ANALYSIS & DESIGN OF ALGORITHMS

(ADA)

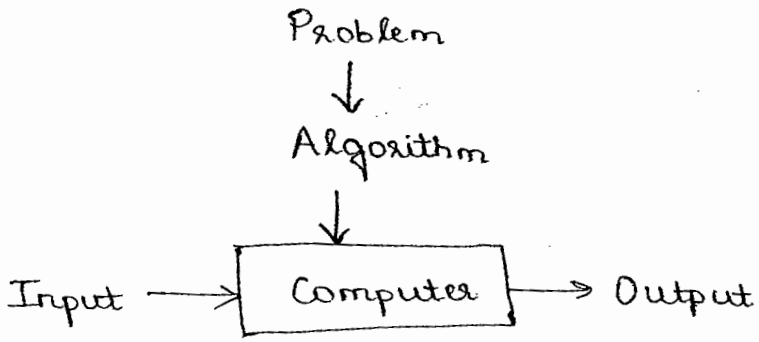
CAMPUS BOOK MART
BMSCE
0% to 25% Discount on Text Books
Buy Back on Text Books Upto 60%
Repurchase Rights are with us

NAMRATHA M
CSE
BMSCE

INTRODUCTION

An algorithm is a sequence of unambiguous instructions for solving a problem i.e for obtaining required output for any legitimate input in a finite amount of time.

Notion of an algorithm:



An algorithm must have certain properties such as:

1. The set of statements of an algorithm must be non-ambiguous.
2. The range of inputs must be specified correctly.
3. It should be effective and definite
4. It should produce the output in finite amount of time.

The same algorithm can be represented in several different ways. There may be several algorithms for solving the same problem. These algorithms may execute with different speeds.

Algorithm to compute GCD of two numbers :

1. Euclid's algorithm : (using mod operation)

m	n	r
10	6	4
6	4	2
4	2	0
2	0	

$\therefore$ GCD (10, 6) = 2

Algorithm GCD (m, n)
// Purpose : To compute GCD using Euclid's algorithm
// Inputs : Two positive integers 'm' and 'n'
// Output : 'm' contains the GCD

```
while n ≠ 0
    r ← m mod n
    m ← n
    n ← r
end while
return m
```


The above algorithm is iterative . To write a recursive algorithm, the recursive definition is as shown below :

$$GCD(m,n) = \begin{cases} m & \text{if } n = 0 \\ GCD(n, m \bmod n) & \text{otherwise} \end{cases}$$

To compute GCD using repetitive subtraction:

Algorithm GCD(m, n)

// Purpose: To compute GCD using Euclid's algorithm

// Inputs: Two positive integers 'm' and 'n'

// Output: 'm' contains the GCD

```
while m ≠ n
    if m > n
        m ← m - n
    else
        n ← n - m
    end if
end while
return m
```

The recursive definition is as shown below:

$$GCD(m,n) = \begin{cases} m \text{ OR } n & \text{if } m = n \\ GCD(m-n, n) & \text{if } m > n \\ GCD(m, n-m) & \text{otherwise} \end{cases}$$

2. Consecutive integer checking method:

m	n	small	m % small	n % small
10	6	6	10 % 6 = 4	6 % 6 = 0
		5	10 % 5 = 0	6 % 5 = 1
		4	10 % 4 = 2	6 % 4 = 2
		3	10 % 3 = 1	6 % 3 = 0
		2	10 % 2 = 0	6 % 2 = 0

∴ GCD(10, 6) = 2

Algorithm GCD(m,n)

// Purpose: To find the GCD using consecutive integer checking method

// Inputs: 2 positive integers 'm' and 'n'

// Output: GCD of 'm' and 'n'

```
small ← min(m,n)
while ( m % small ≠ 0  or  n % small ≠ 0)
    small ← small - 1
end while
return small
```

3. Middle school method:

To compute GCD of 140 and 160

Compute prime factors of 160:

```
2 | 160
2 | 80
2 | 40
2 | 20
2 | 10
    5
```

The prime factors are 2X2X2X2X5

Compute prime factors of 140:

```
2 | 140
2 | 70
5 | 35
    7
```

The prime factors are 2X2X5X7

Compute common prime factors of 160 and 140

2X2X5 = 20 ∴ GCD = 20

The algorithm to find prime numbers:

Step 1: Generate all the numbers from 2 to n and store in the memory locations from a[2] to a[49] where n = 49.

This can be done using the following statement,

```
for i ← 2 to n
    a[i] ← i
end for
```

Step 2: We eliminate multiples of 2 except 2 starting from 2*2 upto n. Then we eliminate multiples of 3 except 3 starting from 3*3 upto n. Note that we are not considering multiples of 4 because multiples of 4 are also multiples of 2 and they have already been eliminated. Eliminate multiples of 5 except 5 starting from 5*5 upto n. Multiples of 6 have not be considered since they are multiples of 2 and they have already been eliminated. We eliminate multiples of 7 except 7 starting from 7*7 to n.

In general, if p is a prime number we have to eliminate all multiples of p except p starting from p*p upto n

i.e $p * p \leq n$

$p^2 \leq n$

$p \leq \sqrt{n}$

The equivalent steps are,

```
for p ← 2 to √n
        if a[p] ≠ 0
```

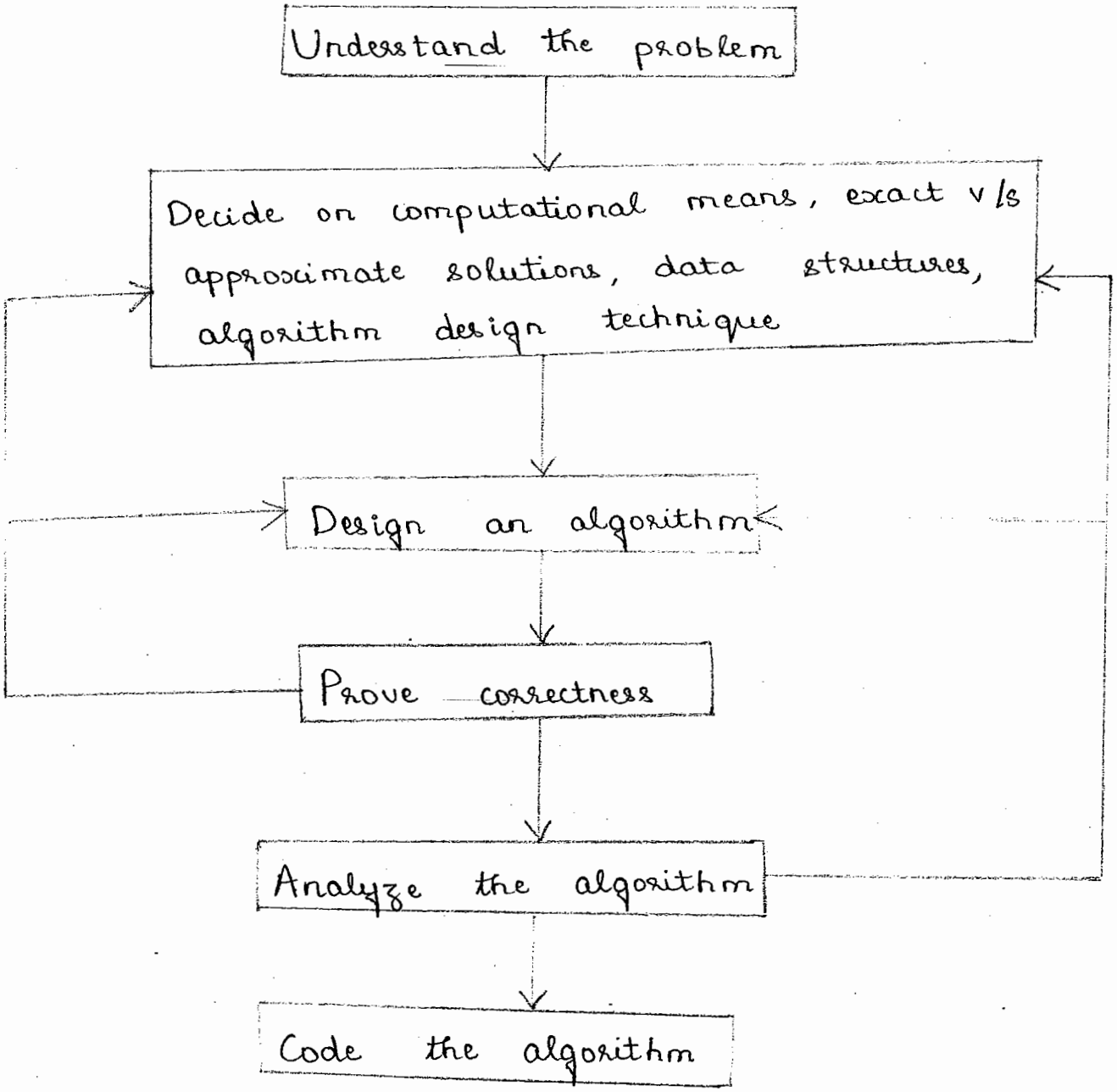
```
        i ← p*p
        while i ≤ n
            a[i] ← 0
            i ← i+p
        end while
    end if
end for
```

The complete algorithm to generate prime numbers within 'n' using Sieve of Eratosthenes method.

```
Algorithm sieve (n, b)
// Purpose : To generate prime numbers within 'n'
// Input : A positive integer 'n'
// Output : 'b' contains the prime numbers within 'n'
    for i ← 2 to n
        a[i] ← i
    end for
    for p ← 2 to √2
        if a[p] ≠ 0
            i ← p*p
            while i ≤ n
                a[i] ← 0
                i ← i+p
            end while
        end if
    end for
    k ← 0
    for i ← 2 to n    // Copy prime numbers to array 'b'
        if a[i] ≠ 0
            b[k] ← a[i]
            k ← k+1
        end if
    end for
```

```
for i ← 0 to K-1      // Output the prime numbers
    print b[i]
end for
```

Fundamentals of algorithmic problem solving:



1. Understanding the problem:

   Read the problem's description and understand the problem clearly without any doubts.

There may be several algorithms available to solve the same problem. But in most of the cases we have to design our own algorithm. An input to an algorithm specifies an instance of the problem that the algorithm solves. The exact range of instances must be specified. A correct algorithm must work correctly for all legitimate inputs.

2. Ascertaining the capabilities of a computational device:

Most of the algorithms usually run on machines that use Von Neumann architecture and hence they must be sequential algorithms. Some newer computers can execute operations concurrently i.e parallely. Mostly we do not give importance to speed and memory of computer but make sure that the algorithm can run on all machines independent of its specification parameters.

Choosing between exact and approximate problem solving:

Problems such as extracting square roots, solving non-linear equations and evaluating definite integrals are all approximation algorithms. Exact algorithms in such case maybe very slow and complex. Hence based on the application either an exact or approximate algorithm may be chosen.

Deciding appropriate data structures:

Based on the algorithm designed a suitable data structure may be used to increase the efficiency of the algorithm.

3. Algorithm design techniques:

An algorithm design technique is a strategy to solving problems algorithmically that is applicable to several problems from several areas of computation. These techniques serve as a method to categorize and study algorithms.

3 methods are used for specifying an algorithm:

a) Natural language

b) Pseudocode

c) Flowchart

Natural language is easy to use but there may be inherent ambiguity because it is difficult to represent certain things clearly.

A pseudocode is a mixture of natural language and programming language like constructs. It is more precise than natural language. There is no single rule to represent variables, assignment, etc when we write pseudocode.

Flowchart uses a collection of connected geometric shapes containing descriptions of the algorithm's steps. This is inconvenient for simple algorithms.

Whether an algorithm is represented using natural language or pseudocode or flowchart it cannot be directly fed to the computer, it has to be converted into a particular programming language understood by the computer.

4. Proving the correctness of an algorithm:

An algorithm must produce the required result for every legitimate input in a finite amount of time. The proof of correctness may be simple or complex. Mathematical induction may be used to prove correctness. To prove that an algorithm is incorrect we just need one instance of input for which the algorithm fails. If an algorithm is incorrect we need to redesign it and may also use different data structures. For approximate algorithms we need to show that the error produced by the algorithm does not exceed the predefined limit.

5. Analyzing the algorithm:

The 2 kinds of algorithm efficiency are,

a) Time efficiency

b) Space efficiency

Time efficiency indicates how fast an algorithm runs and space efficiency indicates how much

extra memory the algorithm needs.

An algorithm must be simple. Simpler algorithms are easy to understand and easy to program. But simple algorithms maybe or may not be efficienct.

An algorithm must be general, i.e generality of the problem it solves and generality of the range of inputs accepted. Sometimes it may be unnecessary or difficult to design such an algorithm.

If at all any of the above mentioned properties are not satisfied by the algorithm then we have to redesign the algorithm.

6. Coding the algorithm:

All the algorithms designed must be implemented as a computer program. After a program is written its' validity is proved using testing. We assume that the inputs to the algorithm lie within the range and hence require no verification. A good algorithm always involves repeated effort and also rework.

Important problem types:

1. Sorting: Rearrange the items of a given list in ascending order (or descending order). An example of sorting is the names listed in a telephone directory in alphabetical order. Sorting helps the searching process. Many algorithms are available for sorting such as bubble sort, selection sort, insertion sort, merge sort, quick sort, heap sort, etc. Each one of the above use different techniques and hence have different efficiency. There is no single algorithm which is best in all cases. Some algorithms maybe simple but slow while some other algorithms maybe complex but relatively fast.
A sorting algorithm is said to be stable if it preserves the relative order of any 2 equal elements in it's input. If the input list contains 2 equal elements at positions i and j such that $i < j$ then in the sorted list they must be at positions i' and j' such that $i' < j'$.
A sorting algorithm is said to be in place if it does not require any extra memory.

2. Searching: To find a given value i.e the search key in a given set of elements. Sequential search uses straight forward linear approach whereas binary search operates only on sorted arrays. Even in case of searching there is no single algorithm that fits the best in all situations. Some algorithms may be fast but may require more memory whereas some algorithms maybe fast but maybe applicable only to sorted arrays. Here stability problem does not occur. When frequent changes are made to the list in that case searching must be performed in conjuction with addition and deletion operations.

3. String processing: A string is a sequence of characters from an alphabet. String matching is an important problem that searches for a given word in a text.

4. Graph problems: Graph is a collection of points called vertices and some of these vertices are connected by line segments called edges. The various graph algorithms include graph traversals, shortest path algorithms, topological sorting for graphs with directed edges.

Ex: The traveling salesman problem asks us to find the shortest tour through 'n' cities that visits every city exactly once.

The graph colouring problem is to find the smallest number of colours needed to colour the vertices of a graph such that no 2 adjacent vertices have the same colour.

5. Combinatorial problems: Traveling salesman problem and graph colouring problem can be considered as combinatorial problems too. These kind of problems involve finding a combinatorial problem / object such as permutation, combination, subset, etc. They are the most difficult problems for computation both theoretically and practically. The number of combinatorial objects grows extremely fast with growing size of a problem. Also there are no algorithms for solving such problems in acceptable amount of time.

6. Geometric problems: Problems that include points, lines, polygons are called geometric problems. These kind of problems are helpful in fields such as computer graphics, robotics, tomography.

Ex: The closest-pair problem is used to find the closest pair among 'n' points of a given plane. The convex-hull problem is used to find the smallest convex polygon that includes all points of a given set.

7. Numerical problems: Problems that involve mathematical objects of continuous nature.

Ex: Solving system of equations
Computing definite integrals
Evaluating functions

Most of these numerical problems can be solved using approximate algorithms. When a large number of arithmetic operations are performed on numbers which are represented approximately can lead to the accumulation of round off error which may sometimes distort the output.

Linear data structures:

A data structure is a particular scheme of organizing related data.

Two elementary data structures:

a) Array
b) Linked list

A array is a sequence of 'n' items of the same data type that are stored contiguously and are accessed using index (ranges from 0 to n-1). Each element of the array can be accessed in the same constant amount of time. Arrays are used to represent strings. Various operations such as computing the length of the string, concatenating 2 strings, finding substring, etc can be performed.

A linked list is a sequence of zero or more elements called nodes containing a data field and an address field which points to the next node in the linked list. In a singly linked list each node points to the next node in the list and the last field contains NULL. To access a particular node we traverse the list starting from the first node. Insertions and deletions can be made efficiently and quickly. The first node in a linked list is called a header.

In a doubly linked list, every node except the first and last contains pointers to both predecessor and successor.

Two special kinds of lists are:

a) Stack

b) Queue

A stack is a data structure in which insertions and deletions can be done at only one end called top of the stack. (Last In First Out - LIFO). Used for implementation of recursive algorithms.

A queue is a data structure in which elements are inserted from the rear end and deleted from front end (First In First Out - FIFO). Queues are used for graph problems.

A priority queue is a data structure in which data items are in a particular order (ascending order or descending order). It can be implemented using a sorted array or a heap. Whenever additions or deletions are done we must make sure that the data items are still in order.

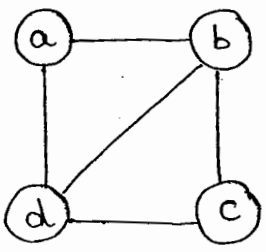
GRAPHS :

A graph $G = (V, E)$ is defined as a finite set of vertices and a set of edges E.

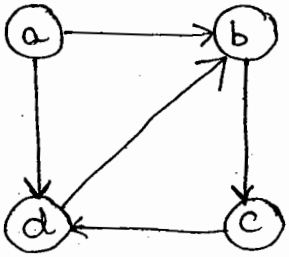
If the pair of vertices are unordered i.e (u,v) is same as (v,u) then (u,v) is called an undirected edge 'u' and 'v' are called endpoints of the edge. 'u' and 'v' are incident on this edge. An undirected graph is a graph in which all edges are undirected.

If the pair of vertices (u,v) is not the same as (v,u) then (u,v) is called a directed edge. 'u' is called the edge's tail and 'v' is called the edge's head. Directed graph or diagraph is a graph in which all edges are directed.

Ex: Undirected graph



Directed graph



The number of possible edges $|E|$ in a graph i.e undirected graph with $|V|$ vertices is

$$0 \leq |E| \leq |V|(|V|-1)/2$$

A complete graph is a graph in which every pair of vertices is connected by an edge. It is denoted as $K_{|V|}$.

A graph in which only a few edges is missing is called dense graph. A graph in which there are a few edges relative to the number of vertices is called a sparse graph.

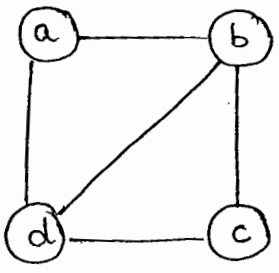
Graph representations:

1. Adjacency matrix
2. Adjacency linked list

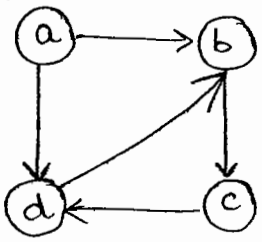
Adjacency matrix of a graph with 'n' vertices is n-by-n boolean matrix in which the ith row and jth column is 1 if there is an edge from vertex i to vertex j and 0 if there is no edge.

The adjacency matrix of an undirected graph is symmetric i.e $A[i,j] = A[j,i]$ for $0 \leq i, j \leq n-1$

Ex:



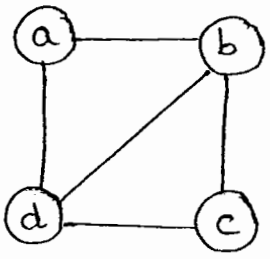
	a	b	c	d
a	0	1	0	1
b	1	0	1	1
c	0	1	0	1
d	1	1	1	0

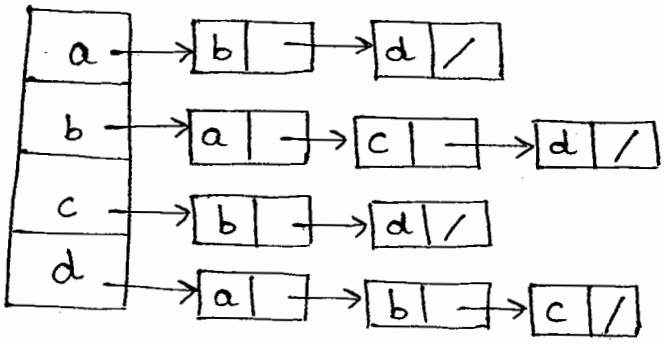


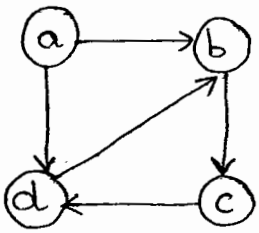
	a	b	c	d
a	0	1	0	1
b	0	0	1	0
c	0	0	0	1
d	0	1	0	0

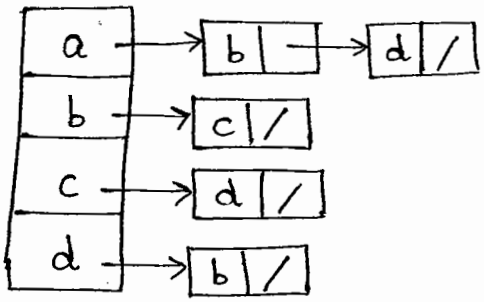
Adjacency linked list of a graph is a collection of linked lists one for each vertex that contains all the vertices adjacent to the list's vertex.

Ex:





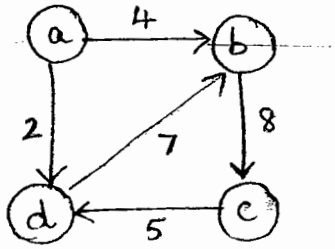




Adjacency linked list is used to represent sparse graphs because it occupies less. Adjacency matrix is used to represent dense graphs because there are more number of edges and hence there are less number of zeros in the matrix but if we use adjacency linked list more number of pointers are needed since there are more number of edges and hence memory is wasted.

Weighted graphs: Is a graph with numbers assigned to its edges. (weights/costs). They can be represented using adjacency matrix (weighted matrix) or adjacency linked lists.

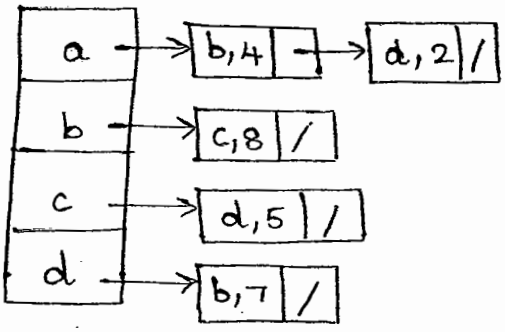
Ex:



Weighted matrix

	a	b	c	d
a	0	4	∞	2
b	∞	0	8	∞
c	∞	∞	0	5
d	∞	7	∞	0

Weighted adjacency list



In the weighted matrix, an entry is ∞ if there is no edge. If there is an edge then the cost of that edge is written in the matrix. An entry is 0 from a vertex to the same vertex.

In adjacency linked list representation, along with the vertices even the cost is entered.

PATH: A path from a vertex u to vertex V of a graph G is a sequence of adjacent vertices that starts from u and ends at v. If all the vertices of a path are distinct then the path is simple.

The length of a path is the total number of vertices minus one or the number of edges in the path.

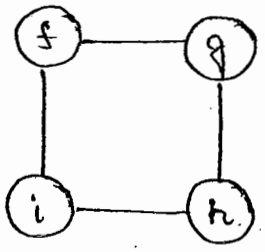
A directed path is a sequence of vertices in which every pair of consecutive vertices is connected by a directed edge.

A graph is said to be connected if for every pair of vertices there is a path.

In an unconnected graph there may be several components (which maybe connected).

CYCLE: A cycle is path (of positive length) that starts and ends at the same vertex and does traverse one edge only once.

Ex:



fghif is a cycle

fghi is a path from f to i

TREES: A tree is a connected acyclic graph.

A forest is an acyclic graph (may not be connected)

The number of edges in a trees is always one less than the number of vertices

$$|E| = |V| - 1$$

ROOTED TREES :

For every 2 vertices in a tree there always exists exactly one simple path from one of these vertices to another. In a rooted tree, the root node is placed at the top of the tree. Trees are used to represent hierarchical structure of files and directories.

For any vertex v in a tree all the vertices in the simple path from root to the vertex are called ancestors of v. The vertex is ancestor of itself. The set of ancestors that exclude the vertex is known as proper ancestors.

If (u,v) is the last edge of the simple path from u to v then u is the parent of v and v is the child of u. Vertices that have the same parent are called siblings. A vertex with no children is called leaf. All the vertices from a vertex v which are below the level of v are called descendants of v.

The depth of a vertex v is the length of the simple path from root to v.

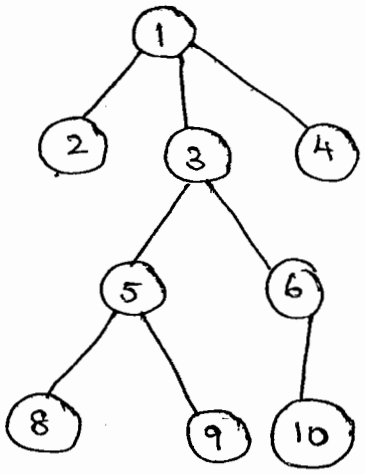
The height of a tree is the length of the longest simple path from the root to a leaf.

An ordered tree is a rooted tree in which all the children of each vertex are ordered. (left to right)

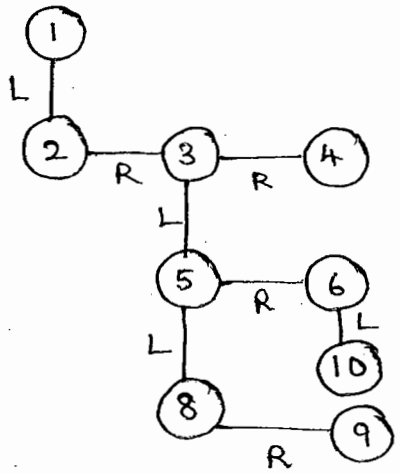
A binary tree can be defined as an ordered tree in which every vertex has a maximum of 2 children (left child or right child).

A binary tree in which the parent is greater than all the values in the left subtree but lesser than all the values in the right subtree is called a binary search tree.

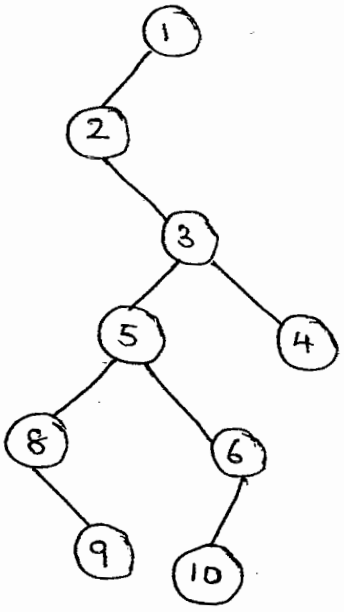
Consider a general tree as shown below:



The above tree can be represented as shown below: (First child - next sibling representation)



The binary tree is,



SETS: A set is an unordered collection of distinct items (elements). The set elements can be specified explicitly or by specifying a property which describe all the elements of a set.

2 ways of implementation:

1. Consider an universal set U, then a subset S can be represented by a bit string of size 'n' known as a bit vector in which an element i is 1 if it is present in S.
2. Representation using list structure:
   A list is an ordered collection of items i.e exactly opposite of sets. But this distinction is not important in many applications.

DICTIONARY: A data structure that implements the 3 operations searching, insertion and deletion is called dictionary.

FUNDAMENTALS OF THE ANALYSIS OF ALGORITHM EFFICIENCY

Time efficiency indicates how fast an algorithm runs. Space efficiency deals with the extra space the algorithm requires.

As the number of inputs increases, the algorithm usually takes larger amount of time to execute.

Units for measuring running time:

We can count the number of times each of the algorithm's operation is executed or we can identify the most important operation of the algorithm called the basic operation (the operation that contributes the most to total running time) and computed the number of times the basic operation is executed.

Order of growth:

Difference in running times for small inputs does not distinguish efficient algorithms from inefficient on

Ex:

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10	33	10^2	10^3	10^3	3.6×10^6
10^2	6.6	10^2	660	10^4	10^6	1.3×10^{30}	9.3×10^{157}
10^3	10	10^3	10^4	10^6	10^9		
10^4	13	10^4	1.3×10^5	10^8	10^{12}		
10^5	17	10^5	1.7×10^6	10^{10}	10^{15}		

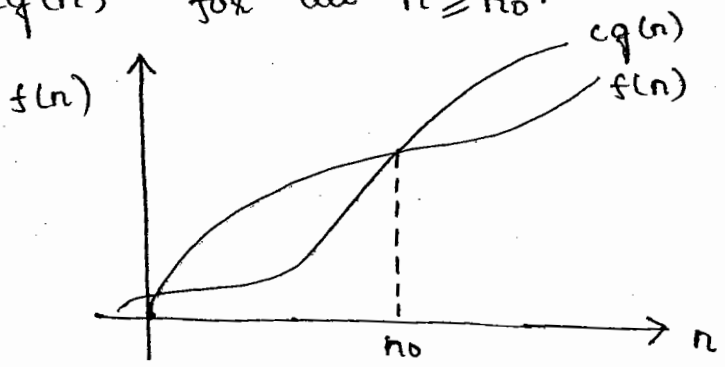
The function which grows the smallest among these is the logarithmic function. Hence it is desirable to design an algorithm which runs in $\log_2 n$ time.

Time efficiency of the algorithm depends on speed of the computer, choice of programming language, compiler used, choice of algorithm, number of inputs and outputs, size of input.

ASYMPTOTIC NOTATIONS :

(1) Big-oh notation :

Let $f(n)$ be the time efficiency of an algorithm. The function $f(n)$ is said to be big-oh of $g(n)$ denoted by $f(n) \in O(g(n))$ OR $f(n) = O(g(n))$ such that there exists a positive constant C and positive integer n_0 satisfying the constraint $f(n) \leq cg(n)$ for all $n \geq n_0$.



$f(n) = cg(n)$ only at n_0

1. Let $f(n) = 100n + 5$. Express $f(n)$ using O.

$cg(n) = f(n) = 100n + 5$

$f(n) \leq c\,g(n) \quad \forall \; n \geq n_0$

$100n + 5 \leq 101n \quad \forall \; n \geq 5$

Here $c = 101$, $n_0 = 5$, $g(n) = n$

$\therefore \; f(n) \in O(n)$

2. Express $f(n)$ using O if $f(n) = 10n^3 + 8$

The constraint to be satisfied is

$$f(n) \leq cg(n) \quad \forall \quad n \geq n_0$$

$$10n^3 + 8 \leq 11n^3 \quad \forall \quad n \geq 8$$

where $c = 11$, $g(n) = n^3$, $n_0 = 8$

By definition, $f(n) \in O(g(n))$ $\therefore f(n) \in O(n^3)$

3. Let $f(n) = 6 \times 2^n + n^2$. Express $f(n)$ using O.

$$f(n) \leq cg(n) \quad \forall \quad n \geq n_0$$

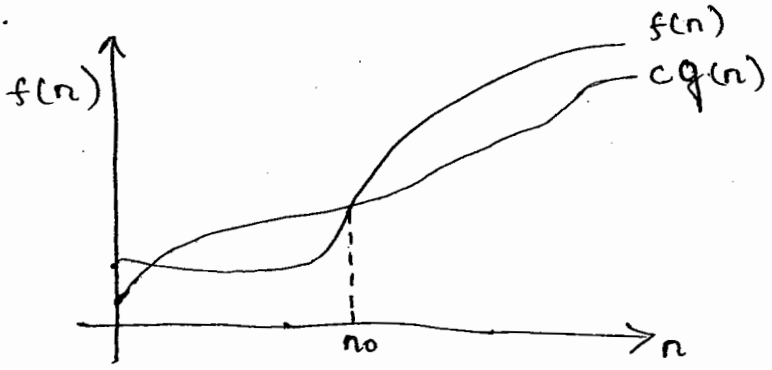
$$6 \times 2^n + n^2 \leq 7 \times 2^n \quad \forall n \geq 4$$

where $c = 7$, $g(n) = 2^n$, $n_0 = 4$

$\therefore f(n) \in O(2^n)$

(2) Big-omega notation:

Let $f(n)$ be the time efficiency of the algorithm. The function $f(n)$ is said to be big-omega of $g(n)$ denoted by $f(n) \in \Omega(g(n))$ or $f(n) = \Omega(g(n))$ such that there exists a positive constant c and positive integer n_0 such that $f(n) \geq cg(n)$ $\forall$ $n \geq n_0$.



1. Let $f(n) = 100n + 5$. Express $f(n)$ using Ω.

The constraint to be satisfied is $f(n) \geq cg(n)$ $\forall$ $n \geq n_0$

$$100n + 5 \geq 100n \quad \forall \quad n \geq 0$$

where $c = 100$, $g(n) = n$, $n_0 = 0$

By definition, $f(n) \in \Omega(g(n))$

(3) Big-theta notation :

Let $f(n)$ be the time complexity of an algorithm. The function $f(n)$ is said to be big-theta of $g(n)$ denoted by $f(n) \in \Theta(g(n))$ or $f(n) = \Theta(g(n))$ such that there exists positive constants c_1 and c_2 and positive integer n_0 satisfying the constraint $c_1 g(n) \leq f(n) \leq c_2 g(n) \; \forall \; n \geq n_0$

Let $f(n) = 6 \times 2^n + n^2$. Express $f(n)$ using Θ.

$c_1 g(n) \leq f(n) \leq c_2 g(n) \; \forall \; n \geq n_0$

$6 \times 2^n \leq 6 \times 2^n + n^2 \leq 7 \times 2^n \; \forall \; n \geq 4$

where $c_1 = 6$, $c_2 = 7$, $g(n) = 2^n$, $n_0 = 4$

By definition, $f(n) \in \Theta(g(n))$ or $f(n) \in \Theta(2^n)$

Mathematical analysis of non-recursive algorithms :

1. Based on the input size determine number of parameters to be considered.
2. Identify basic operation
3. Check whether basic operation depends on the size of input if so compute best and worst case.
4. Obtain total number of times the basic operation is executed.
5. Simplify using standard formulae and obtain time complexity.

Algorithm maximum (n, a)

// Purpose: To find maximum of 'n' elements

// Inputs: n → number of elements
a → list of elements

// Output: big → contains the maximum of 'n' elements

```
big ← a[0]
for i ← 1 to n-1
    if a[i] > big
        big ← a[i]
    end if
end for
return big
```

Time efficiency:

The parameter to be considered is 'n' which is size of input. The basic operation is " if a[i] > big".

$$f(n) = \sum_{i=1}^{n-1} 1$$

$$f(n) = n-1-1+1 = n-1$$

$$\therefore f(n) \in O(n)$$

Algorithm matrix_multiplication (n, a, b, c)

// Purpose: To multiply 2 matrices of size n x n

// Inputs: n → size of matrix
a & b → input matrices

// Output: c contains product of a & b

```
for i ← 0 to n-1
    for j ← 0 to n-1
        sum ← 0
```

```
        for K=0 to n-1
            sum = sum + (a[i,K] * b[K,j])
        end for
        c[i,j] ← sum
    end for
end for
```

Time complexity:

Basic operation is " sum = sum + (a[i,K] * b[K,j]) "

$$f(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{K=0}^{n-1} 1$$

$$= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} (n-1)-0+1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n$$

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

$$= n^2 \sum_{i=0}^{n-1} 1 = n^2 [(n-1)-0+1] = n^3$$

$$f(n) = O(n^3)$$

Algorithm unique_element (n, a)
// Purpose: To check whether the elements are unique or not.
// Inputs: n → number of elements in the list
a → list of elements
// Output: 0 → not unique & 1 → unique

```
for i ← 0 to n-2
    for j ← i+1 to n-1
        if a[i] = a[j]
            return 0
        end if
    end for
```

Time efficiency:

The basic operation is "if a[i]=a[j]"

Best case:

This case occurs if first two items are same and the number of comparisons required = 1

$$f(n) \in \Omega(1)$$

Worst case:

This case occurs if the basic operation is executed maximum number of times.

$$f(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

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

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

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

By neglecting all lower order terms and constants,

$$f(n) \in O(n^2)$$

Mathematical analysis of recursive algorithms:

1. Based on the input size, determine the number of paramet to be considered.
2. Identify the basic operation
3. Check whether the basic operation executed depends on the size of the input.
4. Obtain a recursive relation with an appropriate initial condition.
5. Solve the recurrence relation and obtain the order of growth and express the time efficiency using asymptotic notation.

Algorithm factorial(n)

// Purpose : To compute factorial of n using recursion

// Inputs : n → positive integer

// Output : Factorial of n

```
if n=0 return 1
return n * fact(n-1)
```

Time complexity :

$$f(n) = \begin{cases} 0 & \text{if } n=0 \\ 1+f(n-1) & \text{otherwise} \end{cases}$$

$f(n) = 1+f(n-1)$

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

$= 2+1+f(n-3) = 3+f(n-3)$

$\vdots$

$i = i+f(n-i) \quad (i=n)$

$= n+f(n-n)$

$= n+f(0)$

$= n$

$\therefore f(n) \in O(n)$

Algorithm tower_of_hanoi (n, s, t, d)

// Purpose : Transfer 'n' discs from source to destination

// Input : n → number of discs

// Output : All 'n' discs should be at the destination

```
if n=1
    move disc n from s to d
end if
tower (n-1, s, d, t)
move disc n from s to d
tower (n-1, t, s, d)
```

Time efficiency:

$$f(n) = \begin{cases} 1 & \text{if } n=1 \\ f(n-1)+1+f(n-1) & \text{otherwise} \end{cases}$$

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

$= 2[2f(n-2)+1] = 2^2 f(n-2) + 2+1$

$= 2^2[2f(n-3)+1] + 2+1 = 2^3 f(n-3) + 2^2+2+1$

$\vdots$

$= 2^i f(n-i) + 2^{i-1} + 2^{i-2} + \cdots\cdots 2^3 + 2^2 + 2 + 1$

Let $i = n-1$

$= 2^{n-1} f(1) + 2^{n-2} + 2^{n-3} + \cdots\cdots 2^3 + 2^2 + 2 + 1$

$= 2^{n-1} + 2^{n-2} + \cdots\cdots + 2^3 + 2^2 + 2 + 1$

$= \frac{1(2^n-1)}{2-1} = 2^n - 1$ $\quad \because \frac{a(r^n-1)}{r-1} \rightarrow$ geometric series

$\therefore f(n) \in O(2^n)$

Algorithm count(n)

// Purpose: To count the number of digits in binary number obtained from decimal number

// Input: n → the input number

// Output: The number of digits in binary number

```
if n ≤ 1 return 1
return 1 + f(n/2)
```

Time complexity:

$$f(n) = \begin{cases} 1+f(n/2) & \text{otherwise} \\ 0 & n \le 1 \end{cases}$$

$f(n) = 1 + f(n/2)$

$= 2 + f(n/2^2)$

$= i + f(n/2^i)$

Let $2^i = n$

$= i + f(1)$

$= i$

$2^i = n \Rightarrow i \log_2 2 = \log_2 n$

$\therefore i = \log_2 n$

$\therefore f(n) = O(\log_2 n)$ ——

Algorithm fibonacci(n)
// Purpose : To compute nth fibonacci number using recursion
// Input : n → a positive integer
// Output : nth fibonacci number

```
if (n=0) return 0
if (n=1) return 1
return fib(n-1) + fib(n-2)
```

Algorithm fibonacci(n) using iteration

```
a[0]=0
a[1]=1
for i ← 2 to n
        a[i] ← a[i-1] + a[i-2]
end for
return a[n-1]
```

Explicit formula to find nth fibonacci number:

By definition,

$$f(n) = \begin{cases} 0 & \text{if } n=0 \\ 1 & \text{if } n=1 \\ f(n-1)+f(n-2) & \text{otherwise} \end{cases}$$

From the above relation we have,

$$f(n) - f(n-1) - f(n-2) = 0 \quad \text{——(1)}$$

which is of the form $ax(n) + bx(n-1) + cx(n-2) = 0$ ——(2)

By comparing (1) and (2),

$a = 1, b = -1, c = -1$

(2) can be solved using the following relation,

$$x(n) = \alpha x_1^n + \beta x_2^n \quad \text{——(3)}$$

x_1 and x_2 values can be computed by solving the quadratic equation $ax^2 + bx + c = 0$

We know that $a = 1, b = -1, c = -1$

$\therefore x^2 - x - 1 = 0$

Solving the above equation,

$$x = \frac{1 \pm \sqrt{1 - 4(1)(-1)}}{2 \cdot 1} = \frac{1 \pm \sqrt{5}}{2}$$

$$x_1 = \frac{1+\sqrt{5}}{2} \quad \text{and} \quad x_2 = \frac{1-\sqrt{5}}{2}$$

Substituting in (3) we have,

$$x(n) = \alpha\left(\frac{1+\sqrt{5}}{2}\right)^n + \beta\left(\frac{1-\sqrt{5}}{2}\right)^n \quad \text{——(4)}$$

To solve (4) consider the following initializations

$x(0) = 0$ and $x(1) = 1$.

$\therefore 0 = \alpha + \beta$

$$\alpha = -\beta \quad \text{——(5)}$$

By substituting $x(1) = 1$,

$$1 = \alpha\left(\frac{1+\sqrt{5}}{2}\right)^1 + \beta\left(\frac{1-\sqrt{5}}{2}\right)^1$$

$$\alpha = -\beta \implies 1 = -\beta\left(\frac{1+\sqrt{5}}{2}\right) + \beta\left(\frac{1-\sqrt{5}}{2}\right)$$

$$1 = -\frac{\beta}{2} - \frac{\sqrt{5}\beta}{2} + \frac{\beta}{2} - \frac{\sqrt{5}\beta}{2}$$

$$1 = -\sqrt{5}\beta \quad \therefore \beta = -\frac{1}{\sqrt{5}} \quad \text{and} \quad \alpha = \frac{1}{\sqrt{5}}$$

Substituting these values in (4) we get,

$$x(n) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

Using the above relation we get the nth fibonacci number.

* Prove that every polynomial $p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n^1 + a_0$ with $a_k > 0$ belongs to $O(n^k)$

PROOF:

It is given that $f(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n^1 + a_0$ where a_k, a_{k-1} are the co-efficients of the polynomi

$$|f(n)| = |a_k n^k + a_{k-1} n^{k-1} + \cdots a_1 n^1 + a_0|$$
$$\leq |a_k| n^k + |a_{k-1}| n^{k-1} + \cdots + |a_1| n^1 + |a_0|$$
$$\leq n^k \left[|a_k| + \frac{|a_{k-1}|}{n} + \frac{|a_{k-2}|}{n^2} + \cdots \frac{|a_0|}{n^k} \right]$$

By neglecting lower order terms and constants,

$$|f(n)| \leq n^k |a_k|$$

By definition, if $f(n) \leq c\,g(n) \Rightarrow f(n) \in O(g(n))$ where $c = a_k$ and $g(n) = n^k$

$$\therefore f(n) \in O(n^k)$$

* If $f_1(n)$ belongs to $O(g_1(n))$ and $f_2(n)$ belongs to $O(g_2(n))$ then $f_1(n) + f_2(n) \in O(\max(g_1(n), g_2(n)))$

PROOF:

Note: Consider 4 arbitrary numbers a_1, b_1, a_2, b_2 if $a_1 \leq b_1$ and $a_2 \leq b_2$ then,

$$a_1 + a_2 \leq b_1 + b_2$$

i.e. $a_1 + a_2 \leq 2 \max(b_1, b_2)$

It is given that $f_1(n) \in O(g_1(n))$

By definition, $f_1(n) \leq c_1 g_1(n)$ ——(1)

It is also given that $f_2(n) \in O(g_2(n))$

By definition, $f_2(n) \leq c_2 g_2(n)$ ——(2)

By adding (1) and (2),

$$f_1(n) + f_2(n) \leq c_1 g_1(n) + c_2 g_2(n)$$

$$\leq c g_1(n) + c g_2(n)$$

where $c = 2\max(c_1, c_2)$

$$\leq c\,[g_1(n) + g_2(n)]$$

$$f_1(n) + f_2(n) \leq c \cdot 2 \max(g_1(n), g_2(n)) \quad \text{——(3)}$$

(3) is of the form $f(n) \leq c g(n)$ where $f(n) = f_1(n) + f_2(n)$

$c = 2c$, $g(n) = \max(g_1(n), g_2(n))$

By definition, $f(n) \in O(g(n))$ i.e

$\therefore f_1(n) + f_2(n) \in O(\max(g_1(n), g_2(n)))$

Using limits for comparing orders of growth:

$$\lim_{n \to \infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & t(n) \text{ has smaller order of growth than } g(n) \\ c>0 & t(n) \text{ has same order of growth as } g(n) \\ \infty & t(n) \text{ has larger order of growth than } g(n) \end{cases}$$

First two cases means $t(n) \in O(g(n))$.

Last two mean that $t(n) \in \Omega(g(n))$

Second case means that $t(n) \in \Theta(g(n))$

By L'Hospital's rule we know that,

$$\lim_{n \to \infty} \frac{t(n)}{g(n)} = \lim_{n \to \infty} \frac{t'(n)}{g'(n)}$$

Ex: 1. Compare the orders of growth of $\frac{1}{2}n(n-1)$ and n^2

$$\lim_{n\to\infty} \frac{\frac{1}{2}n(n-1)}{n^2} = \frac{1}{2} \lim_{n\to\infty} \frac{n^2-n}{n^2}$$

$$= \frac{1}{2} \lim_{n\to\infty} \left(1 - \frac{1}{n}\right) = \frac{1}{2}$$

Since the limit is equal to positive constant both the functions have same order of growth.

2. Compare the orders of growth of $\log_2 n$ and $\sqrt{n}$.

$$\lim_{n\to\infty} \frac{\log_2 n}{\sqrt{n}} = \lim_{n\to\infty} \frac{(\log_2 n)'}{(\sqrt{n})'}$$

$$= \lim_{n\to\infty} \frac{(\log_2 e)\frac{1}{n}}{\frac{1}{2\sqrt{n}}}$$

$$= 2\log_2 e \lim_{n\to\infty} \frac{\sqrt{n}}{n}$$

$$= 0$$

Since the limit is equal to 0, $\log_2 n$ has smaller order of growth than $\sqrt{n}$.

