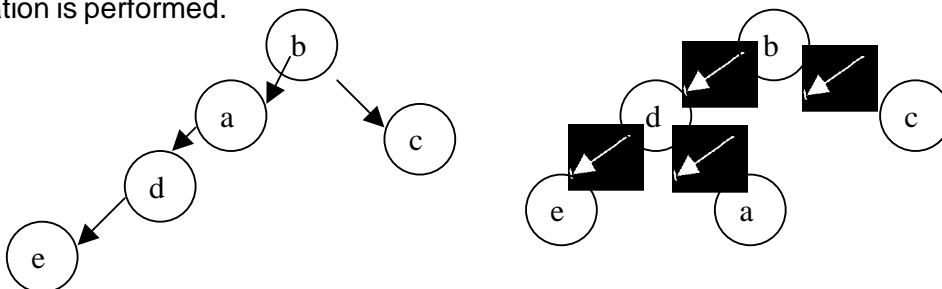
Inserting a and b results bst shown in fig 1



When c is inserted into the tree, the height of the right subtree of a becomes 2 and left subtree is 0. To rebalance the tree a rotation is performed.



Again introducing d and e makes tree unbalanced. To rebalance the tree another rotation is performed.



## 10.2 2 - 3 TREES:

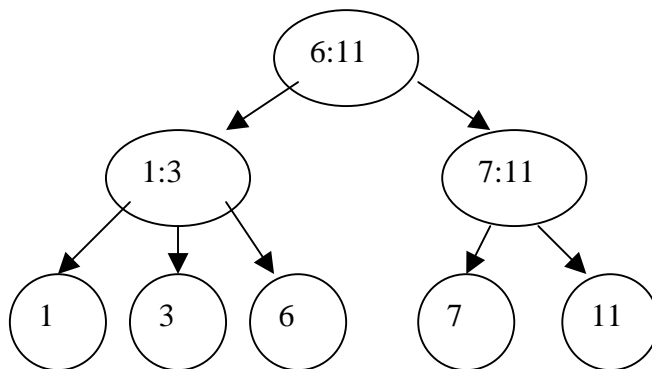
It is a tree that can have nodes of two kinds: 2-nodes and 3-nodes.

A 2 node contains a single key  $K$  and has two children: the left child serves as the root of a subtree whose keys are less than  $K$  and the right child serves as the root of a subtree whose keys are greater than  $K$ .

A 3-node contains two ordered keys  $K_1$  and  $K_2$  and has three children. The left most child serves as the root of a subtree with key less than  $k_1$ , the middle child serves as the root of a subtree with keys between  $K_1$  and  $K_2$ , and the right most child serves as the root of a subtree with keys greater than  $K_2$ .

Also all its leaves must be on the same level i.e., a 2-3 tree is always height balanced: the length of a path from the root of the tree to a leaf must be the same for every leaf.

Ex:



In the above example a non leaf node 1:3 can be obtained as follows. It has 3 children 1,3,6 among which, largest left subtree value is 1, middle subtree value is 3, so we will get 1:3 similarly 7:11 obtained. Next we will get 6:11 the largest left subtree value is 6 largest middle subtree value is 11 no right subtree exists.

Searching in 2-3 tree:

For example if we want to search an element is present in 2-3 tree or not, first 7 is compared with root node, i.e, 7 is greater than 6 and less than 11. So move down the middle branch again 7 is compared with 7:11 node, here 7 is less than or equal to first value i.e., 7, so move down the left subtree. Now here node is leaf node, the value of leaf node i.e., 7 is compared with 7, so matching occurs.

Algorithm

1. Repeat while not a leaf node
2. If key  $\leq$  the first search value in the node  
Then move down the left branch

NOTES

Else if key ≤ the second search value in the node

Then move down the middle branch

Else if a right branch exists

Then move down the right branch

Else unsuccessful search and return

3. If the key has been found

then successful search and return

else unsuccessful search and return.

Insertion in AVL Tree:

When we are inserting there are 4 cases,

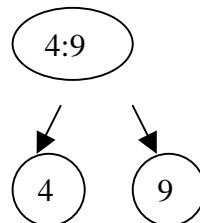
**Case 1:** The tree before insertion is empty. In this case we create the node to be inserted, and make this the root of the tree.