BRUTE FORCE

Selection sort:

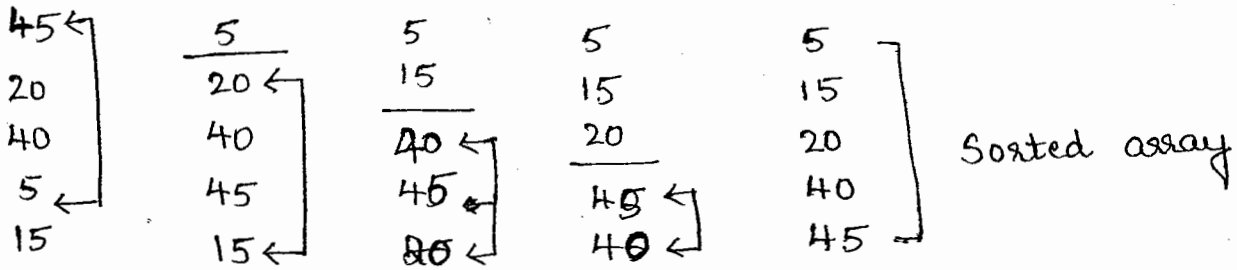
Here we scan the entire array to find its smallest element and exchange it with the first element and hence putting the smallest element in the first position of the list.

Brute force techniques use simple logic and are suitable for smaller values of 'n'.

```
Algorithm Selection Sort (a, n)
// Purpose : To sort the given array
// Input : An array 'a' to be sorted
// Output : Array 'a' sorted in ascending order

  for i = 0 to n-2  // Number of passes = n-1
        pos ← i
        small ← a[i]
        for j = i+1 to n-1
              if a[j] < small
                    small = a[j]
                    pos = j
              end if
        end for
        temp = a[i]
        a[i] = a[pos]
        a[pos] = temp
  end for
```

The basic operation here is "if a[j] < small"

Ex: Consider the array elements 45, 20, 40, 5, 15



Time complexity is given by,

$$O(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

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

$$= \sum_{i=0}^{n-2} n-i-1$$

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

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

$$= \frac{n^2}{2} - \frac{n}{2}$$

Neglecting all the lower order terms and constants

$$\therefore O(n) \approx n^2$$

Pattern matching: (String matching)

Consider a text string of length 'n' and pattern string of length 'm'.

Algorithm Pattern Matching (text[], pattern[])

// Purpose: To search for the pattern

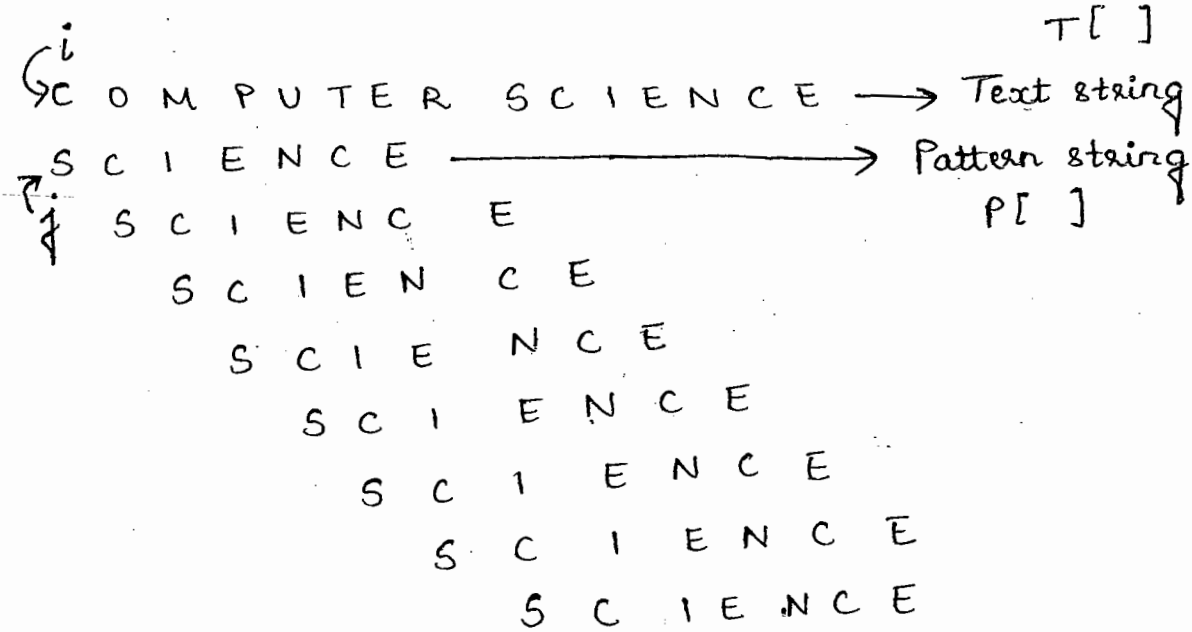
// Input: Text string text[] and the

pattern string pattern[ ]

// Output: The index of the first character in the text that matches the substring and -1 for unsuccessful search

```
for i=0 to n-m // Can also use while loop here
                                    i=0 , while i<=n-m
    j ← 0
    while ( j<m && text[i+j] = pattern[j])
        j++   or   j ← j+1
    end while
    if j=m return i
end for
return -1
```

Ex:



We keep comparing the character in the text string with the pattern string. If match is not found shift the pattern string towards right by one position.

The time complexity is given by,

Worst case:

$$O(n) = \sum_{i=0}^{n-m} \sum_{j=0}^{m-1} 1$$

$$= \sum_{i=0}^{n-m} (m-1) - 0 + 1$$

$$= \sum_{i=0}^{n-m} m$$

$$= m \sum_{i=0}^{n-m} 1$$

$$= m[(n-m) - 0 + 1]$$

$$= mn - m^2 + m$$

Neglecting the lower order terms and for sufficiently large values of 'n'

$$O(n) = mn \quad \therefore \underline{O(n,m) = mn}$$

Best case:

If the pattern string is present at the beginning of the text string the total number of comparisons needed are:

$$\Omega(n) = \sum_{j=0}^{m-1} 1$$

$$= (m-1) - 0 + 1$$

$$\Omega(n) = m$$

$$\therefore \underline{\Omega(n,m) = m}$$

Bubble sort:

Algorithm Bubble Sort (a, n)

// Purpose : To sort the given elements

// Inputs : n → positive integer

a → list of elements to be sorted

// Output : a → contains the sorted list

```
for j ← 0 to n-2   or   j ← 1 to n-1
    for i ← 0 to n-j-2   or   i ← 0 to n-j-1
        if a[i] > a[i+1]
            exchange a[i] and a[i+1]
        end if
    end for
end for
```

The time complexity is, Input size = n

The basic operation is if "a[i] > a[i+1]"

In the best case and the worst case the number of times the basic operation is executed remains the same. No of passes = n-1 and no of comparisons in each pass is n-j-1

$$C(n) = \sum_{j=1}^{n-1} \sum_{i=0}^{n-j-1} 1$$

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

$$= \sum_{j=1}^{n-1} n-j$$

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

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

By neglecting all the lower order terms and constants we get,

$$O(n) = n^2 \quad \text{or}$$

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

If the elements are arranged in ascending order the number of comparisons remain the same. But once we know that the elements are arranged in ascending order there is no need to go to the next pass. But in this algorithm even if elements are arranged in ascending order it goes to the next pass. This disadvantage can be overcome by using the following algorithm:

```
for j ← 1 to n-1
    flag = 0
    for i ← 0 to n-j-1
        if a[i] > a[i+1]
            exchange a[i] and a[i+1]
            flag = 1
        end if
    end for
    if flag = 0 break
end for
```

Time complexity is given by,

Best case: This case occurs when the elements are already sorted in ascending order.

```
j=1
- - - - -
- - - - -
for i ← 0 to n-j-1
  - - - -
  - - - -
```

$$\therefore f(n) = \sum_{i=0}^{n-2} 1$$

$$= n-2-0-1$$

$$= n-1$$

$$\therefore f(n) = \Omega(n)$$

Worst case:

This remains the same as previous algorithm

$$f(n) = \sum_{j=1}^{n-1} \sum_{i=0}^{n-j-1} 1$$

$$\therefore f(n) \in O(n^2)$$

Sequential search / Linear search:

```
Algorithm Linear search (item, n, a)
// Purpose: To search for the item in the given list
// Inputs: item → element to be search
           n → the number of elements
           a → the list of elements
// Output: position if item found
           -1 if item not found
for i ← 0 to n-1
    if item = a[i] return i
end for
return -1
```

Time efficiency:

The basic operation is " if item = a[i] "

Best case:

Successful search:

Occurs when the item is present in the first position so number of comparisons = 1

$\therefore f(n) = \Omega(1)$

Unsuccessful search:

If item is not present in the list, the minimum number of comparisons = n

$\therefore f(n) = \Omega(n)$

Worst case:

Successful search:

Item is present at the end of the list. Hence number of comparisons = n

$f(n) = O(n)$

Unsuccessful search:

Item is not present in the list. Hence total number of comparisons = n

$f(n) = O(n)$

Average case:

Successful search and unsuccessful search

$f(n) = \Theta(n)$

Tracing of Bubble Sort Algorithm:

Given Array:

89 45 68 90 29

Pass 1:

89 ↑	45	45	45	45
45 ↓	89 ↑	68	68	68
68	68 ↓	89 } no swap	89	89
90	90	90 } no swap	90 ↑	29
29	29	29	29 ↓	90

Pass 2:

45 } no swap	45	45	45	45
68 } no swap	68 } no swap	68	68	68
89	89 } no swap	89 ↑	29	29
29	29	29 ↓	89 } no swap	89
90	90	90	90 } no swap	90

Pass 3:

45 } no swap	45	45	45	45
68 } no swap	68 ↑	29	29	29
29	29 ↓	68 } no swap	68	68
89	89	89 } no swap	89 } no swap	89
90	90	90	90 } no swap	90

Pass 4 :

45 ↑	29	29	29	29
29 ↓	45 } no swap	45	45	45
68	68	68 } no swap	68	68
89	89	89	89 } no swap	89
90	90	90	90	90

∴ The sorted array is,

29 45 68 89 90

Number of passes required = n-1

Here value of 'n' i.e number of array elements = 5

∴ Number of passes = 5-1 = 4

$P \rightarrow S \rightarrow Q \rightarrow R \rightarrow P = 9+4+8+6 = 27$

$P \rightarrow S \rightarrow R \rightarrow Q \rightarrow P = 9+2+8+3 = 22$

The optimal solutions are,

$P \rightarrow Q \rightarrow S \rightarrow R \rightarrow P$ and

$P \rightarrow R \rightarrow S \rightarrow Q \rightarrow P$ (tours)

The optimal cost is 15 (length)

The time complexity is $(n-1)!$

$$f(n) = O[(n-1)!]$$

2. Knapsack problem:

We are given 'n' items of known weights $w_1, w_2 \cdots w_n$ and profits $v_1, v_2, \cdots v_n$ and a bag of capacity M. Here we generate all the subsets of the 'n' items and compute the total weight of each subset. We identify the feasible subsets and among them we find the optimal solution.

Ex: $M=40$

$n=3$

$w_1 = 20, w_2 = 25, w_3 = 10$

$v_1 = 30, v_2 = 40, v_3 = 35$

The various subsets for $n=3$ are,

$\{\}$

$\{1\}, \{2\}, \{3\}$

$\{1,2\}, \{1,3\}, \{2,3\}$

$\{1,2,3\}$

Exhaustive search:

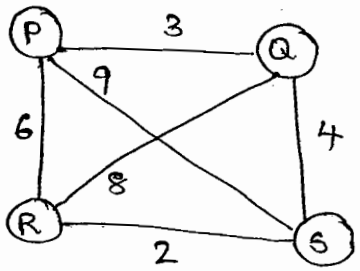
Is a brute force technique to solve combinatorial problems. It generates each and every element of the problem's domain, then select those which satisfy the constraints and then find an optimal solution.

1. Traveling Salesman problem:

To find the shortest tour through a given set of 'n' cities, visit each city exactly once before returning to the city where we started. Hence the problem is to find the shortest Hamiltonian circuit of the graph. We assume that all the circuits start and end at a particular vertex.

We can get the tour by generating the permutations of all (n-1) intermediate cities.

Ex:



We can consider P as the start vertex.

Hence the various combinations of other 3 vertices (Q,R,S) are QRS, QSR, RQS, RSQ, SQR, SRQ

$P \to Q \to R \to S \to P = 3+8+2+9 = 22$

$P \to Q \to S \to R \to P = 3+4+2+6 = 15$

$P \to R \to Q \to S \to P = 6+8+4+9 = 27$

$P \to R \to S \to Q \to P = 6+2+4+3 = 15$

Subset or Objects selected	Total weight	Total profit	Feasible or not
{ }	0	0	Feasible
{1}	20	30	Feasible
{2}	25	40	Feasible
{3}	10	35	Feasible
{1,2}	45	70	Not feasible
{1,3}	30	65	Feasible
{2,3}	35	75	Feasible
{1,2,3}	55	105	Not feasible

Feasible solution → Solution which satisfies the given constraint

Optimal solution → Solution which gives maximum profit

Time complexity : $O(2^n)$

Job Assignment problem :

There are 'n' people who are to execute 'n' jobs, one person for each job. The problem is to find the assignment with minimum cost.

(1) Consider 3 jobs and 3 person as follows:

	J_1	J_2	J_3
P_1	10	8	15
P_2	8	8	8
P_3	8	12	18

Position represents the person and the number represents the job number.

$\langle 3,1,2 \rangle = 15+8+12 = 35$

$\langle 1,3,2 \rangle = 10+8+12 = 30$

$\langle 1,2,3 \rangle = 10+8+18 = 36$

$\langle 2,1,3 \rangle = 8+8+18 = 34$

$\langle 2,3,1 \rangle = 8+8+8 = 24$

$\langle 3,2,1 \rangle = 15+8+8 = 31$

Optimal solution:

The minimum cost 24 occurs when person 1 does second job, person 2 does third job, person 3 does first job.

The time complexity is $O(n!)$ = number of feasible solutions

In exhaustive search, as the value of 'n' increases the number of possible solutions increases exponentially and hence difficult to perform for very large values of 'n'.

(2) Find the optimal solution for the given assignment problem:

	J_1	J_2	J_3	J_4
P_1	10	3	8	9
P_2	7	5	4	8
P_3	6	9	2	9
P_4	8	7	10	5

The various feasible solutions are:

$\langle 1,2,3,4 \rangle = 10+5+2+5 = 22$

$\langle 1,2,4,3 \rangle = 10+5+9+10 = 34$

$\langle 1,3,2,4 \rangle = 10+4+9+5 = 28$

$\langle 1,3,4,2 \rangle = 10+4+9+7 = 30$

$\langle 1,4,2,3 \rangle = 10+8+9+10 = 37$

$\langle 1,4,3,2 \rangle = 10+8+2+7 = 27$

$\langle 2,1,3,4 \rangle = 3+7+2+5 = \boxed{17}$

$\langle 2,1,4,3 \rangle = 3+7+9+10 = 29$

$\langle 2,3,1,4 \rangle = 3+4+6+5 = 18$

$\langle 2,3,4,1 \rangle = 3+4+9+8 = 24$

$\langle 2,4,1,3 \rangle = 3+8+6+10 = 27$

$\langle 2,4,3,1 \rangle = 3+8+2+8 = 21$

$\langle 4,3,1,2 \rangle = 9+4+6+7 = 26$

$\langle 4,3,2,1 \rangle = 9+4+9+8 = 30$

$\langle 4,1,2,3 \rangle = 9+7+9+10 = 35$

$\langle 4,1,3,2 \rangle = 9+7+2+7 = 25$

$\langle 4,2,1,3 \rangle = 9+5+6+10 = 30$

$\langle 4,2,3,1 \rangle = 9+5+2+8 = 24$

$\langle 3,1,2,4 \rangle = 8+7+9+5 = 29$

$\langle 3,1,4,2 \rangle = 8+7+9+7 = 31$

$\langle 3,2,1,4 \rangle = 8+5+6+5 = 24$

$\langle 3,2,4,1 \rangle = 8+5+9+8 = 30$

$\langle 3,4,1,2 \rangle = 8+8+6+7 = 29$

$\langle 3,4,2,1 \rangle = 8+8+9+8 = 33$

Number of feasible solutions = 24

The solution can be expressed as a n-tuple using a single dimensional array,

$$(J_1, J_2, J_3, \text{----} J_n)$$

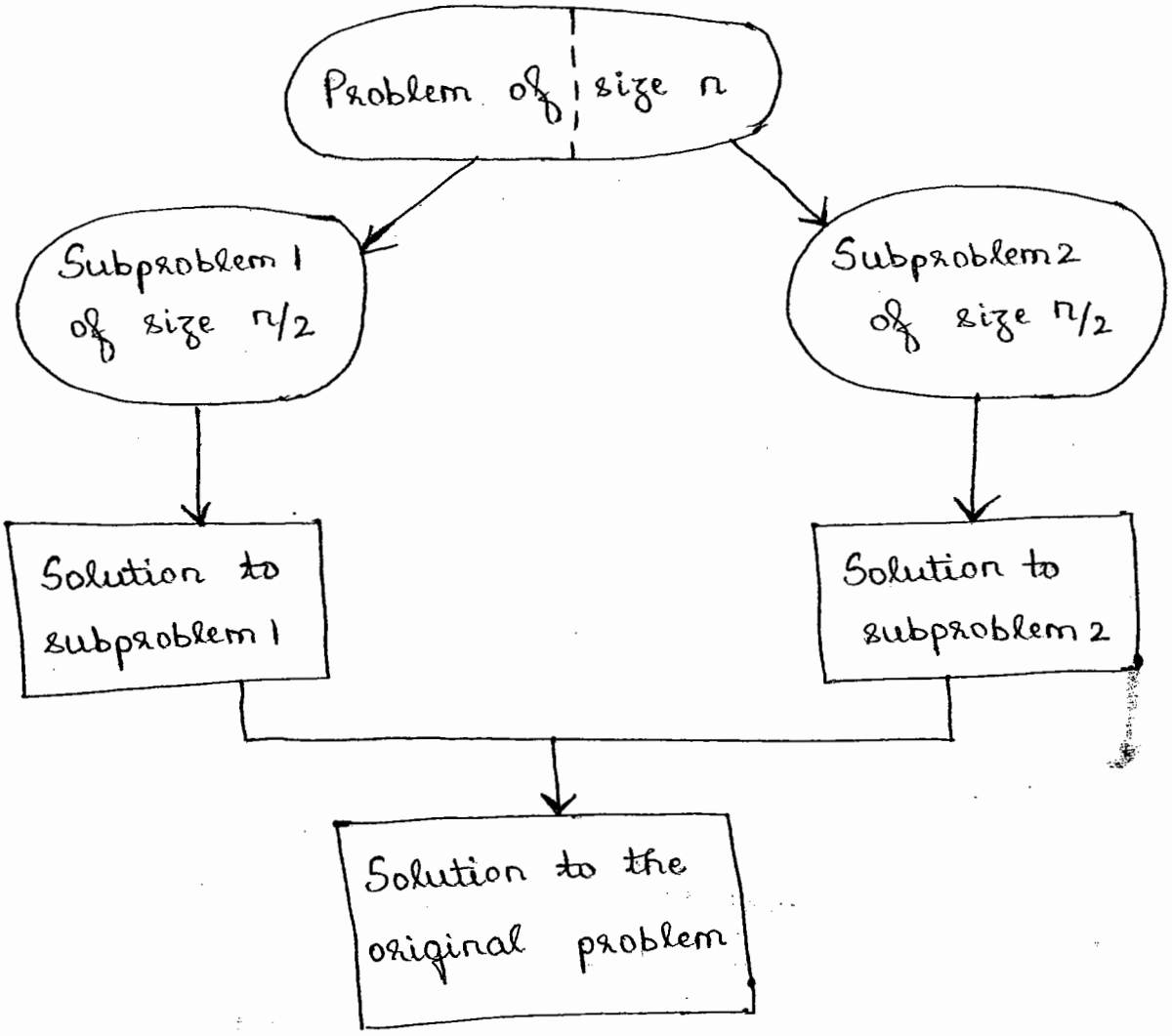
∴ The optimal solution is, $(2,1,3,4)$

i.e Minimum cost 17 occurs when person 1 does job 2, person 2 does job 1, person 3 does job 3 and person 4 does job 4.

Optimal cost = 17

DIVIDE AND CONQUER

In the divide and conquer approach, a given problem is divided into subproblems. If these subproblems are larger they are again divided into subproblems. If the subproblem is small, the solutions for the subproblems are obtained and they are combined to get the solution for the original problem. They are normally solved using recursion.



MERGE SORT :

A function merge is written to divide the array into 2 halves recursively until there is a single element. A function merge_sort is used to merge the sorted arrays. 2 pointers are made to point to the first elements of the arrays being merged. The elements pointed to are compared and the smaller one is added to a new array and index of smaller element is incremented to point to its successor in the original array. These operations are done repeatedly for all elements in the array.

```
Algorithm merge (low, mid, high, a)
i ← low
j ← mid+1
k ← low
while i ≤ mid and j ≤ high
      if a[i] < a[j]
          c[k] ← a[i]
          k ← k+1
          i ← i+1
      else
          c[k] ← a[j]
          k ← k+1
          j ← j+1
      end if
```

```
while i≤mid
    c[k] ← a[i]
    k ← k+1
    i ← i+1
end while
while j≤ high
    c[k] ← a[j]
    k ← k+1
    j ← j+1
end while
for i ← low to high
    a[i] ← c[i]
end for
```

```
Algorithm merge_sort (low, high, a)
// Purpose : To sort the elements of the array 'a'
// Inputs : a → list of elements to be sort
// Output : a → contains the sorted list
if low < high
    mid = (low + high)/2
    merge_sort (low, mid, a)
    merge_sort (mid+1, high, a)
    merge (low, mid, high, a)
end if
```

The time efficiency is,

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

$T(n/2)$ → left part of array; $T(n/2)$ → right part of array; n → input size

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

$$= 2\left[2T(n/4) + n/2\right] + n = 2^2T(n/2^2) + 2n$$

$$= 2^2\left[2T(n/8) + n/4\right] + 2n$$

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

$$\vdots$$

$$= 2^iT(n/2^i) + in$$

Let $2^i = n$ to get terminal condition $T(1) = 0$

$$T(n) = 2^iT(1) + in$$

$$= 0 + in$$

Now we know $n = 2^i$

Taking logarithm to base 2 on both sides

$$\log_2 n = \log_2 2^i$$

$$\log_2 n = i \log_2 2$$

$$\log_2 n = i \quad \text{since } \log_2 2 = 1$$

$\therefore T(n) = \log_2 n \times n$

$$T(n) = O(n \log_2 n)$$

MASTER'S THEOREM:

The general recurrence relation for divide and conquer method is given by,

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

The time efficiency can be obtained as follows:

$$T(n) = \begin{cases} \theta(n^d) & \text{if } a < b^d \\ \theta(n^d \log n) & \text{if } a = b^d \\ \theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Here 'd' is the power of 'n' in $f(n)$

The time complexity for merge sort is given by the following recurrence relation,

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

By comparing it with the equation $T(n) = aT(n/b) + f(n)$

$a=2,\ b=2,\ f(n) = n^1,\ d=1$

Here $a = b^d$ $(2 = 2^1)$

$\therefore\ T(n) = n^1 \log n$

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

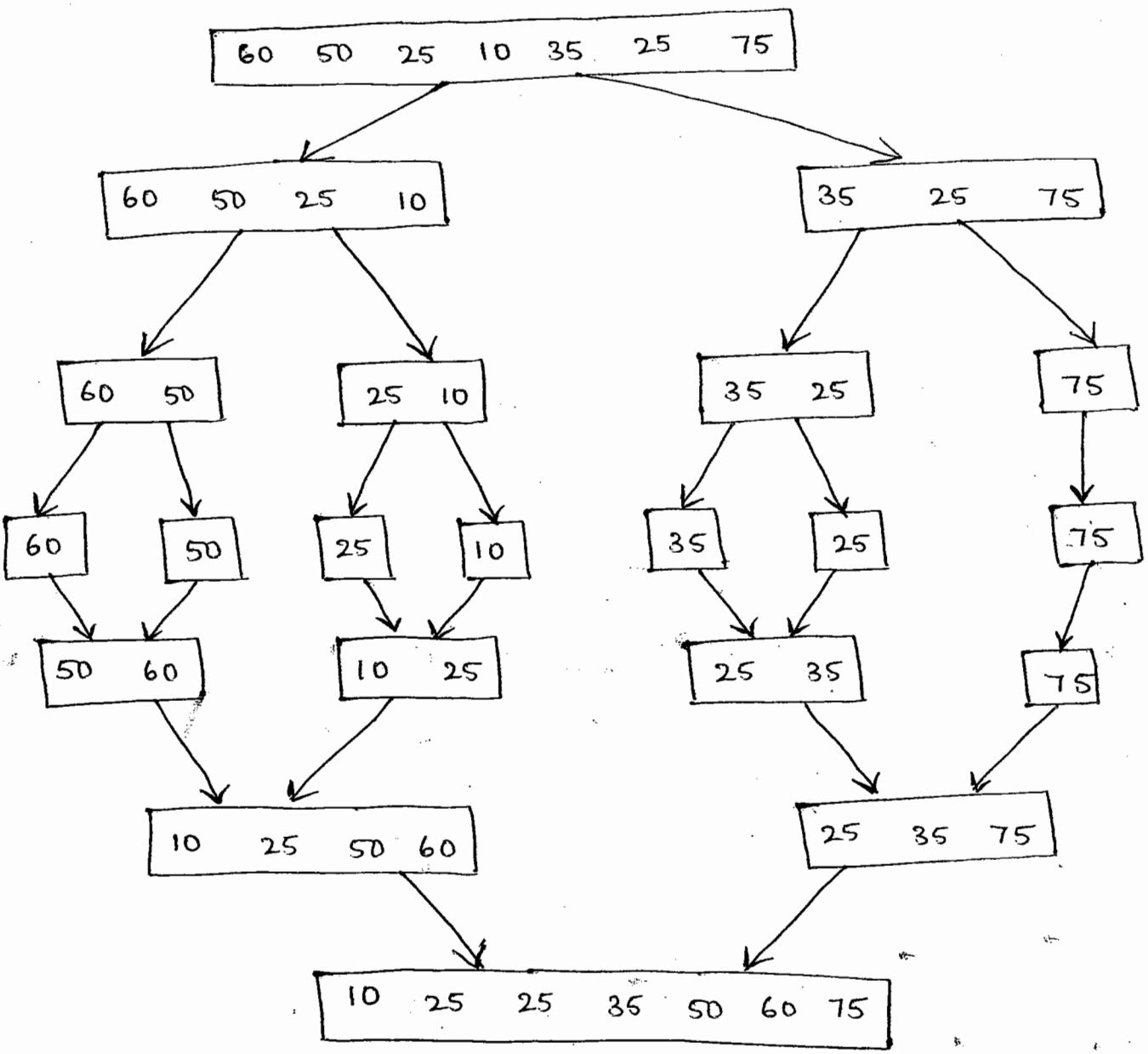
Ex:

Sort the following elements in ascending order using merge sort

60, 50, 25, 10, 35, 25, 75

low mid high

0	1	2	3	4	5	6
60	50	25	10	35	25	75



QUICK SORT :

Merge sorts divides the elements based on their position in the array. Quick sort divides them according to their value. It rearranges elements of a given array A to achieve its partition where all elements before the position j are smaller than or equal to A[j] and all elements after j are greater than or equal to A[j]. We select an element with respect to its value we divide the subarray, this element is called pivot.

```
Algorithm partition (low, high, a)
// Purpose : Partitions the subarray using the first
   element as a pivot
// Input : Input array 'a' of size 'n'
// Output : The position where the array is partitioned

 i ← low
 j ← high+1
 pivot ← a[low]
 while(i<j) do
 i ← i+1   while pivot >=a[i]
 do
    j ← j-1   while   pivot < a[j]
 if (i<j)
      exchange a[i] and a[j]  or swap(a[i], a[j])
 end while
 exchange a[low] and a[j]  // When i>j
 return j

Algorithm quick_sort (low, high, a)
// Purpose : To sort the given array using quick sort
// Input : The input array 'a' of size 'n'
// Output : The sorted array 'a'
    if (low < high)
    j ← partition (low, high, a)
     quick_sort (low, j-1, a)  // Sort the left part of
                                  the array recursively
     quick_sort (j+1, high, a) // Sort the right part of the
```

The time complexity is,

Best case:

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

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

$$= 2\left[2T(n/4) + n/2\right] + n$$

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

$$\vdots$$

$$= 2^i T(n/2^i) + in$$

Let $2^i = n$ to get terminal condition $T(1) = 0$

$$= 2^i \underbrace{T(1)}_{0} + in = 0 + in = in$$

We know that $2^i = n$

$$\log_2 n = \log_2 2^i = i \underbrace{\log_2 2}_{1}$$

$$\therefore i = \log_2 n$$

$$\therefore T(n) = \log_2 n \times n = n \log_2 n$$

$$\therefore T(n) = \Omega(n \log_2 n)$$

Average case: $T(n) = \theta(n \log_2 n)$

Worst case:

$$T(n) = \begin{cases} 0 & \text{if } n=1 \\ T(n-1) + n & \text{otherwise} \end{cases}$$

↓

$n-1$ elements on left side or right side and no elements on the other side

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

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

$= T(n-i) + n + (n-1) + (n-2) + \cdots (n-i)$

Put $i = n-1$ to get terminal condition $T(1) = 0$

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

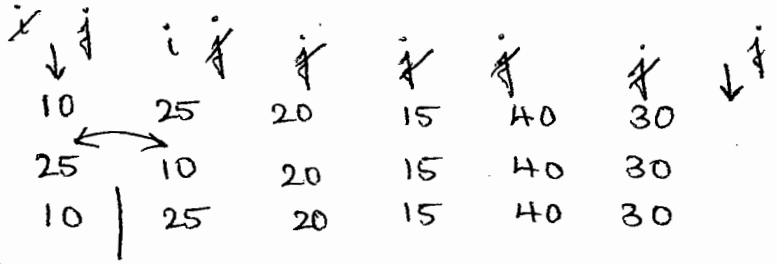
$= n + (n-1) + (n-2) + \cdots 1$

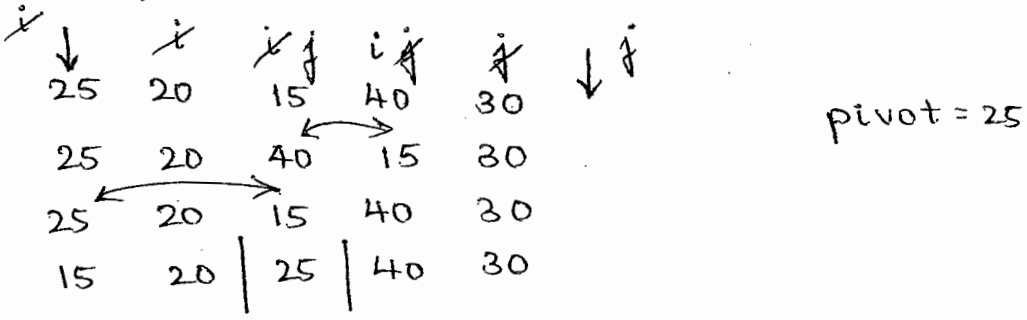
$= \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$

By neglecting all lower order terms and constants,

$\therefore T(n) = O(n^2)$

Ex:





and

Hence the sorted array is,

10 15 20 25 30 40

When middle element of the array is chosen as pivot - the efficiency is better. (than first element)

BINARY SEARCH:

Binary search is an efficient algorithm to search in a sorted array. Here we compare the key element K with the middle element. If a match is found then the search is stopped else repeated recursively for the left part of the array if K< middle element else for the right part.

```
Algorithm binary-search (item, n, a)
// Purpose: To search for the item in the given list
// Inputs: item → element to be searched
           n → number of elements
           a → list of elements
// Output: Successful search if item found
           Unsuccessful search if item not found

low ← 0
high ← n-1
while low <= high
      mid = (low + high) / 2
      if item = a[mid]
          print Successful search
          return
      end if
      if item < a[mid]
          high = mid - 1
      else
          low = mid + 1
      end if
end while
```

The time complexity is,

BEST CASE : This case occurs if the item to be searched is present in the middle of the array. So the number of comparisons required = 1.

$$f(n) = \Omega(1)$$

WORST CASE :

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n/2) + 1 & \text{otherwise} \end{cases}$$

$$T(n) = T(n/2) + 1$$
$$= T(n/4) + 1 + 1$$
$$= T(n/8) + 1 + 1 + 1$$
$$= T(n/2^3) + 3$$
$$\vdots$$
$$= T(n/2^i) + i$$

Let $2^i = n$ to get terminal condition $T(1) = 1$

$$= T(1) + i$$
$$= 1 + i$$

We know that $2^i = n$

$$i \log_2 2 = \log_2 n$$
$$i = \log_2 n$$

$$\therefore T(n) = 1 + \log_2 n$$
$$\approx \log_2 n$$

$$\therefore T(n) = O(\log_2 n)$$

The divide and conquer technique divides a problem into several subproblems each of which needs to be solved. In binary search only one of the two subproblems (left part / right part of the array) need to be solved hence it is better considered as a decrease by half algorithm.

MULTIPLICATION OF LARGE INTEGERS:

In few modern applications such as cryptography require manipulation of long integers. If we use the normal method to multiply two integers ('n' digits) each of the 'n' digits of the first number must be multiplied with each of the 'n' digits of second number.

Ex: Consider $a = 1234$, $b = 5678$

$a_1 = 12 \quad a_0 = 34 \quad b_1 = 56 \quad b_0 = 78$

$a = a_1 * 10^2 + a_0 \qquad b = b_1 * 10^2 + b_0$

$a * b = (a_1 * 10^2 + a_0) * (b_1 * 10^2 + b_0)$

$a * b = (a_1 b_1 * 10^4 + a_1 b_0 * 10^2 + a_0 b_1 * 10^2 + a_0 b_0)$

$a * b = a_1 b_1 10^4 + 10^2 (a_1 b_0 + a_0 b_1) + a_0 b_0 \quad \text{——(1)}$

By assuming 10^4 and 10^2 can be obtained by shifting the decimal digits towards the left, the time to compute 10^4 and 10^2 is negligible. Hence in (1) the number of multiplications = 4.

To reduce the number of multiplications, consider

$(a_1+a_0)(b_1+b_0) = a_1b_1 + a_0b_1 + a_1b_0 + a_0b_0$

$a_0b_1 + a_1b_0 = (a_1+a_0)(b_1+b_0) - a_0b_0 - a_1b_1$

Substituting this value in (1) we get,

$a*b = a_1b_1 10^4 + [(a_1+a_0)(b_1+b_0) - a_0b_0 - a_1b_1] 10^2 + a_0b_0$

In the above equation, a_1b_1 is computed once and the value is reused when we get a_1b_1 again. Similarly a_0b_0 is computed only once. So total number of multiplications = 3

The time efficiency is,

$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ 3*T(n/2) & \text{otherwise} \end{cases}$$

$T(n) = 3*T(n/2)$

$= 3\left[3*T\left(n/2^2\right)\right]$

$= 3^2*T\left(n/2^2\right)$

$= 3^2\left[3*T\left(n/2^3\right)\right]$

$= 3^3*T\left(n/2^3\right)$

$\vdots$

$= 3^i*T\left(n/2^i\right)$

Let $2^i = n$ to get terminal condition $T(1)=1$

$= 3^i*T(1) = 3^i*1 = 3^i$

We know that $2^i = n$

$i = \log_2 n$

$T(n) = 3^{\log_2 n} = n^{\log_2 3}$ $\quad \because a^{\log b} = b^{\log a}$ if $n=1$

$\therefore T(n) = n^{1.585}$

STRASSEN'S MATRIX MULTIPLICATION :

Two matrices can be multiplied using the formulae:

$$\begin{bmatrix} c_1 & c_2 \\ c_3 & c_4 \end{bmatrix} = \begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \end{bmatrix} \begin{bmatrix} b_1 & b_2 \\ b_3 & b_4 \end{bmatrix}$$

$$= \begin{bmatrix} m_1+m_4-m_5+m_7 & m_3+m_5 \\ m_2+m_4 & m_1+m_3-m_2+m_6 \end{bmatrix}$$

where $m_1 = (a_1+a_4) * (b_1+b_4)$

$m_2 = (a_3+a_4) * b_1$

$m_3 = a_1 * (b_2-b_4)$

$m_4 = a_4 * (b_3-b_1)$

$m_5 = (a_1+a_2) * b_4$

$m_6 = (a_3-a_1) * (b_1+b_2)$

$m_7 = (a_2-a_4) * (b_3+b_4)$

To multiply two 2X2 matrices, this algorithm requires 7 multiplications and 18 additions / subtractions. The brute force algorithm requires 8 multiplications and 4 additions. But the algorithm gains importance for very large values of 'n'.

The time complexity is,

$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ 7*T(n/2) & \text{otherwise} \end{cases}$$

$$T(n) = 7*T(n/2)$$

$$= 7\left[7*T(n/4)\right] = 7^2*T(n/2^2)$$

$$= 7^2 * \left[7*T(n/2^3)\right]$$

$$= 7^3*T(n/2^3)$$

⋮

$$= i \quad \ldots$$

Let $n=2^i$ to get terminal condition $T(1)=1$

$$= 7^i * T(1) = 7^i$$

We know that $2^i = n$

$$i = \log_2 n$$

$$\therefore T(n) = 7^{\log_2 n} = n^{\log_2 7}$$

$$\therefore T(n) = n^{2.806}$$

$$T(n) = O(n^{2.806})$$

BACKTRACKING

Construct solutions for each component at a time and evaluate partially constructed solutions. If partially constructed solution can be developed further without violating the problem's constraints – take the legitimate option for the next component. If there is no legitimate option for next component, consider the remaining components. No alternatives for remaining components – backtrack to replace last component of partially constructed solution with it's next option.

State-space tree: For the above process we construct a tree for the choices made.

Root node – initial state before the search for a solution begins.

Nodes in the first level of the tree – choices made for the first component of a solution.

Promising node – partially constructed solution that may lead to a complete solution.

Nonpromising node – partially constructed solution that may not lead to a complete solution.

Leaf nodes – nonpromising dead ends or complete solutions found.

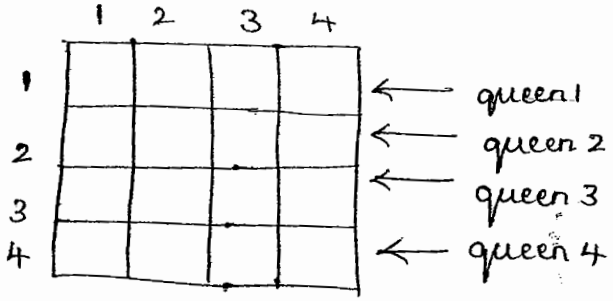
n-Queens Problem:

To place 'n' queens on n-by-n chessboard so that no two queens are placed in the same row or in the same column or on the same diagonal.

$n=1 \rightarrow$ no discrepancy - trivial solution

$n=2$, $n=3$ } No solution.

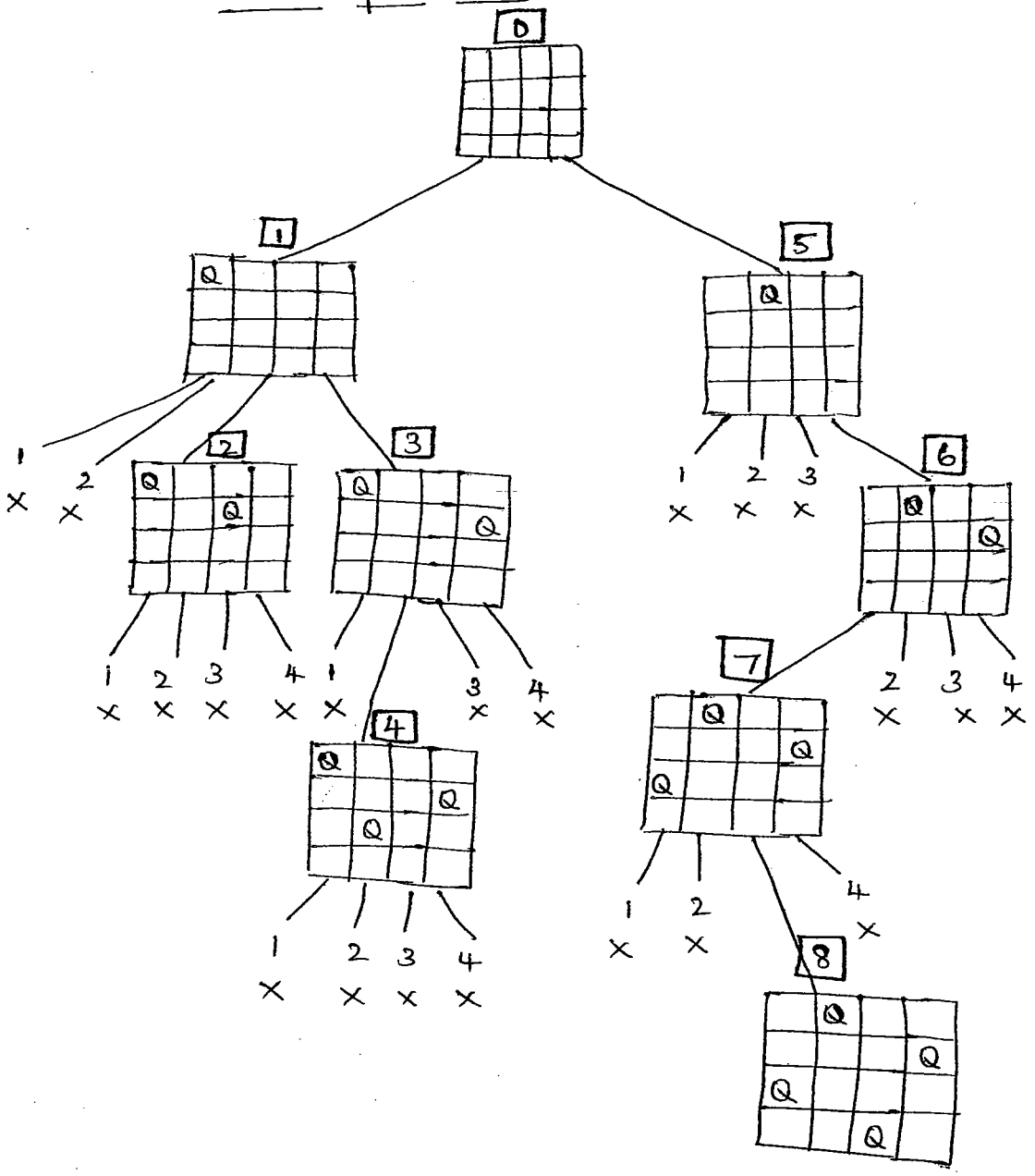
Four queens problem:



Start with empty board. Queen 1 is placed in first row of column 1. Then we take queen 2 - it cannot be placed in first row since queen 1 is already placed. Hence, we come to second row. We cannot place in column 1 since queen 1 is present in the same column. We cannot place in column 2 since it is on the same diagonal. We can place it in column 3 or 4 and so on.

When queen 2 is placed in row 2 column 3 — we cannot place queen 3 ∴ It becomes dead end and we backtrack to queen 2 and place it in row 2 and column 4.

State - space tree.



× → unsuccessful attempt to place a queen

[1-8] → order in which nodes are generated

Subset - Sum Problem:

Find a subset of a given set $S = \{s_1, s_2, \ldots s_n\}$ of 'n' positive integers whose sum is equal to a given positive integer 'd'.

Ex: (1) $S = \{1, 2, 5, 6, 8\}$

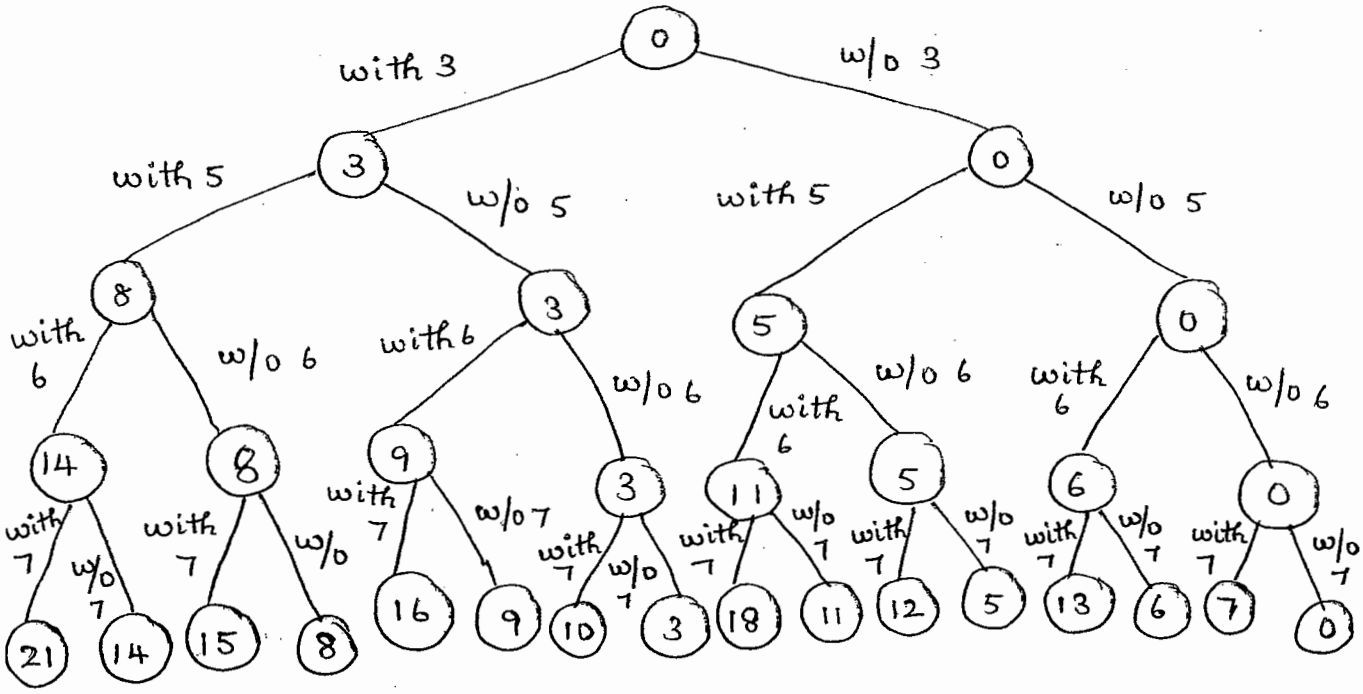
$d = 9$

The solutions are $\{1, 2, 6\}$ and $\{1, 8\}$

State-space tree can be used to find all possible solutions using backtracking

(2) $S = \{3, 5, 6, 7\}$

$d = 15$



The solution is $\{3, 5, 7\}$

Left subtree at each point - inclusion

Right subtree at each point - exclusion

Non-promising node : If the sum > d or sum < d

```
Algorithm n-queen ( int x[ ], K)
    K ← 1
    x[K] ← 0
    while K ≠ 0
        x[K] ← x[K]+1
        while x[K] ≤ n and ! place (x, K)
            x[K] ← x[K]+1
        end while
        if x[K] ≤ n
            if K = n
                print solution x
            else
                K ← K+1
                x[K] ← 0
            end if
        else
            K ← K-1
        end if
    end while

Algorithm place (x[ ], K)
    for i ← 1 to K-1
        if x[i] = x[K] or abs (i-K) ←
                abs (x[i] - x[K])
            return 0
        end if
    end for
    return 1
```

```
Algorithm subset ( weight_so_far , k,  total_remaining_
                                       weight )

x[k] ← 1
if (weight_so_far + w[k] = m )
        print solution found
        for i ← 0 to k
            if x[i] = 1
                print w[i]
            end if
        end for
end if
if ( weight_so_far + w[k] + w[k+1] <= m)
        subset ( weight_so_far + w[k], k+1,
            total_remaining_weight - w[k])
end if
if ( weight_so_far + total_remaining_weight
            - w[k] ≥ m  and
    weight_so_far + w[k+1] ≤ m)
            x[k] ← 0
        subset ( weight_so_far , k+1,
        total_remaining_weight - w[k])
end if
```

LIMITATIONS OF ALGORITHM POWER

Lower Bound Arguments :

Worst case time complexity is usually used to measure the efficiency of an algorithm.

Selection sort $\longrightarrow O(n^2)$

Tower of Hanoi $\longrightarrow O(2^n)$

∴ Selection sort is better than Tower of Hanoi as seen from their time complexity. But not true, since both are used to solve different kinds of problems.

In general, time complexity is used to measure efficiency of algorithms that solve the same problem.

Lower bound - how much improvement is required to obtain a better algorithm for a given problem. If we obtain a tight lower bound, the existing algorithm is in the same efficiency class as lower bound ∴ Maximum of only a constant-factor improvement.

Trivial Lower Bounds :

Simplest method - count no of items in the problem's input to be processed and also count no of output items to be produced.

Ex: 1. Generation of all permutations of 'n' distinct items

$= \Omega(n!)$

Since size of output = $n!$

2. Evaluation of polynomial of degree 'n'

$p(x) = a_n x^n + a_{n-1} x^{n-1} + \text{----} + a_0$

All co-efficients must be processed = 'n'

$\therefore$ Time complexity = $\Omega(n)$

3. Product of two n-by-n matrices

Produces n^2 elements of the product

$\therefore$ Time complexity = $\Omega(n^2)$

Trivial lower bounds are not very useful - we need to determine which parts of inputs must be processed

Searching algorithm - does not require processing of all inputs.

Information - Theoretic Arguments:

Generates lower bound based on the amount of information to be produced.

Use for comparison problems such as sorting and searching.

Ex: "Deduce a positive integer between 1 and n selected by a person by asking yes or no questions"

Time complexity = $\lceil \log_2 n \rceil$

Each answer yields at most one bit of information about the algorithm's output.

Adversary Arguments:

Generates lower bound by measuring the amount of work needed to shrink a set of potential inputs to a single input along the most time-consuming path.

Ex: Merging two sorted lists of size 'n'

$a_1 < a_2 < \cdots a_n$ and

$b_1 < b_2 < \cdots b_n$

All a's and b's are distinct.

Merge sort - no of comparisons $= 2n-1$

Using adversary method we prove that $2n-1$ is a lower bound.

True if $a_i < b_j$ if and only if $i < j$.

∴ Producing the following list,

$b_1 < a_1 < b_2 < a_2 \_\_\_\_ < b_n < a_n$

To obtain this correct list - compare $2n-1$ adjacent pair of elements.

Suppose a_1 is not compared to b_2, then the list is

$b_1 < b_2 < a_1 < a_2 \cdots < b_n < a_n$

Problem Reduction:

Obtain an algorithm for problem P by reducing it to another problem Q solvable with a known algorithm.

Ex: Euclidean minimum spanning tree problem

Given 'n' points in the Cartesian plane, construct a tree of minimum total length whose vertices are the given points.

We reduce the given problem to element uniqueness problem.

$(x_1, 0), (x_2, 0), \ldots (x_n, 0)$ are the 'n' points in the Cartesian plane - 'y' co-ordinates are 0.

Let T be the minimum spanning tree for the above set of points. T contains a shortest edge - checks whether T contains a zero-length edge. This is similar to checking uniqueness of numbers.

$\therefore$ Lower bound = $n \log n$

(1) Consider,

$$x \cdot y = \frac{(x+y)^2 - (x-y)^2}{4}$$

and $x^2 = x \cdot x$

Both belong to the same complexity class even though the second one seems simpler.

(2) Multiplying two symmetric matrices and multiplying two arbitrary square matrices [illegible] same complexity class.

Multiplying two symmetric matrices is a special case of multiplying two arbitrary square matrices.

Multiplying two arbitrary square matrices can be reduced to the problem of multiplying two symmetric matrices as shown below,

$$X = \begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix} \quad \text{and} \quad Y = \begin{bmatrix} 0 & B^T \\ B & 0 \end{bmatrix}$$

A^T and B^T → transposes of A and B

$$A^T[i,j] = A[j,i]$$
$$B^T[i,j] = B[j,i]$$

$$XY = \begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix} \begin{bmatrix} 0 & B^T \\ B & 0 \end{bmatrix} = \begin{bmatrix} \boxed{AB} & 0 \\ 0 & A^TB^T \end{bmatrix}$$

needed product

DECISION TREES :

Used to study performance of algorithms which involve comparison such as sorting and searching.

Decision tree to find minimum of 3 numbers :

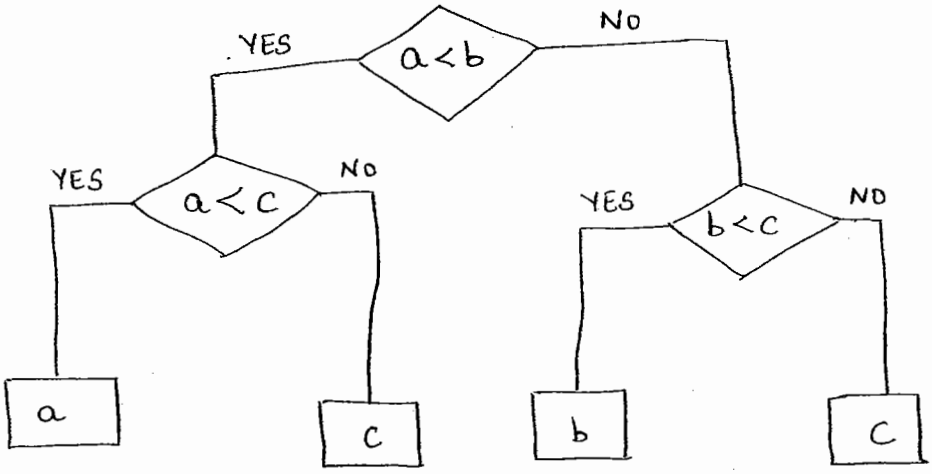
Internal node - key comparison $K < K'$

Node's left subtree - subsequent comparisons made if $K < K'$

Node's right subtree - subsequent comparisons made if $K > K'$

Leaf nodes - possible outcome of the algorithm's run on some input of size 'n'.

No of leaf nodes >= No of possible outcomes



Number of comparisons made can be traced by a path from root to leaf in a decision tree = number of edges in the path.

∴ Number of comparisons in worst case = height of algorithm's decision tree.

$$2^h \geq l$$

$h \longrightarrow$ height of the tree

$l \longrightarrow$ no of leaves

$$\therefore h \geq \lceil \log_2 l \rceil$$

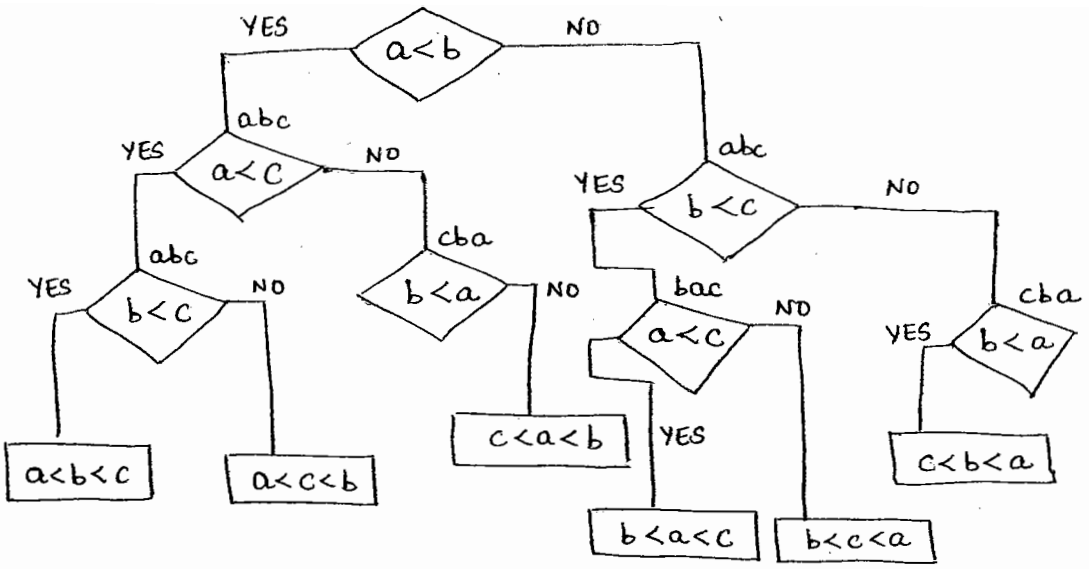
↓

lower bound → information-theoretic

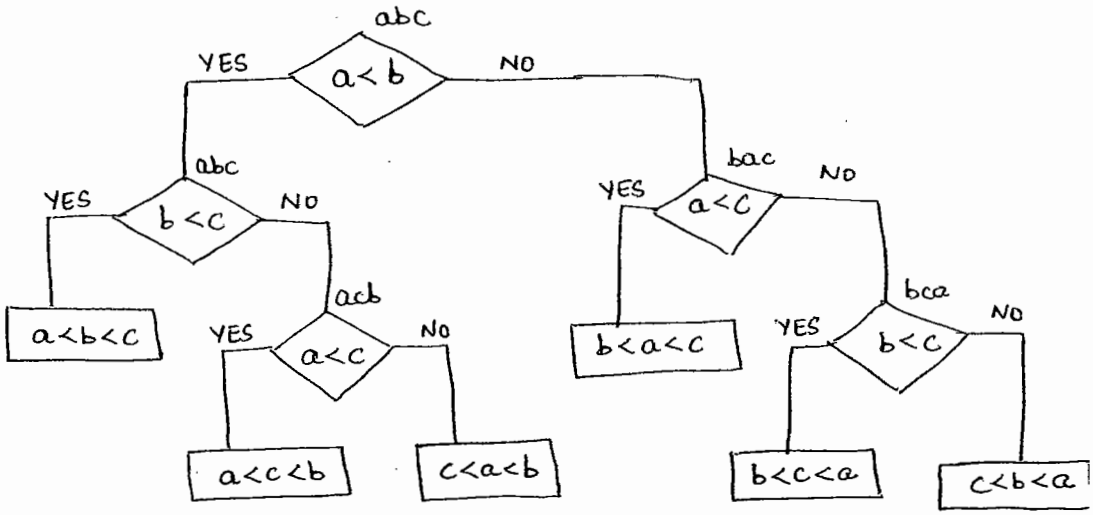
Decision Trees for Sorting Algorithms:

(1) Three-element selection sort:

Leaf nodes may have same possible outcome in more than one place based on the comparisons performed.



(2) Three-element insertion sort:



Number of possible outcomes = $n!$

Since $a<c<b$ can be considered as 1, 3, 2 permutation.

$$C_{worst}(n) \geq \lceil \log_2 n! \rceil$$

Using Stirling's formula for $n!$,

$$\lceil \log_2 n! \rceil \simeq \log \sqrt{2\pi n}(n/e)^n = n\log_2 n - n\log_2 e + \frac{\log_2 n}{2} + \frac{\log_2 2\pi}{2}$$

$$\simeq n\log_2 n$$

$\downarrow$

tight asymptotic lower bound

(merge sort)

Average-case behaviour → average path length from root to the leaves.

For three-element insertion sort,

$$(2+3+3+2+3+3)/6 = 2^2/3$$

$\therefore \quad C_{avg}(n) \geq \log_2 n!$

Decision trees for searching a sorted array:

Consider the sorted array,

$A[0] < A[1] < \cdots A[n-1]$

The algorithm usually used for the above problem is binary search.

$$C^{bs}_{worst}(n) = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2 (n+1) \rceil$$

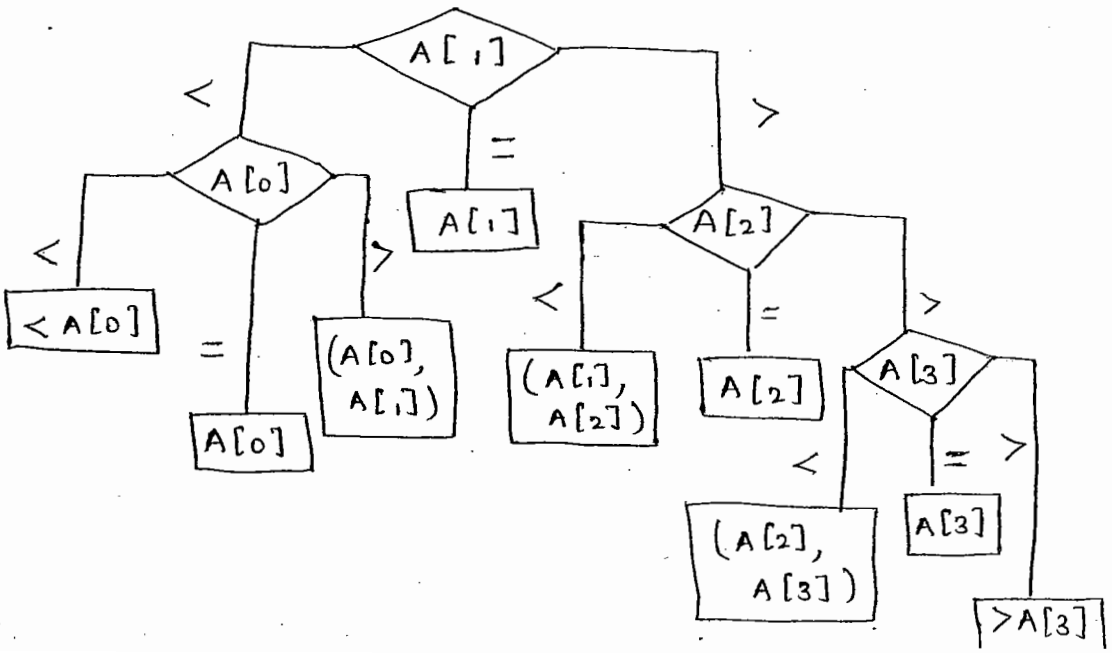
Binary search uses three-way comparison,

Key element $K < A[i]$

$K = A[i]$

$K > A[i]$

$\therefore$ We use ternary decision trees.



No of leaf nodes = $2n+1$ $(n+n+1)$

where $n \rightarrow$ successful searches

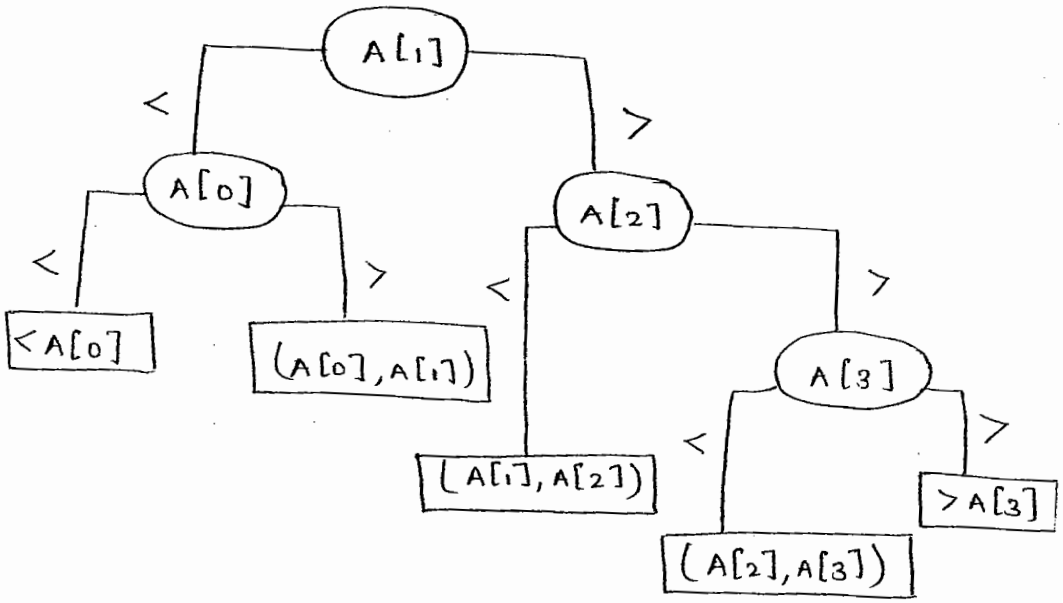
$n+1 \rightarrow$ unsuccessful searches

Minimum height of a tree $h = \lceil \log_3 l \rceil$

$\therefore C_{worst}(n) \geq \lceil \log_3 (2n+1) \rceil$

This value is smaller than $\lceil \log_2 (n+1) \rceil$ which is the worst case time complexity of binary search

By using binary decision tree,



Now $C_{worst}(n) \geq \lceil \log_2 (n+1) \rceil$

P, NP and NP – complete problems :

An algorithm can solve a problem in polynomial time if it's worst-case time efficiency $\in O(p(n))$ where $p(n) \rightarrow$ polynomial of the problem's input size n.

Such problems are called tractable.

Problems which cannot be solved in polynomial time are called intractable.

Intractable problems cannot be solved in a reasonable amount of time.

Polynomial time algorithms with degree of polynomial greater than 3 take large amount of time for execution but are not usually used.

Polynomials that bound running time of algorithms usually don't have extremely large coefficients.

Sum and composition of polynomials are always polynomials only.

All these properties led to development of computational complexity - classifies problems based on their inherent difficulty.

Class P is a class of decision problems that can be solved in polynomial time by deterministic algorithms - polynomial.

Every decision problem cannot be solved in polynomial time. Such decision problems cannot be solved by any algorithm - undecidable.

Ex: Halting problem

Given a computer program and an input to it, determine whether the program will halt on that input or continue working indefinitely on it.

Proof by contradiction:

$$A(P, I) = \begin{cases} 1 & \text{if program } P \text{ halts on input } I \\ 0 & \text{if program } P \text{ does not halt on input } I \end{cases}$$

A: any algorithm; P: program; I: input

Consider program P as an input to itself and use the output of algorithm A for pair (P, P) to construct a program Q,

$$Q(P) = \begin{cases} \text{halts if } A(P,P) = 0 \text{ i.e if program P does not halt on input P} \\ \text{does not halt if } A(P,P) = 1 \text{ i.e if program P halts on input P.} \end{cases}$$

By substituting Q for P,

$$Q(Q) = \begin{cases} \text{halts if } A(Q,Q) = 0 \text{ i.e if program Q does not halt on input Q} \\ \text{does not halt if } A(Q,Q) = 1 \text{ i.e if program Q halts on input Q.} \end{cases}$$

This is a contradiction since 2 outcomes for Q is possible.

∴ Proved.

No polynomial - time algorithms exist for the following : (decision problems - exponential growth)

(1) Hamiltonian circuit - a path that starts and ends at the same vertex and passes through all other vertices only once.

(2) Traveling salesman - find shortest tour through 'n' cities with known distances between them.
(finding shortest Hamiltonian circuit in a complete graph)

(3) Knapsack problem - find most valuable subset of 'n' items of given positive integer weights and values that fit into a knapsack of a given positive integer capacity.

(4) Partition problem – Given n positive integers, determine whether it is possible to partition them into 2 disjoint subsets with same sum.

(5) Bin packing – Given 'n' items whose sizes are positive rational numbers not larger than 1, put them into the smallest number of bins of size 1.

(6) Graph coloring – For a given graph, find it's chromatic number – smallest no of colors that need to be assigned to the graph's vertices so that no adjacent vertices are assigned to the same color.

(7) Integer linear programming – Find the maximum or minimum value of a linear function of several integer-valued variables subject to a finite set of constraints in the form of linear equalities or inequalities.

Non-deterministic algorithm:

Guessing stage and Verification stage

Guess a solution atleast once.

Decision problems such that for every yes instance of the problem it returns yes on some execution.

Non-deterministic polynomial algorithm:

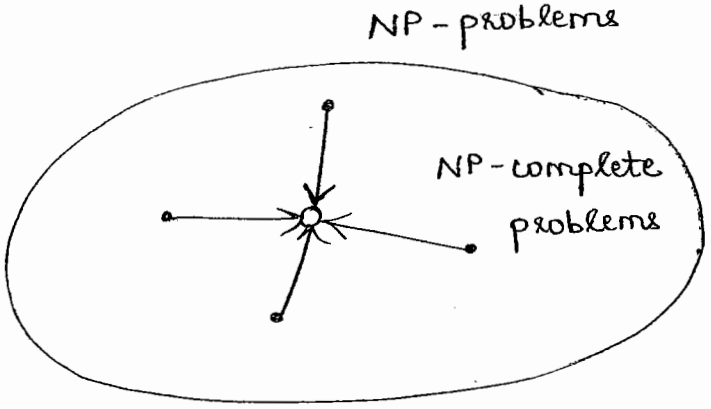
Time efficiency for the verification stage is polynomial.

Class NP is the class of decision problems that can be solved by non-deterministic polynomial algorithms.

$$P \subseteq NP$$

NP-complete problems:

Belongs to NP class and every problem in NP is polynomially reducible to a decision problem D.



A decision problem D_1 is said to be polynomially reducible to a decision problem D_2 if there exists a function 't' that transforms instances of D_1 to instances of D_2 such that:

(1) 't' maps all yes instances of D_1 to yes instances of D_2 and all no instances of D_1 to no instances of D_2.

(2) 't' is computable by a polynomial-time algorithm.

Approximation algorithms for NP-hard problems:

NP-hard problems are problems that are atleast as hard as NP-complete problems. No polynomial-time algorithms exist for these problems. Hence, use approximation problem algorithms.

Heuristic - common sense rule drawn from experience rather than from a mathematically proved assertion.

To measure the accuracy of approximation algorithms for finding an optimal solution,

$$re(s_a) = \frac{f(s_a) - f(s^*)}{f(s^*)}$$

where $s^* \longrightarrow$ exact solution to the problem

$s_a \longrightarrow$ approximate solution to the problem

$$re(s_a) = \frac{f(s_a)}{f(s^*)} - \frac{f(s^*)}{f(s^*)} = \frac{f(s_a)}{f(s^*)} - 1$$

$$\therefore re(s_a) = \frac{f(s_a)}{f(s^*)}$$

where $re(s_a) \longrightarrow$ accuracy ratio

A polynomial-time approximation algorithm is said to be a C-approximation algorithm, where $C \geq 1$, if the accuracy ratio of the approximation it produces does not exceed C for any instance of the problem $r(s_a) \leq C$

Performance ratio : Smallest value of c for which the inequality $r(s_a) \leq c$ holds for all instances of the problem. (R_A)

R_A is close to 1 – desirable

1. Traveling Salesman Problem :

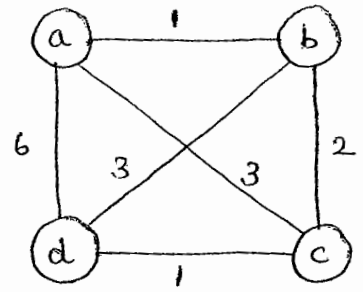
(1) Nearest – neighbour algorithm : (Greedy algorithm)

Step 1 : Choose an arbitrary city as the start

Step 2 : Repeat the following operation until all the cities have been visited - go to the unvisited city nearest the one visited last

Step 3 : Return to the starting city

Ex :



'a' → starting vertex

$a - b - c - d - a = 1 + 2 + 1 + 6 = 10$ ↓ optimal solution

From a, to b = 1 ✓

to c = 3

to d = 6

From b, to c = 2 ✓

to d = 3

From c, to d = 1 ✓

From d, to a = 6 ✓

By using exhaustive search,

$a-b-c-d-a = 1+2+1+6 = 10$

$a-b-d-c-a = 1+3+1+3 = 8$ ✓

$a-c-b-d-a = 3+2+3+6 = 14$

$a-c-d-b-a = 3+1+3+1 = 8$ ✓

$a-d-b-c-a = 6+3+2+3 = 14$

$a-d-c-b-a = 6+1+2+1 = 10$

Optimal solution = 8

Accuracy ratio $r(s_a) = \frac{f(s_a)}{f(s^*)} = \frac{10}{8} = 1.25$

The tour s_a is 25% longer than the tour s^*

(2) Multifragment - heuristic algorithm:
(Greedy technique)

Step 1: Sort the edges in increasing order of their weights.
Initialize set of tour edges to be constructed to the empty set.

Step 2: Repeat this step until a tour of length 'n' is obtained where 'n' is the number of cities.
Add the next edge on the sorted edge list to the set of tour edges provided this addition does not create a vertex of degree 3 or a cycle of length $< n$. Else skip the edge.

Step 3: Return the set of tour edges.

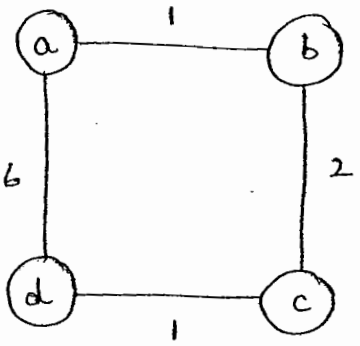
For the above example,

Set of edges in increasing order of weights

$\{(a,b), (c,d), (b,c), (a,c), (b,d), (a,d)\}$

The set of edges in the tour,

$\{(a,b), (c,d), (b,c), (a,d)\}$



(a,c) and (b,d) are not added since they form a cycle of length < 4.

The multifragment - heuristic algorithms tends to produce significantly better tours than the nearest - neighbour algorithms.

Accuracy of nearest - neighbour algorithms and multifragment - heuristic algorithms for Euclidean which satisfies:

(1) Triangle inequality

$d[i,j] \leq d[i,k] + d[k,j]$ $\quad i, j, k \rightarrow$ cities

(2) Symmetry

$d[i,j] = d[j,i]$ $\quad i, j \rightarrow$ cities

$$\frac{f(s_a)}{f(s^*)} \leq \frac{1}{2}\left(\lceil \log_2 n \rceil + 1\right)$$

(3) Minimum-spanning-tree-based algorithm:

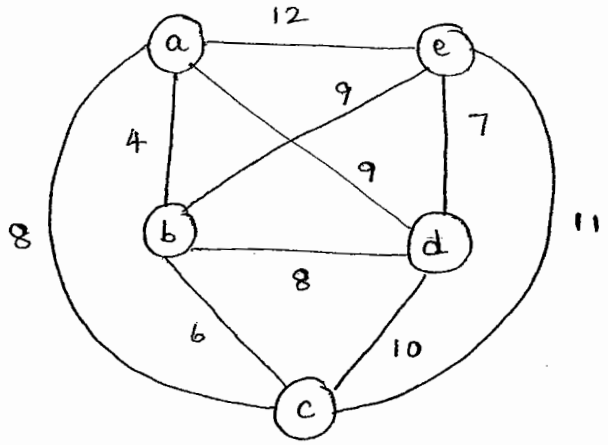
(a) Twice-around-the-tree algorithm:

Step 1: Construct a minimum spanning tree of the graph.

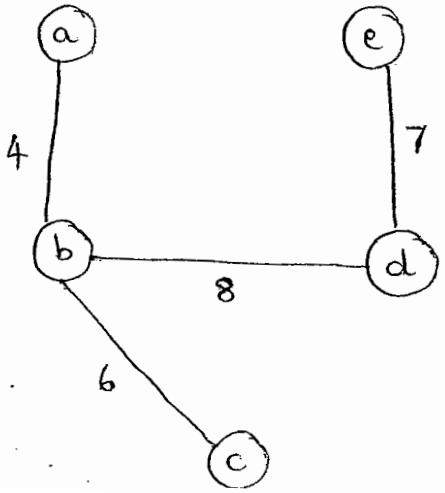
Step 2: Start at an arbitrary vertex, perform a walk around the minimum spanning tree recording all the vertices passed by.

Step 3: Scan the vertex list obtained in step 2 and eliminate from it all repeated occurences of the same vertex except the starting one at the end of the list.

Ex:



Minimum spanning tree consists of $\{(a,b), (b,c), (e,d), (b,d)\}$



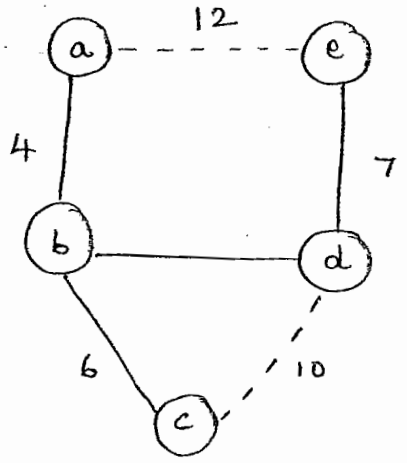
Walk that starts and ends at a

a, b, c, b, d, e, d, b, a

Eliminate second and third 'b', second 'd'

$\Longrightarrow$ a, b, c, d, e, a

Length = 4 + 6 + 10 + 7 + 12 = 39

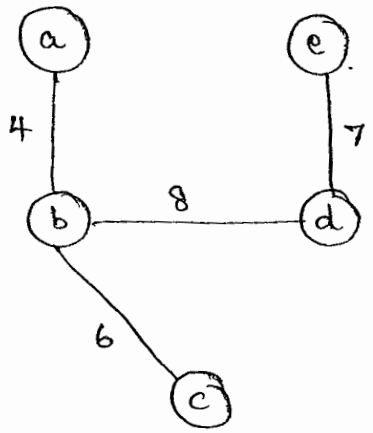


(b) Christofides algorithm :

Better performance ratio than twice-around-the-tree algorithm for Euclidean TSP.

For the above example,

Minimum spanning tree is,



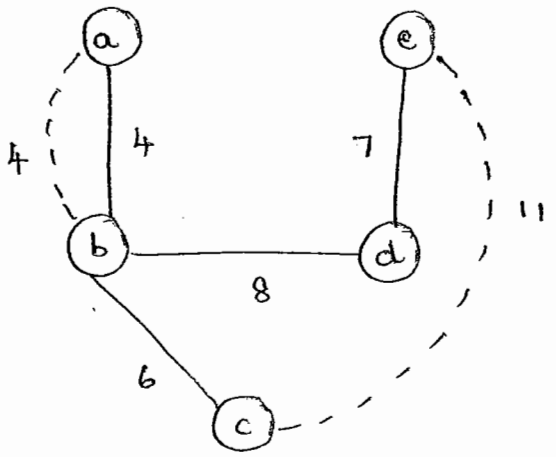
Vertices with odd-degree : a, b, c, e.

Eulerian circuit exists for a connecting multigraph with only even-degree vertices.

∴ Add the edge with least weight to the odd-degree vertices.

From a, b, c, e ⟹ (a,b) (a,c) (a,e) (b,c) (b,e) (c,e)

Add (a,b) and (c,e)



Eulerian circuit : a-b-c-e-d-b-a

Eliminating second 'b' ⟹ a-b-c-e-d-a

= 4+6+11+7+9

Length = 37 → optimal solution

Eliminating first 'b' ⟹ a-c-e-d-b-a

= 8+11+7+8+4

Length = 38

(4) Local search heuristics.

Iterative - improvement algorithms

(a) 2-opt algorithm
(b) 3-opt algorithm
(c) Lin-Kernighan algorithm

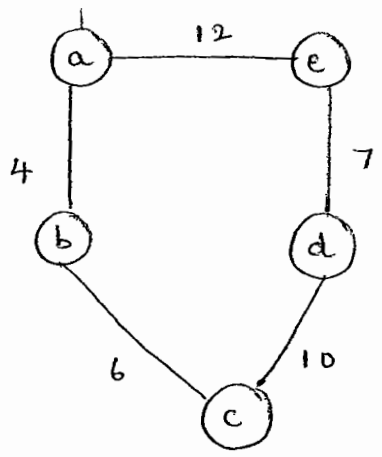
(a) 2-opt algorithm:

Start with an initial tour constructed randomly or using nearest - neighbour algorithm.

For the above example graph,

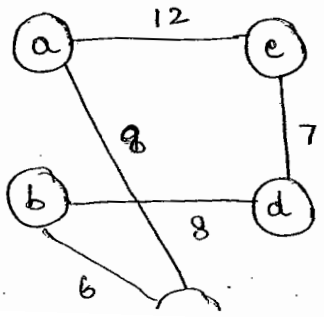
Initial tour : a-b-c-d-e-a = 4+6+10+7+12

Length = 39



Delete a pair of non-adjacent edges in the tour and reconnect their endpoints by different pair of edges to obtain another tour. This operation is called 2-change.

Case 1: By deleting ab and cd

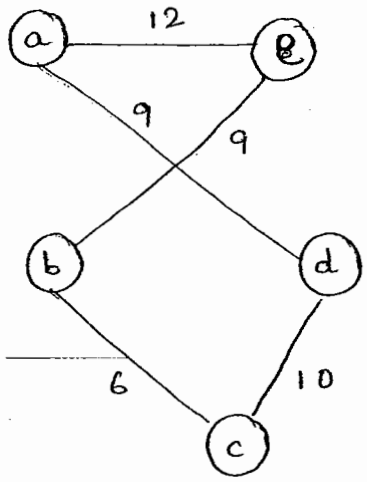


Add ac and bd

Length = 41 > 39

(a-c-b-d-e-a)

Case 2 : By deleting ab and de

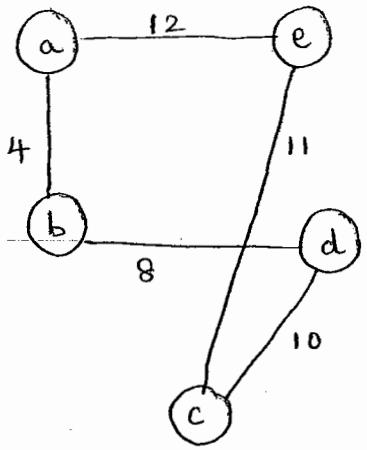


Add ad and be

Length = 46 > 39

(a-d-c-b-e-a)

Case 3 : By deleting bc and de

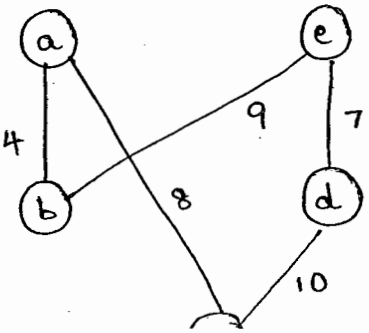


Add bd and ce

Length = 45 > 39

(a-b-d-c-e-a)

Case 4 : By deleting bc and ae



Add ac and be

We don't add ab and ce because edge ab is already present.

Length = 38 < 39

(a-b-e-d-c-a)

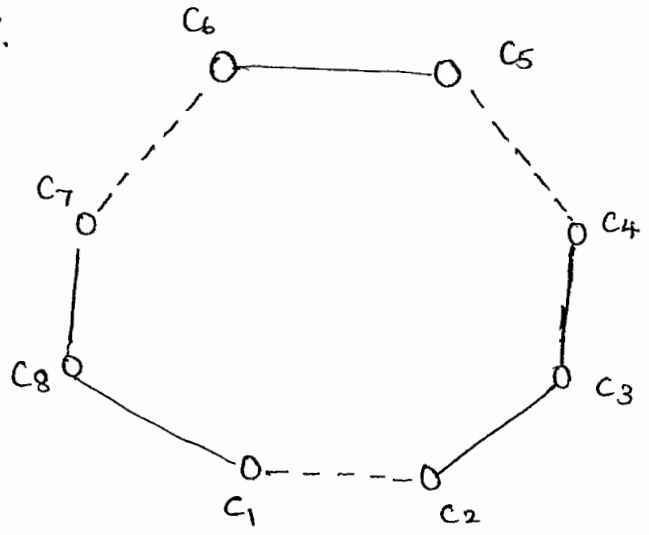
The new optimal tour is:

a-b-e-d-c-a

Optimal length = 38

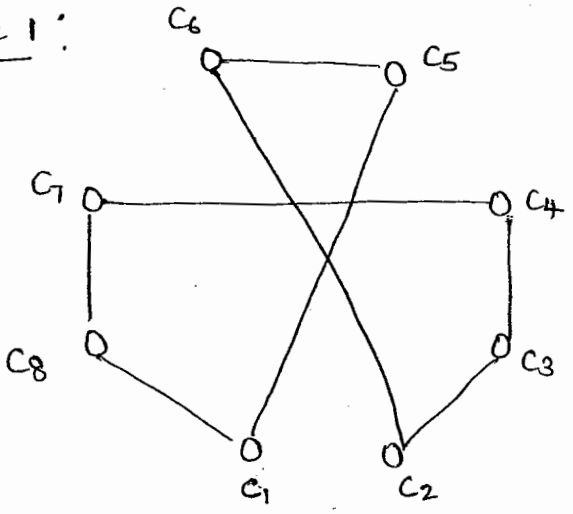
(b) 3-opt algorithm:

Ex:



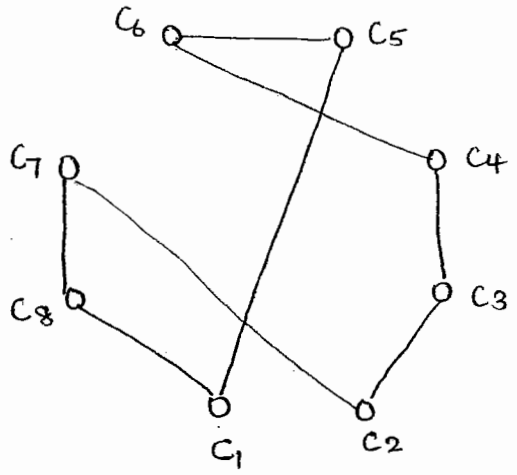
By deleting 3 non-adjacent edges C_1C_2, C_4C_5 and C_6C_7

Case 1:



Add C_1C_5, C_2C_6 and C_4C_7

Case 2: Add C_3C_5, C_1C_4 and C_2C_6



(c) Lin-Kernighan algorithm:

Variable-opt algorithm — 3-opt move followed by a sequence of 2-opt moves.

Empirical results of TSP:

Lower bound - Held-Karp bound is used to measure average tour quality.

The algorithm's accuracy ratio,

$$r(s_a) = \frac{f(s_a)}{HK(s^*)}$$

where $f(s_a) \longrightarrow$ length of heuristic tour s_a

$HK(s^*) \longrightarrow$ Held-Karp lower bound on the shortest-tour length.

2. Knapsack problem:

(1) Greedy algorithm for discrete knapsack problem:

Step 1: Compute the value-to-weight ratios or profit-to-weight ratios $r_i = v_i / w_i$ for $i = 1, 2, \ldots n$ for the given items.

Step 2: Sort the items in non-increasing or decreasing order of ratios computed in step 1.

Step 3: Repeat the following operation until no item is left in the sorted list. If the current item on the list fits into the knapsack, place it else proceed to the next item.

Ex: $n = 4$, $m = 10$

Item	Weight	Value	Value / Weight
1	4	40	10
2	7	42	6
3	5	25	5
4	3	12	4

Item 1 of weight 4 is placed.

Remaining capacity = 6

∴ Can't place item 2 (weight = 7)

Next, item 3 of weight 5 is placed.

Remaining capacity = 1

∴ Can't place item 4 (weight = 3)

∴ Optimal solution = 65

Objects selected = 1 and 3

(2) Greedy algorithm for the continuous knapsack problem:

Step 1: Compute the value-to-weight ratios v_i/w_i for $i = 1, 2, \ldots n$

Step 2: Sort the items in decreasing order of the ratios computed in step 1.

Step 3: Repeat the operation until the knapsack is filled to it's full capacity or no item is left in the sorted list.

If the current item on the list fits into the knapsack entirely, take it and proceed to the next item, else take it's largest fraction to fill the knapsack to it's full capacity and stop.

For the above example,

Item 1 of weight 4 is entirely placed

Remaining capacity = 6

Next, 6/7 of item 2 is placed to fill it's capacity.

∴ Optimal solution = 76

$(40 + 6/7 \times 42)$

Approximation schemes:

Discrete version of knapsack problem:

$$\frac{f(s^*)}{f(s_a^{(K)})} \leq 1 + 1/K$$

$0 \leq K < n$, K and n are integers

The algorithm generates all subsets of K items or less and for each one that fits into the knapsack, it adds the remaining items as the greedy algorithm would. Subset with highest value is returned as output.

Ex: $n = 4$, $m = 10$

Item	Weight	Value	Value / Weight
1	4	40	10
2	7	42	6
3	5	25	5
4	1	4	4

Assume $K = 2$,

Subsets	Added items	Value
{ }	1, 3, 4	69
{1}	3, 4	69
{2}	4	46
{3}	1, 4	69
{4}	1, 3	69

Subsets	Added items	Value
{1, 2}	Not feasible Weight = 11 > m	—
{1, 3}	4	69
{1, 4}	3	69
{2, 3}	Not feasible Weight = 12 > m	—
{2, 4}	None	46
{3, 4}	1	69

∴ Optimal solution = 69

Items = {1, 3, 4}

TRANSFORM AND CONQUER

The 3 variations are:

1. Instance simplification - transform to a simpler or more convenient instance of the same problem.
2. Representation change - transform to a different representation of the same instance.
3. Problem reduction - transform to an instance of a different problem for which an algorithm is already available.

PRESORTING:

It is sometimes considered to be easier to perform certain operations on the list when the list is sorted.

Ex: 1. Checking element uniqueness in an array:

The brute force algorithm to find whether elements are unique or not has a time complexity of $O(n^2)$.

If at all the array is already sorted then we can use the following algorithm:

```
Algorithm Unique(a,n)
// Purpose: To find element uniqueness in an array
// Input: The array 'a'
// Output: 'true' if elements are unique else
           'false'
```

Initially sort the array

```
for i ← 0 to n-2
    if a[i] = a[i+1] return false
end for
return true
```

If at all we use a good sorting algorithm such as merge sort then the overall time complexity is,

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

2. Computing a mode value:

A mode is a value that occurs most frequently in the list. The worst case time complexity for this algorithm is $O(n^2)$.

The algorithm to find mode in a sorted array is,

```
Algorithm mode (a,n)
// Purpose : To compute the mode
// Input : The sorted array 'a'
// Output : Mode value
    i ← 0
    mode_frequency ← 0
    while i ≤ n-1
        runlength ← 1
        runvalue ← a[i]
        while i+runlength ≤ n-1 and a[i+runlength]
```

```
            runlength ++
        end while
        if runlength > modefrequency
            modefrequency ← runlength
            modevalue ← runvalue
        end if
        i ← i + runlength
    end while
    return modevalue
```

Suppose we have the list,

10, 15, 20, 20, 25, 25, 25, 25, 40

Mode = 25

Even here the the worst case time efficiency is better when compared to the brute force algorithm.

3. Searching problem:

For example, consider binary search where the elements are to be sorted. The time efficiency is,

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

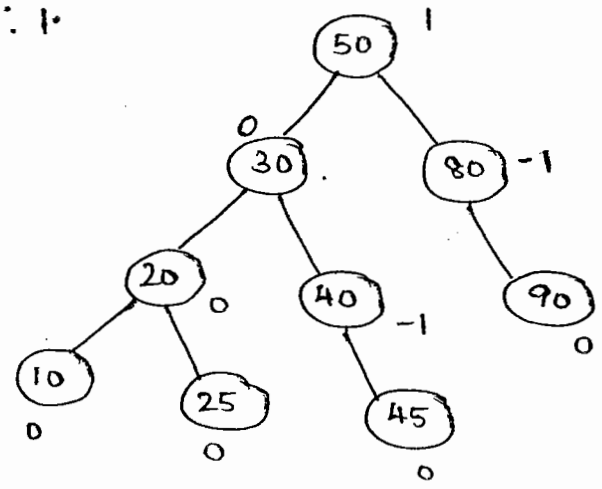
If the same list is to be searched several times then it's very useful.

BALANCED SEARCH TREES:

1. AVL trees : (Instance simplification)

An AVL tree is a binary search tree in which the balance factor of every node which is the difference between the height of the node's left and right subtrees is either 0 or +1 or ~~-1~~.

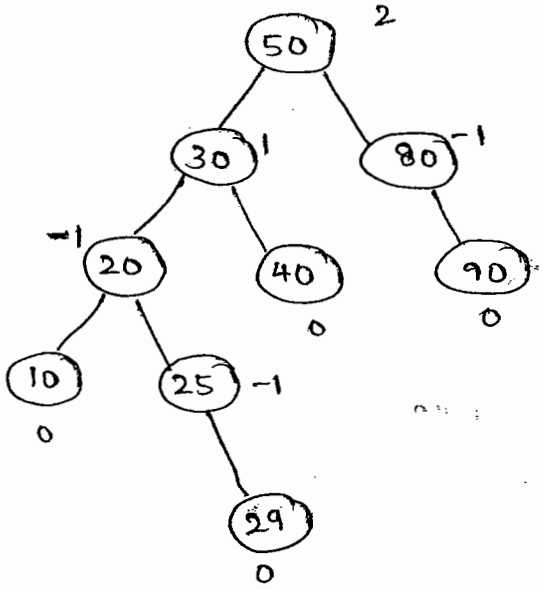
Ex: 1.



→ AVL tree

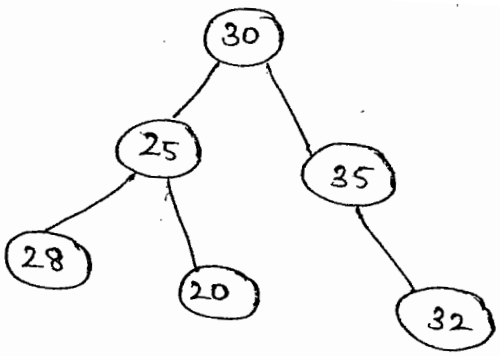
The balance factor of all leaf nodes = 0

2.