Ex: Insert 4, initially 2-3 tree is empty, so node is created.



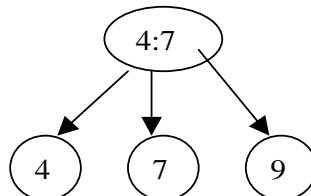
**Case 2:** The tree contains just one node. In this case a non leaf node is created, with the previous root node and the node being inserted as the new nodes children.

Ex: Insert 9 in above 2-3 tree. So we have to create a non leaf node with first value as 4, second value as 9 i.e., 4:9.



**Case 3:** If the parent has two children, insert the new node into its proper position below the parent. If its key value is less than that of the left child, the new node becomes the left child; if its key value is greater than that of the middle child, the new node becomes the right child, otherwise, it becomes the middle child.

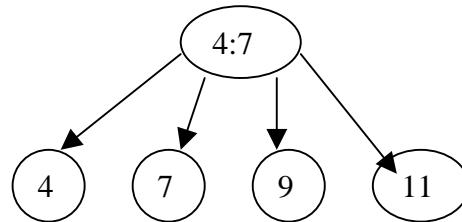
Ex: Insert 7 in the above tree. Here for non leaf node, largest left subtree value is 4, largest middle subtree value is 7. so it is 4:7



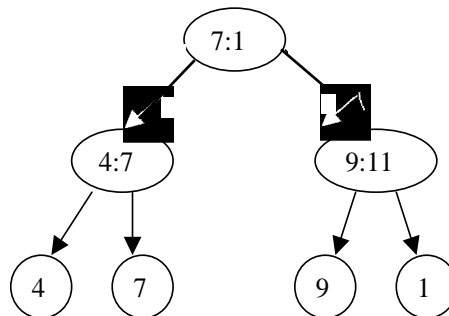
child, in its proper position, of the parent. A new non leaf node is created and it becomes a brother to the immediate right of the parent node. The two right most children of the parent node now become the offspring of the new non leaf node. If the addition of the new non leaf node causes its parent to have four children the

process is repeated on this node.

Ex: Insert 11 in above 2-3 tree



Any non leaf node in 2-3 tree can have either 2 or 3 leaf nodes, so we have to split the nodes



Algorithm

INSERT(V)

```

{
create a new vertex v;
make the two right most sons of v the left and right sons of v;
if v has no father then
    {
    create a new root r;
    make v the left son and v the right son of r
    }
else
    {
    let f be the father of v;
    make v a son of f immediately to the right of v
    if f now has four sons then INSERT(f)
    }
}

```

NOTES

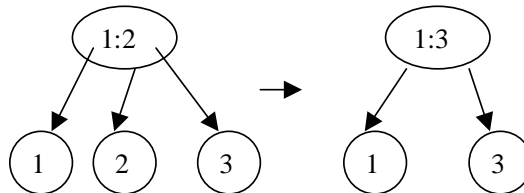


Deletion of 2-3 Tree:

While deleting a node from 2-3 tree there are 3 cases

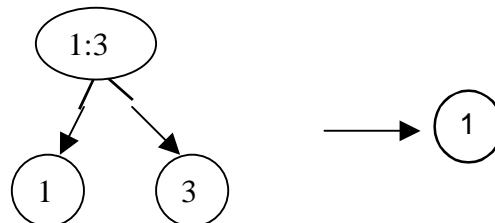
**Case 1:** If X is the root remove x.

**Case 2:** If x is the son of a vertex having three sons remove x.



**Case 3:** If x is the son of a vertex f having two sons s and x then there are two possibilities:

- a) f is the root. Remove x and f and leave the remaining son s as the root.
- b) F is not the root. Suppose f has a brother g to its left. A brother to the right is handled similarly. If g has only two sons, make s the rightmost son of g, remove x and call the selection procedure recursively to delete f. If g has 3 sons, make the rightmost son of g be the left son of f and remove x from tree.



Time Complexity

A 2-3 tree of height h with the smallest number of nodes is a full tree of 2 nodes. Therefore for any 2-3 tree of height h with n nodes,

$$n \geq 1 + 2 + \dots + 2^h = 2^{h+1} - 1 \text{ and hence } h \leq \log_2(n+1) - 1$$

On the other hand 2-3 tree of height h with the largest number of nodes is a full tree of 3 nodes each with two keys and three children. For any 2-3 tree with n nodes

$$n \leq 2 + 2 \cdot 3 + \dots + 2 \cdot 3^h = 2(1 + 3 + \dots + 3^h) = 3^{h+1} - 1 \text{ and hence } h \leq \log_3(n+1) - 1$$

Thus searching, insertion and deletion are all in  $\theta(\log n)$

NOTES

### **10.3 DICTIONARY:**

A data structure that implements adding a new item, deleting an item and searching for a given item from a collection is called dictionary. A dictionary is a storage structure, having an identifier and information. The information is associated with the identifier.

The important aspect of the dictionary is that nay stored information can be retrieved at any time. We can create an empty dictionary then store pairs (id, inf) in it. The dictionary is very useful in the design of dynamic programming.

### **10.4 PRIORITY QUEUE:**

A priority queue is a structure with some aspects of a FIFO but in which element order is related to each elements priority rather than its chronological arrival time.

Each element is inserted into a priority queue , conceptually it is inserted in order of its priority. The one element that can be removed is the most important element currently in the priority queue.

The principal operations on a priority queue are finding largest/smallest element, deleting its largest/smallest element and adding a new element. The largest/smallest element can be found in  $O(\log n)$  time. Therefore any sequence of  $n$  INSERT, DELETE operations can be done in  $O(n \log n)$  steps. Two other data structures that can be used to implement an  $O(n \log n)$  priority queue are the heap and AVL tree.

### **10.5 HEAP:**

A heap can be defined as a binary tree with keys assigned to its nodes provided the following conditions are met

- 1.It must be essentially complete, all its levels are full except possibly the last level, where only some right most leaves may be missing.
2. The key at each node is greater than or equal to the keys at its children. The key values in a heap are ordered top down.

Properties of heap

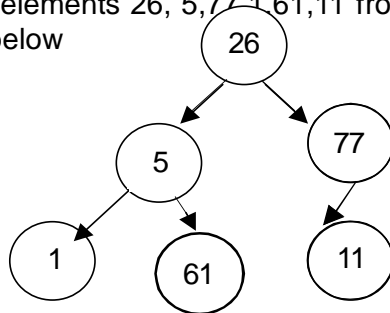
1. There exists exactly one complete binary tree with  $n$  nodes. Its height is  $\log n$ .

NOTES

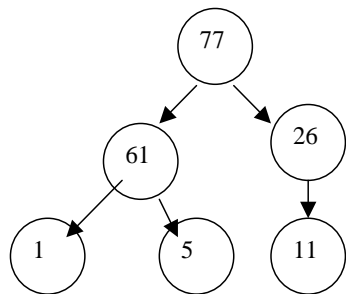
2. The root of a heap always contains its largest element.
3. A node of a heap considered with all its descendants is also a heap.

**NOTES**

Ex: let us consider the elements 26, 5, 77, 1, 61, 11 from which a binary tree is constructed as shown below



Applying heap properties for all nodes in this tree, we get



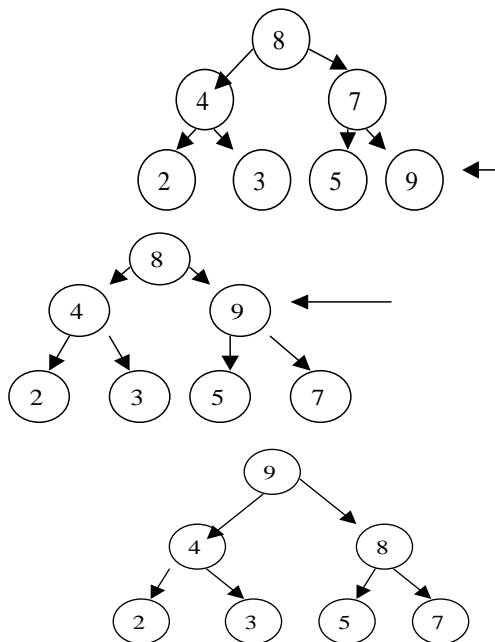
**Insertion into a Heap**

Each new element is added at the bottom of the heap and then compares it with its parent, grand parent and so on until it is less than or equal to one of these values.