→ Not AVL tree

3.

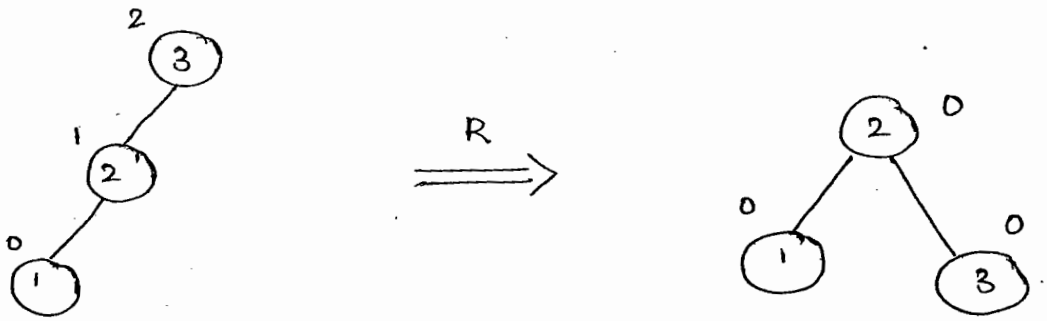


Not an AVL tree since it is not a binary search tree.

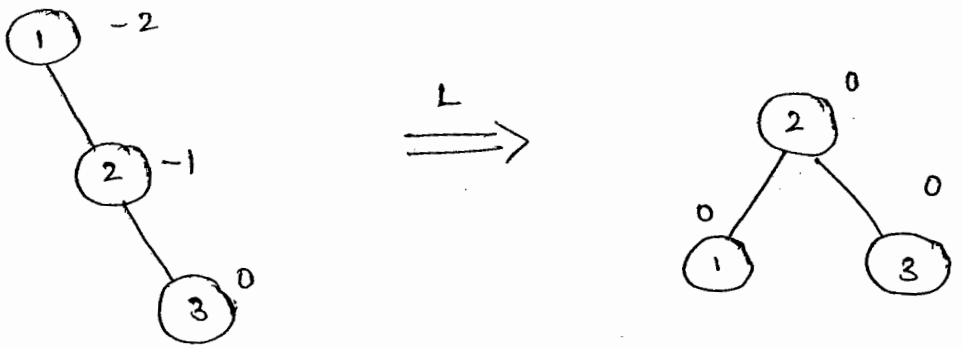
If an insertion of new node makes the AVL tree unbalanced we transform the tree by rotation. Rotation in an AVL tree is a local transformation of it's subtree whose balance factor is -2 or +2

The 4 types of rotation are:

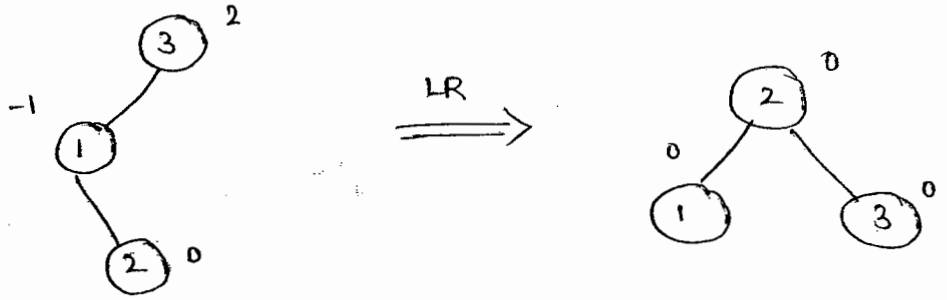
1. Single right rotation / R- rotation:



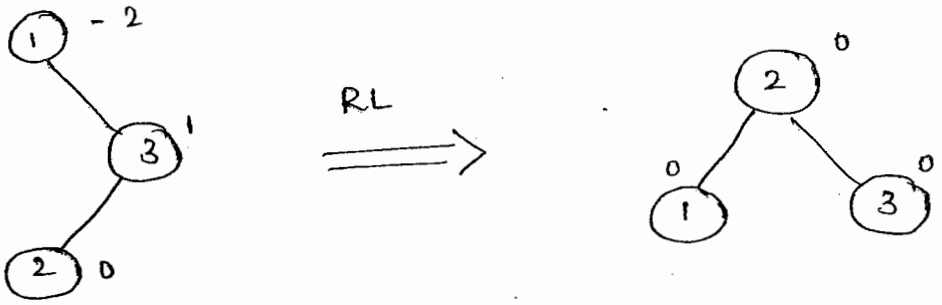
2. Single left rotation / L- rotation:



3. Double left-right rotation / LR-rotation :



4. Double right-left rotation / RL-rotation :



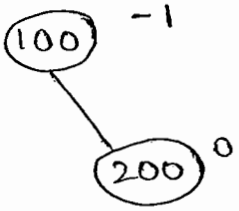
1. Construct an AVL tree by inserting the following elements successively from an empty tree.

100, 200, 300, 250, 270, 70, 40

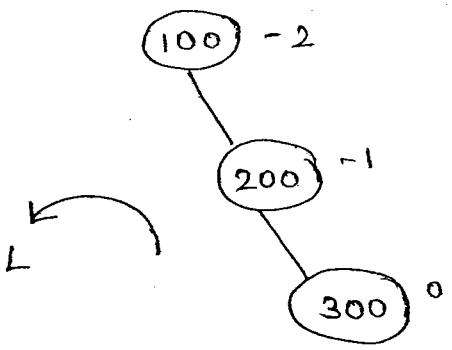
Step 1: Item = 100. Since tree is empty, 100 is made as the root node.

BF = 0

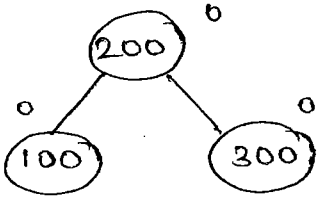
Step 2: Item = 200. It is inserted as shown below



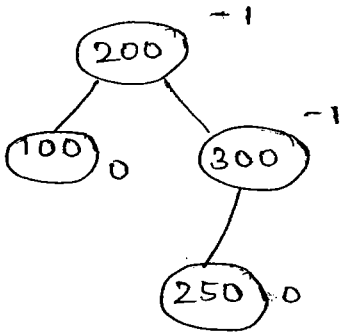
Step 3: Item = 300



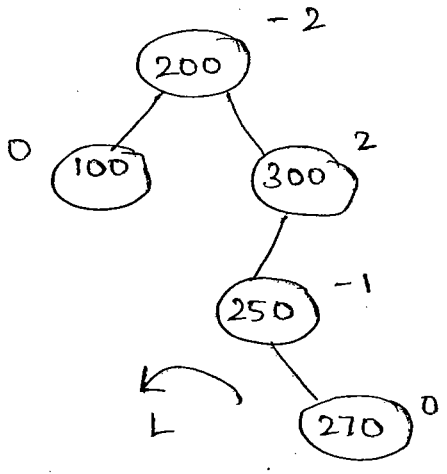
Since Bf of 100 is -2 the tree is not balanced. Rotate the tree towards left as it is more populated on the right.



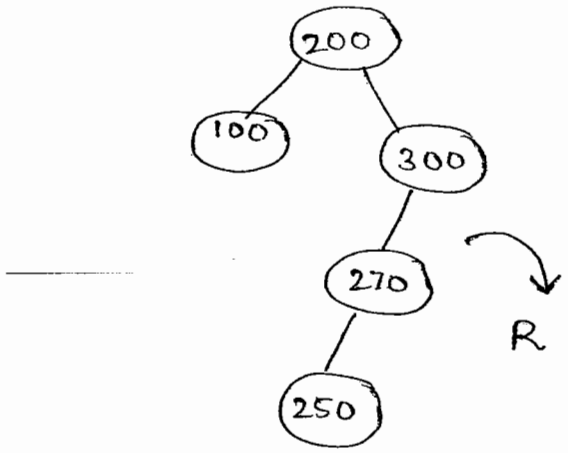
Step 4: Item = 250

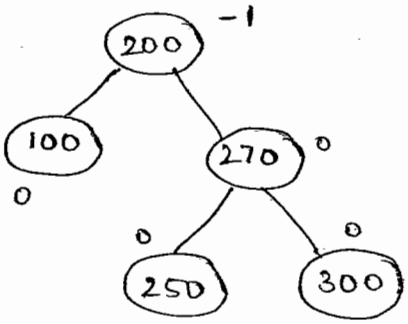


Step 5: Item = 270

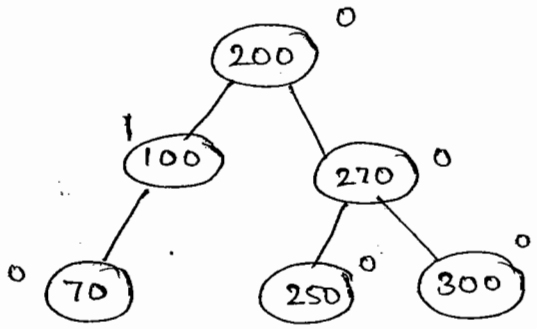


The BF of 200 is -2 and BF of 300 is 2.
Hence not balanced.

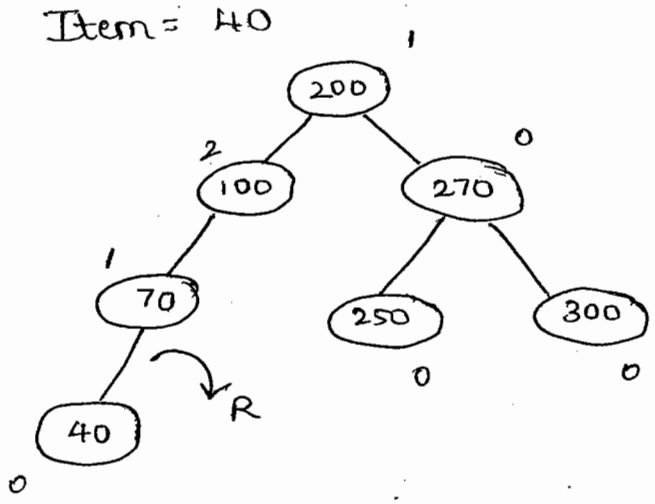




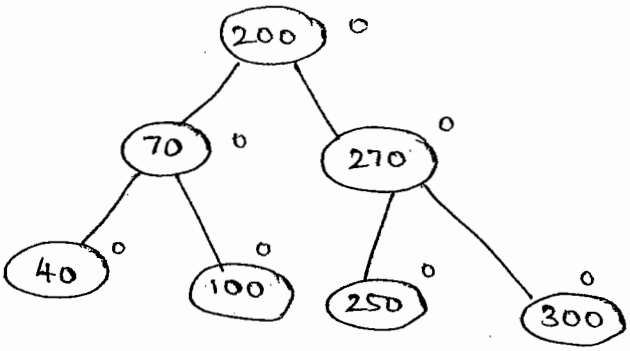
Step 6: Item = 70



Step 7: Item = 40

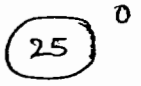


Since BF of 100 is 2 the tree is not balanced.
Rotate the tree towards right.

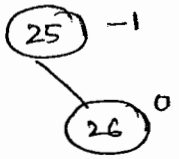


2. Construct an AVL tree : 25, 26, 28, 23, 22, 24, 27

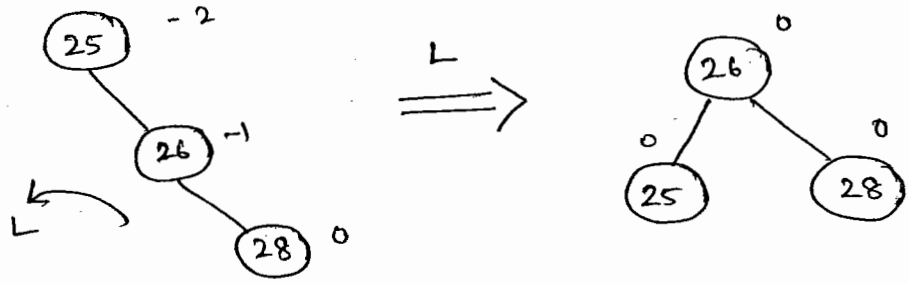
Step 1: Item = 25



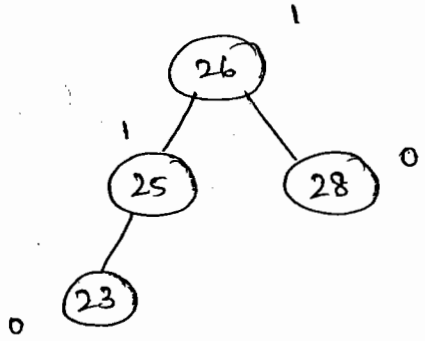
Step 2: Item = 26



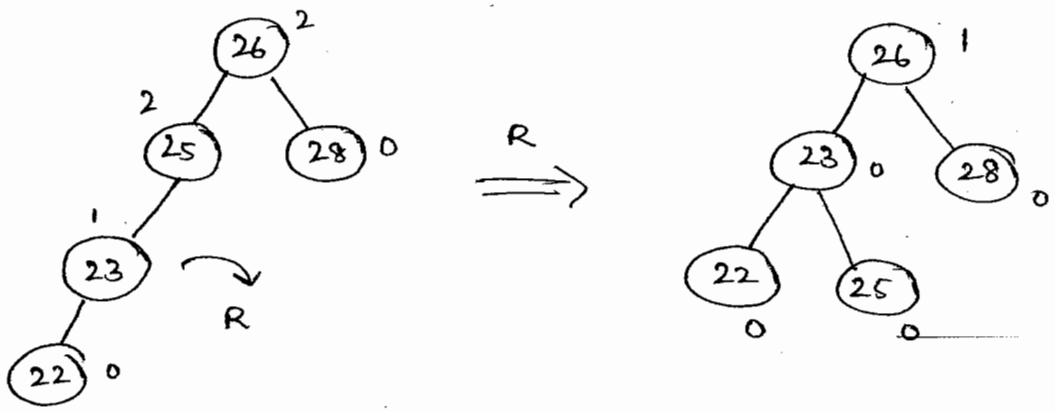
Step 3: Item = 28



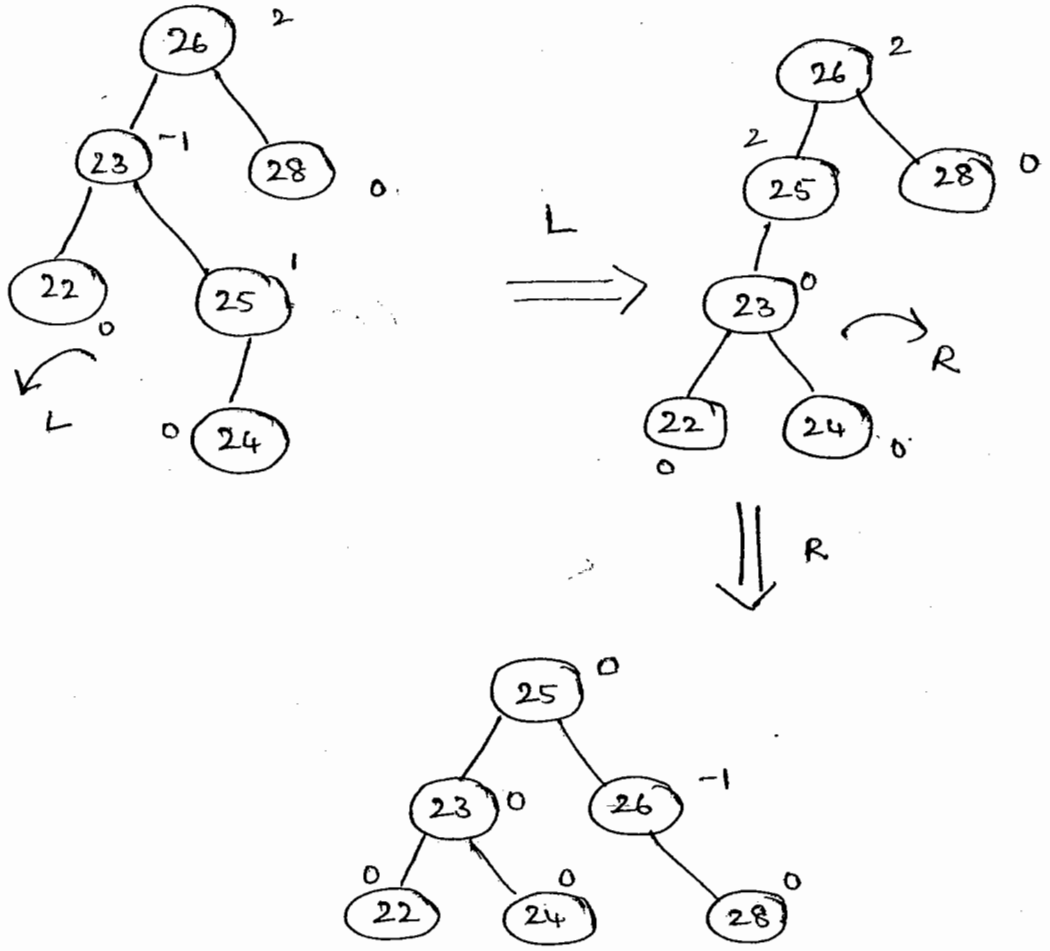
Step 4: Item = 23



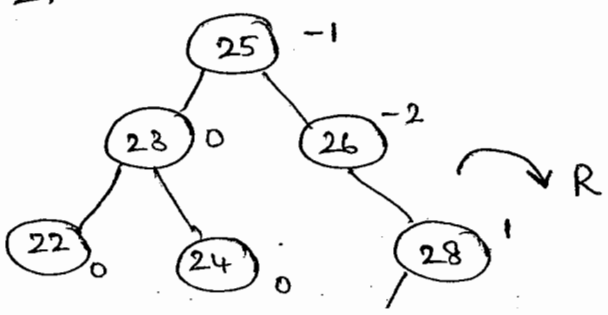
Step 5: Item = 22

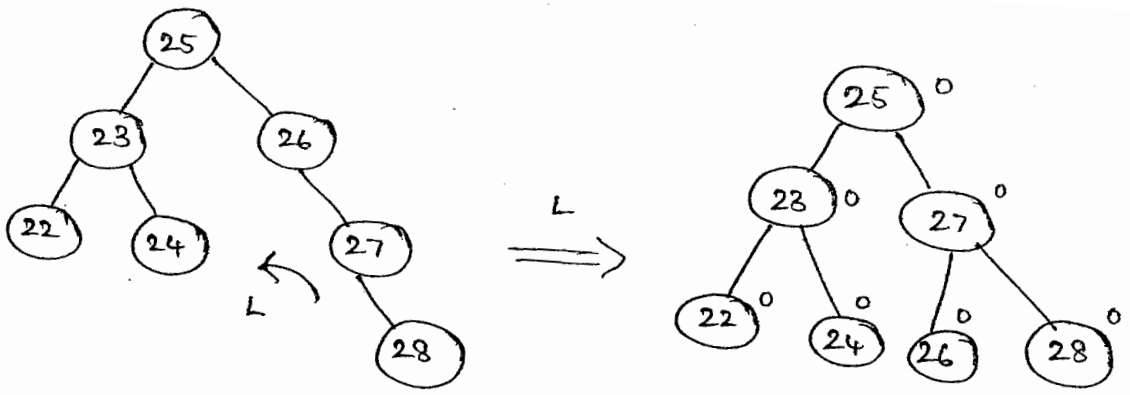


Step 6: Item = 24



Step 7: Item = 27





Searching and inserting in ~~an~~ AVL tree has time complexity O(log n). Searching in an AVL tree requires almost the same number of comparisons as in binary search.

2-3 TREES : (Representation change)

A 2-3 tree can have 2 or 3 children. All it's leaves must be on the same level i.e it is height balanced.

1. Construct a 2-3 tree for the list C, O, M, P, U, T, I, N, G

Step 1 : Item = C

It is inserted into the tree as a root node.

(C)

Step 2 : Item = O

This is inserted to the right of C as shown,

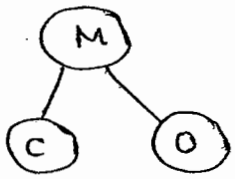
(C : O)

Step 3 : Item = M

This is inserted between C and O as shown,

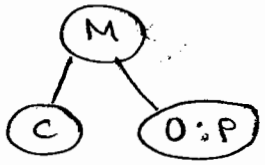
(C : M : O)

Since a node cannot have more than 2 items, move the middle element to the parent position. The resulting tree is shown below.



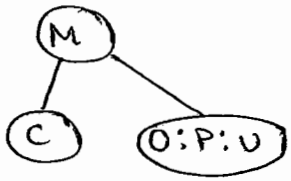
Step 4 : Item = P

It has to be inserted to the right of O.

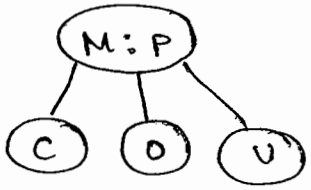


Step 5 : Item = U

It is inserted to the right of P.

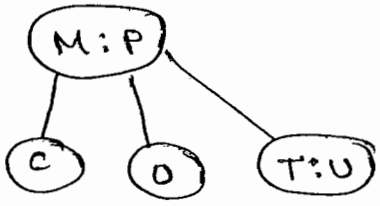


Since a node cannot have more than 2 items, move the middle element P to the parent position.



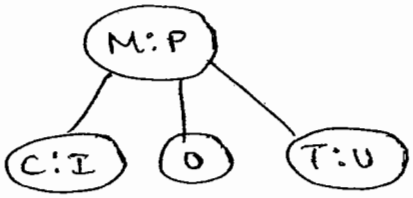
Step 6 : Item = T

It is inserted towards the left of U.



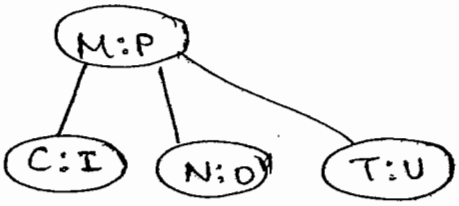
Step 7 : Item = I

It is inserted towards the right of C .



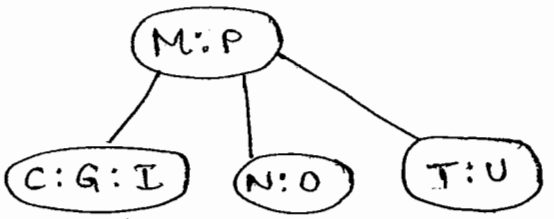
Step 8 : Item = N

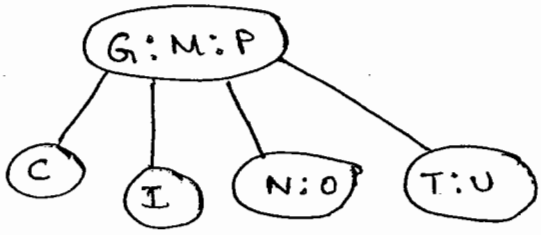
It is inserted towards the left of O .

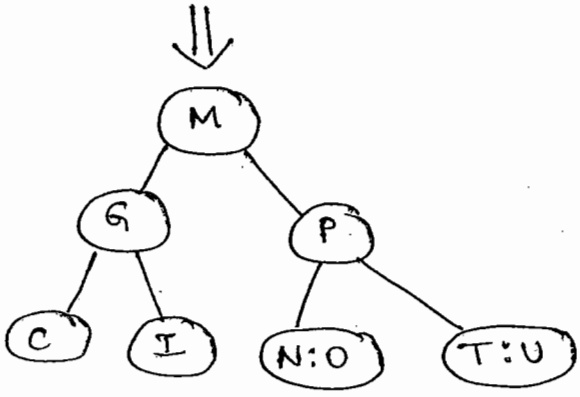


Step 9 : Item = G

It is inserted between C and I as shown,

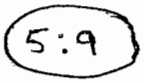


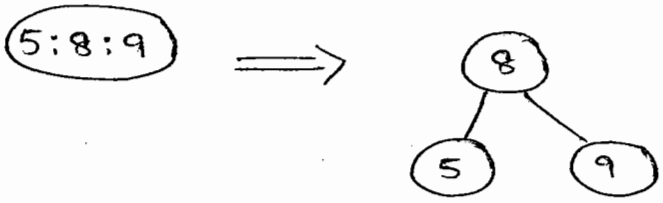


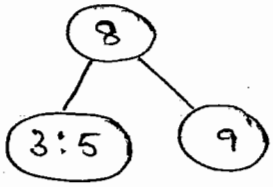


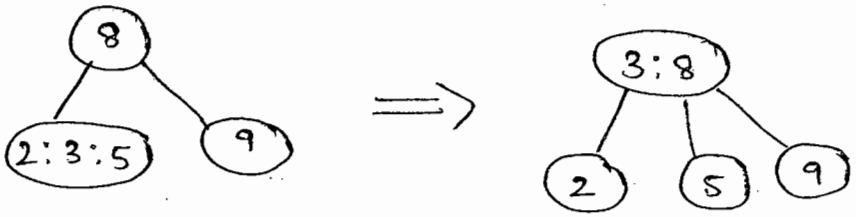
2. Construct a 2-3 for 9, 5, 8, 3, 2, 4, 7

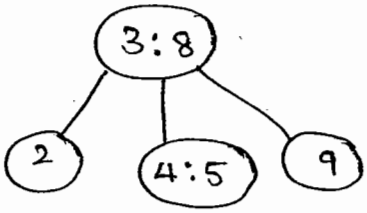
(1) 

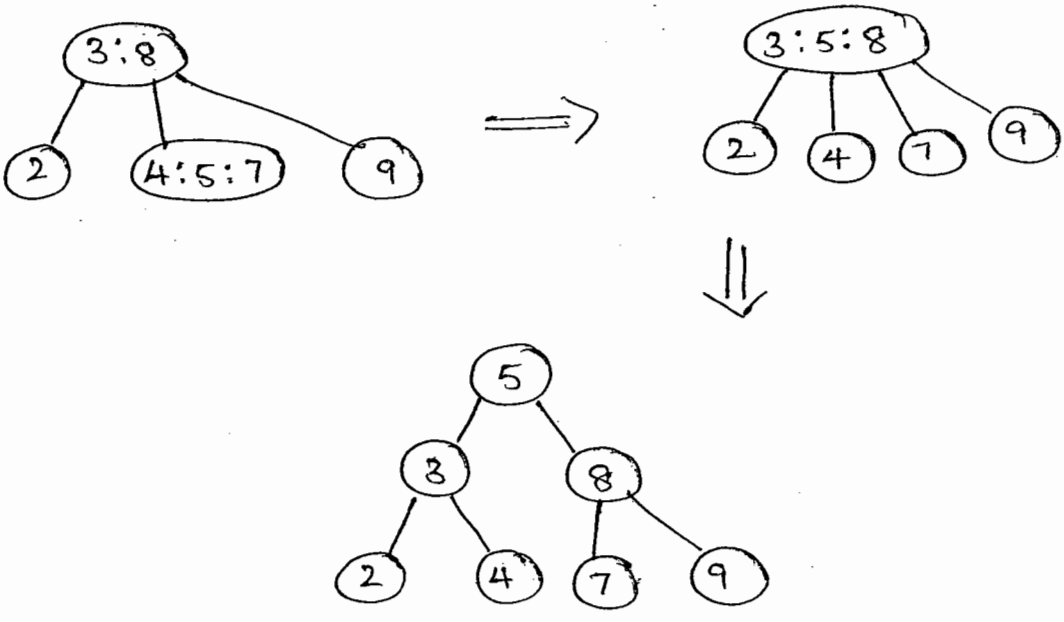
(2) 

(3) 

(4) 

(5) 

(6) 

(7) 

Time complexity :

Since a 2-3 tree is always height balanced, the efficiency on any operation depends on the height of the tree. So to find the time complexity we find the lower bound and upper bound.

Upper bound : Consider a 2-3 tree with each node exhibiting the property of 2-node (tree is similar to a binary search tree). Assume each node has only 1 item. The relation between the number of items and number of nodes is,

$$n = 2^0 + 2^1 + 2^2 + \text{---} + 2^i$$

But the leaf nodes may have more than 1 item. In such a situation the number of items $\geq$ numbe of nodes.

$$n \geq 2^0 + 2^1 + 2^2 + \text{---} + 2^i$$

$$n \geq 1\,\frac{(2^{i+1}-1)}{2-1} \qquad \because S = \frac{a(2^n-1)}{2-1}$$

$$n \geq 2^{i+1} - 1$$

$$n \geq 2^h - 1$$

$$n+1 \geq 2^h$$

$$2^h \leq n+1$$

Taking log on both sides,

$$h \cdot \log_2 2 \leq \log_2 (n+1)$$

$$h \leq \log_2 (n+1) \text{ ——— (1)}$$

Lower bound:

Consider a 2-3 tree with each node exhibiting the property of 3-node such that a node has a maximum of 2 items

$$n = 2\times3^0 + 2\times3^1 + 2\times3^2 + \text{----} + 2\times3^i$$

In case a leaf has only one item then the above relation can be written as,

$$n \leq 2\times3^0 + 2\times3^1 + \text{---} + 2\times3^i$$

$$n \leq 2\left[3^0 + 3^1 + 3^2 + \text{----} + 3^i\right]$$

$$n \leq 2\left[1\left(\frac{3^{i+1}-1}{3-1}\right)\right]$$

$$n \leq 3^{i+1} - 1$$

$$n+1 \leq 3^{i+1}$$

$$3^{i+1} \geq n+1$$

$$3^h \geq n+1$$

Taking log on both sides,

$$h \log_3 3 \geq \log_3(n+1)$$

$$h \geq \log_3(n+1) \quad \text{——(2)}$$

Using (1) and (2) we have,

$$\log_3(n+1) \leq h \leq \log_2(n+1)$$

By neglecting all lower order terms and constants,

$$h \approx \log n$$

$$\boxed{\therefore f(n) \in O(\log n)}$$

HEAP:

Heap is almost complete binary tree with parental dominance. Parent node is always less than left child or right child.

Given the parent position p, the position of left child is given by $2p+1$ and position of right child is given by $2p+2$. Given child position c, the parent position is given by $\frac{c-1}{2}$.

Heap can be constructed using 2 different techniques

1. Bottom up approach
2. Top down approach

1. Bottom up approach:

```
Algorithm bottom-up-heapify(n, a)
// Purpose: To create descending heap
// Inputs: n → number of elements
           a → list of elements
// Output: a → contains the descending heap
    for i = (n-1)/2 down to 0
        p ← i
        item ← a[p]
        c ← 2p+1
        while c<n
            if c+1 <n
                if a[c] < a[c+1]
                    c ← c+1
                end if
            end if
```

```
        if item < a[c]
            a[p] ← a[c]
            p ← c
            c ← 2p+1
        else
            break
        end if
    end while
    a[p] ← item
end for
```

Time complexity:

Inserting an element at the appropriate place depends on the height of the tree which is given by,

$$h = \log_2 n$$

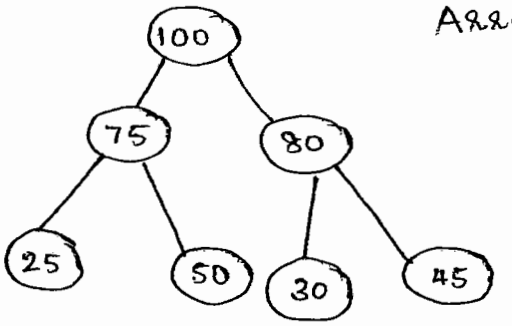
While creating a heap, the number of comparisons required to move an item from the parent to child position is equal to 2. So the total number of comparisons required to move a single node from level i to leaf level is,

$$f(n) = \sum_{i=0}^{h-1} 2 \times (h-i) \times 2^i$$

After simplification we get,

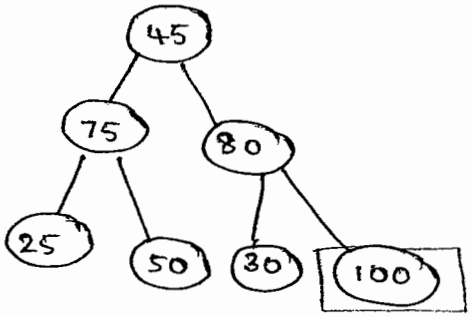
$$f(n) = O(n)$$

Consider the heap shown below which is created using bottom up approach.



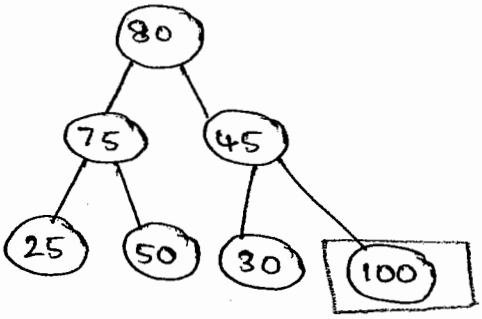
Array = 100, 75, 80, 25, 50, 30, 45

Exchange a[0] and a[6]

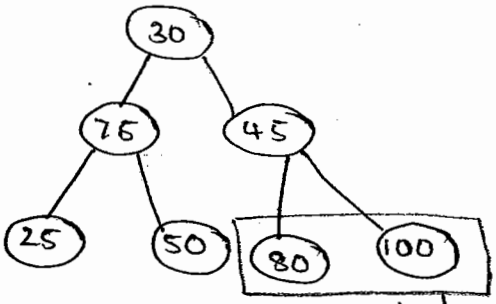


sorted

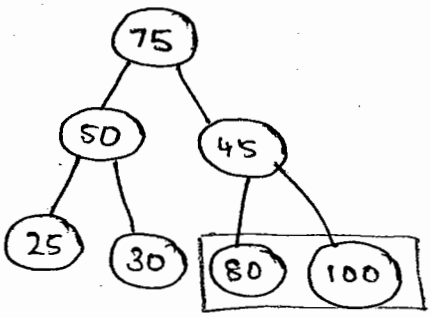
Recreate a heap for n = 6



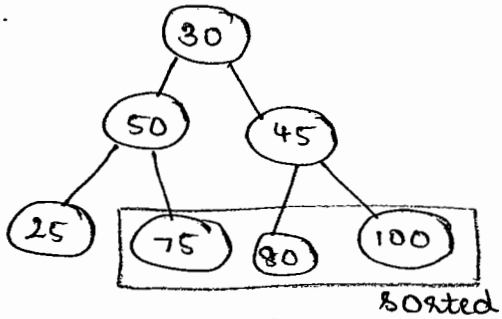
exchange a[0] - a[5]



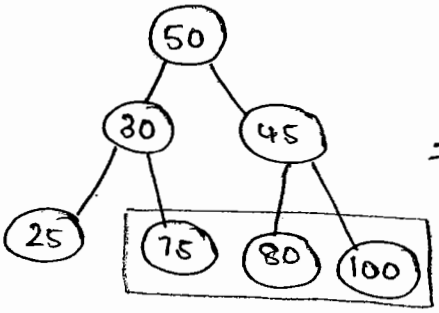
sorted



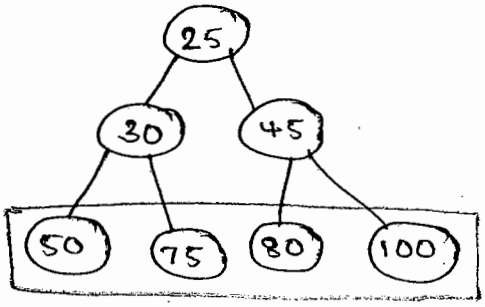
exchange a[0] - a[4]

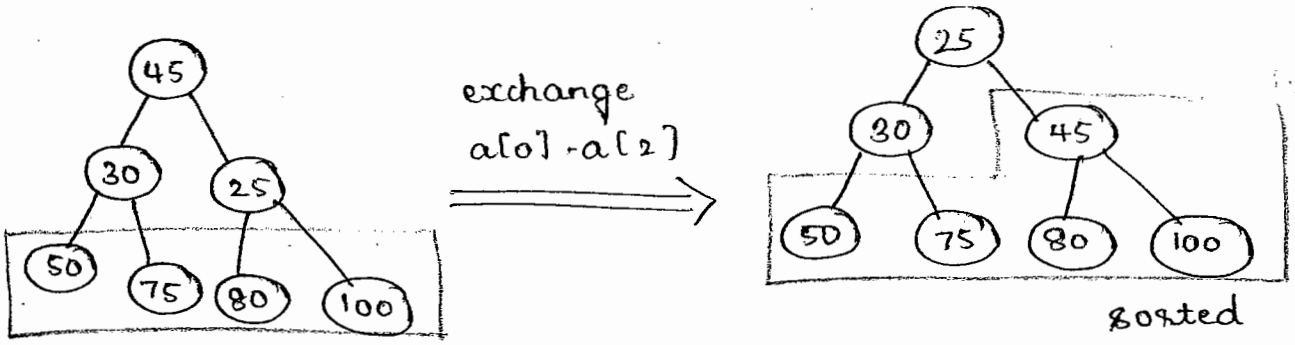


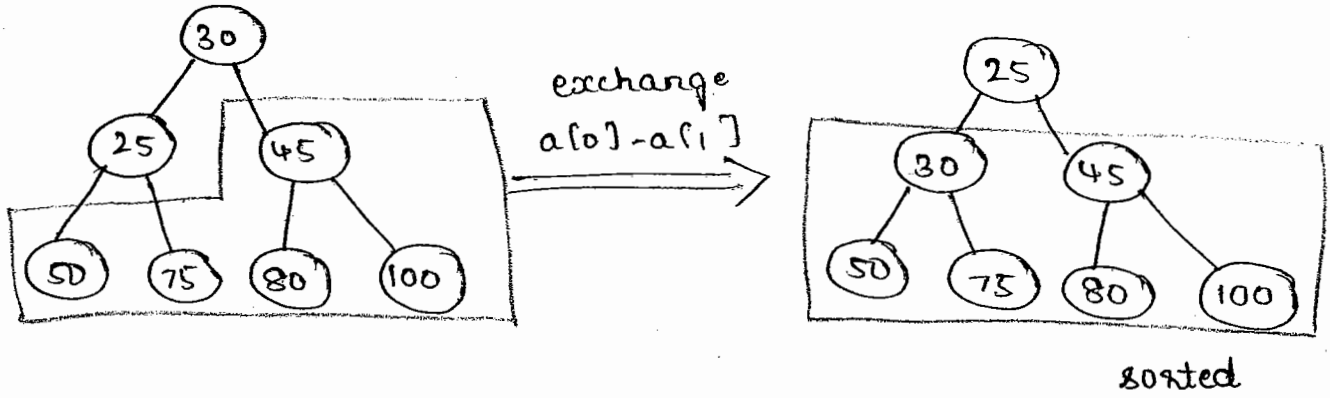
sorted



exchange a[0] - a[3]







Algorithm heap-sort (n, a)

// Purpose: To arrange the elements in ascending orde

// Inputs: n → number of elements

a → descending heap

// Output: a → sorted list

```
bottom-up-heapify (n, a)
for i ← n-1 down to 1
        exchange a[0], a[i]
        bottom-up-heapify (a, i)
end for
```

Time complexity:

$f(n) = f_1(n) + f_2(n)$

where $f_1(n)$ is the time to create the heap

$$\boxed{f_1(n) \in O(n)}$$

$f_2(n)$ is the time to sort the elements

The relation between 'n' and 'h' is given by

$h = \log_2 n$

So time to create the heap for 'n' elements

$= \log_2 n$

After exchanging we recreate the heap for n-1 elements whose time complexity is given by

$\log_2(n-1)$

$$f_2(n) = \log_2 n + \log_2(n-1) + \text{----} + \log_2 1$$

$$f_2(n) = \sum_{i=0}^{n-1} \log_2(n-i)$$

$$f_2(n) \leq n \log_2 n$$

$$\therefore f_2(n) \approx n \log_2 n$$

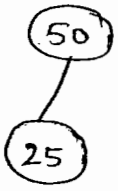
To arrange elements in descending order create an ascending heap. $\therefore f(n) = n + n \log_2 n \approx n \log_2 n$

Creating heap using top down approach:

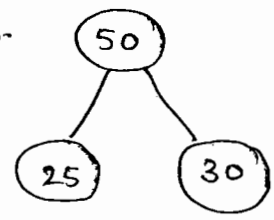
50, 25, 30, 75, 100, 45, 80

1.

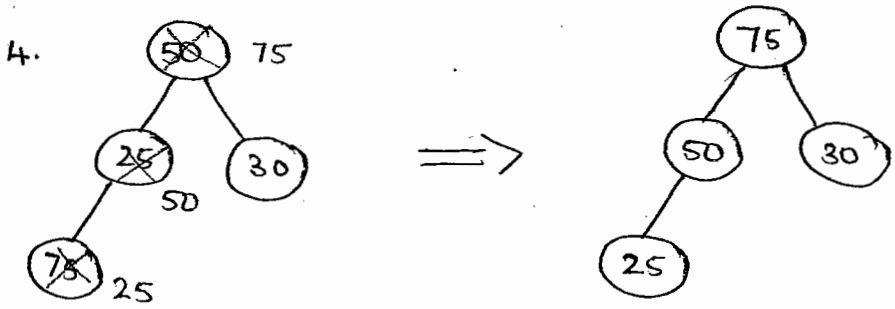
2.



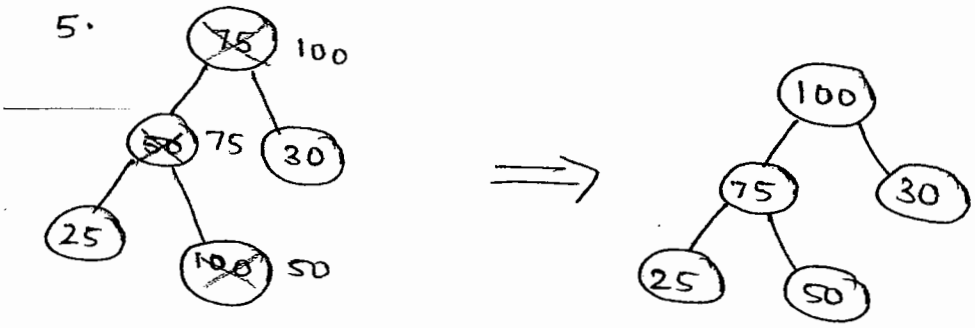
3.



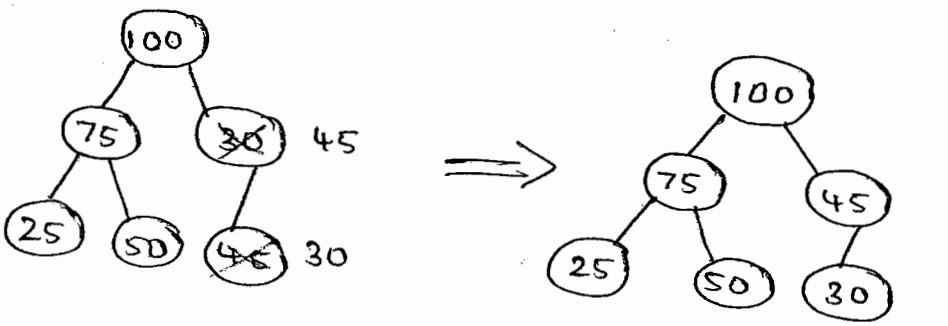
4.



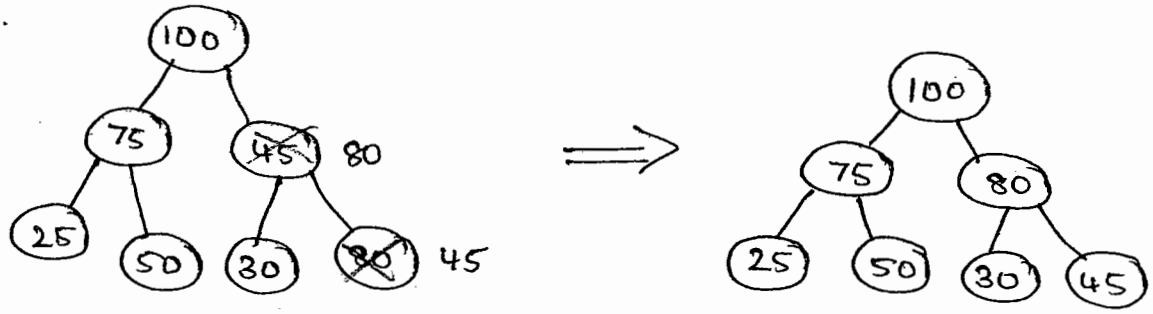
5.



6.



7.



Algorithm top_down_heapify (n,a)

// Purpose : To create a heap using top down approach

// Inputs : n → number of elements

a → list of elements

// Output: a → descending heap

for i ← 0 to n-1

c = i

```
        p = (c-1)/2
        while c!=0 and item > a[p]
            a[c] = a[p]
            c = p
            p = (c-1)/2
        end while
        a[c] = item
    end for
```

PROBLEM REDUCTION:

1. Computing LCM:

LCM (Least Common Multiple) of two positive integers m and n is the smallest integer that is divisible by both m and n.

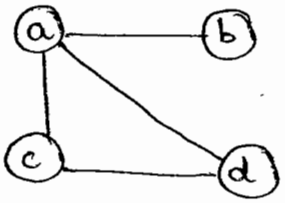
To find LCM we have to compute prime factors and hence it is a drawback as in middle school method of finding GCD. We know that Euclid's algorithm to find GCD of 2 numbers is efficient.

$$\text{Hence, } LCM(m,n) = \frac{m \times n}{GCD(m,n)}$$

2. Counting paths in a graph:

For a given graph, its adjacency matrix A gives the paths of length 1 and its square gives the paths of length 2.

Ex:



For 'a', paths of length 2 are:
a-b-a, a-c-a,
a-d-a ∴ 3 paths

Paths of length 1 = $A =$

	a	b	c	d
a	0	1	1	1
b	1	0	0	0
c	1	0	0	1
d	1	0	1	0

Paths of length 2 = $A^2 =$

	a	b	c	d
a	3	0	1	1
b	0	1	1	1
c	1	1	2	1
d	1	1	1	2

3. To find minimum : (Reduction of optimization problems)

The function maximum (a, n) exists

Consider the numbers,

25, 35, 10, 5, 40, 60, 20

Take negative form,

-25, -35, -10, -5, -40, -60, -20

Now find the maximum

max = -5

Print -(max) = 5

Algorithm minimum (a, n)

for i ← 0 to n-1

a[i] = -a[i]

end for

p = maximum (a, n)

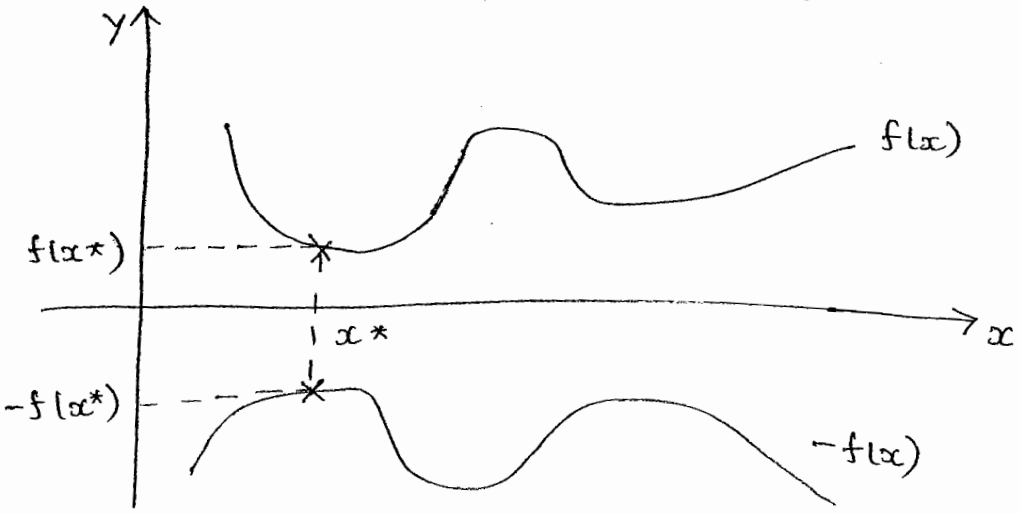
print (-p)

Relationships:

$$\min f(x) = -\max [-f(x)]$$

and

$$\max f(x) = -\min [-f(x)]$$



Relationship between minimization and maximization problems is shown above.

4. Reduction to graph problems:

Most of the puzzles and games can be solved by reduction to one of the standard graph problems. Vertices of a graph represent possible states of the problem and edges represent permitted transition among such states. One of the graph's vertex represents an initial state and another represents goal of the problem. Such a graph is called state-space graph.

SPACE AND TIME TRADEOFFS

One technique that exploits space for time trade offs simply uses extra space for faster and flexible access to data. This is called prestructuring. Another technique is input enhancement.

SORTING BY COUNTING:

Comparison counting Sort:

```
Algorithm comparison_counting_sort (n, a, b)
// Purpose : To sort the given elements using sorting by counting
// Inputs : n → number of elements
            a → list of elements to be sorted
// Output : b → sorted array

for i ← 0 to n-1
    c[i] ← 0
end for
// Compute the number of elements less than a[i]
for i ← 0 to n-2
    for j ← i+1 to n-1
        if a[i] < a[j]
            c[j] ← c[j]+1
        else
            c[i] ← c[i]+1
        end if
    end for
end for
```

```
for i ← 0 to n-1
    b[c[i]] = a[i]
end for
```

The time complexity is,

The basic operation is "if a[i] < a[j]"

$$f(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

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

$$= \sum_{i=0}^{n-2} n-i-1$$

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

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

By neglecting all lower order terms and constants we get,

$$\therefore f(n) \in O(n^2)$$

Ex:

	a[]	57	28	13	65	76
	c[]	0	0	0	0	0
i=0	c[]	2	0	0	1	1
i=1	c[]	2	1	0	2	2
i=2	c[]	2	1	0	3	3
i=3	c[]	2	1	0	3	4
	b[]	13	28	57	65	76

DISTRIBUTION COUNTING:

Algorithm sorting-by-distribution (n, a, b)

// Purpose: To arrange the numbers in ascending order using distribution counting

// Inputs: n → number of elements

a → list of elements to be sorted

// Output: b → sorted array

```
lb = minimum (n, a)
ub = maximum (n, a)

for i ← 0 to ub-lb
        D[i] ← 0
end for

// Find the frequency of each item
for i ← 0 to n-1
    j = a[i] - lb
    D[j] = D[j] + 1
end for

// Find the accumulated frequency
    for i ← 1 to ub-lb
        D[i] = D[i-1] + D[i]
    end for

// Scan each item from right to left, decrement the
   accumulated frequency, insert the item into this
   position.
```

```
for i ← n-1 down to 0
    j = a[i] - lb
    D[j] = D[j] - 1
    B[D[j]] = a[i]
end for
```

Time complexity is given by,

$$f(n) = \sum_{i=n-1}^{0} 1$$

$$= n-1-0+1$$

$$= n$$

$$\therefore f(n) \in O(n)$$

Since the time complexity is linear it is also called linear sorting algorithm.

HASHING :

Hashing is based on distribution of keys in an one-dimensional array called hash table. The distribution is done by computing for each of the keys a predefined function h called the hash function. This function is used to assign an integer between 0 to m-1 called the hash address to a key. For example, $h(k) = k \bmod m$ can be a hash function. A hash function needs to distribute keys among the cells of the hash table evenly and a hash function must be easy to compute.

OPEN HASHING (SEPARATE CHAINING)

Here keys are stored in linked lists attached to cells of a hash table. 3 operations – insertion, deletion and searching

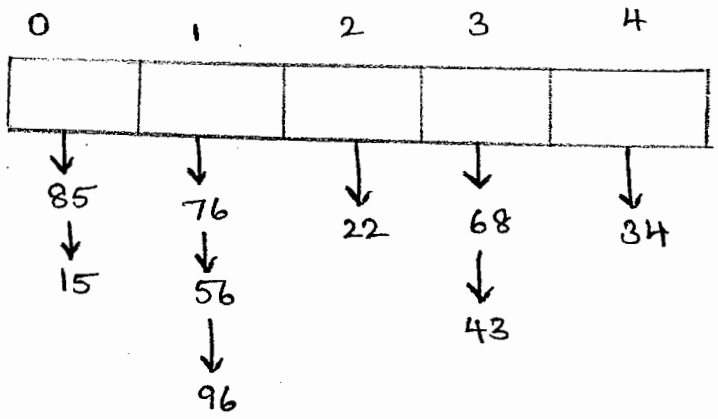
Consider the following numbers:

68, 76, 34, 56, 85, 15, 43, 96, 22

Let the hash function be mod 5.

Keys	68	76	34	56	85	15	43	96	22
Hash addresses	3	1	4	1	0	0	3	1	2

The hash table is as shown,



The efficiency of searching depends on the length of linked list. The efficiency also depends on the load factor α ($\alpha = n/m$ where 'n' is the number of keys and 'm' is the number of cells)

The number of pointers inspected for successful search is $1 + \frac{\alpha}{2}$ and for unsuccessful search is α. Insertion is normally done at the [illegible] the list.

CLOSED HASHING (OPEN ADDRESSING):

All keys are stored in the hash table without using linked lists.

The simplest one is called linear probing where the key is entered in the next empty cell whenever collision occurs. If the end of the hash table is reached the search is wrapped to the beginning of the table.

Ex: Hash function = mod 5

Keys	98	67	45	32	23	12	24	95	43
Hash Addresses	3	2	0	2	3	2	4	0	3

0	1	2	3	4
45	23	67	98	32

When we choose hash function as mod 5 we do not have sufficient space in the table to fill up all the entries. Hence choose the hash function appropriately.

Suppose now we choose the hash function as mod 9.

0	1	2	3	4	5	6	7	8
45	95	67	98	32	23	12	24	43

Here the average number of pointers inspected in successful search is $\frac{1}{2}(1 + 1/(1-\alpha))$ and unsuccessful search is $1/2\,(1 + 1/(1-\alpha)^2)$

If the load factor is too small then most of the space in the list / table is used. It is good to have load factor around 1.

A cluster in linear probing is a sequence of contiguously occupied cells.

Double hashing:

Here we use another hash function to determine the amount by which the increment has to be done whenever collision occurs. The performance is better than linear probing but does not work well when the table is almost full. In such cases the current table maybe scanned and all the keys can be relocated into a larger table. This is called as rehashing.

Suppose collision occurs at a location $l = h(K)$, use another hash function $s(K)$ to determine increment,

$(l + s(K)) \bmod m$, $(l + 2s(K)) \bmod m$ ----

$s(K)$: secondary hash function

m: primary hash function

INPUT ENHANCEMENT IN STRING MATCHING :

The brute force algorithm used for pattern matching has a time complexity of mn in the worst case and m in the best case where 'm' is the length of pattern string and 'n' is the length of text string.

HORSPOOL'S ALGORITHM :

Suppose we want to search for the pattern string BARBER in the string JIM SAW ME IN A BARBER SHOP ,

The number of shifts required to move the pattern string towards right is,

0	1	2	3	4	5
B	A	R	B	E	R

s[0]	1	2	...	A 65	B 66	...	E 69	...	R	... 12
6	6	6		~~6~~ 4	~~6~~ ~~5~~ 2		~~6~~ 1		~~6~~ 3	6

The string barber can be searched in the given text string as shown below :

P[0] = B is at a distance of 5 (5-0) from R
P[1] = A is at a distance of 4 (5-1) from R
P[2] = R is at a distance of 3 (5-2) from R
P[3] = B is at a distance of 2 (5-3) from R
P[4] = E is at a distance of 1 (5-4) from R

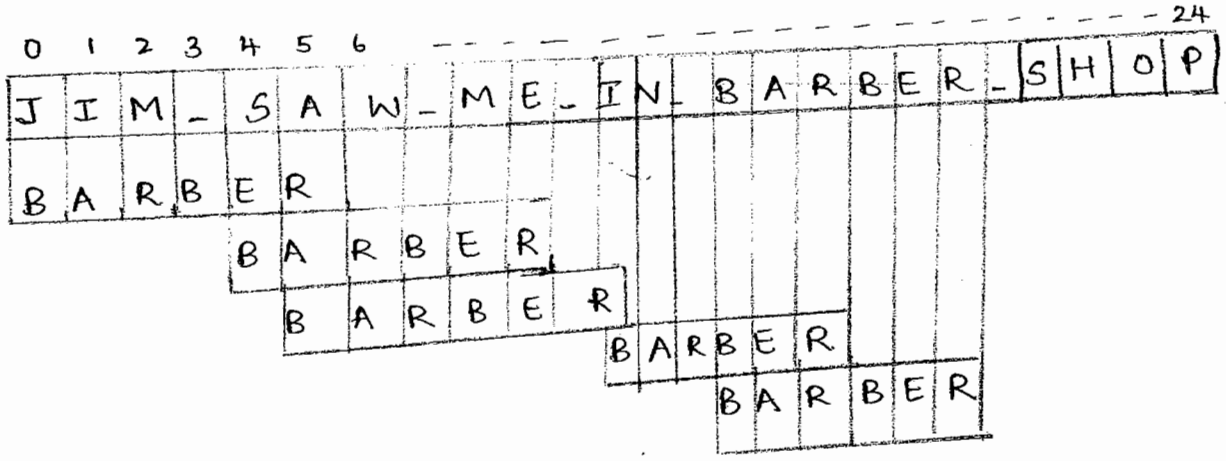
In general,

$P[i] = 5 - i$

$= 6 - 1 - i$

$= m - 1 - i$

where 'm' is the length of the pattern string



The shift table can be computed as shown below:

Algorithm shift_table (p, s)

// Purpose: To fill the shift table based on the pattern string

// Input: p → pattern string

// Output: s → table which contains the number of shifts required

```
m = length(p)
for i = 0 to 127
    S[i] = m
end for
for i = 0 to m-2
    S[p[i]] ← m-1-i
```

Algorithm horspool (p,t)

// Purpose: To check whether the pattern string is present in the text string

// Inputs: p → pattern to be searched

t → text string where searching takes plac

// Output: position of the pattern in the text string if found else -1

```
shift_table (p, s)
n = length(t)
m = length(p)
i = m-1
while i < n
        k=0
        while k<m && t[i-k] = p[m-1-k]
                k ← k+1
        end while
        if k=m  return i-(m-1)
            i = i + s[t[i]]
end while
return -1
```

Analysis:

Worst case:

The inputs are 'm' - length of pattern and
'n' - length of text

The basic operation is comparison of pattern and text $t[i-k] = p[m-1-k]$

$$T(n) = \sum_{i=m-1}^{n-1} \sum_{k=0}^{m-1} 1$$

$$= \sum_{i=m-1}^{n-1} m - k - 0 + k$$

$$= m \sum_{i=m-1}^{n-1} 1$$

$$= m \left[n - k - (m-1) + k \right]$$

$$= m (n - m + 1)$$

$$= mn - m^2 + 1$$

Neglecting constants and for large values of 'n'

$n >> m$

$$\therefore T(n) \approx O(mn)$$

Best case:

Occurs when the pattern is present at the beginning of the text string.

∴ Number of comparisons = Length of pattern

$$\therefore T(n) = \Omega(m)$$

Two approaches for space and time tradeoffs:

1. Input enhancement:

Ex: (1) Sorting by counting
(2) Distribution counting
(3) Horspool's algorithm for string matching

General idea:

Preprocess the problem's input as a whole or part and store the additional information obtained to accelerate problem solving.

2. Prestructuring:

Ex: (1) Hashing
(2) Indexing with B-trees

General idea:

Uses extra space to facilitate faster and flexible access to data.

Dynamic programming - uses space and time trade-off idea to record solutions to overlapping subproblems of a given problem in a table from which a solution to the given problem is obtained.

Example for Distribution Counting:

Consider the following elements:

12, 13, 10, 12, 10, 12, 11, 10, 13

Arrange these numbers, using sorting by distribution or linear sorting method:

0	1	2	3
10	11	12	13

The frequency of occurences are,

f

0	1	2	3
3	1	3	2

The cumulative frequencies are,

D

0	1	2	3
3	4	7	9

The array elements are,

a

0	1	2	3	4	5	6	7	8
12	13	10	12	10	12	11	10	13

0	1	2	3	
3	4	7	9	D
3	4	7	8	a[8] = 13
2	4	7	8	a[7] = 10
2	3	7	8	a[6] = 11
2	3	6	8	a[5] = 12
1	3	6	8	a[4] = 10
1	3	5	8	a[3] = 12
0	3	5	8	a[2] = 10
0	3	5	7	a[1] = 13
0	3	4	7	a[0] = 12

	0	1	2	3	4	5	6	7	8
0									13
1			10						
2				11					
3							12		
4		10							
5						12			
6	10								
7								13	
8					12				
9	10	10	10	11	12	12	12	13	13

↓

Sorted array

DECREASE AND CONQUER

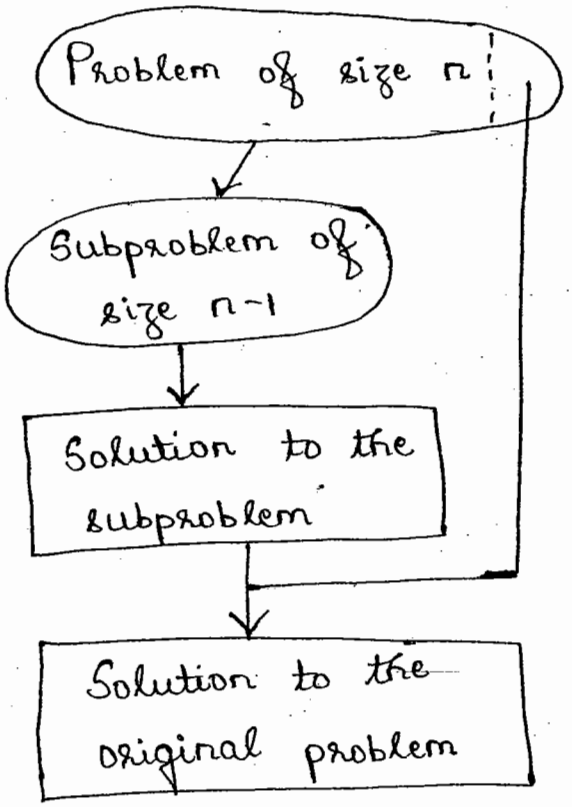
3 kinds are:

1. Decrease by a constant:

The size of an instance is reduced by the same constant on each iteration. Usually this constant is 1.

To compute $f(n) = a^n$ we can use top down approach (recursion) as follows:

$$f(n) = \begin{cases} f(n-1) \cdot a & \text{if } n > 1 \\ a & \text{if } n = 1 \end{cases}$$



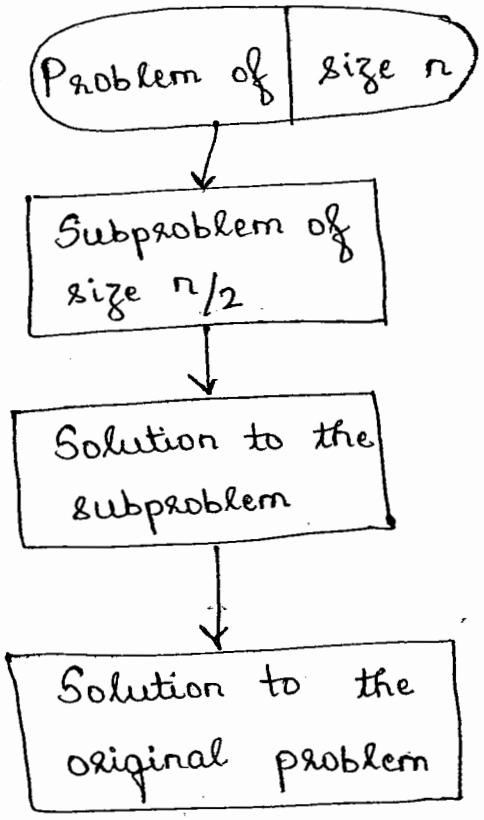
2. Decrease by a constant factor:

Reduce the problem's instance by the same constant factor on each iteration of the algorithm.

To compute $f(n) = a^n$ we can use $a^n = (a^{n/2})^2$. But this does not work when 'n' is odd. Hence we can compute as follows:

$$a^n = \begin{cases} (a^{n/2})^2 & n \text{ is even} \\ (a^{(n-1)/2})^2 \cdot a & n \text{ is odd} \\ a & n = 1 \end{cases}$$

The efficiency of the above algorithm must be $O(\log n)$ because on each iteration the problem size is reduced by half.



3. Variable size decrease:

The reduction in size varies from one iteration of an algorithm to another.

Ex: Euclid's algorithm for computing GCD

GCD (10, 6)

1. m = 10 n = 6 r = 4
2. m = 6 n = 4 r = 2
3. m = 4 n = 2 r = 0

Divisor is the GCD when remainder becomes 0.

∴ GCD (10, 6) = 2

Insertion Sort:

3 ways : 1. Scan the sorted subarray from left to right until the first element is greater than or equal to A[n-1] and insert it before the right element.

2. Scan the sorted subarray from right to left until the first element is smaller than or equal to A[n-1] and insert it right after the element.

3. Use binary search to find an appropriate position for A[n-1] in the sorted portion of the array. This is called as binary insertion sort.

Algorithm Insertion_Sort (A[ ], n)

// Purpose: To sort the given array

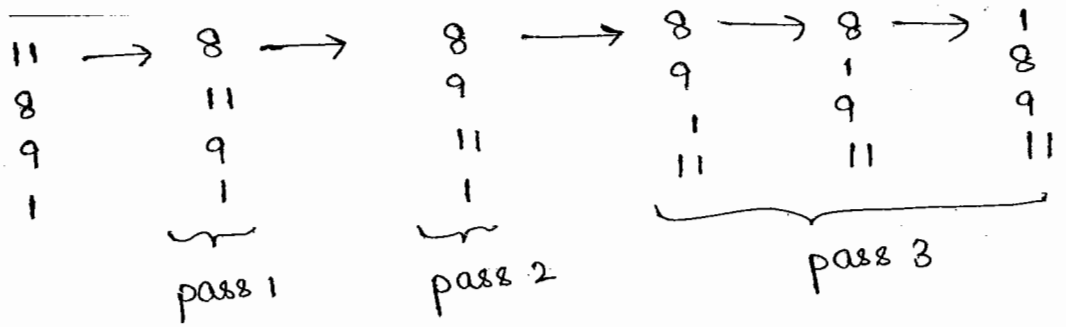
// Input: An array A[ ] of unsorted elements

// Output: Sorted array [ ]

```
for i = 1 to n-1
    item = A[i]
    j = i-1
```

```
while j>=0 && A[j]> item
        A[j+1] ← A[j]
        j ← j-1
end while
A[j+1] ← item
end for
```

Ex:



The worst time complexity is :

$$O(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{n-1} 1$$

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

$$= \sum_{i=1}^{n-1} n$$

$$= n \sum_{i=1}^{n-1} 1$$

$$= n[(n-1) - 1 + 1]$$

$$= n(n-1)$$

$$= n^2 - n$$

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

The average case time complexity is :

$$\theta(n) = n^2$$

The best case time complexity is

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

$$\therefore \Omega(n) = n$$

The best case occurs if the while loop is executed only once for each iteration i.e the array is already sorted.

For 'n' elements, (n-1) passes are required to sort.

Depth First Search (DFS):

Start visiting the vertices of a graph in an order such that on each iteration go to the adjacent unvisited vertex until it becomes a dead vertex. Once a vertex becomes dead go back to the previous vertex and continue. The algorithm ends when it goes back to the starting vertex. If there are still any unvisited vertices left out then start the traversal again from one of these vertices.

DFS uses a stack to push the vertex on to stack when reached for the first time and to pop a vertex when it becomes a dead vertex.

Whenever a new unvisited vertex is reached for the first time, the edge which connects the source vertex to this vertex is called tree edge. When an already visited vertex is reached it is called back edge.

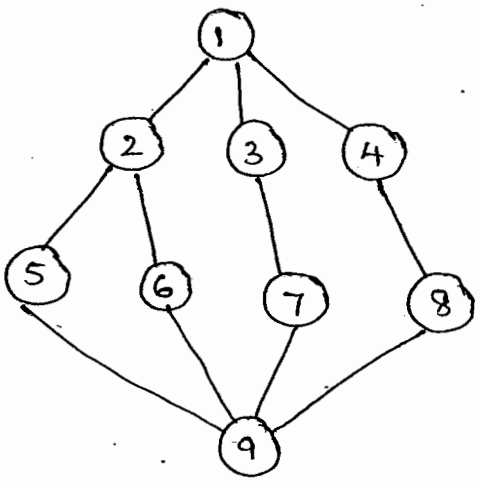
Algorithm DFS(G)

// Purpose: To traverse the vertices of a graph

// Input: Graph $G=(V,E)$

// Output: Graph G with DFS traversal

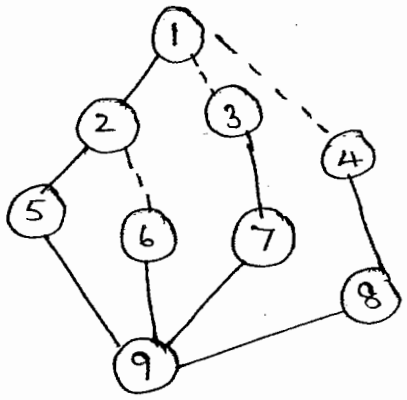
```
mark each vertex V as unvisited i.e V=0
for each vertex V
    if V is marked 0
        dfs(v)
    end if
end for

dfs(v)
Print v
for each vertex w adjacent to V
    if w is marked 0
        dfs(w)
    end if
end for
```



The DFS traversal for the above graph is,

125967384

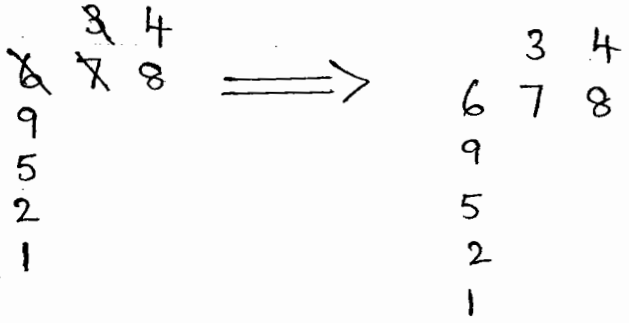
DFS uses stack (LIFO structure). Hence there will be 2 ordering of vertices.



——→ tree edge

- - - - → back edge

The stack structure is as shown,



The time complexity,

Adjacency matrix representation : $O(|V|^2)$

Adjacency linked list representation : $O(|V|+|E|)$

V → number of vertices

E → number of edges

Applications of DFS :

1. Checking connectivity of a graph
2. Checking acyclicity of a graph
3. Finding articulation points in a graph

(A vertex of a connected graph is said to be an articulation point if its removal along with the edges divides the graph into disjoint subgraphs)

Breadth First Search (BFS):

Start from a particular vertex and visit all the adjacent vertices. Repeat this in a concentric manner. If there are unvisited vertices left then restart the procedure for those vertices.

We use a queue structure. Initially put the start vertex onto the queue. On each iteration all unvisited vertices from the start vertex which are adjacent are visited and inserted to the queue, then remove the vertex (from which we started).

Whenever a new unvisited vertex is reached for the first time, then it is called as tree edge.

When an already visited vertex is reached, then such an edge is called as cross edge.

Algorithm BFS(G)

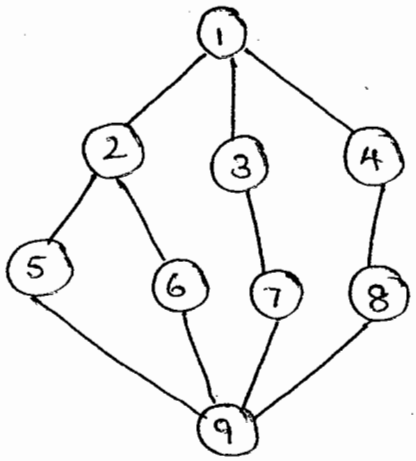
// Purpose: To traverse the vertices of a graph in order

// Input: Graph G = (V, E)

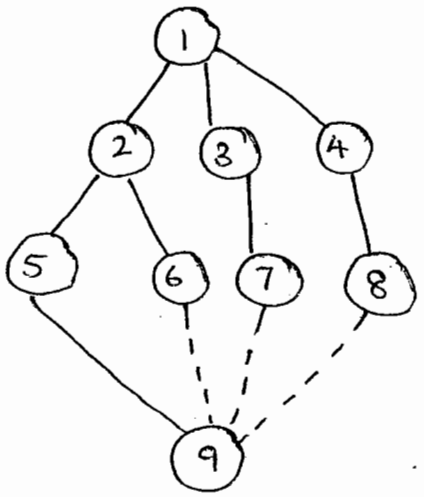
// Output: Graph G along with it's BFS traversal

```
mark each vertex V as unvisited i.e V=0
for each vertex V
        if V is marked 0
            bfs(v)
        end if
```

```
bfs(v)
while queue not empty
        print the vertex which is visited.
        for each vertex w adjacent to v
                if w is marked 0
                        mark w as 1
                        add w to queue
                end if
        end for
        remove the front vertex from queue
end while
```



The BFS traversal for the above graph is,

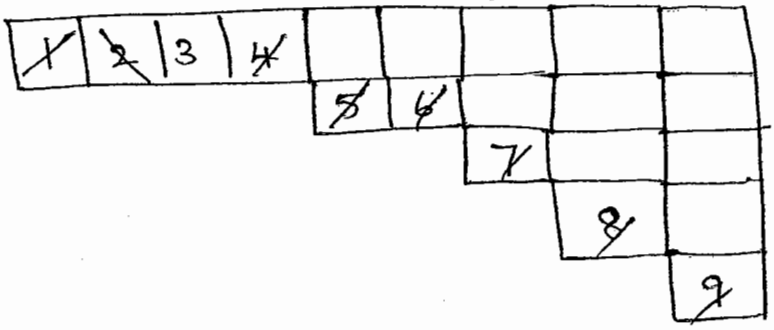


——→ tree edge

----→ cross edge

1 2 3 4 5 6 7 8 9

The queue structure is as shown,



Time efficiency :

Adjacency matrix representation : $O(|V|^2)$

Adjacency linked list representation : $O(|V|+|E|)$

Since BFS uses queue (FIFO structure) there will be a single ordering of vertices.

Applications of BFS :

1. Checking connectivity of a graph
2. Checking acyclicity of a graph
3. To find minimum edge paths in a graph

Topological Sorting:

To perform topological sorting on a graph it must not have a directed cycle.

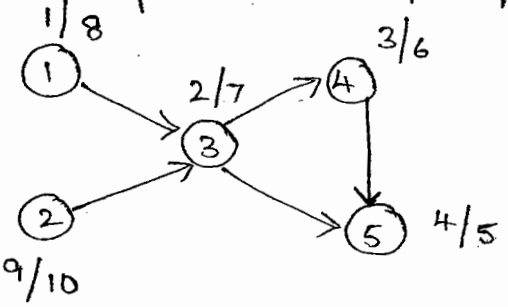
Two methods :

1. Using DFS :

Perform DFS traversal of a graph, Once a particular vertex becomes a dead vertex store it an array. The decreasing order in which the vertices have

become dead gives the topological order.

Ex:



21345 is the topological order

d[v] / f[v]

where d[v] → DFS traversal

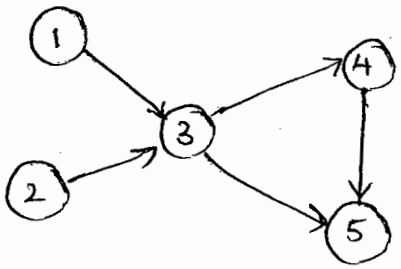
f[v] → dead vertex ordering

Hence arrange the numbers stored in f[v] in decreasing order to obtain the topological order.

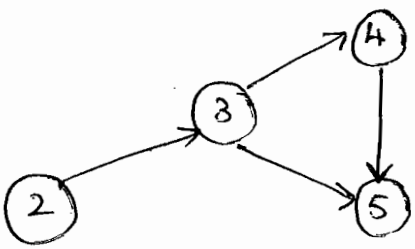
2. Source removal method:

Identify the vertex which has no incoming edges. Then delete it along with all the edges outgoing from it.

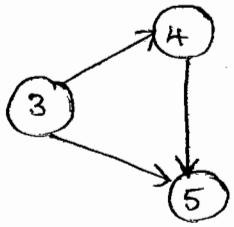
Ex:



Delete vertex 1



Delete vertex 2



Delete vertex 3

Delete vertex 4

Delete vertex 5

The order in which the vertices are deleted gives the topological order.

Generating permutations:

Two methods:

1. Bottom up minimal change algorithm:

Each permutation can be obtained from its immediate predecessor by exchanging two elements in it.

Step 1: Start

1

Step 2: Insert 2 into 1 R→L

12 21

Step 3 : Insert 3 into 21 R → L

123 132 312

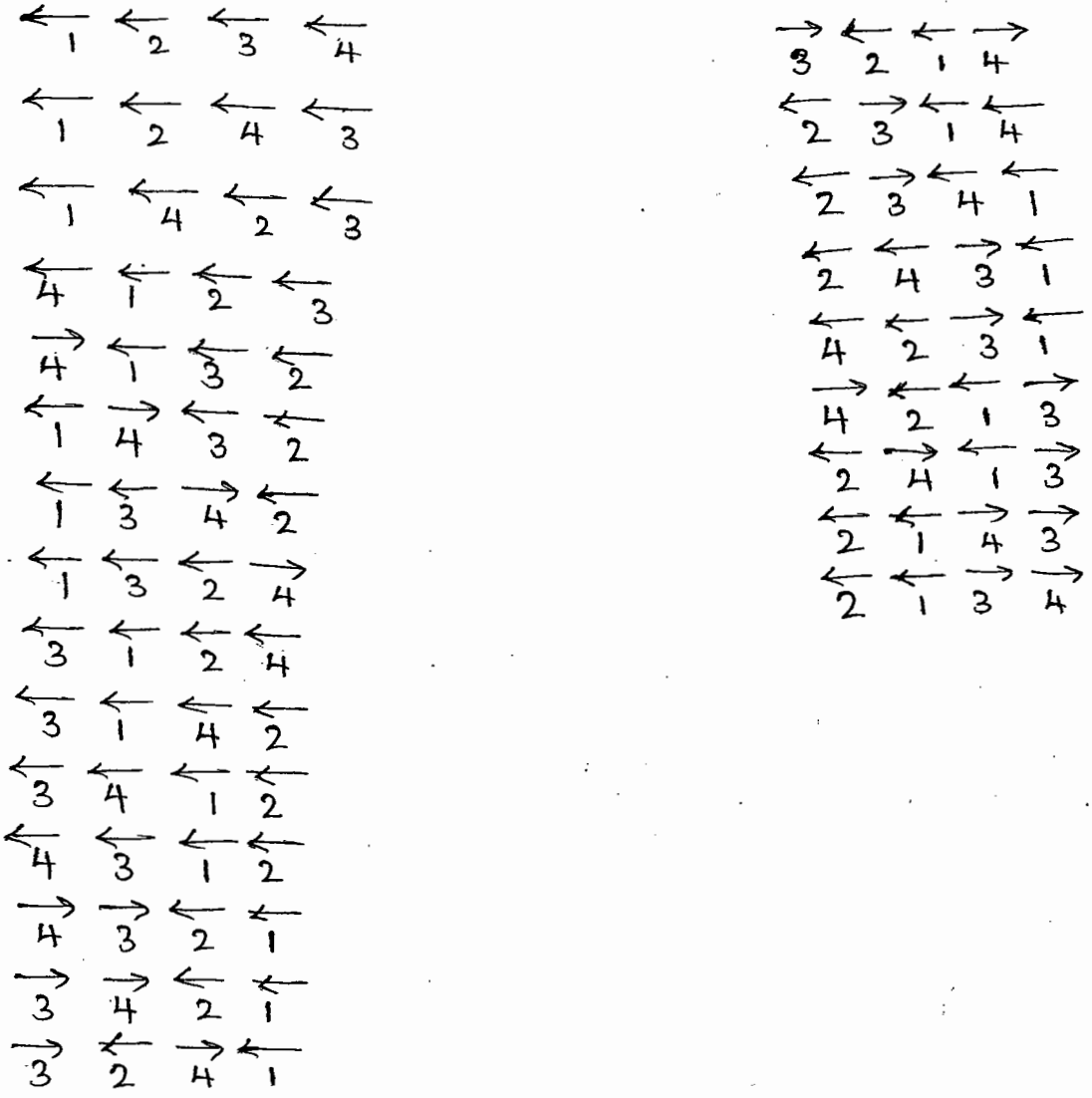
Insert 3 into 21 L → R

321 231 213

2. Johnson Trotter algorithm :

In the above method to find permutation of n, the permutations for all (n-1) are required starting from 1. This disadvantage can be overcome by using Johnson Trotter algorithm.

To find permutation of 4 :



Algorithm Johnson Trotter (n)

// Purpose : To find permutation of n

// Input : A positive integer n

// Output : List of all permutations of n

Initialize the first permutation with

$\overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{3}\ \cdots\cdots\ \overleftarrow{n}$

while the last permutation has a mobile component

find largest mobile component k

exchange the mobile component k with its immediate adjacent integer pointed by mobile

change the direction of all components which is larger than k

end while

If an element k points to a smaller number adjacent to it, then the element k is said to be mobile.

We can also find the permutations in lexicographic order. (The order in which representation is made in dictionary)

To find permutation of 3 :

123	132	213
231	312	321

Generation of subset:

For a set $A = \{a_1, a_2, ----an\}$ we have 2^n subsets. For $n = 3$,

$n=0$ $\{\}$

$n=1$ $\{\}, \{a_1\}, \{a_2\}, \{a_3\}$

$n=2$ $\{\}, \{a_1\}, \{a_2\}, \{a_3\}, \{a_1, a_2\}, \{a_1, a_3\}, \{a_2, a_3\}$

$n=3$ $\{\}, \{a_1\}, \{a_2\}, \{a_3\}, \{a_1, a_2\}, \{a_1, a_3\}, \{a_2, a_3\}, \{a_1, a_2, a_3\}$

To generate the subsets, we use a bit string bi,

$b_i = 1$ if a_i belongs to the subset

$b_i = 0$ if a_i does not belong to the subset

Bit string 000 corresponds to $\{\}$

$111 \longrightarrow \{a_1, a_2, a_3\}$

Hence, the correspondence is as follows

$000 \longrightarrow \{\}$

$001 \longrightarrow \{a_3\}$

$010 \longrightarrow \{a_2\}$

$011 \longrightarrow \{a_2, a_3\}$

$100 \longrightarrow \{a_1\}$

$101 \longrightarrow \{a_1, a_3\}$

$110 \longrightarrow \{a_1, a_2\}$

$111 \longrightarrow \{a_1, a_2, a_3\}$

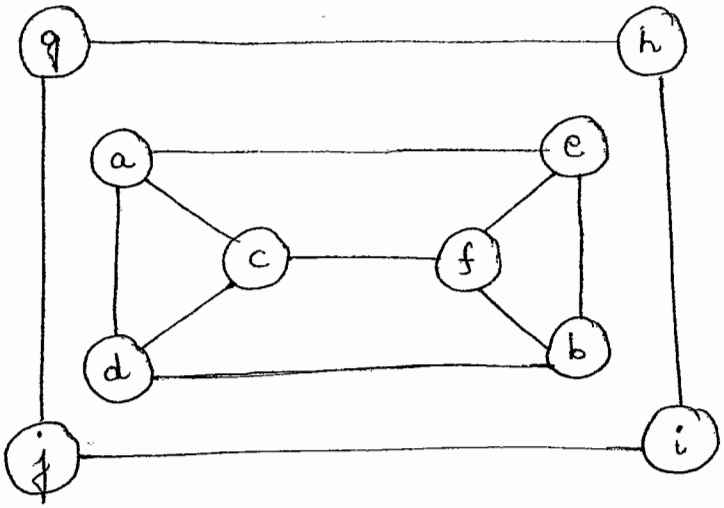
Suppose we want to find the subsets in such a way that every subset differs from its immediate predecessor by either addition or deletion (not both) of a single element.

Ex: For n=3,

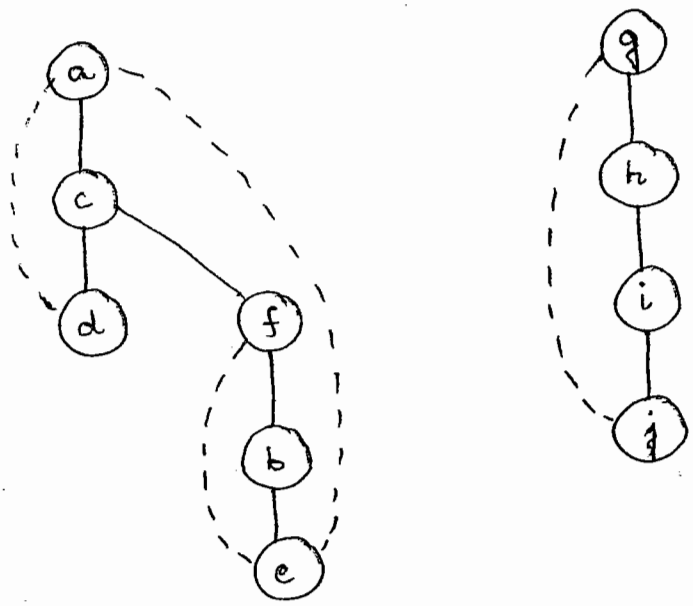
000
001
011
010
110
111
101
100

Such a sequence of bit strings is called binary reflected gray code.

Given graph:



(i) DFS:



——→ Tree edges

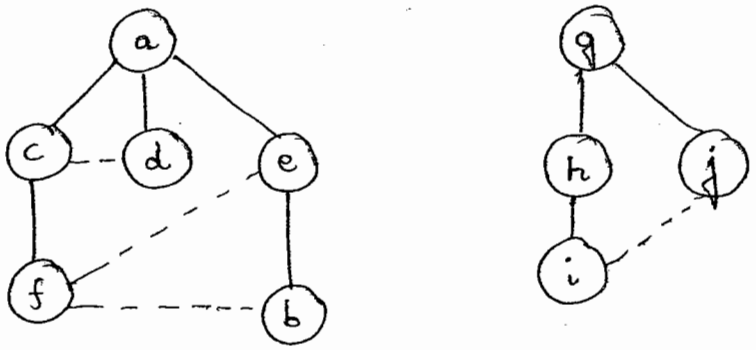
- - - → Back edges

DFS traversal: a c d f b e g h i j

Stack traversal:

	$e_{6,2}$	
	$b_{5,3}$	
$d_{3,1}$	$f_{4,4}$	$j_{10,7}$
$c_{2,5}$		$i_{9,8}$
$a_{1,6}$		$h_{8,9}$
		$g_{7,10}$

(2) BFS:



——→ Tree edges
- - - → Cross edges

BFS traversal:

acdefbghjji

Queue traversal:

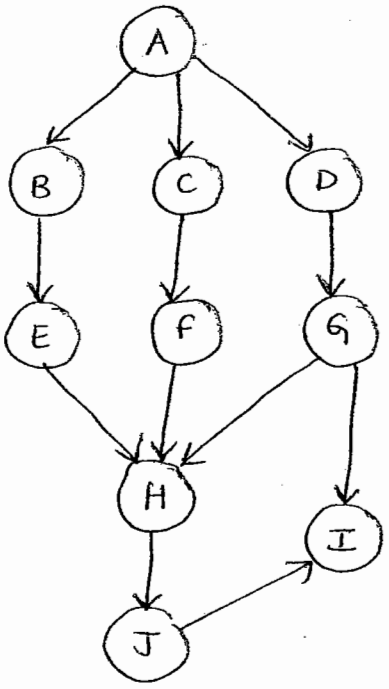
$a_1\ c_2\ d_3\ e_4\ f_5\ b_6\ g_7\ h_8\ j_9\ i_{10}$

In queue traversal, subscript denotes the order in which vertices are inserted into the queue.

In stack traversal, first subscript denotes the order in which vertices are pushed onto the stack and second subscript denotes the order in which vertices are popped from the stack.