**Algorithm for Insertion**  
**INSERT(A,n)**

```

{
  integer l,j,n;
  j=n;
  l=n/2;
  Item=A(n);
  While(l>0 and A(l)<Item)
  {
    A(j)=A(l);
    J=l;
    l=l/2;
  }
}
    
```



```

    }
    A(j)=item;
}

```

NOTES

Time complexity:

There are at most  $2^{i-1}$  nodes on level  $i$  of a complete binary tree,  $1 \leq i \leq \log(n+1)$ . For a node on level  $i$  the distance to the root is  $i-1$ . Thus the worst case time for heap creation is

$$\begin{aligned}
 \sum (i+1)2^{i-1} &< \log(n+1)2\log(n+1) \\
 &= \log(n+1)(n+1) \\
 &= \log^n(n+1) \\
 &= n \log n + \log n = O(n \log n)
 \end{aligned}$$

Deletion from a heap

Since we are deleting largest element from the heap, the element at the root is removed. After removing the root element, it is replaced by last element and again heap is adjusted using below algorithm.

HEAP Adjust

ADJUST(A,i,n)

```

{
    integer j,k;
    k=A(i)
    j=2i;
    while(j<=n)
    {
        if(j<n)
            if (A(j)<A(j+1)) then j=j+1;
        If(k>=A(j)) exit;
        Else
        {
            A(j/2)=A(j);
            J=2j;
        }
    }
}

```

## 10.6 HEAP SORT:

Sorting can be performed using heap. IF we delete every time root element and adjust the heap until all the elements in the heap are deleted, we get a sequence of elements arranged in descending order.

```

HEAPSORT(A,n)
{
    integer i,t;
    for (l=n/2;i>0;i--)
        ADJUST(A,i,n);
    For(i=n-1;i>0i--)
    {
        t=A(i+1);
        A(l+1)=A(i);
        A(i)=t;
        ADJUST(A,1,i);
    }
}

```

Time complexity

In the above algorithm , every time adjust function is called, the time required for this procedure is  $\log n$ .

So time complexity  $T(n) = (n-1)\log n$   
 $= n\log n - \log n$   
 $= O(n\log n)$

### MERGEABLE HEAP

In this heap insertion, deletions are executed in  $O(n\log n)$  time. The smallest element in the set S can be found by starting at the root of T and walking down the tree as follows.

If we are at an interior vertex v, we next visit the son of v with the lowest value of SMALLEST. Then if T has n leaves the instruction MIN requires  $O(\log n)$ .

To merge two sets S1 and S2 we call the procedure IMPLANT(T1,T2) where T1 and T2 are the 2-3 trees representing S1 and S2.

NOTES

Let us suppose that  $H_1$ , the height of  $T_1$  is greater than or equal to  $H_2$  the height of  $T_2$ . IMPLANT finds on the right most path in  $T_1$ , the vertex  $v$  which is of height  $H_2$  and makes the root of  $T_2$  the right most brother of  $v$ ,  $f$  be a father of  $v$ , now  $f$  has four sons. IMPLANT calls the function of INSERT( $f$ ).

Time complexity

The procedure IMPLANT combines  $T_1$  and  $T_2$  into a single 2-3 tree in time  $O(h_1 - h_2)$

Algorithm IMPLANT

IMPLANT( $T_1, T_2$ )

```
{
if HEIGHT( $T_1$ )=HEIGHT( $T_2$ )
    {
    create a new root  $r$ ;
    make ROOT( $T_1$ ) and ROOT( $T_2$ ) the left and right sons of  $r$ 
    }
else
    assume HEIGHT( $T_1$ )>HEIGHT( $T_2$ ) otherwise interchange  $T_1$  and  $T_2$ 
    {
    let  $v$  be the vertex on the right most path of  $T_1$  such that
    DEPTH ( $v$ )=HEIGHT( $T_1$ )-HEIGHT( $T_2$ )
    Let  $f$  be a father of  $v$ ;
    Make ROOT( $T_2$ ) a son of  $f$  immediately to the right of  $v$ ;
    If  $f$  now has four sons then INSERT( $f$ )
    }
}
```

\*\*\*\*\*