Obtain DFS and BFS traversal for the following:

(1)



DFS:

Traversal: A B E H J I C F D G

Stack Traversal:

I 6,1			
J 5,2			
H 4,3			
E 3,4	F 8,6	G 10,8	
B 2,5	C 7,7	D 9,9	
A 1,10			

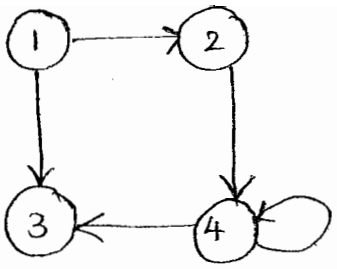
BFS:

Traversal: A B C D E F G H I J

Queue Traversal:

A_1 B_2 C_3 D_4 E_5 F_6 G_7 H_8 I_9 J_{10}

(2)



DFS :

Traversal : 1243

Stack traversal :

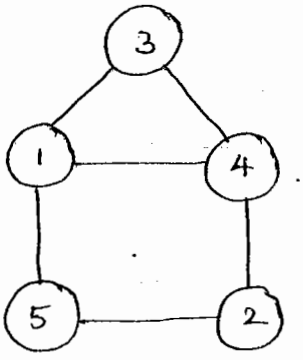
3	4, 1
4	3, 2
2	2, 3
1	1, 4

BFS :

Traversal : 1 2 3 4

Queue traversal : 1_1 2_2 3_3 4_4

(3)



DFS :

Traversal : 31524

4	5, 1
2	4, 2
5	3, 3
1	2, 4

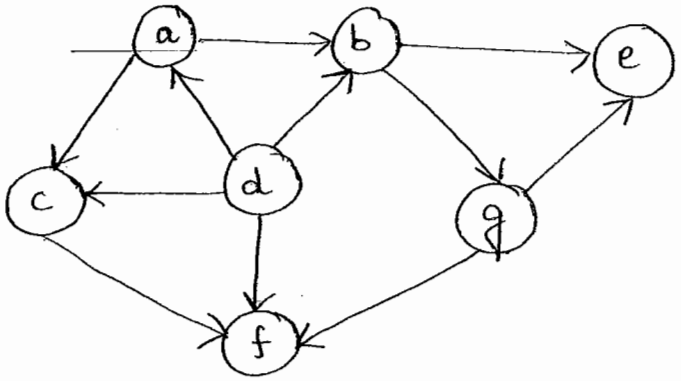
BFS :

Traversal : 3 1 4 5 2

Queue traversal :

3_1 1_2 4_3 5_4 2_5

(4)



DFS :

Traversal : a b e g f c d

Stack traversal :

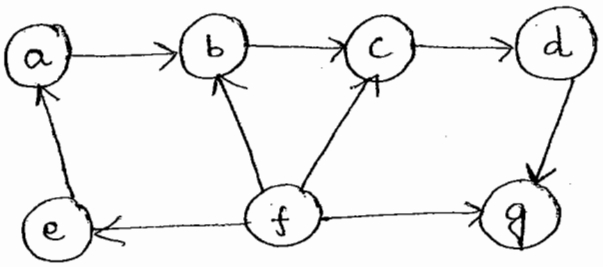
e 3,1	f 5,2		
b 2,4	g 4,3		
a 1,6		c 6,5	d 7,7

BFS :

Traversal : d a b c f g e

d_1 a_2 b_3 c_4 f_5 g_6 e_7

(5)



DFS :

Traversal : a b c d g f e

Stack traversal :

g 5, 1
d 4, 2
c 3, 3
b 2, 4
a 1, 5
e 7, 6
f 6, 7

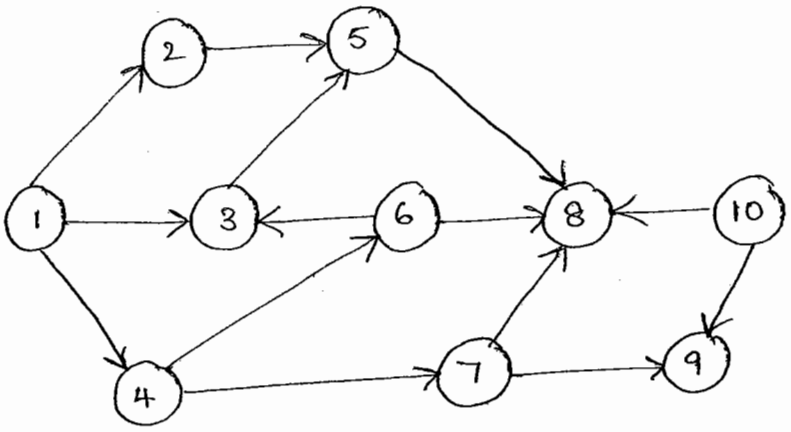
BFS :

Traversal : f b c e g d a

f_1 b_2 c_3 e_4 g_5 d_6 a_7

Queue traversal

(6)



DFS :

Traversal : 1 2 5 8 3 4 7 9 6 10

Stack traversal:

8 4,1
5 3,2
2 2,3
1 1,9 3 5,4 9 8,5 7 7,6 4 6,8 6 9,7 10 10,10

BFS:

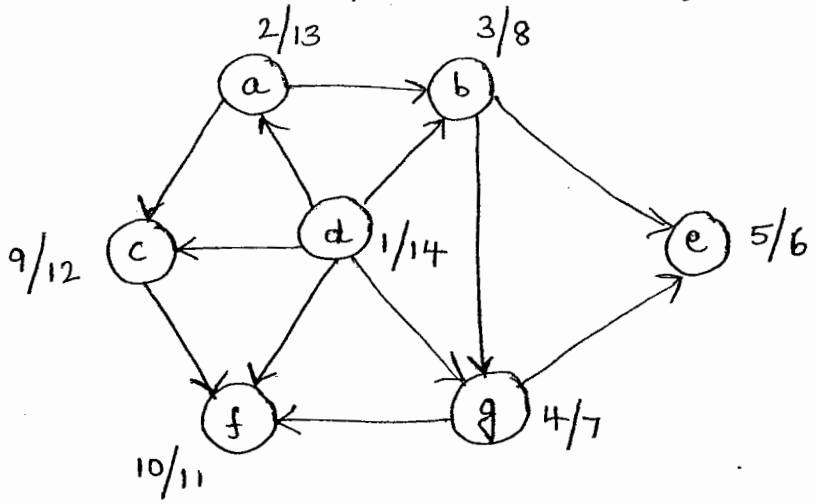
Traversal: 1 2 3 4 5 6 7 8 9 10

Queue traversal:

1, 22 33 44 55 66 77 88 99 1010

Obtain the topological sorting for the following:

(1)

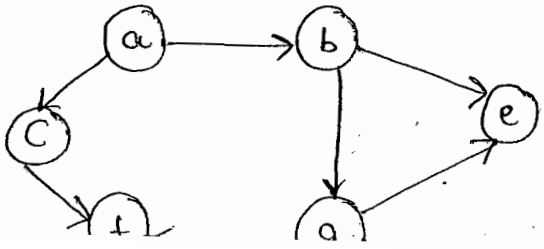


1. Using DFS method,

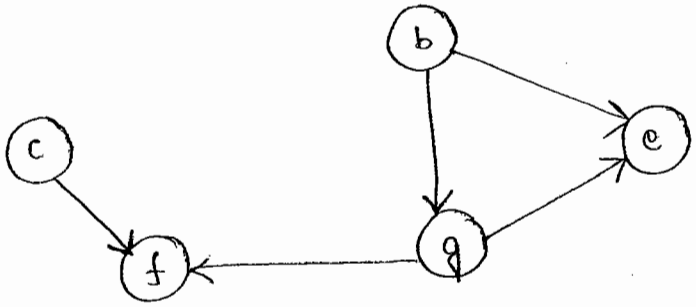
d a c f b g e

2. Using source removal method,

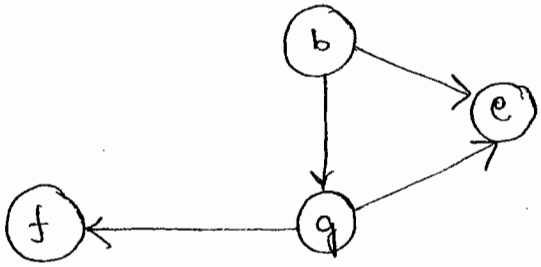
Delete 'd'



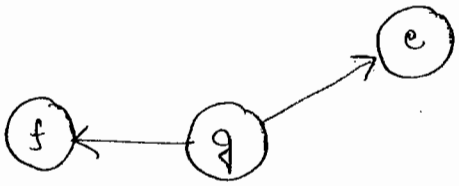
Delete 'a'



Delete 'c'



Delete 'b'



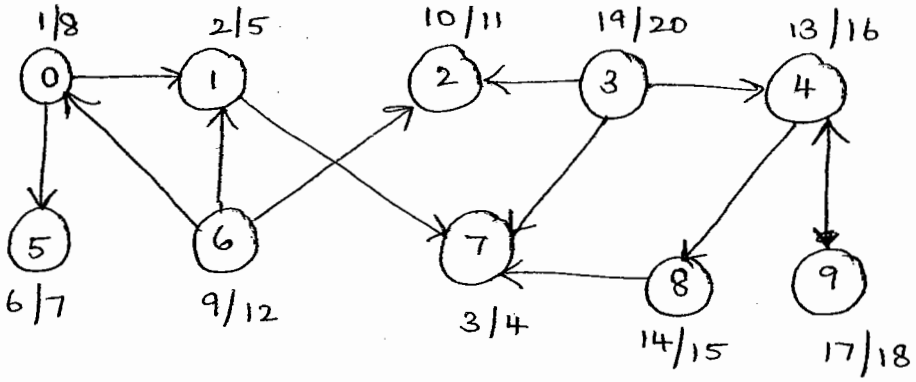
Delete 'g'

Delete 'f'

Delete 'e'

Topological order : dacbgfe

(2)

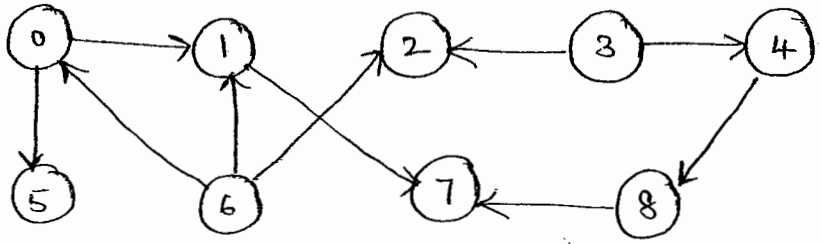


1. Using DFS method,

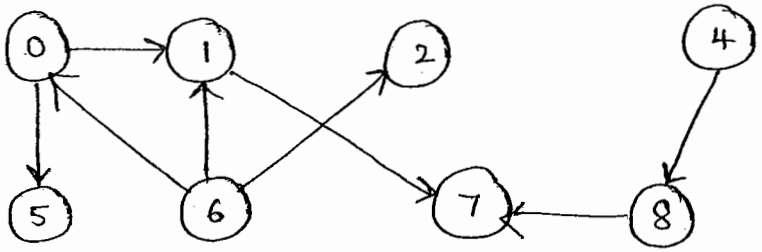
3 9 4 8 6 2 0 5 1 7

2. Using source removal method,

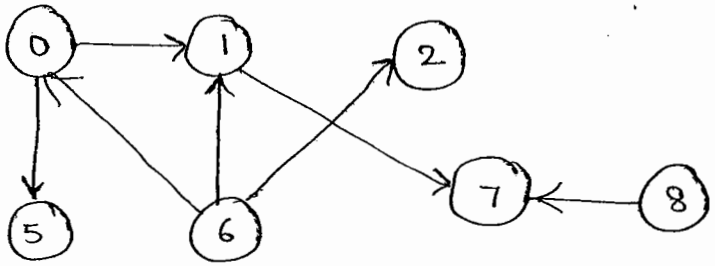
Delete 9



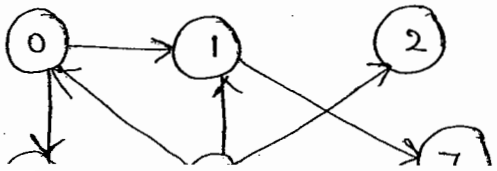
Delete 3



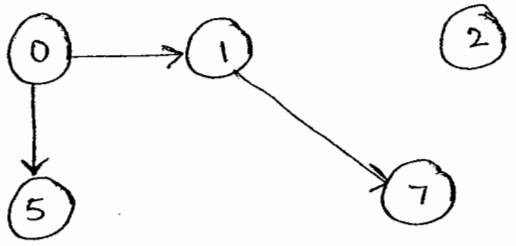
Delete 4



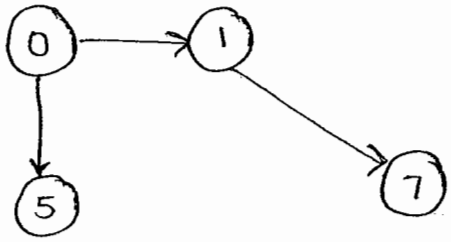
Delete 8



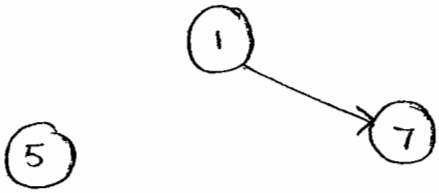
Delete 6



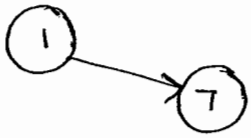
Delete 2



Delete 0



Delete 5

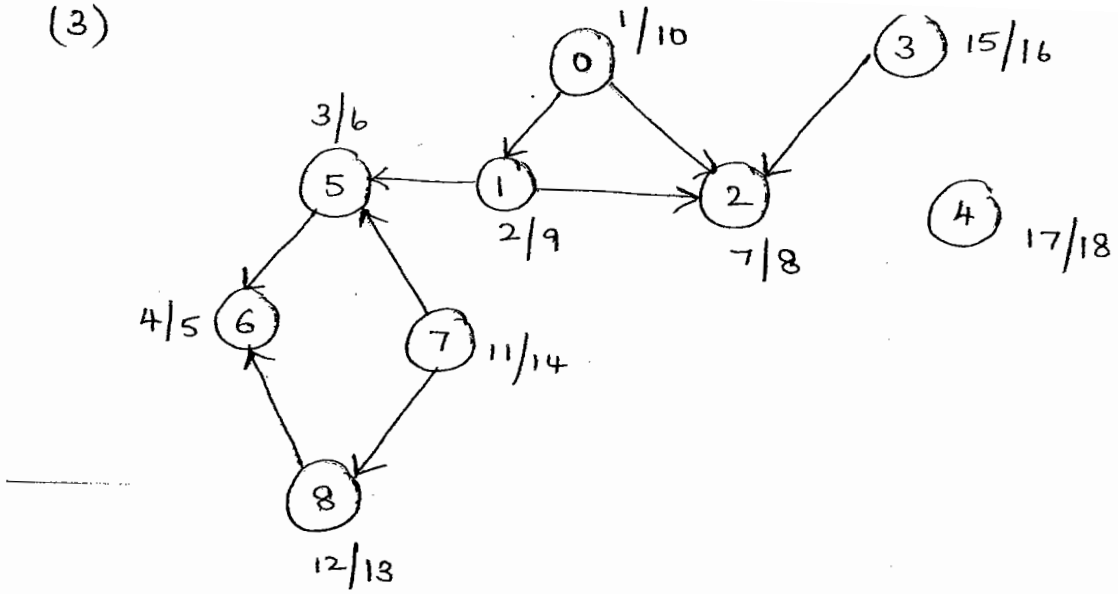


Delete 1

Delete 7

Topological order: 9 3 4 8 6 2 0 5 1 7

(3)

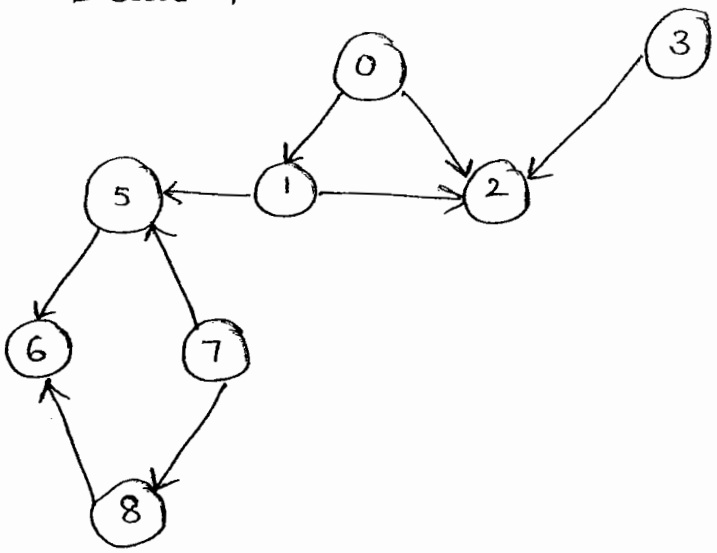


1. Using DFS method,

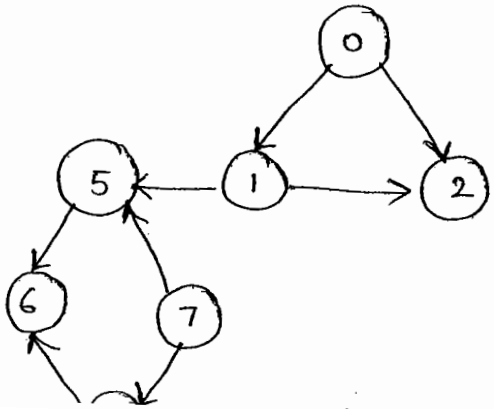
4 3 7 8 0 1 2 5 6

2. Using source removal method,

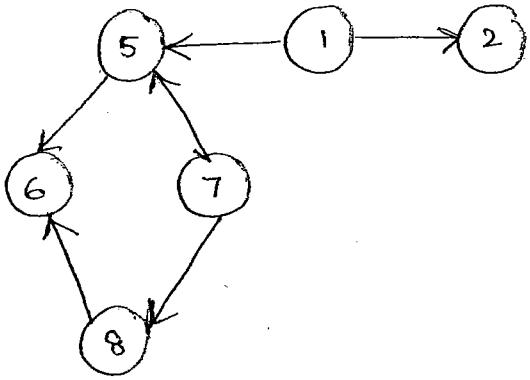
Delete 4



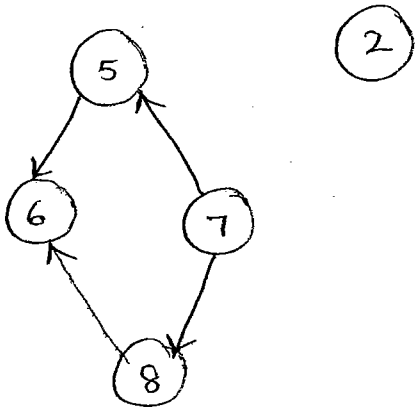
Delete 3



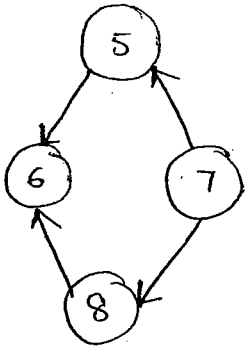
Delete 0



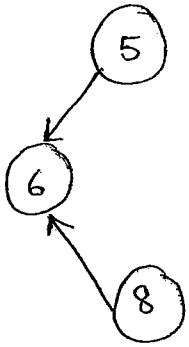
Delete 1



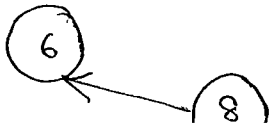
Delete 2



Delete 7



Delete 5



Delete 8

(6)

Delete 6

Topological order: 4 3 0 1 2 7 5 8 6

DYNAMIC PROGRAMMING

Used for solving problems with overlapping subproblems. These subproblems arise from a recurrence relating to a solution to a given problem with solutions to it's smaller subproblems of the same type.
Here, each subproblem is solved only once and results are recorded in a table and these values are used to obtain the solution.

(1) Binomial Co-efficient:

Denoted by $C(n,K)$ or $\binom{n}{K}$ is the number of combinations of K elements from an n-element set.
$(0 \leq K \leq n)$

$$C(n,K) = C(n-1, K-1) + C(n-1, K) \text{ for } n \geq K \geq 0$$
$$C(n,0) \text{ and } C(n,n) = 1$$

i.e

$$C(n,K) = \begin{cases} 1 & \text{if } n=K \text{ or } K=0 \\ C(n-1,K-1) + C(n-1,K) & \text{if } n \geq K \geq 0 \end{cases}$$

Ex: (1) Computation of $6C_3$

Construct a table with $n+1$ rows and $K+1$ colum

Here, $n=6$
$K=3$

$n+1 = 6+1 = 7$ rows (0 to 6)
$K+1 = 3+1 = 4$ columns (0 to 3)

$c[0,0] = \emptyset, c[0,1] = c[0,2] = c[0,3] =$ can't be computed

$c[1,0] = c[2,0] = c[3,0] = c[4,0] = c[5,0] = c[6,0] = \emptyset$

$c[1,1] = c[2,2] = c[3,3] = 1$

$c[1,2] = c[1,3] =$ can't be computed

$c[2,3] =$ can't be computed

$c[2,1] = c[1,0] + c[1,1]$

$= 1+1 = 2$

$c[3,1] = c[2,0] + c[2,1]$

$= 1+2 = 3$

$c[3,2] = c[2,1] + c[2,2]$

$= 2+1 = 3$

$c[4,1] = c[3,0] + c[3,1]$

$= 1+3 = 4$

$c[4,2] = c[3,1] + c[3,2]$

$= 3+3 = 6$

$c[4,3] = c[3,2] + c[3,3]$

$= 3+1 = 4$

$c[5,1] = c[4,0] + c[4,1]$

$= 1+4 = 5$

$c[5,2] = c[4,1] + c[4,2]$

$= 4+6 = 10$

$c[5,3] = c[4,2] + c[4,3]$

$= 6+4 = 10$

$c[6,1] = c[5,0] + c[5,1]$

$= 1+5 = 6$

$c[6,2] = c[5,1] + c[5,2]$

$= 5+10 = 15$

$c[6,3] = c[5,2] + c[5,3]$

$= 10+10 = 20$

(Values of j) →

n ↓ (Values of i)	0	1	2	3
0	1	X	X	X
1	1	1	X	X
2	1	2	1	X
3	1	3	3	1
4	1	4	6	4
5	1	5	10	10
6	1	6	15	**20**

Bottom-up approach

(2) Compute the value of $^7C_4$.

	0	1	2	3	4
0	1	X	X	X	X
1	1	1	X	X	X
2	1	2	1	X	X
3	1	3	3	1	X
4	1	4	6	4	1
5	1	5	10	10	5
6	1	6	15	20	15
7	1	7	21	35	**35**

Algorithm Binomial_coefficient(n, K)

// Purpose : To compute the value of nCK

// Inputs : n and K are positive integers

$$n \geq K \geq 0$$

// Output : Value of nCK or C(n, K)

```
for i ← 0 to n
    for j ← 0 to min(i, K)
        if (i = j or j = 0)
            c[i, j] ← 1
        else
            c[i, j] ← c[i-1, j-1] + c[i-1, j]
        end if
    end for
end for
```

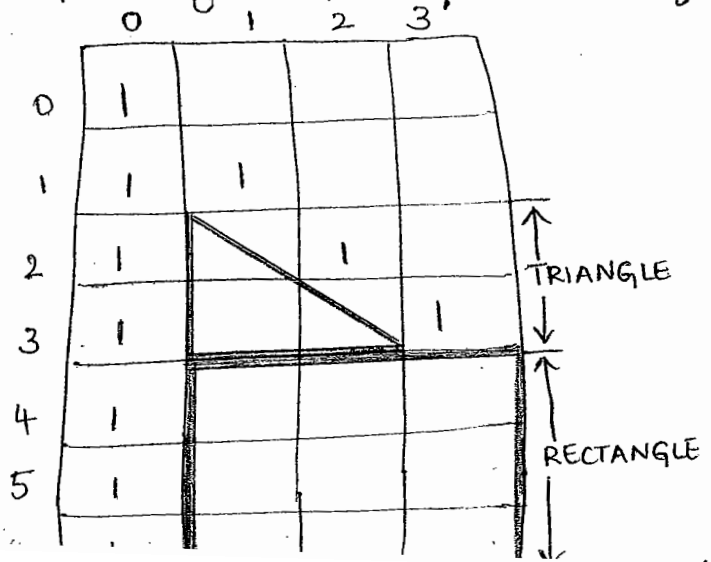
Analysis :

The parameters to be considered are 'n' and 'K'

The basic operation is,

$$c[i, j] = c[i-1, j-1] + c[i-1, j]$$

For the example of computing value of 6C3,



$T(n,K)$ = Time efficiency to compute values for triangular portion + Time efficiency to compute values for rectangular portion

i.e $T(n,K) = T_1(n,K) + T_2(n,K)$ ——— (1)

To compute $T_1(n,K)$:

$i=2$ $j=1$ to 1

$i=3$ $j=1$ to 2

for $i \leftarrow 2$ to K

  for $j \leftarrow 1$ to $i-1$

$$T_1(n,K) = \sum_{i=2}^{K} \sum_{j=1}^{i-1} 1$$

$$= \sum_{i=2}^{K} i-1-1+1$$

$$= 1+2+ \cdots (k-1)$$

$$= \frac{(K-1)\cdot K}{2} \approx K^2$$

$$\boxed{T_1(n,K) \approx O(K^2)} \quad \text{——— (2)}$$

To compute $T_2(n,K)$:

$i=4$ $j=1$ to 3

$i=5$ $j=1$ to 3

$i=6$ $j=1$ to 3

for $i \leftarrow K+1$ to n

  for $j \leftarrow 1$ to K

$$T_2(n,K) = \sum_{}^{n} \sum_{}^{K} 1$$

$$T_2(n,K) = \sum_{i=K+1}^{n} K-X+Y$$

$$= K \sum_{i=K+1}^{n} 1$$

$$= K\left[n-(K+1)+1\right]$$

$$= K(n-K-X+Y)$$

$$= nK-K^2$$

$$\boxed{T_2(n,K) = O(nK-K^2)} \quad \text{——— (3)}$$

Substituting (2) and (3) in (1),

$$T(n,K) = K^2 + nK - K^2 = nK$$

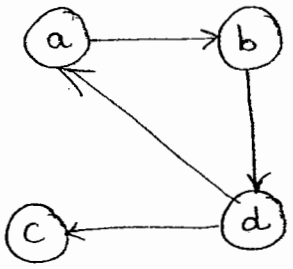
$$\boxed{\therefore T(n,K) \approx O(nK)}$$

(2) Warshall's Algorithm:

Adjacency matrix $A = \{a_{ij}\}$ of a directed graph is a boolean matrix that has 1 in it's ith row and jth column if and only if there is a directed edge from ith vertex to the jth vertex.

Transitive closure or path matrix of a directed graph with 'n' vertices is a n-by-n boolean matrix $T = \{t_{ij}\}$, in which the element in ith row and jth column is 1 if there is a directed path (nontrivial directed path) of positive length from ith vertex to jth vertex else $t_{ij} = 0$.

Ex: (1) Given graph:



Solution:

Adjacency Matrix A = $R^{(0)}$

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

Step 1:

Consider entries which are 1 in first column and first row of the adjacency matrix.

$(d,a) = 1$ and $(a,b) = 1 \Longrightarrow (d,b) = 1$

The resulting matrix is,

$A^a = R^{(1)}$

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	1	1	0

Step 2:

Consider entries which are 1 in second column and second row of the resulting matrix of step 1.

$(a,b) = 1$ and $(b,d) = 1 \Longrightarrow (a,d) = 1$

$(d,b) = 1$ and $(b,d) = 1 \Longrightarrow (d,d) = 1$

The resulting matrix is,

A^b / $R^{(2)}$ =

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

Step 3:

Consider the entries which are 1 in third column and third row of the resulting matrix of step 2.

Since no entries are 1 in the third row, the resulting matrix is,

A^c / $R^{(3)}$ =

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

Step 4:

Consider the entries which are 1 in fourth column and fourth row of the resulting matrix of step 3.

$(a,d)=1$ and $(d,a)=1 \Longrightarrow (a,a)=1$

$(b,d)=1$ and $(d,a)=1 \Longrightarrow (b,a)=1$

$(a,d)=1$ and $(d,b)=1 \Longrightarrow (a,b)=1$

$(b,d)=1$ and $(d,b)=1 \Longrightarrow (b,b)=1$

$(a,d)=1$ and $(d,c)=1 \Longrightarrow (a,c)=1$

$(b,d)=1$ and $(d,c)=1 \Longrightarrow (b,c)=1$

$(a,d)=1$ and $(d,d)=1 \Longrightarrow (a,d)=1$

$(b,d)=1$ and $(d,d)=1 \Longrightarrow (b,d)=1$

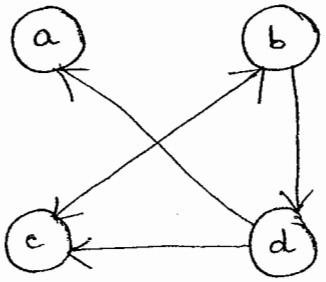
} These values are already 1

NOTE: Do not separately consider intersecting cells

$$A^d = R^{(4)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

$\longrightarrow$ Path matrix or Transitive Closure

(2) Given graph:



Solution:

Adjacency matrix $A = R^{(0)}$

$$\begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Step 1:

No entries are 1 in first row.

The resulting matrix is,

$$A^a = R^{(1)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Step 2:

$(c,b) = 1$ and $(b,c) = 1 \Longrightarrow (c,c) = 1$

$(c,b) = 1$ and $(b,d) = 1 \Longrightarrow (c,d) = 1$

The resulting matrix is,

$$A^b = R^{(2)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Step 3:

$(b,c)=1$ and $(c,b)=1 \Rightarrow (b,b)=1$

$(b,c)=1$ and $(c,c)=1 \Rightarrow (b,c)=1$

$(b,c)=1$ and $(c,d)=1 \Rightarrow (b,d)=1$

$(d,c)=1$ and $(c,b)=1 \Rightarrow (d,b)=1$

$(d,c)=1$ and $(c,c)=1 \Rightarrow (d,c)=1$

$(d,c)=1$ and $(c,d)=1 \Rightarrow (d,d)=1$

The resulting matrix is,

$$A^c = R^{(3)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Step 4:

$(b,d)=1$ and $(d,a)=1 \Rightarrow (b,a)=1$

$(b,d)=1$ and $(d,b)=1 \Rightarrow (b,b)=1$

$(b,d)=1$ and $(d,c)=1 \Rightarrow (b,c)=1$

$(b,d)=1$ and $(d,d)=1 \Rightarrow (b,d)=1$

$(c,d)=1$ and $(d,a)=1 \Rightarrow (c,a)=1$

$(c,d)=1$ and $(d,b)=1 \Rightarrow (c,b)=1$

$(c,d)=1$ and $(d,c)=1 \Rightarrow (c,c)=1$

$(c,d)=1$ and $(d,d)=1 \Rightarrow (c,d)=1$

The path matrix or transitive closure is,

$R^{(4)} = A^d =$

	a	b	c	d
a	0	0	0	0
b	1	1	1	1
c	1	1	1	1
d	1	1	1	1

Pseudocode:

```
Algorithm Warshall(a, p)
// Purpose: To implement Warshall's algorithm for
            computing the transitive closure.
// Input: Adjacency matrix A of the given directed
          graph G = (V, E)
// Output: Path matrix or transitive closure of the
           given graph.
for i ← 1 to n
    for j ← 1 to n
        p[i,j] ← a[i,j]
    end for
end for
for k ← 1 to n
    for i ← 1 to n
        for j ← 1 to n
            if (p[i,j] ≠ 1 and (p[i,k] = 1 and
                                p[k,j] = 1))
                p[i,j] ← 1
            end if
        end for
    end for
end for
```

Analysis:

The parameter to be considered is 'n'.

The basic operation is,

if (p[i,j] ≠ 1 and (p[i,k] = 1 and p[k,j] = 1))

$$T(n) = \sum_{k=1}^{n} \sum_{i=1}^{n} \sum_{j=1}^{n} 1$$

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

$$= n \sum_{k=1}^{n} \sum_{i=1}^{n} 1$$

$$= n \sum_{k=1}^{n} n-1+1$$

$$= n \times n \sum_{k=1}^{n} 1$$

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

$$= n^2 \times n$$

$$= n^3$$

$$\therefore T(n) \approx O(n^3)$$

(3) Flyod's algorithm: (All-pairs shortest-paths problem)

Distance matrix: n-by-n matrix D, the element dij in the ith row and jth column of this matrix indicates the length of the shortest path from ith vertex to jth vertex.

$(1 \leq i \text{ and } j \leq n)$

$(i \leq n \text{ and } 1 \leq j)$

Applicable to,

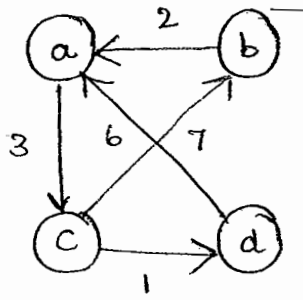
(1) Directed weighted graphs

(2) Undirected weighted graphs

Condition:

Must not contain cycle of negative length.

Ex:



To compute shortest path from c to a,

$c \rightarrow d \rightarrow a = 1+6 = 7$

$c \rightarrow b \rightarrow a = 7+2 = 9$

Even though 'a' can be reached from 'c' in two different ways, the path through vertex 'd' is chosen since it has the least weight i.e 7.

Flyod's algorithm always computes the shortest path between two vertices, it can be a direct edge or direct path or a path through one or more intermediate vertices.

The distance matrix is computed for a graph with 'n' vertices through a series of n-by-n matrices,

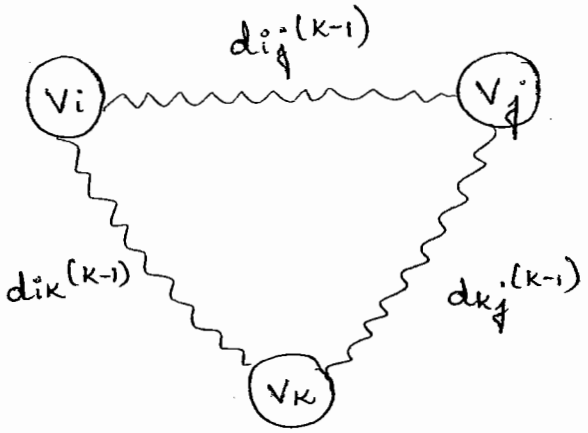
$$D^{(0)}, \ldots D^{(k-1)}, D^{(k)}, \ldots D^{(n)}$$

$D^{(0)}$ = cost adjacency matrix

Each element of the matrix $D^{(k)}$ can be computed from its immediate predecessor $D^{(k-1)}$.

Let $d_{ij}^{(k)}$ be the element in ith row and jth column of matrix $D^{(k)}$.

$d_{ij}^{(k)}$ = length of shortest path among all paths from the ith vertex v_i to the jth vertex v_j with their intermediate vertices numbered not higher than k.



Underlying idea of Flyod's algorithm

Algorithm Flyod (a, n)

// Purpose: To find the shortest distance from each vertex to remaining vertices of a graph

// Input: Cost adjacency matrix of the weighted graph G = (V, E)

// Output: The distance matrix of the shortest

```
for i=1 to n
    for j=1 to n
        d[i,j] ← a[i,j]
    end for
end for
for k=1 to n
    for i=1 to n
        for j=1 to n
            d[i,j] = min{ d[i,j], d[i,k] + d[k,j] }
        end for
    end for
end for
```

The time complexity is,

The basic operation is " $d[i,j] = \min\{ d[i,j], d[i,k] + d[k,j] \}$ "

$$f(n) = \sum_{k=1}^{n} \sum_{i=1}^{n} \sum_{j=1}^{n} 1$$

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

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

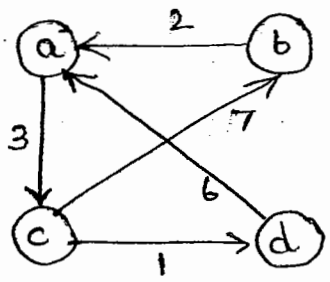
$$= n^2 \sum_{k=1}^{n} 1$$

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

$$= n^3$$

$$\therefore f(n) = O(n^3)$$

Ex:



The cost adjacency matrix is as shown below:

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

Step 1: Through vertex 'a'

$(b,a)=2 \quad (a,c)=3 \quad \therefore (b,c) = \min(\infty, 2+3) = 5$

$(d,a)=6 \quad (a,c)=3 \quad \therefore (d,c) = \min(\infty, 6+3) = 9$

The resultant matrix is,

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	∞	7	0	1
d	6	∞	9	0

Step 2: Through vertex 'b'

$(c,b)=7 \quad (b,a)=2 \quad \therefore (c,a) = \min(\infty, 7+2) = 9$

$(c,b)=7 \quad (b,c)=5 \quad \therefore (c,c) = \min(0, 7+5) = 0$

The resultant matrix is,

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	9	7	0	1
d	6	∞	9	0

Step 3: Through vertex 'c'

$(a,c)=3 \quad (c,a)=9 \quad \therefore (a,a)=\min(0,3+9)=0$

$(a,c)=3 \quad (c,b)=7 \quad \therefore (a,b)=\min(\infty,3+7)=10$

$(a,c)=3 \quad (c,d)=1 \quad \therefore (a,d)=\min(\infty,3+1)=4$

$(b,c)=5 \quad (c,a)=9 \quad \therefore (b,a)=\min(2,5+9)=2$

$(b,c)=5 \quad (c,b)=7 \quad \therefore (b,b)=\min(0,5+7)=0$

$(b,c)=5 \quad (c,d)=1 \quad \therefore (b,d)=\min(\infty,5+1)=6$

$(d,c)=9 \quad (c,a)=9 \quad \therefore (d,a)=\min(6,9+9)=6$

$(d,c)=9 \quad (c,b)=7 \quad \therefore (d,b)=\min(\infty,9+7)=16$

$(d,c)=9 \quad (c,d)=1 \quad \therefore (d,d)=\min(0,9+1)=0$

The resultant matrix is,

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	9	7	0	1
d	6	16	9	0

Step 4: Through vertex 'd'

$(a,d)=4 \quad (d,a)=6 \quad \therefore (a,a)=\min(0,4+6)=0$

$(a,d)=4 \quad (d,b)=16 \quad \therefore (a,b)=\min(10,4+16)=10$

$(a,d)=4 \quad (d,c)=9 \quad \therefore (a,c)=\min(3,4+9)=3$

$(b,d)=6 \quad (d,a)=6 \quad \therefore (b,a)=\min(2,6+6)=2$

$(b,d)=6 \quad (d,b)=16 \quad \therefore (b,b)=\min(0,6+16)=0$

$(b,d)=6 \quad (d,c)=9 \quad \therefore (b,c)=\min(5,6+9)=5$

$(c,d)=1 \quad (d,a)=6 \quad \therefore (c,a)=\min(9,1+6)=7$

$(c,d)=1 \quad (d,b)=16 \quad \therefore (c,b)=\min(7,1+16)=7$

$(c,d)=1 \quad (d,c)=9 \quad \therefore (c,c)=\min(0,1+9)=10$

The resultant matrix is,

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	7	7	0	1
d	6	16	9	0

KNAPSACK PROBLEM :

Given :

'n' items of weights $w_1, w_2, ----w_n$

'n' items of values / profits $v_1, v_2, ----v_n$

Knapsack of capacity M

We are supposed to find the most valuable subset of items that fit into the knapsack.

To obtain the solution to the above problem using dynamic programming we use the following recursive relation :

$$v[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ v[i-1,j] & \text{if } w_i > j \\ \max\{ v[i-1,j], v[i-1, j-w_i]+p_i \} & \text{if } w_i \leq j \end{cases}$$

where $v[i,j]$ represents the optimal solution

$v[i-1,j]$ represents ith object not selected and

$v[i-1, j-w_i]$ represents ith object selected

Ex: (1)

Item	Weight	Value / Profit
1	2	12
2	1	15
3	3	25
4	2	10

Knapsack capacity M=5

SOLUTION :

We construct a table with n+1 rows (5) and M+1 columns (6). The table is filled using the following relation,

$$V[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ V[i-1,j] & \text{if } w_i > j \\ \max\{V[i-1,j],\ V[i-1,j-w_i]+P_i\} & \text{if } w_i \le j \end{cases}$$

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	15	15	27	27	27
3	0	15	15	27	40	40
4	0	15	15	27	40	(40)

→ M

↓ n

→ optimal solution

When i=0 : $V[0,0], V[0,1], V[0,2], V[0,3], V[0,4], V[0,5] = 0$

When j=0 : $V[0,0], V[1,0], V[2,0], V[3,0], V[4,0] = 0$

Step 1 : When $i=1$, $W_1=2$, $P_1=12$

$j=1$ $V[1,1] = V[0,1] = 0$

$j=2$ $V[1,2] = \max\{V[0,2], V[0,0]+12\} = 12$

$j=3$ $V[1,3] = \max\{V[0,3], V[0,1]+12\} = 12$

$j=4$ $V[1,4] = \max\{V[0,4], V[0,2]+12\} = 12$

$j=5$ $V[1,5] = \max\{V[0,5], V[0,3]+12\} = 12$

Step 2 : When $i=2$, $W_2=1$, $P_2=15$

$j=1$ $V[2,1] = \max\{V[1,1], V[1,0]+15\} = 15$

$j=2$ $V[2,2] = \max\{V[1,2], V[1,1]+15\} = 15$

$j=3$ $V[2,3] = \max\{V[1,3], V[1,2]+15\} = 27$

$j=4$ $V[2,4] = \max\{V[1,4], V[1,3]+15\} = 27$

$j=5$ $V[2,5] = \max\{V[1,5], V[1,4]+15\} = 27$

Step 3 : When $i=3$, $W_3=3$, $P_3=25$

$j=1$ $V[3,1] = V[2,1] = 15$

$j=2$ $V[3,2] = V[2,2] = 15$

$j=3$ $V[3,3] = \max\{V[2,3], V[2,0]+25\} = 27$

$j=4$ $V[3,4] = \max\{V[2,4], V[2,1]+25\} = 40$

$j=5$ $V[3,5] = \max\{V[2,5], V[2,2]+25\} = 40$

Step 4 : When $i=4$, $W_4=2$, $P_4=10$

$j=1$ $V[4,1] = V[3,1] = 15$

$j=2$ $V[4,2] = \max\{V[3,2], V[3,0]+10\} = 15$

$j=3$ $V[4,3] = \max\{V[3,3], V[3,1]+10\} = 27$

$j=4$ $V[4,4] = \max\{V[3,4], V[3,2]+10\} = 40$

$j=5$ $V[4,5] = \max\{V[3,5], V[3,3]+10\} = 40$

The optimal solution is given by $v[n,m] = v[4,5]$

$= 40$

To get the optimal solution let us express the solution using n-tuple,

x_1	x_2	x_3	x_4
0	0	0	0

If ith object has been selected then $v[i,j] \neq v[i-1,j]$

Since $v[4,5] \neq v[3,5]$, 4 th object is not selected

So now $i=3$, $j=5$

Since $v[3,5] \neq v[2,5]$, 3rd object is selected

So now $i=2$, $j=5-3=2$

Since $v[2,2] \neq v[1,2]$, 2nd object is selected

So now $i=1$, $j=2-1=1$

Since $v[1,1] = v[0,1]$, 1 st object is not selected

So now $i=0$, $j=1$

The solution now is,

(Since $v[0,1]$ cannot be considered, stop when i becomes 0)

$(x_1, x_2, x_3, x_4) = (0,1,1,0)$

∴ Objects or item 2 and 3 are selected to obtain the optimal solution.

Algorithm knapsack (n, M, W, P, V, x)

// Purpose : To obtain the optimal solution for the knapsack problem

// Inputs : n → number of items
M → capacity of knapsack
W → weights of 'n' objects
P → profits of 'n' objects

// Outputs : V → optimal solution
x → contains the objects selected

```
for i ← 0 to n
    for j ← 0 to m
        if i=0 or j=0
            v[i, j] = 0
        else if wi > j
            v[i, j] ← v[i-1, j]
        else
            v[i, j] = max{ v[i-1, j], v[i-1, j-wi] + Pi }
        end if
    end for
end for

for i ← 0 to n
    x[i] ← 0
end for
i ← n
j ← m
```

```
while i ≠ 0 and j ≠ 0
    if V[i,j] ≠ V[i-1,j]
        x[i] ← 1
        j ← j - wi
    end if
    i ← i - 1
end while
for i ← 1 to n
    if x[i] ≠ 1
        print Object i selected
    else
        print Object i not selected
end for
return V[n,m]
```

Knapsack problem using memory functions:

The top down approach which we used for solving the knapsack problem solves common subproblems more than once and hence is inefficient.

Hence we can use the bottom up approach since it fills the table in such a way that all subproblems are solved only once. This algorithm uses memory functions.

We consider the same example,

Item	Weight	Value
1	2	12
2	1	15
3	3	25
4	2	10

$$V[4,5] = \max\{V[3,5], V[3,3]+10\} \quad \text{——(1)}$$

To compute $V[3,5]$

$$V[3,5] = \max\{V[2,5], V[2,2]+25\} \quad \text{——(2)}$$

To compute $V[3,3]$

$$V[3,3] = \max\{V[2,3], V[2,0]+25\} \quad \text{——(3)}$$

To compute $V[2,5]$

$$V[2,5] = \max\{V[1,5], V[1,4]+15\} \quad \text{——(4)}$$

To compute $V[2,2]$

$$V[2,2] = \max\{V[1,2], V[1,1]+15\} \quad \text{——(5)}$$

To compute $V[2,3]$

$$V[2,3] = \max\{V[1,3], V[1,2]+15\} \quad \text{——(6)}$$

To compute $V[1,5]$

$$V[1,5] = \max\{V[0,5], V[0,3]+12\} \quad \text{——(7)}$$

To compute $V[1,4]$

$$V[1,4] = \max\{V[0,4], V[0,2]+12\} \quad \text{——(8)}$$

To compute $v[1,2]$

$$v[1,2] = \max\{ v[0,2], v[0,0]+12 \} \quad \text{——(9)}$$

To compute $v[1,1]$

$$v[1,1] = v[0,1] = 0 \quad \text{——(10)}$$

To compute $v[1,3]$

$$v[1,3] = \max\{ v[0,3], v[0,1]+12 \} \quad \text{——(11)}$$

$v[1,3] = 12$

$v[1,2] = 12$

$v[1,4] = 12$

$v[1,5] = 12$

$v[2,3] = \max(12, 12+15) = 27$

$v[2,2] = \max(12, 0+15) = 15$

$v[2,5] = \max(12, 12+15) = 27$

$v[3,3] = \max(27, 0+25) = 27$

$v[3,5] = \max(27, 15+25) = 40$

$v[4,5] = \max(40, 27+10) = 40$

The computed values are stored in the table as shown below,

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0		15	27		27
3	0			27		40
4	0					40

The time complexity is given by,

$$T(n) = \sum_{i=0}^{n} \sum_{j=0}^{m} 1$$

$$= \sum_{i=0}^{n} m-0+1$$

$$= (m+1) \sum_{i=0}^{n} 1$$

$$= (m+1)(n-0+1)$$

$$= mn+m+n+1$$

Neglecting all lower order terms and constants

$$T(n) \approx mn$$

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

where $n \rightarrow$ number of items

$m \rightarrow$ capacity of knapsack

Ex: (2)

Item	Weight	Profit / Value
1	3	25
2	2	20
3	1	15
4	4	40
5	5	50

Capacity $M = 6$

Solution:

$V[0,0] = V[0,1] = V[0,2] = V[0,3] = V[0,4] = V[0,5]$

$= V[0,6] = V[1,0] = V[2,0] = V[3,0] = V[4,0]$

$= V[5,0] = 0$

$V[1,1] = V[0,1] = 0$

$V[1,2] = V[0,2] = 0$

$V[1,3] = \max\{ V[0,3], V[0,0] + 25 \}$

$= \max\{ 0, 25 \} = 25$

$V[1,4] = \max\{ V[0,4], V[0,1] + 25 \}$

$= \max\{ 0, 25 \} = 25$

$V[1,5] = \max\{ V[0,5], V[0,2] + 25 \}$

$= \max\{ 0, 25 \} = 25$

$V[1,6] = \max\{ V[0,6], V[0,3] + 25 \}$

$= \max\{ 0, 25 \} = 25$

Step 2 : $i = 2, w_2 = 2, P_2 = 20$

$V[2,1] = V[1,1] = 0$

$V[2,2] = \max\{ V[1,2], V[1,0] + 20 \}$

$= \max\{ 0, 0+20 \} = 20$

$V[2,3] = \max\{ V[1,3], V[1,1] + 20 \}$

$= \max\{ 25, 0+20 \} = 25$

$V[2,4] = \max\{ V[1,4], V[1,2] + 20 \}$

$= \max\{ 25, 0+20 \} = 25$

$V[2,5] = \max\{ V[1,5], V[1,3] + 20 \}$

$= \max\{ 25, 25+20 \} = 45$

$V[2,6] = \max\{ V[1,6], V[1,4] + 20 \}$

$= \max\{ 25, 25+20 \} = 45$

Step 3 : $i = 3, w_3 = 1, P_3 = 15$

$V[3,1] = \max\{ V[2,1], V[2,0] + 15 \}$

$= \max\{ 0, 0+15 \} = 15$

$V[3,2] = \max\{V[2,2], V[2,1]+15\}$

$= \max\{20, 0+15\} = 20$

$V[3,3] = \max\{V[2,3], V[2,2]+15\}$

$= \max\{25, 20+15\} = 35$

$V[3,4] = \max\{V[2,4], V[2,3]+15\}$

$= \max\{25, 25+15\} = 40$

$V[3,5] = \max\{V[2,5], V[2,4]+15\}$

$= \max\{45, 25+15\} = 45$

$V[3,6] = \max\{V[2,6], V[2,5]+15\}$

$= \max\{45, 45+15\} = 60$

Step 4 : $i=4$, $W_4=4$, $P_4=40$

$V[4,1] = V[3,1] = 15$

$V[4,2] = V[3,2] = 20$

$V[4,3] = V[3,3] = 35$

$V[4,4] = \max\{V[3,4], V[3,0]+40\}$

$= \max\{40, 0+40\} = 40$

$V[4,5] = \max\{V[3,5], V[3,1]+40\}$

$= \max\{45, 15+40\} = 55$

$V[4,6] = \max\{V[3,6], V[3,2]+40\}$

$= \max\{60, 20+40\} = 60$

Step 5 : $i=5$, $W_5=5$, $P_5=50$

$V[5,1] = V[4,1] = 15$

$V[5,2] = V[4,2] = 20$

$V[5,3] = V[4,3] = 35$

$V[5,4] = V[4,4] = 40$

$V[5,5] = \max\{V[4,5], V[4,0]+50\}$

$= \max\{55, 0+50\} = 55$

$V[5,6] = \max\{V[4,6], V[4,1]+50\}$

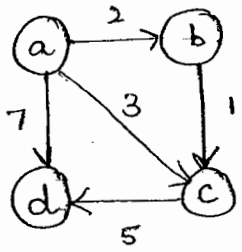
$= \max\{60, 15+50\} = 65$

n \ M	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	25	25	25	25
2	0	0	20	25	25	45	45
3	0	15	20	35	40	45	60
4	0	15	20	35	40	55	60
5	0	15	20	35	40	55	65

→ optimal solution

Problems on Flyod's algorithm:

(1) Given graph:



Solution:

Adjacency matrix A =

	a	b	c	d
a	0	2	3	7
b	∞	0	1	∞
c	∞	∞	0	5
d	∞	∞	∞	0

Step 1: Through vertex 'a'

Since all the entries in the first column are 0 and ∞ ∴ No change.

$$A^a = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & a \end{array} \\ \begin{bmatrix} 0 & 2 & 3 & 7 \\ \infty & 0 & 1 & \infty \\ \infty & \infty & 0 & 5 \\ \infty & \infty & \infty & 0 \end{bmatrix} \end{array}$$

Step 2 : Through the vertex 'b'

$(a,b) = 2$ and $(b,c) = 1$

$\therefore (a,c) = \min\{3, 3\} = 3$

$$A^b = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 2 & 3 & 7 \\ \infty & 0 & 1 & \infty \\ \infty & \infty & 0 & 5 \\ \infty & \infty & \infty & 0 \end{bmatrix} \end{array}$$

Step 3 : Through the vertex 'c'

$(a,c) = 3$ and $(c,d) = 5$

$\therefore (a,d) = \min\{7, 3+5\} = 7$

$(b,c) = 1$ and $(c,d) = 5$

$\therefore (b,d) = \min\{\infty, 1+5\} = 6$

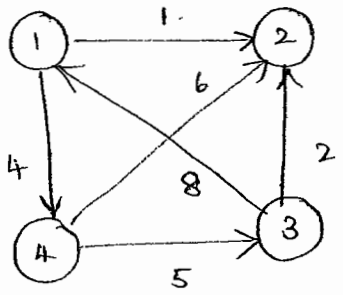
$$A^c = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 2 & 3 & 7 \\ \infty & 0 & 1 & 6 \\ \infty & \infty & 0 & 5 \\ \infty & \infty & \infty & 0 \end{bmatrix} \end{array}$$

Step 4 : Through the vertex 'd'

No changes since all entries in last row are 0 and ∞

$\therefore$ The distance matrix $D = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 2 & 3 & 7 \\ \infty & 0 & 1 & 6 \\ \infty & \infty & 0 & 5 \\ \infty & \infty & \infty & 0 \end{bmatrix} \end{array}$

(2) Given graph:



Solution:

The cost adjacency matrix A =

	1	2	3	4
1	0	1	∞	4
2	∞	0	∞	∞
3	8	2	0	∞
4	∞	6	5	0

Step 1: Through the vertex '1'

$(3,1) = 8$ and $(1,2) = 1$

$\therefore (3,2) = \min\{2, 1+8\} = 2$

$(3,1) = 8$ and $(1,4) = 4$

$\therefore (3,4) = \min\{\infty, 8+4\} = 12$

A^1 =

	1	2	3	4
1	0	1	∞	4
2	∞	0	∞	∞
3	8	2	0	12
4	∞	6	5	0

Step 2: Through the vertex '2'

No changes since all entries in second row are 0 and ∞.

A^2 =

	1	2	3	4
1	0	1	∞	4
2	∞	0	∞	∞
3	8	2	0	12
4	∞	6	5	0

Step 3 : Through the vertex '3'

$(4,3) = 5$ and $(3,1) = 8$

$\therefore (4,1) = \min\{\infty, 5+8\} = 13$

$(4,3) = 5$ and $(3,2) = 2$

$\therefore (4,2) = \min\{6, 5+2\} = 6$

$(4,3) = 5$ and $(3,4) = 12$

$\therefore (4,4) = \min\{0, 5+12\} = 0$

$A^3 =$

	1	2	3	4
1	0	1	∞	4
2	∞	0	∞	∞
3	8	2	0	12
4	13	6	5	0

Step 4 : Through the vertex '4'

$(1,4) = 4$ and $(4,1) = 13$

$\therefore (1,1) = \min\{0, 4+13\} = 0$

$(1,4) = 4$ and $(4,2) = 6$

$\therefore (1,2) = \min\{1, 4+6\} = 1$

$(1,4) = 4$ and $(4,3) = 5$

$\therefore (1,3) = \min\{\infty, 4+5\} = 9$

$(3,4) = 12$ and $(4,1) = 13$

$\therefore (3,1) = \min\{8, 12+13\} = 8$

$(3,4) = 12$ and $(4,2) = 6$

$\therefore (3,2) = \min\{2, 12+6\} = 2$

$(3,4) = 12$ and $(4,3) = 5$

$\therefore (3,3) = \min\{0, 12+5\} = 0$

∴ The distance matrix is,

$$D = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 9 & 4 \\ \infty & 0 & \infty & \infty \\ 8 & 2 & 0 & 12 \\ 13 & 6 & 5 & 0 \end{bmatrix} \end{array}$$

(1)

GREEDY TECHNIQUE

In this technique, we construct the solution through a sequence of steps. The partially obtained solutions after each step is used to obtain the complete solution.

The solution should satisfy the following 3 conditions:

1. Feasible - the problem should satisfy the constraints specified.
2. Locally optimal - it should be the best solution among all the available solutions.
3. Irrevocable - once a decision is made it cannot be changed on subsequent steps of the algorithm.

PRIM'S ALGORITHM:

A spanning tree of a connected graph is it's connected acyclic subgraph that contains all the vertices of the graph.

A minimum spanning tree of a weighted connected graph is it's spanning tree of the smallest weight. Weight is sum of the weights on it's edges.

Initially we have a subtree with a single vertex and no edges. We keep adding an edge with minimum weight and also it's corresponding vertex. Once all the vertices have been included

we stop. For 'n' vertices n-1 iterations are needed. A spanning tree always has n-1 edges where 'n' is the number of vertices.

Algorithm prims (G)

// Purpose : To construct minimum spanning tree

// Input : A weighted connected graph

// Output : The set of edges which form the minimum spanning tree.

$V_T \leftarrow \{v_0\}$, $E_T \leftarrow \phi$

for i ← 1 to |V|-1 do

find an edge $e^* = (v^*, u^*)$ with minimum weight among all the edges $e = (v, u)$ such that v^* is in V_T and u^* is in $V - V_T$

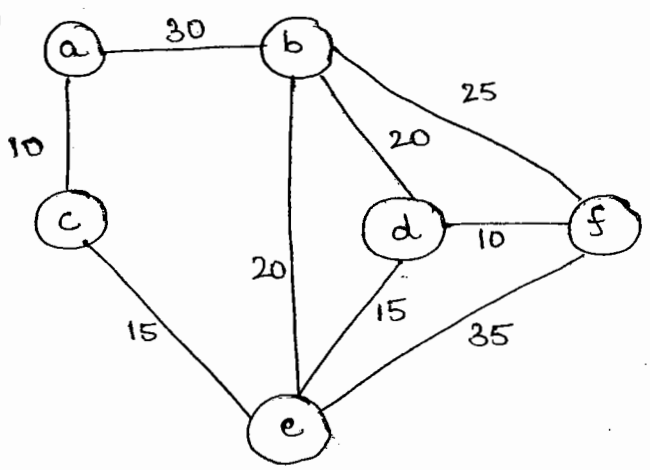
$V_T \leftarrow V_T \cup \{u^*\}$

$E_T \leftarrow E_T \cup \{e^*\}$

end for

return E_T

Ex: 1)



Initially,

$V_T = \{a\}$ $E_T = \phi$

The cost adjacency matrix for the above graph is,

	a	b	c	d	e	f
a	0	30	10	∞	∞	∞
b	30	0	∞	20	20	25
c	10	∞	0	∞	15	∞
d	∞	20	∞	0	15	10
e	∞	20	15	15	0	35
f	∞	25	∞	10	35	0

$\langle a,b \rangle = 30$ $\underline{\langle a,c \rangle = 10}$ $\langle a,d \rangle = \infty$ $\langle a,e \rangle = \infty$ $\langle a,f \rangle = \infty$

The least weight among these edges is 10

$\therefore V_T = \{a,c\}$ $E_T = \{(a,c)\}$

$\langle a,b \rangle = 30$ $\langle a,d \rangle = \infty$ $\langle a,e \rangle = \infty$ $\langle a,f \rangle = \infty$

$\langle c,b \rangle = \infty$ $\langle c,d \rangle = \infty$ $\underline{\langle c,e \rangle = 15}$ $\langle c,f \rangle = \infty$

The least weight is 15

$\therefore V_T = \{a,c,e\}$ $E_T = \{(a,c),(c,e)\}$

$\langle a,b \rangle = 30$ $\langle a,d \rangle = \infty$ $\langle a,f \rangle = \infty$

$\langle c,b \rangle = \infty$ $\langle c,d \rangle = \infty$ $\langle c,f \rangle = \infty$

$\langle e,b \rangle = 20$ $\underline{\langle e,d \rangle = 15}$ $\langle e,f \rangle = 35$

$\therefore V_T = \{a,c,d,e\}$

$E_T = \{(a,c),(c,e),(e,d)\}$

$\langle a,b \rangle = 30$ $\langle a,f \rangle = \infty$ $\langle c,b \rangle = \infty$ $\langle c,f \rangle = \infty$

$\langle d,b \rangle = 20$ $\underline{\langle d,f \rangle = 10}$ $\langle e,b \rangle = 20$ $\langle e,f \rangle = 35$

$\therefore V_T = \{a, c, d, e, f\}$

$E_T = \{(a,c), (c,e), (e,d), (d,f)\}$

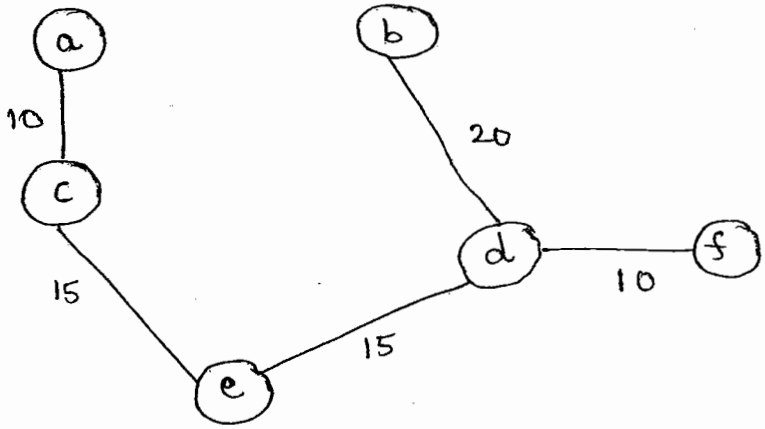
$\langle a,b \rangle = 30 \quad \langle c,b \rangle = \infty \quad \langle d,b \rangle = 20 \quad \langle e,b \rangle = 20$

$\langle f,b \rangle = 25$

$\therefore V_T = \{a, b, c, d, e, f\}$

$E_T = \{(a,c), (c,e), (e,d), (d,f), (d,b)\}$

The minimum cost spanning tree is,



The minimum cost is,

$10 + 15 + 15 + 10 + 20 = 70$ units

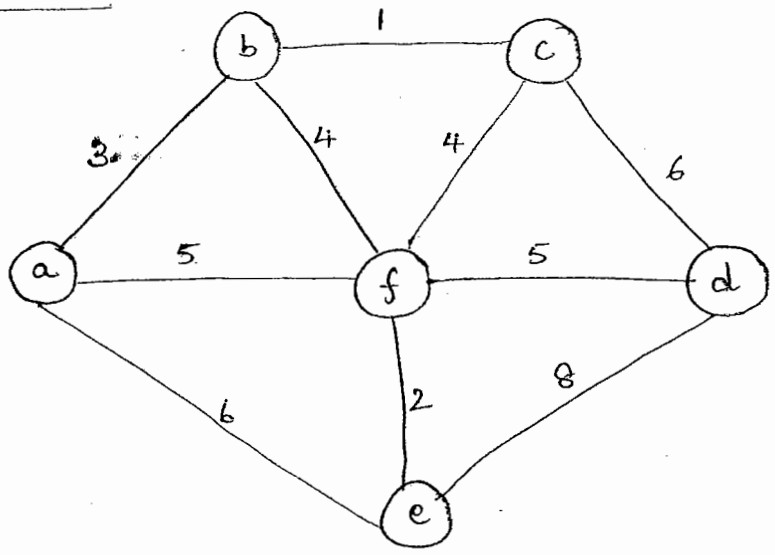
The time complexity is $|V|^2$ if adjacency matrix representation is used and priority queue is unordered.

$$f(n) = O(|V|^2)$$

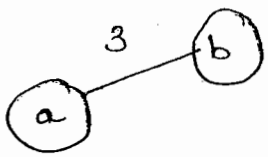
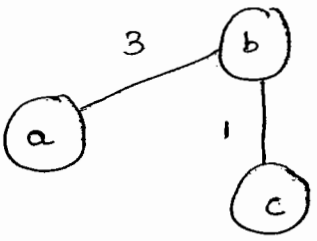
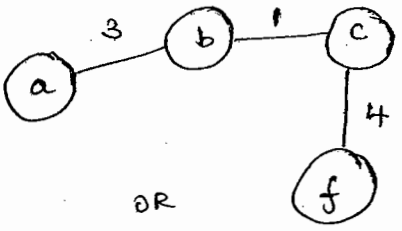
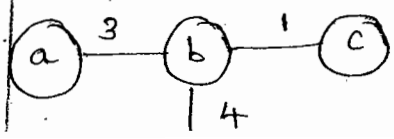
If adjacency linked list representation is used and priority queue is implemented as min heap then the time complexity is,

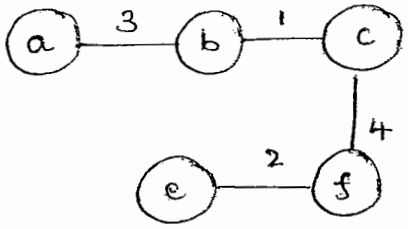
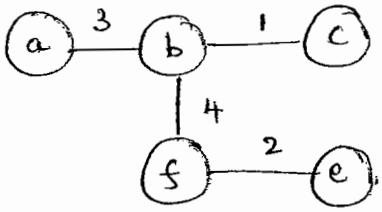
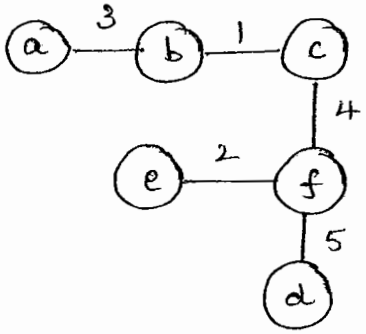
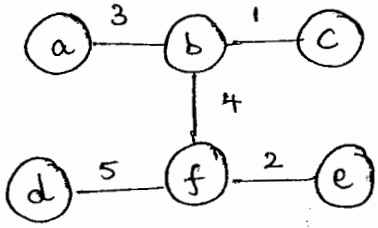
$$f(n) = O(|E| \log |V|)$$

(2) Problem:

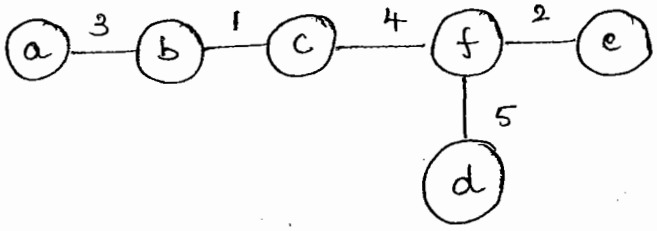


Solution:

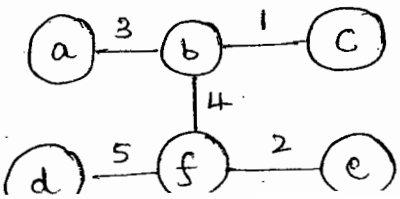
Tree Vertices	Remaining vertices	Resulting graph
Step 1: a(-, -)	**b(a,3)** c(-, ∞) d(-, ∞) e(a,6) f(a,5)	
Step 2: b(a,3)	**c(b,1)** d(-, ∞) e(a,6) f(b,4)	
Step 3: c(b,1)	d(c,6) e(a,6) **f(c,4)** OR **f(b,4)**	 OR 

Tree Vertices	Remaining vertices	Resulting graph
Step 4: f(c,4)	d(f,5) **e(f,2)**	
OR		OR
f(b,4)	d(f,5) **e(f,2)**	
Step 5: e(f,2)	**d(f,5)**	
		OR
		

Minimum spanning tree,



OR

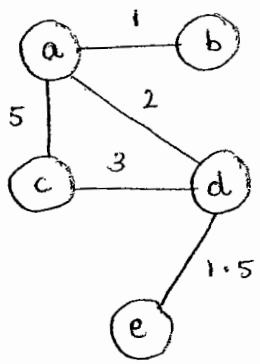


Minimum cost =

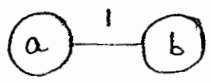
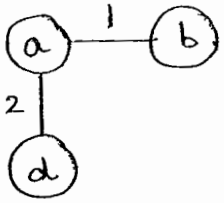
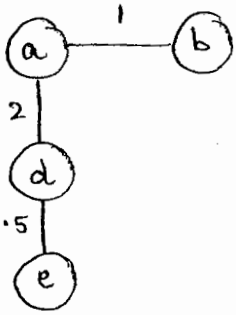
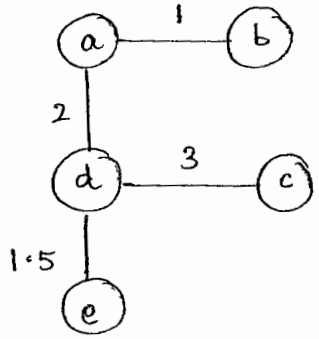
3+1+4+2+5

= 15 units

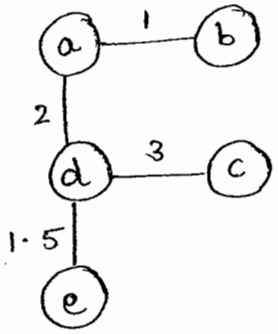
(3) Problem:



Solution:

Tree Vertices	Remaining vertices	Resulting graph
1. a(-,-)	**b (a,1)** c (a,5) d (a,2) e (-,∞)	
2. b(a,1)	c(a,5) c (b,-∞) **d(a,2)** d (b,∞) e(-,∞) e (b,∞)	
3. d(a,2)	c(a,5) c(b,∞) e(-,∞) e(b,∞) c (d,3) **e (d,1.5)**	
4. e(d,1.5)	c(a,5) c(b,∞) **c(d,3)** c (e,∞)	
5. c (d,3)		

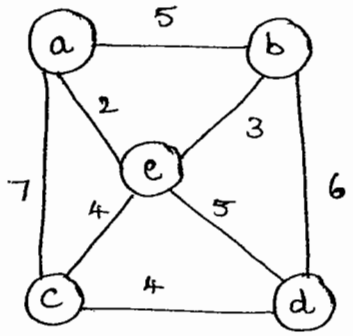
Minimum spanning tree is,



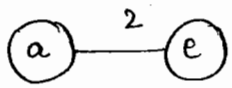
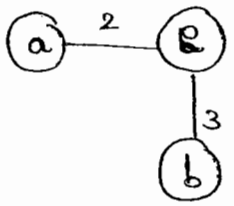
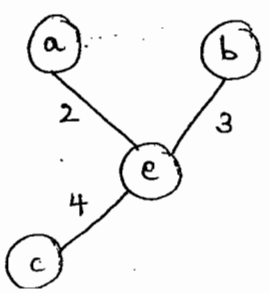
Minimum cost = 1+2+3+1·5

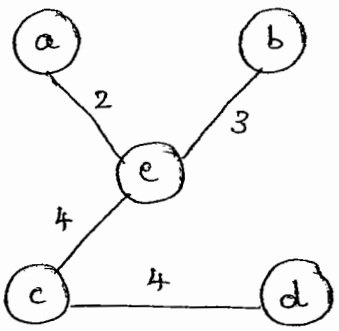
= 7·5 units

(4) Problem:

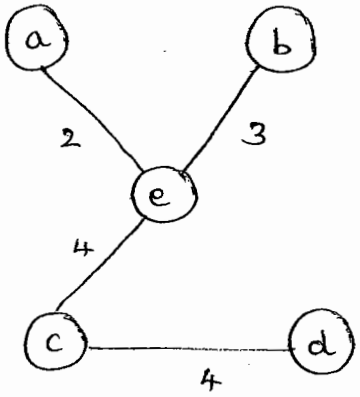


Solution:

Tree vertices	Remaining vertices	Resulting graph
1. a(–,–)	b(a,5) c(a,7) d(a,∞) [e(a,2)]	
2. e(a,2)	b(a,5) [b(e,3)] c(a,7) c(e,4) d(a,∞) d(e,5)	
3. b(e,3)	c(a,7) [c(e,4)] d(a,∞) d(e,5) c(b,∞) d(b,6)	

Tree Vertices	Remaining vertices	Resulting graph
4. c(e, 4) 5. d(c, 4)	d(a, ∞) d(e, 5) d(b, 6) **d(c, 4)**	

Minimum spanning tree:

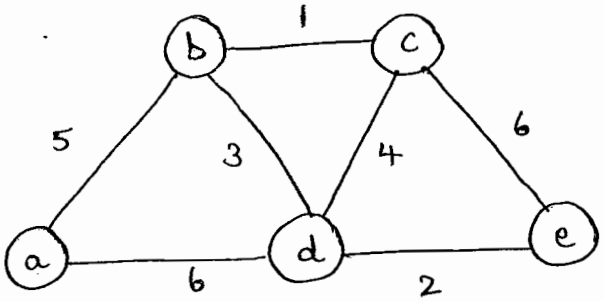


Minimum cost

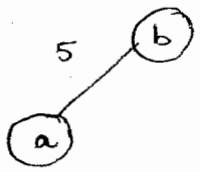
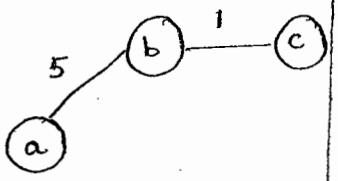
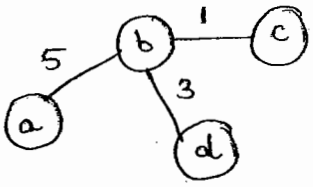
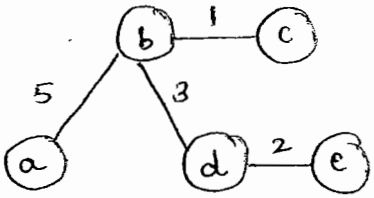
$= 2+3+4+4$

$= 13$ units

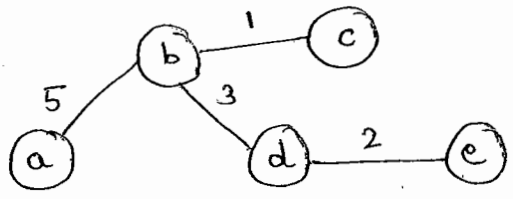
(5) Problem:



Solution:

Tree Vertices	Remaining vertices	Resulting graph
1. a(-,-)	**b (a,5)** c (-,∞) d (a,6) e (-,∞)	
2. b(a,5)	c(-,∞) **c(b,1)** d(a,6) d(b,3) e(-,∞) e(-,∞)	
3. c(b,1)	d(a,6) **d(b,3)** e(-,∞) e(-,∞) d(c,4) e(c,6)	
4. d(b,3)	e(-,∞) e(-,∞) e(c,6) **e(d,2)**	
5. e(d,2)		

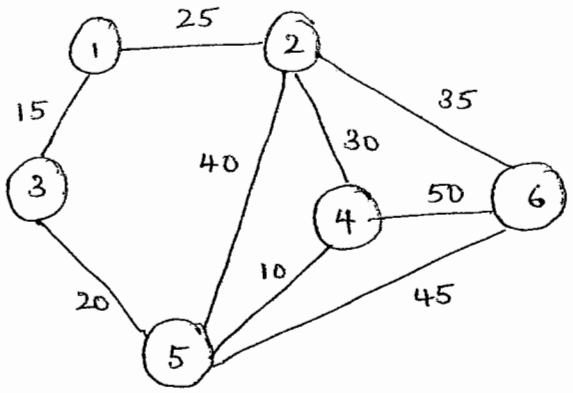
Minimum spanning tree:



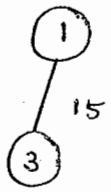
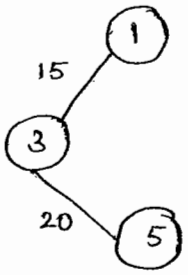
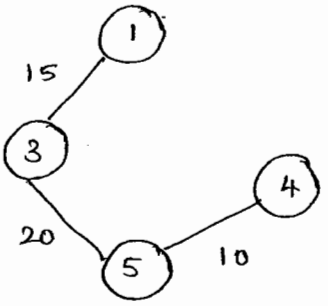
Minimum cost = 5+1+3+2

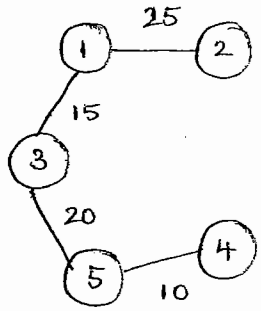
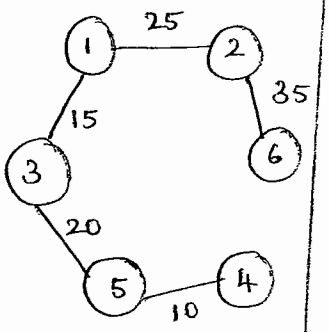
= 11 units

(6) Problem:

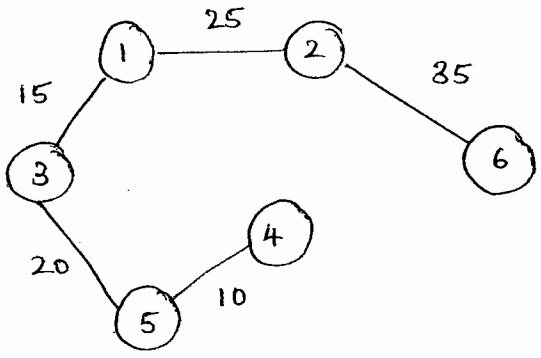


Solution:

Tree vertices	Remaining vertices	Resulting graph
1. 1(-,-)	2 (1, 25) 3 (1, 15) 4 (-, ∞) 5 (-, ∞) 6 (-, ∞)	
2. 3(1,15)	2 (1,25) 2 (-, ∞) 4 (-, ∞) 4 (-, ∞) 5 (-, ∞) 5 (3,20) 6 (-, ∞) 6 (-, ∞)	
3. 5(3,20)	2(1,25) 2(-, ∞) 4(-, ∞) 4(-, ∞) 6(-, ∞) 6(-, ∞) 2 (5,40) 4 (5,10) 6 (5,45)	

Tree Vertices	Remaining Vertices		Resulting graph
4. 4(5,10)	2(1,25)	2(-,∞)	
	6(-,∞)	6(-,∞)	
	2(5,40)	2(4,30)	
	6(5,45)	6(4,50)	
5. 2(1,25)	6(-,∞)		
	6(-,∞)		
	6(5,45)		
	6(4,50)		
	6(2,35)		

Minimum spanning tree,



Minimum cost = 25+35+15+20+10

= 105 units

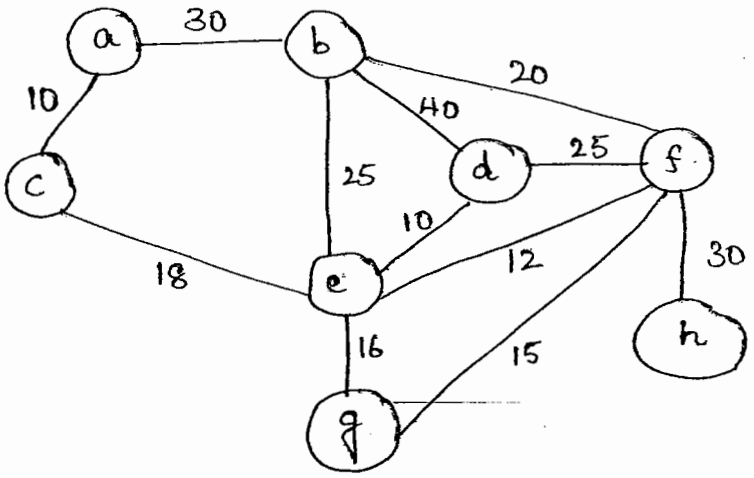
KRUSKAL'S ALGORITHM:

First we sort the graph's edges in increasing order of their weights. We start with an empty subgraph and keep adding the next edge in the sorted list if a cycle is not formed.

```
Algorithm Kruskal (G)
// Purpose: To find minimum spanning tree
// Input : A weighted connected graph G
// Output: The set of edges that form the
   minimum spanning tree.
   E_T ← ϕ
   count ← 0
   K ← 0
   while count < |V|-1
         K ← K+1
         if E_T U {e_ik} is acyclic
               E_T ← E_T U {e_ik}
               count ← count+1
         end if
   end while
   return E_T
```

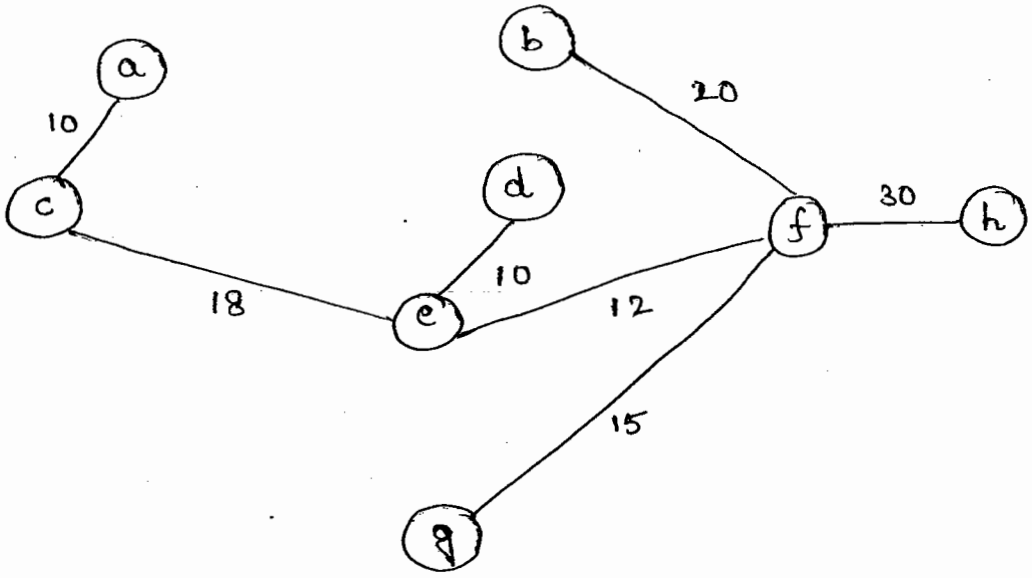
The Kruskal's algorithm may seem simpler than the Prim's algorithm but in Kruskal's algorithm each time before adding an edge we should check if it forms a cycle or not.

Ex: (1)



Arrange the edges in increasing order of their weights.

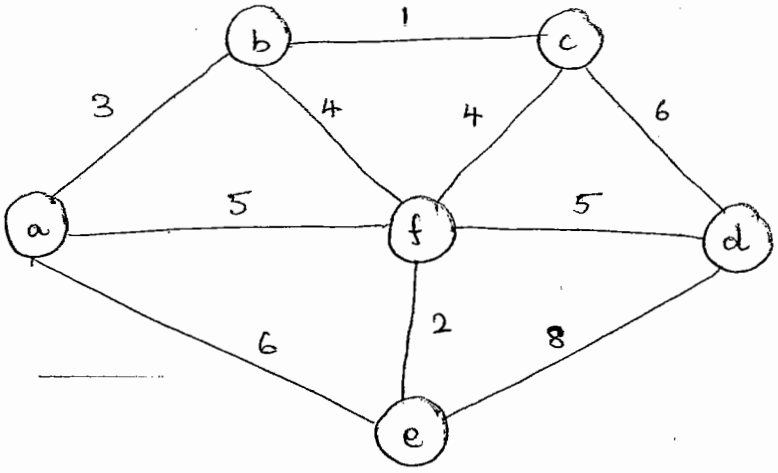
ac	de	ef	gf	eg	ce	bf	df	be	fh	ab	bd
10	10	12	15	16	18	20	25	25	30	35	40



The total cost of the minimum spanning tree is,

$10+18+10+12+15+20+30 = 115$ units

(2) Problem:

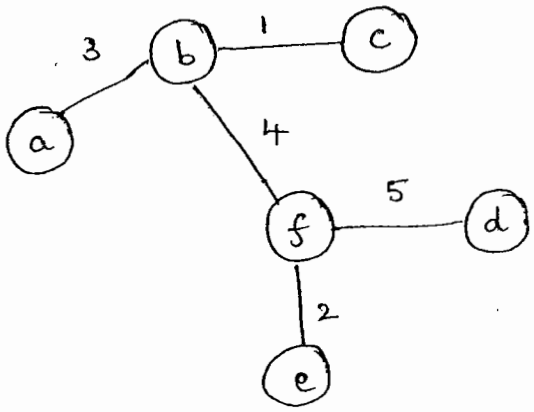


Solution:

Arrange the edges in increasing order of their weights.

✓	✓	✓	✓	×	×	✓			
bc	ef	ab	bf	cf	af	df	ae	cd	de
1	2	3	4	4	5	5	6	6	8

not considered since they form a cycle



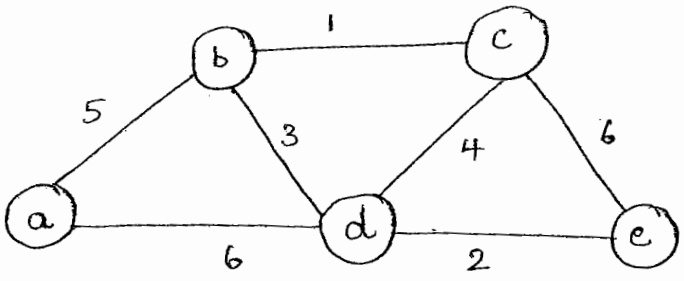
Minimum Spanning Tree

Minimum cost

= 1+3+4+2+5

= 15 units

(3) Problem :



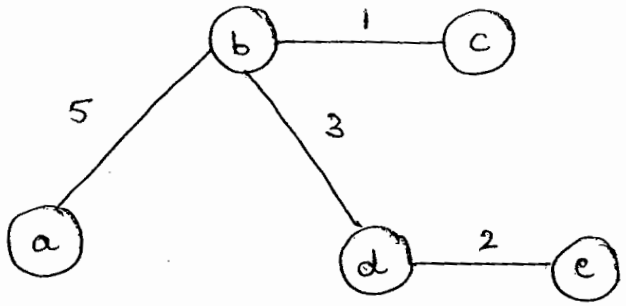
Solution :

Arrange the edges in increasing order of their weights.

bc	de	bd	cd	ab	ad	ce
✓	✓	✓	×	✓		
1	2	3	4	5	6	6

↓

not included since it forms a cycle



Minimum Spanning Tree

Minimum cost

$= 1+2+3+5$

$= 11$ units

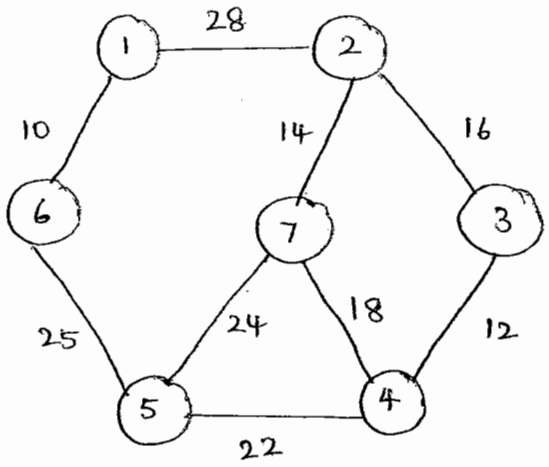
Number of vertices $= 5\ (n)$

∴ Number of edges in the spanning tree

$= n-1$

$= 5-1 = 4$

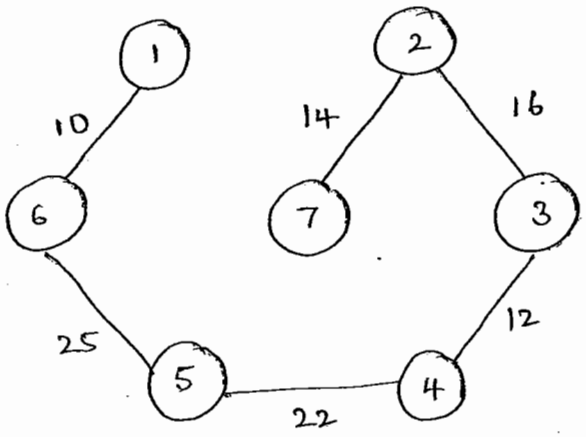
(4) Problem:



Solution:

Arrange the edges in increasing order of their weights.

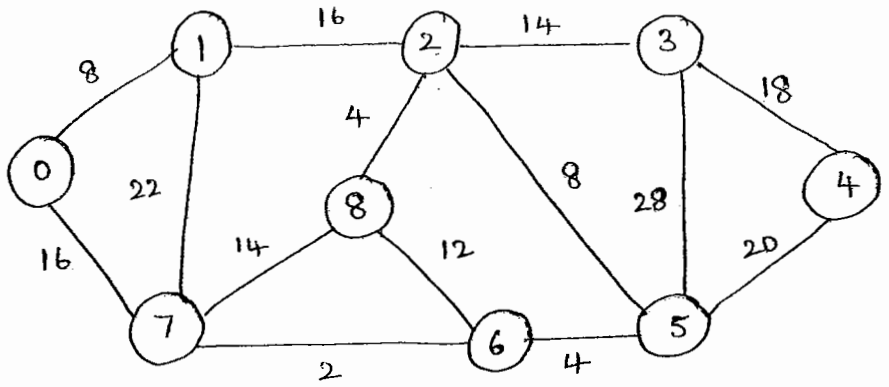
✓	✓	✓	✓	×	✓	×	✓	
16	34	27	23	47	45	57	56	12
10	12	14	16	18	22	24	25	28
				↓		↓		
				not included since it forms a cycle				



Minimum cost = 10 + 12 + 14 + 16 + 22 + 25

= 99 units

(5) Problem :

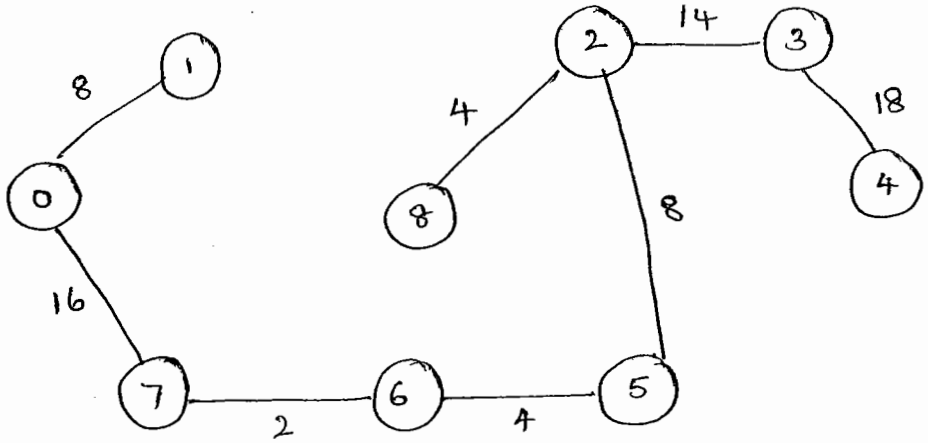


Solution :

Arrange the edges in increasing order of their weights.

✓	✓	✓	✓	✓	X	X	✓	✓	X	✓			
67	56	28	25	01	68	78	23	07	12	34	45	17	35
2	4	4	8	8	12	14	14	16	16	18	20	22	28
					↓	↓			↓				

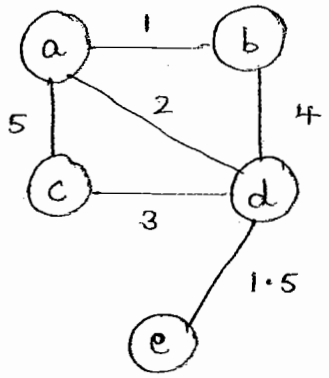
not included since it forms a cycle



Minimum cost = 8+16+2+4+8+4+14+18

= 74 units

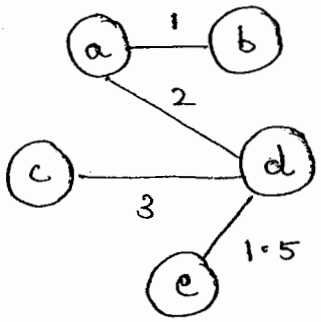
(6) Problem :



Solution :

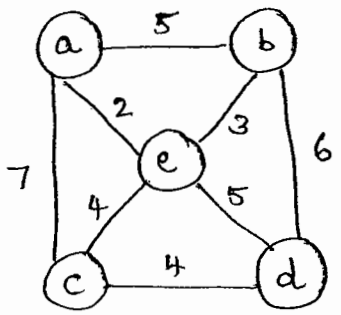
Arrange the edges in increasing order of their weights.

ab	de	ad	cd	bd	ac
1	1·5	2	3	4	5



Minimum cost
= 1 + 1·5 + 2 + 3
= 7·5 units

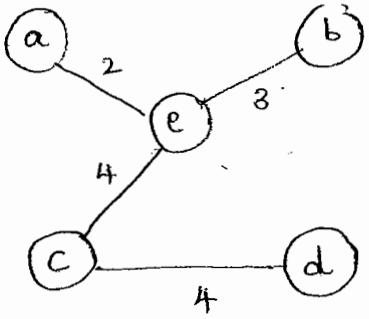
(7) Problem :



Solution :

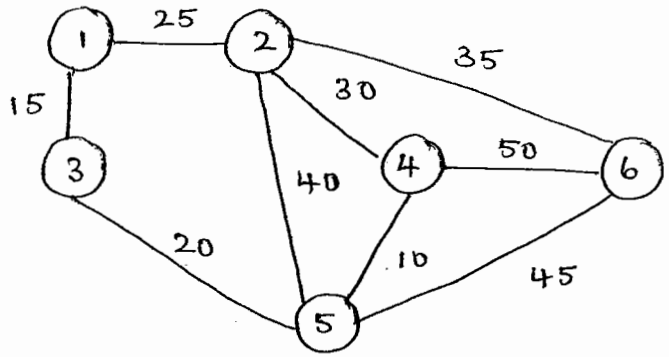
Arrange the edges in increasing order of their weights.

ae	be	ce	cd	de	ab	bd	ac
2	3	4	4	5	5	6	7



Minimum cost
$= 2+3+4+4$
$= 13$ units

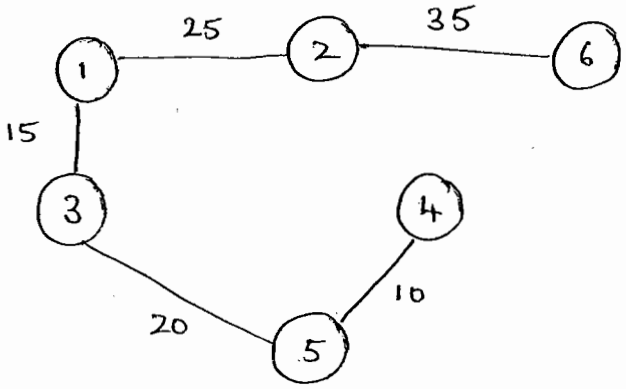
(8) Problem :



Solution :

Arrange the edges in increasing order of their weights.

✓	✓	✓	✓	×	✓			
45	13	35	12	24	26	25	56	46
10	15	20	25	30	35	40	45	50



Minimum cost = $15+20+10+25+35$

$= 105$ units

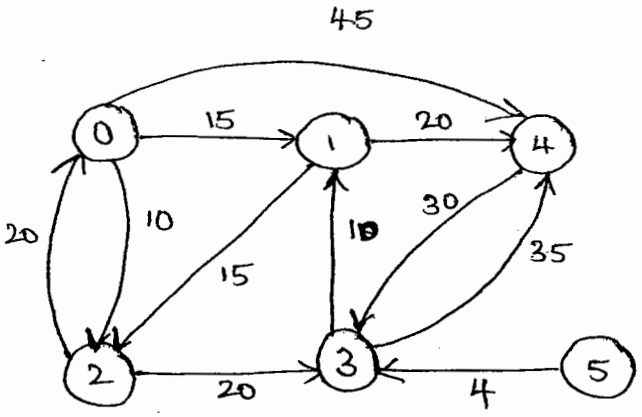
DIJKSTRA'S ALGORITHM : (Single-source shortest-paths problem)

For a given source vertex in a weighted connected graph find the shortest path to all other vertices.

Here we find the shortest path from the source to a vertex nearest to it first. Hence when the ith iteration begins, the shortest paths to other i-1 vertices nearest to source is found.

The set of vertices adjacent to the vertices present in the shortest path graph is called as fringe vertices.

Ex: (1)



Consider 5 as the source vertex

(1) 0(-,∞) 1(-,∞) 2(-,∞) **3(5,4)** 4(-,∞) 5(-,0

(2) 3(5,4) **1(3,14)** 2(-,∞) 4(3,39) 0(-,∞)

(3) 1(3,14) **2(1,29)** 4(1,34) 0(-,∞)

(4) 2(1,29) **4(1,34)** 0(2,49)

(5) 4(1,34) **0(2,49)**

(6) 0(2,49)

The shortest paths are,

$5 \rightarrow 3 \rightarrow 1 \rightarrow 2 \rightarrow 0 = 49$

$5 \rightarrow 3 \rightarrow 1 = 14$

$5 \rightarrow 3 \rightarrow 1 \rightarrow 2 = 29$

$5 \rightarrow 3 = 4$

$5 \rightarrow 3 \rightarrow 1 \rightarrow 4 = 34$

Algorithm dijkstra(n,a)

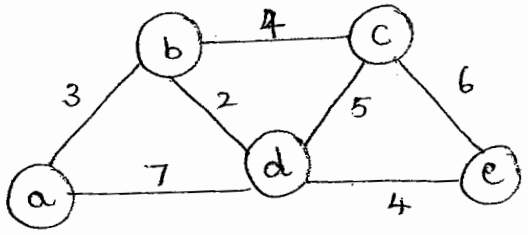
// Purpose: To find the shortest distance and the corresponding path

// Inputs: n → number of nodes in the graph
a → cost adjacency matrix

// Output: Shortest distance and the path

```
for i ← 0 to n-1
    d[i] = a[source, i]
    p[i] = source
    s[i] = 0
end for
s[source] = 1
for i ← 1 to n-1
    find u and d[u] such that d[u] is minimum
    and u ∈ V-S
    s[u] = 1
    for every v ∈ V-S
        if d[u] + a[u][v] < d[v]
            d[v] = d[u] + a[u][v]
            p[v] = u
```

(2) Given graph:



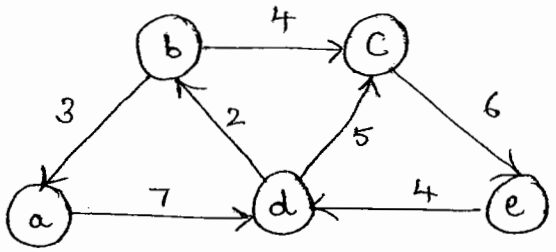
Consider 'a' as the source vertex

Tree vertices	Remaining vertices
(1) a(-,0)	b(a,3) c(-,∞) d(a,7) e(-,∞)
(2) b(a,3)	c(b,3+4) = c(b,7) d(b,3+2) = d(b,5) e(-,∞)
(3) d(b,5)	c(b,7) e(d,5+4) = e(d,9)
(4) c(b,7)	e(d,9)
(5) e(d,9)	

The shortest paths from 'a' are,

1. a to b = $a \rightarrow b = 3$
2. a to c = $a \rightarrow b \rightarrow c = 7$
3. a to d = $a \rightarrow b \rightarrow d = 5$
4. a to e = $a \rightarrow b \rightarrow d \rightarrow e = 9$

(3) Given graph:



Solution:

Consider 'a' as the source vertex

Tree vertices	Remaining vertices
1. a(-,0)	b(-,∞) c(-,∞) **d(a,7)** e(-,∞)
2. d(a,7)	b(d,7+2) = **b(d,9)** c(d,7+5) = c(d,12) e(-,∞)
3. b(d,9)	**c(d,12)** e(-,∞)
4. c(d,12)	e(c,12+6) = **e(c,18)**
5. e(c,18)	

The shortest paths are,

1. a to b = a → d → b = 9
2. a to c = a → d → c = 12
3. a to d = a → d = 7
4. a to e = a → d → c → e = 18

HUFFMAN ENCODING :

1. Fixed length encoding
2. Variable length encoding

In fixed length encoding, each character is assigned a bit string of same length $m >= \log_2 n$ where 'n' is the number of characters.

Ex: A B C D

00 01 10 11

$m = \log_2 4 = 2$

Each character here is replaced by 2 bits. Suppose the above text contains 4 characters and is transmitted 100 times then

$2 * 4 * 100 = 800$ bits

Suppose encoding was not used then,

$4 * 100 = 400$

$400 * 8 = 3200$ bits

Hence file size is compressed when we use encoding.

In variable length encoding, we assign codewords of different lengths to different characters. We use prefix codes i.e no codeword is a prefix of a codeword of another character.

Huffman's algorithm:

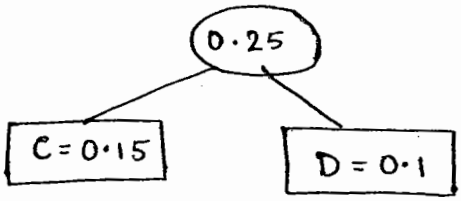
Step 1: Initialize n one-node trees and label them with the characters of the alphabet. Record the frequency of each character in the root of the tree to indicate the weight of the tree.

Step 2: Repeat until a single tree is obtained. Find 2 trees with smallest weights and record the sum of their weights in the root of the new tree.

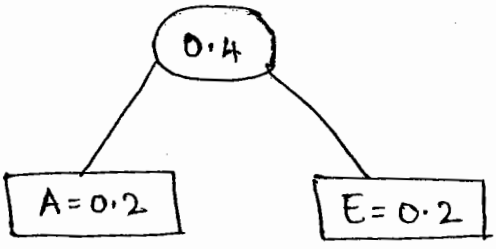
Ex: (1)

Character	A	B	C	D	E
Probability	0.2	0.35	0.15	0.1	0.2

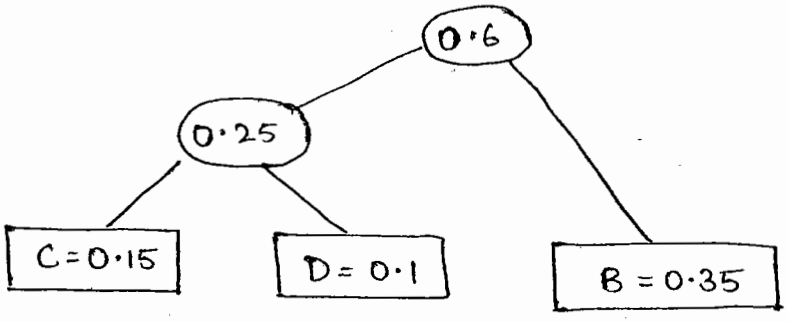
The least values here are C = 0.15 and D = 0.1

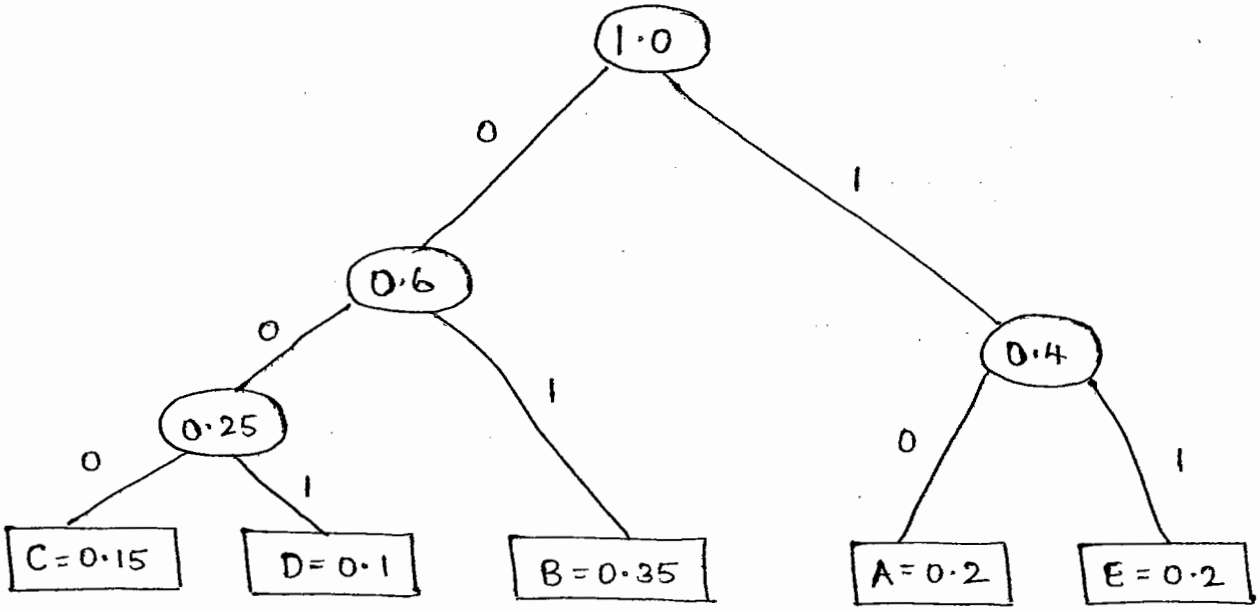


Then the least values are A = 0.2 and E = 0.2



Then the least values are 0.25 and B=0.35





Hence the codewords are,

A = 10
B = 01
C = 000
D = 001
E = 11

Usually characters that are used less frequency will have longer codeword whereas characters with high frequency will have smaller codeword.

For the above example,

A, B, E → 2 bits of codeword

C, D → 3 bits of codeword

2+2+2+3+3 = 12

Suppose it is transmitted 100 times then,

12 * 100 = 1200 bits

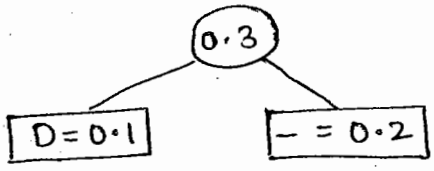
Dynamic Huffman encoding - coding tree is updated each time a new character is read from the source text.

(2) Given:

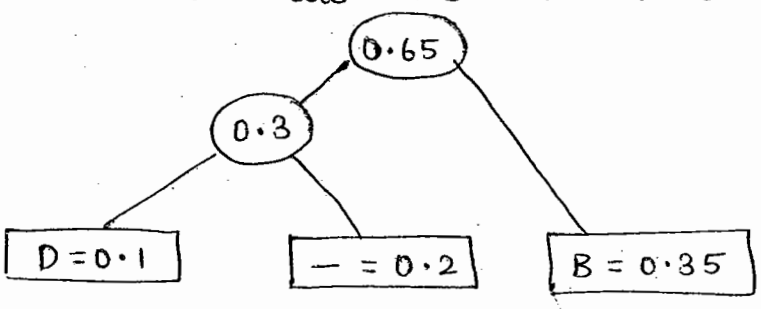
Character	A	B	C	D	E	–
Probability	0.5	0.35	0.5	0.1	0.4	0.2

Solution:

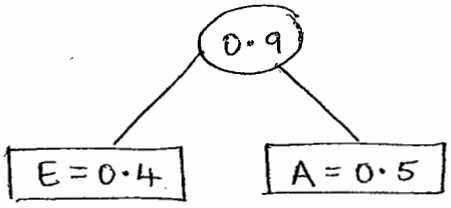
The least values are D = 0.1 and – = 0.2



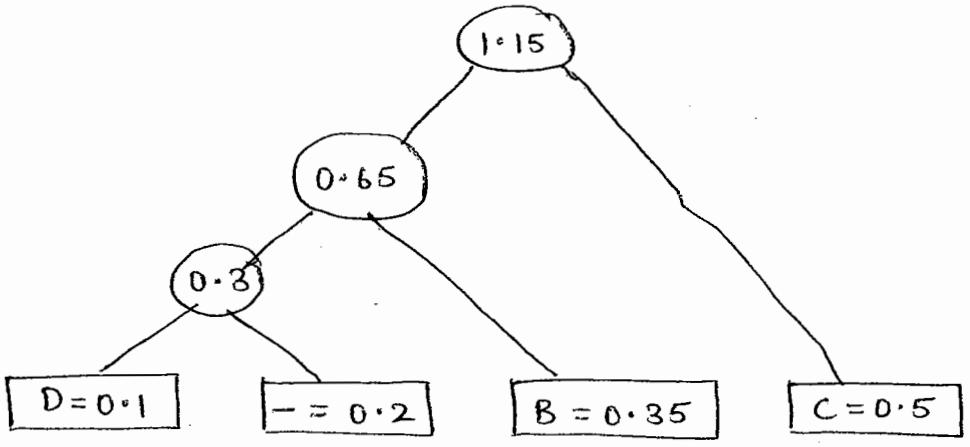
Next least values are 0.3 and B = 0.35



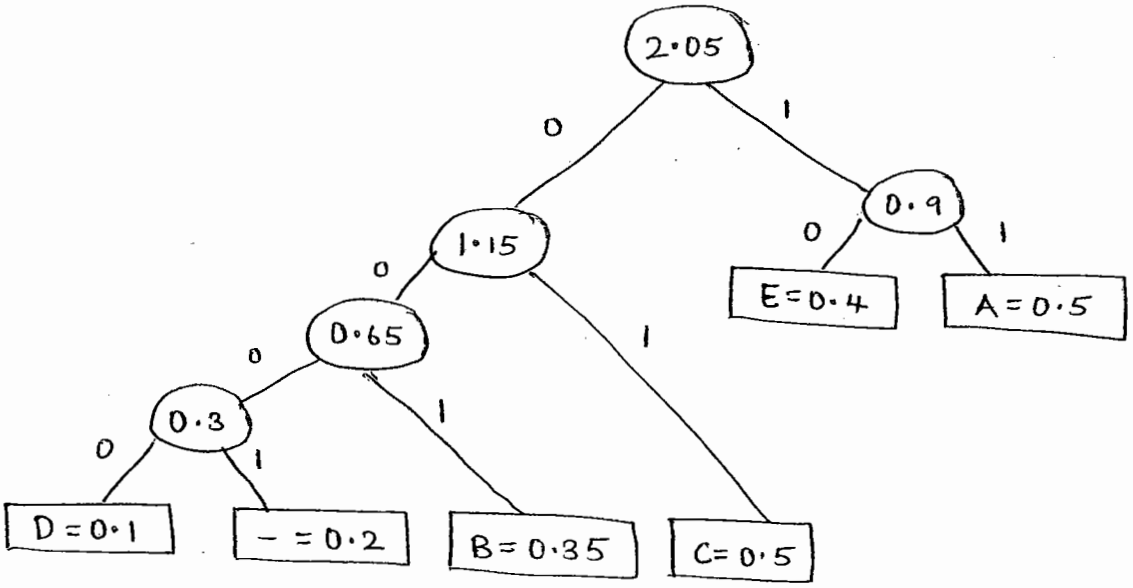
Next least values are E=0.4 and A=0.5



Next least values are C=0.5 and 0.65



Combining the above two we get,



The codewords are,

A = 11

B = 001

C = 01

D = 0000

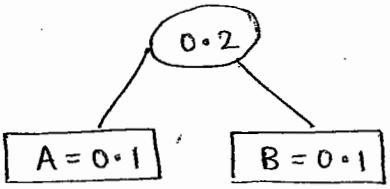
E = 10

(3) Given :

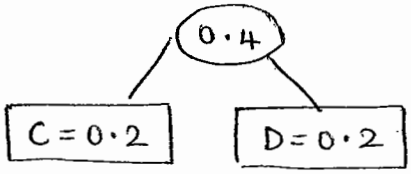
Character	A	B	C	D	E
Frequency	0·1	0·1	0·2	0·2	0·4

Solution :

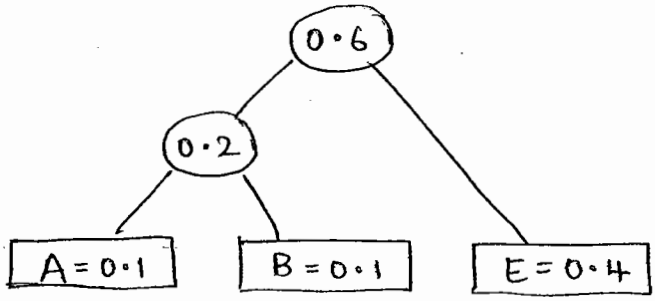
The least values are A = 0·1 and B = 0·1



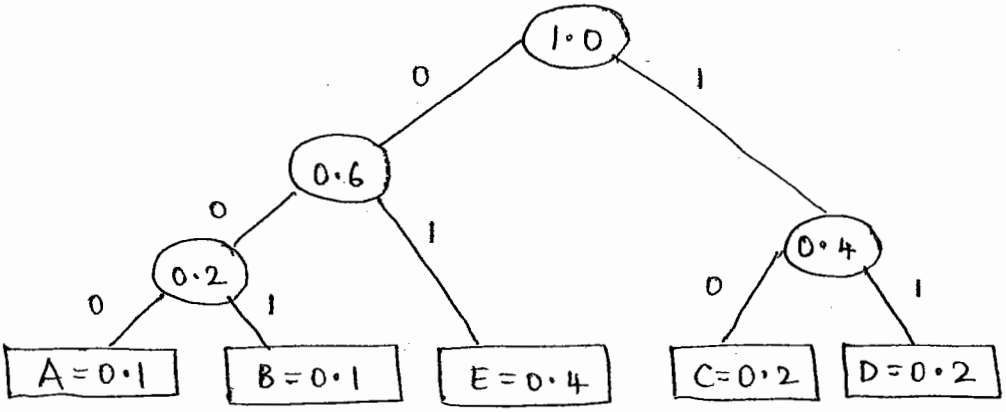
Next least values are C = 0·2 and D = 0·2



Next least values are 0·2 and E = 0·4



Next least values are 0·4 and 0·6



The codewords are,

A = 000

B = 001

C = 10

D = 11

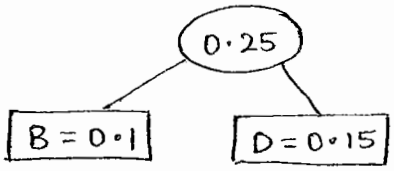
E = 01

(4) Given:

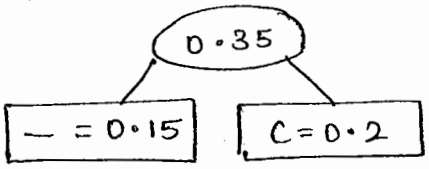
Character	A	B	C	D	_
Probability	0.4	0.1	0.2	0.15	0.15

Solution:

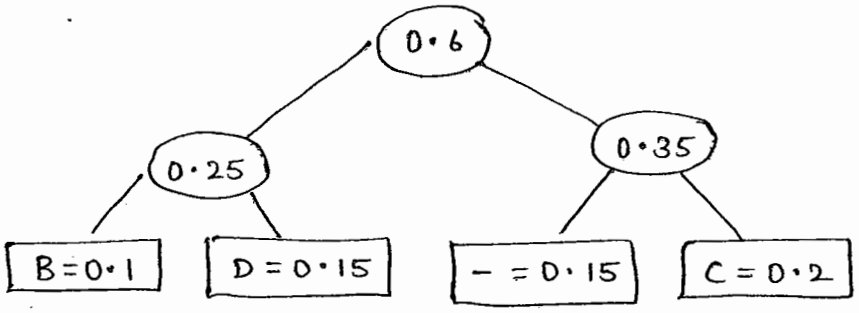
The least values are B and D,



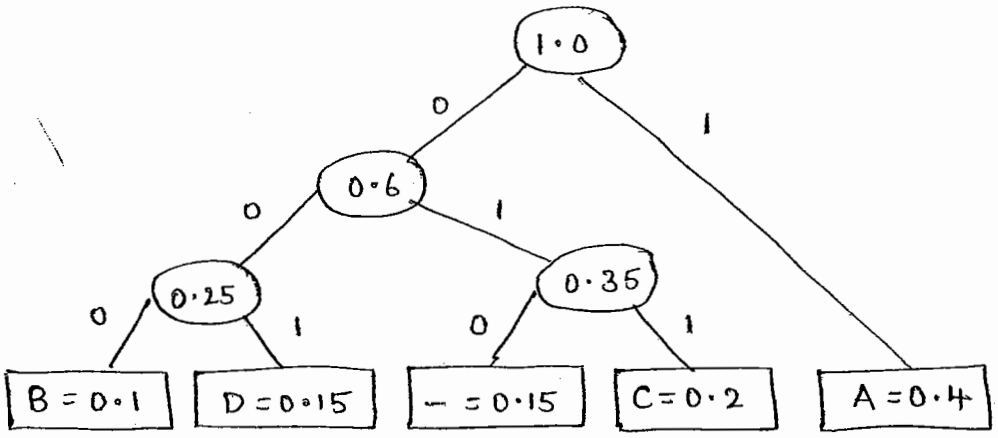
The next least values are _ = 0.15 and C = 0.2,



The next least values are 0.25 and 0.35,



The next least values are A=0.4 and 0.6



The codewords are,

A = 1

B = 000

C = 011

D = 001

– = 010

Encode the text ABACABAD

Encoded text : 1000101110001001

Decode the text : 1000101110010 10

Decoded text :

1	000	1	011	1	001	010
A	B	A	C	A	D	–

The decoded text is,

ABACAD–

KNAPSACK PROBLEM USING GREEDY TECHNIQUE :

If a fraction x_i, $0 \leq x_i \leq 1$ of object i is placed in the knapsack then a profit of $p_i x_i$ is earned.

$$\text{maximize} \sum_{1 \leq i \leq n} p_i x_i$$

$$\text{subject to} \sum_{1 \leq i \leq n} w_i x_i \leq m$$

$$0 \leq x_i \leq 1, \quad 1 \leq i \leq n$$

Ex: $n = 3$, $m = 20$

$(p_1, p_2, p_3) = (25, 24, 15)$

$(w_1, w_2, w_3) = (18, 15, 10)$

	(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum p_i x_i$
1.	$(1/2, 1/3, 1/4)$	16.5	24.25
2.	$(1, 2/15, 0)$	20	28.2
3.	$(0, 2/3, 1)$	20	31
4.	$(0, 1, 1/2)$	20	31.5

The optimal solution is 4.

```
Algorithm knapsack (m, n)
for i ← 1 to n
    x[i] ← 0
for i ← 1 to n
    if w[i] > m  break
    x[i] ← 1
    m = m - w[i]
```

```
if i<=n
x[i] = m/w[i]
end if
```

OPTIMAL STORAGE ON TAPES:

'n' programs are to be stored on a computer tape of length 'l'. Whenever a program is to be retrieved from this tape, the tape is initially positioned at the front. In optimal storage, we have to find a permutation for 'n' programs so that when they are stored on the tape mean retrieval time is minimized.

Ex: $n=3$

$(l_1, l_2, l_3) = (5, 10, 3)$

The various orderings are,

$\langle 1,2,3 \rangle = 5+5+10+5+10+3 = 38$

$\langle 1,3,2 \rangle = 5+5+3+5+3+10 = 31$

$\langle 2,1,3 \rangle = 10+10+5+10+5+3 = 43$

$\langle 2,3,1 \rangle = 10+10+3+10+3+5 = 41$

$\langle 3,1,2 \rangle = 3+3+5+3+5+10 = 29$

$\langle 3,2,1 \rangle = 3+3+10+3+10+5 = 34$

The optimal ordering is 3,1,2.

```
Algorithm optimal_storage(n,m)
// Inputs: n → number of programs
           m → number of tapes

j=0
for i ← 1 to n
    print "append program", i "to permutation for
```

$j \leftarrow (j+1) \bmod m$

end for

OPTIMAL MERGE PATTERNS :

Two sorted files containing 'n' and 'm' records could be merged together to obtain a sorted file in time $O(n+m)$

If files x_1, x_2, x_3, x_4 are to be merged, we may first merge x_1 and x_2 to get a file y_1. Then we could merge x_3 and y_1 to get file y_2. Then we merge x_4 and y_2 to get the desired sorted file. Given 'n' sorted files there are many ways to pair them and merge them into a sorted file.

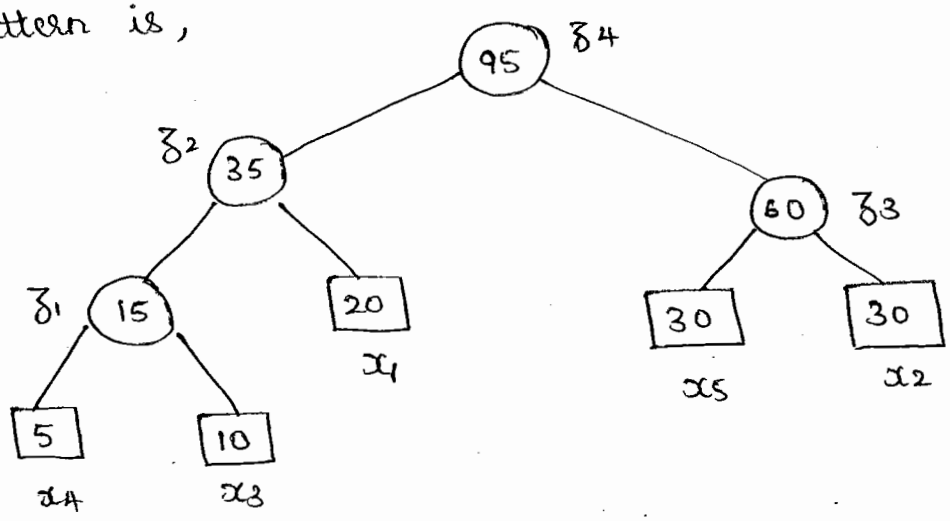
Ex:1. Files x_1, x_2, x_3 are 3 sorted files of length 30, 20, 10

If we merge x_1 and x_2 and then the result with x_3, we required $50+60 = 110$ moves.

Instead if we first merge x_2 and x_3 and then with x_1 we require $30+60 = 90$ moves.

2. $(x_1, x_2, x_3, x_4, x_5) = (20, 30, 10, 5, 30)$

The binary merge tree representing the merge pattern is,



An optimal two-way merge pattern corresponds to a binary merge tree with minimum weighted external path length.

```
treenode = record {
    treenode * lchild
    treenode * rchild
    weight
}
```

```
Algorithm tree(n)
    for i ← 1 to n-1
        pt = new treenode
      (pt → lchild) = least(list)
      (pt → rchild) = least(list)
      (pt → weight) = ((pt → lchild) → weight) +
                      ((pt → rchild) → weight)
      insert(list, pt)
    end for
    return least(list)
```

JOB SEQUENCING WITH DEADLINES :

n=4 and each job takes one unit of time.

$(p_1, p_2, p_3, p_4) = (100, 10, 15, 27)$

$(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$

Feasible solution	Sequence	Value
(1,2)	2,1	110
(1,3)	1,3 or 3,1	115
(1,4)	4,1	127
(2,3)	2,3	25

Feasible solution	Sequence	Value
(1)	1	100
(2)	2	10
(3)	3	15
(4)	4	27

Optimal solution is (1,4) = 127

or (4,1)

Algorithm job_sequencing (d, j, n)

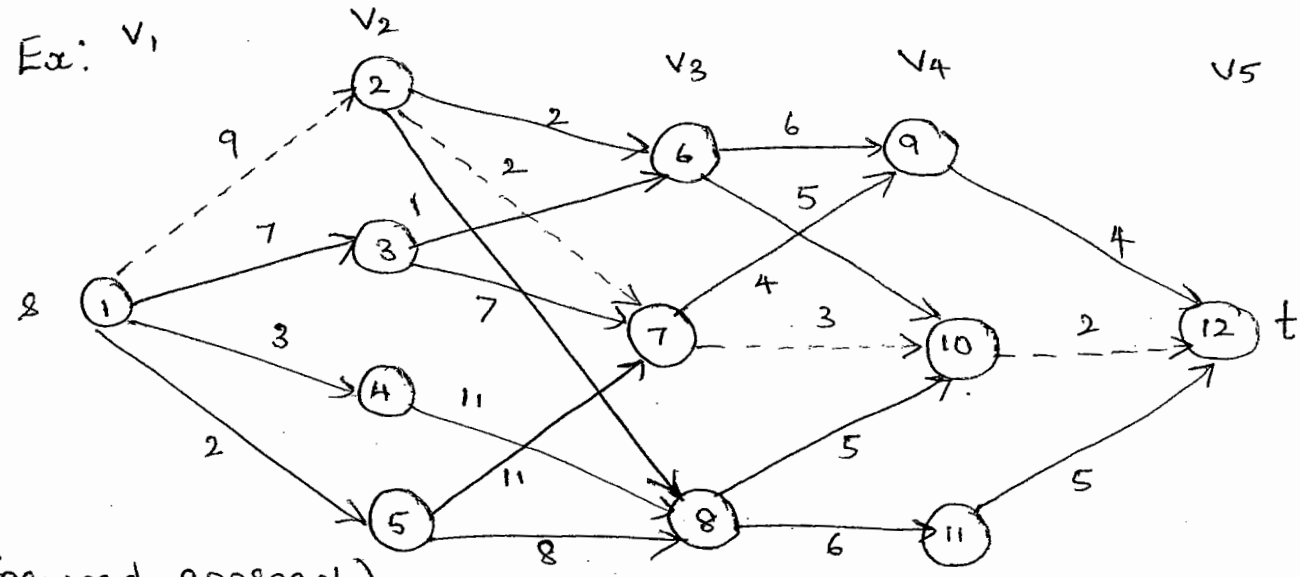
```
d[0] = J[0] = 0
J[1] = 1
K = 1
for i ← 2 to n
    r ← K
    while d[J[r]] > d[i] and d[J[r]] ≠ r
        r = r - 1
    if d[J[r]] <= d[i] and d[i] > r
        for q ← K to r+1
            J[q+1] ← J[q]
        J[r+1] ← i
        K ← K+1
    end for
end if
end for
return K
```

For the above example, select the job with max. profit initially i.e $p_1 = 100$, $d_1 = 2$, then $p_4 = 27$, $d_4 = 1$. But p_4 must be placed before p_1 to satisfy deadline constraints.

MULTISTAGE GRAPHS :

A multistage graph $G=(V,E)$ is a directed graph in which the vertices are partitioned into $K>=2$ disjoint sets V_i, $1 \le i \le K$. If (u,v) is an edge in E then $u \in V_i$ and $v \in V_{i+1}$. The sets V_1 and V_K are such that $|V_1| = |V_K| = 1$. Let s and t be vertices in V_1 and V_K. The vertex s is source and t is sink. Let $c(i,j)$ be the cost of edge (i,j). The cost of path from s to t is the sum of costs of the edges on the path. We have to find the minimum cost path.

Ex:



(Forward approach)

$$cost(i,j) = \min_{l \in V_{i+1}} \{ c(j,l) + cost(i+1,l) \}$$

$cost(3,6) = \min \{ 6 + cost(4,9), 5 + cost(4,10) \} = 7$

$cost(3,7) = \min \{ 4 + cost(4,9), 3 + cost(4,10) \} = 5$

$cost(3,8) = 7$

$cost(2,2) = \min \{ 4 + cost(3,6), 2 + cost(3,7), 1 + cost(3,8) \} = 7$

$cost(2,3) = 9$

$cost(2,4) = 18$

$cost(2,5) = 15$

$\min \{ 9 + cost(2,2), 7 + cost(2,3),$

$3+cost(2,4),\ 2+cost(2,5)\} = 16$

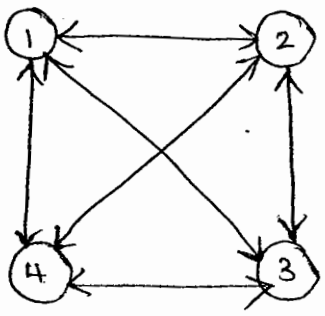
Resource allocation problem in which 'n' units of resource are to be allocated to 'r' projects is a multistage graph.

Algorithm multistage graph (G, k, n, p)

```
cost[n] ← 0
for j ← n-1 down to 1
    let r be a vertex such the (j,r) is an
    edge of G and c[j,r] + cost[r] is minimum
    cost[j] ← c[j,r] + cost[r]
    d[j] ← r
end for
p[1] ← 1
p[k] ← n
for j ← 2 to k-1
    p[j] ← d[p[j-1]]
```

TRAVELING SALESMAN PROBLEM USING DYNAMIC PROGRAMMING

Ex:



	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

We know that $g(i,S) = \min_{j \in S} \{c_{ij} + g(j, S-\{j\})\}$

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

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

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

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

$g(3,\{2\}) = 18 = c_{32} + g(2,\phi) = 13+5 = 18$

$g(4,\{2\}) = 13 = c_{42} + g(2,\phi) = 8+5 = 13$

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

$g(3,\{4\}) = 20 = c_{34} + g(4,\phi) = 12+8 = 20$

$g(4,\{3\}) = \boxed{15} = c_{43} + g(3,\phi) = 9+6 = 15$

We compute $g(i,S)$ with $|S|=2$, $i \neq 1$, $1 \notin S$ and $i \notin S$

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

$= 25 \qquad \min\{29, \boxed{25}\}$

$g(3,\{2,4\}) = \min\{c_{32} + g(2,\{4\}), c_{34} + g(4,\{2\})\}$

$= 25$

$g(4,\{2,3\}) = \min\{c_{42} + g(2,\{3\}), c_{43} + g(3,\{2\})\}$

$= 23$

$g(1,\{2,3,4\}) = \min\{c_{12} + g(2,\{3,4\}), c_{13} + g(3,\{2,4\}), c_{14} + g(4,\{2,3\})\}$

$= \min\{\boxed{35}, 40, 43\}$

$= 35$

The optimal tour is $1 \xrightarrow{10} 2 \xrightarrow{10} 4 \xrightarrow{9} 3 \xrightarrow{6} 1 = 35$

Time complexity : $O(n^2 2^n)$

Space needed : $O(n2^n)$ $\longrightarrow$ too large and hence a drawback.

The optimal tour starts at 1 - then reach 2 (since min. value is 35) - then reach 4 from 2 (since min. value is 25) - then reach 3 from 4 and then back to 1 from 3.

MULTISTAGE GRAPHS :

Directed graph $G = (V, E)$ in which vertices are partitioned into $K \geq 2$ disjoint sets V_i, $1 \leq i \leq K$.

If $\langle u, v \rangle$ is an edge in E, then $u \in V_i$ and $v \in V_{i+1}$ for some i, $1 \leq i \leq K$.

The sets V_1 and V_K are such that $|V_1| = |V_K| = 1$.

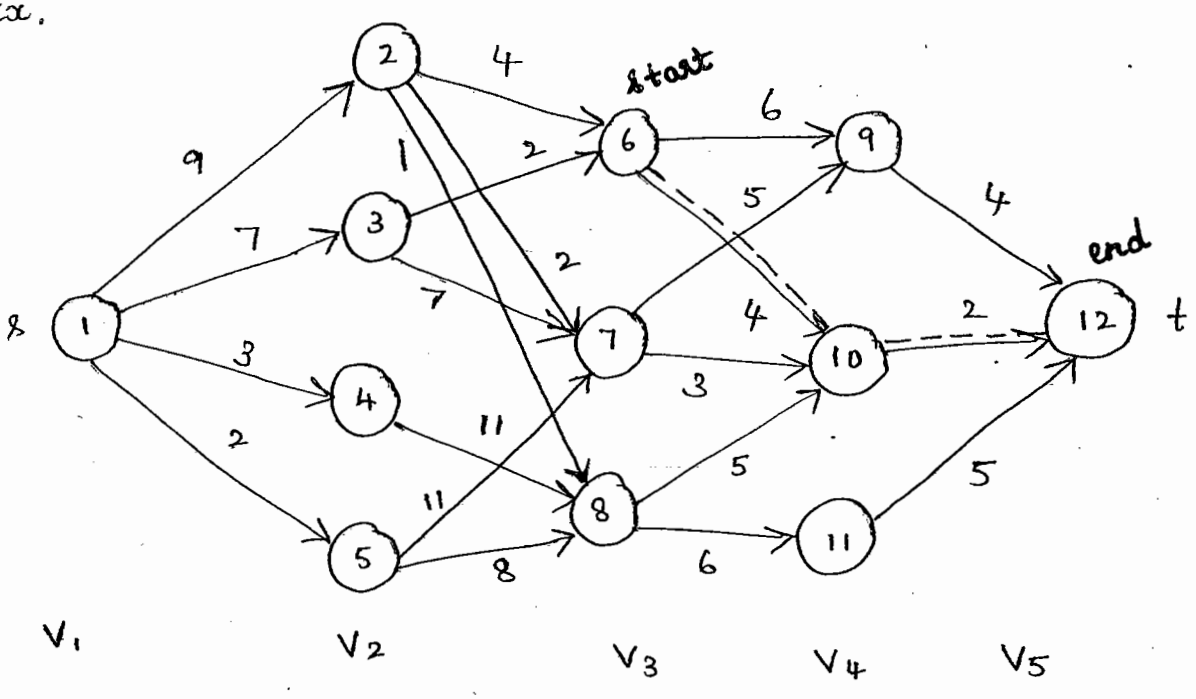
Let s = source $\in V_1$

t = sink / destination $\in V_K$.

The cost of a path from s to t is the sum of costs of all edges on the path.

Multistage graph problem is to find a minimum-cost path from s to t.

Ex:



Using forward approach,

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

For the above graph,

Consider cost (3, 6)
i j

where we want the distance from vertex 6 (j) in stage 3 (i) to 12 (t)

$$cost(3,6) = \min \{ c(6,9) + cost(4,9), \; c(6,10) + cost(4,10) \}$$

$$= \min \{ 6 + cost(4,9), \; 5 + cost(4,10) \}$$

cost (i, j) = cost (4, 9) denotes distance from vertex 9 (j) in stage 4 to 12 (t)

cost (4, 9) = 4 (only one path from 9 → 12)

Similarly cost (4, 10) = 2

(only one path from 10 → 12)

$$\therefore cost(3,6) = \min \{ 6+4, \; 5+2 \}$$

$$= \min \{ 10, 7 \}$$

$$= 7$$

Algorithm Multistage-Graph (G, K, n, p)

// Purpose: To find minimum cost path from source to sink for a multistage graph.

// Inputs: K-stage graph G = (V, E)

n = number of vertices

p[1:K] = minimum cost path

// Output: Minimum cost path

```
cost[n] ← 0
for j ← n-1 down to 1
        Let r be a vertex such that <j, r> is
        an edge of G and
        c[j, r] + cost[r] is minimum
        cost[j] ← c[j, r] + cost[r]
        d[j] ← r
end for
p[1] = 1
p[K] = n
for j ← 2 to K-1
        p[j] ← d[p[j-1]]
end for
```

JOB SEQUENCING WITH DEADLINES

Given: 'n' jobs

Associated with job 'i' is a deadline $d_i \geq 0$ and a profit $p_i > 0$.

For any job i, the profit p_i is earned iff the job is completed by it's deadline. To complete a job, one has to process the job on a machine for one unit of time. Only one machine is available for processing jobs.

Feasible solution: Subset of jobs such that each job in this subset is completed by it's deadline.

Value of a feasible solution = sum of profits of all jobs in the subset.

Ex: $n = 4$

$(p_1, p_2, p_3, p_4) = (100, 10, 15, 27)$

$(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$

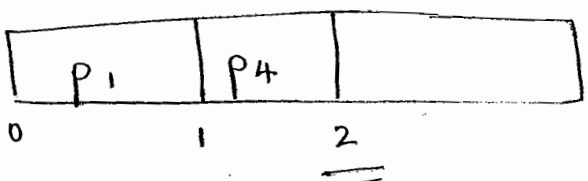
Using greedy technique,

Start with the job with maximum profit.

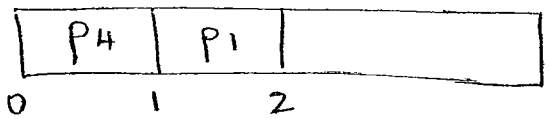
$p_1 = 100$, $d_1 = 2$

Then select p_4

$p_4 = 27$, $d_4 = 1$ ⟶ deadline constraint not satisfied



But when we place job 4 and then job 1,



The constraints are satisfied.

Profit = 100 + 27 = 127

↓

optimal solution

Other possible feasible solutions are,

$\langle 2,1 \rangle$	=	110
$\langle 1,3 \rangle$ or $\langle 3,1 \rangle$	=	115
$\langle 2,3 \rangle$	=	25
$\langle 4,3 \rangle$	=	42
$\langle 1 \rangle$	=	100
$\langle 2 \rangle$	=	10
$\langle 3 \rangle$	=	15
$\langle 4 \rangle$	=	27

Algorithm Job_Sequencing (d, J, n)

// Purpose: To find optimal profit for the given set of jobs in the deadline

// Inputs: Jobs' profits and deadlines

// Output: Subset of jobs to obtain maximum profit.

```
J ← {1}
for i ← 2 to n
    if all jobs in J ∪ {i} can be completed by
        their deadlines
            J ← J ∪ {i}
    end if
end for
```

OR

```
d[0] ← J[0] ← 0
J[1] ← 1   // By default select job 1
K ← 1
for i ← 2 to n
        r ← K
        while ( d[J[r]] > d[i]  and
                d[J[r]] ≠ r )
        end while      r ← r - 1
        if (d[J[r]] ≤ d[i]  and  d[i] > r)
                for q ← K to r+1
                        J[q+1] ← J[q]
                        J[r+1] ← i
                        K ← K+1
                end for
        end if
        return K
end for
```

Optimal storage on Tapes:

'n' programs are to be stored on a tape of length 'l'. Associated with each program 'i' is length 'li', $1 \le i \le n$.

Whenever a program is to be retrieved, the tape is always positioned at the front i.e 0.

Mean Retrieval Time (MRT): If programs are stored in the order $I = i_1, i_2, \ldots$ in the time needed to retrieve program i_j is proportional to $\sum_{1 \le K \le j} l_{iK}$.

If all programs are retrieved equally often,

$$MRT = \frac{1}{n} \sum_{1 \le j \le n} t_j$$

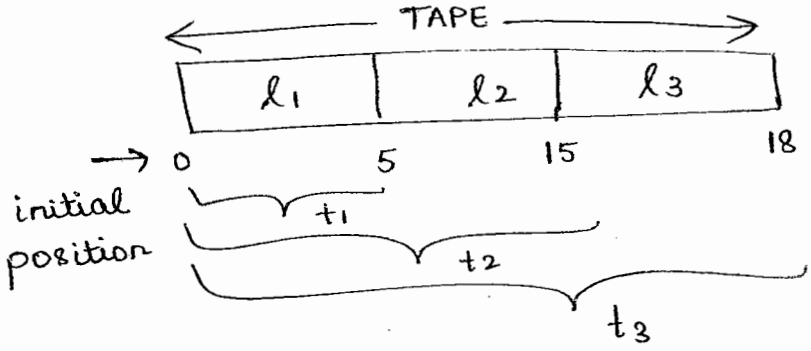
Ex: $n = 3$

$(l_1, l_2, l_3) = (5, 10, 3)$

There are $3! = 6$ possible orderings.

1. $\langle 1,2,3 \rangle = 5 + 5 + 10 + 5 + 10 + 3 = 38$
2. $\langle 1,3,2 \rangle = 5 + 5 + 3 + 5 + 3 + 10 = 31$
3. $\langle 2,1,3 \rangle = 10 + 10 + 5 + 10 + 5 + 3 = 43$
4. $\langle 2,3,1 \rangle = 10 + 10 + 3 + 10 + 3 + 5 = 41$
5. $\langle 3,1,2 \rangle = 3 + 3 + 5 + 3 + 5 + 10 = \boxed{29}$ → optimal solution
6. $\langle 3,2,1 \rangle = 3 + 3 + 10 + 3 + 10 + 5 = 34$

For the combination $<1,2,3>$)



t_1 = Retrieval time for l_1 (5)

t_2 = Retrieval time for l_2 (15 = 5+10)

t_3 = Retrieval time for l_3 (18 = 5+10+3)

Greedy method : Store programs in increasing order of their lengths.

Time complexity $\approx O(n \log n)$

Algorithm Optimal_Storage (n, m)

// Purpose : To obtain optimal storage of 'n' programs on 'm' tapes.

// Inputs : n → number of programs
m → number of tapes

// Output : Optimal solution with least value of retrieval time.

```
j ← 0
for i ← 1 to n
        append program 'i' to permutation for
        tape 'j'
        j ← (j+1) mod m
```

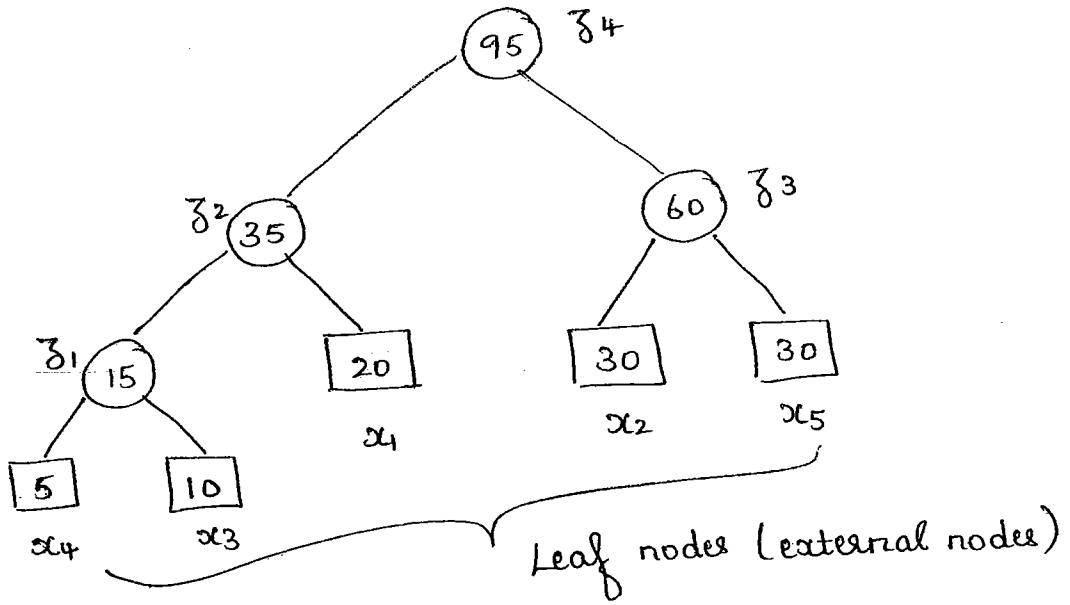
Optimal Merge Patterns:

Given 'n' sorted files, there are several ways of pairwise merging them into single sorted file. Different pairings require different amounts of computing time.

We need to obtain an optimal way - requiring least number of comparisons to pairwise merge 'n' sorted files.

Ex: $n = 5$

$(x_1, x_2, x_3, x_4, x_5) = (20, 30, 10, 5, 30)$



Greedy technique: At every step, two files with smallest sizes are merged.

Also known as two-way merge pattern.

Represented using binary merge tree.

Leaf nodes / external nodes → x_1, x_2, x_3, x_4, x_5 squares

Internal nodes → $\zeta_1, \zeta_2, \zeta_3, \zeta_4$

The total number of record moves for this binary merge tree is $\sum_{i=1}^{n} d_i q_i$

$d_i \rightarrow$ distance from root to external node for file x_i

$q_i \rightarrow$ length of x_i

For example, x_4 is at a distance of 3 from the root node 4.

Weighted external path length of the tree

$$= \sum_{i=1}^{n} d_i q_i$$

Algorithm:

```
treenode = record {
                treenode * lchild
                treenode * rchild
                integer weight
           }

for i ← 1 to n-1
        p ← new treenode
        (p → lchild) = Least (list)
        (p → rchild) = Least (list)
        (p → weight) = ((p → lchild) → weight) +
                       ((p → rchild) → weight)
        Insert (list, p)
end for
```

Knapsack Problem:

Given 'n' objects and knapsack of capacity 'm'.
Object 'i' has weight w_i and profit p_i.
If a fraction x_i, $0 \le x_i \le 1$ of object 'i' is placed in the knapsack, profit of $p_i x_i$ is earned.

$$\text{maximize} \sum_{1 \le i \le n} p_i x_i$$

$$\text{such that} \sum_{1 \le i \le n} w_i x_i \le m$$

$$\text{and} \quad 0 \le x_i \le 1, \quad 1 \le i \le n$$

Ex: $n = 3$, $m = 20$

$(p_1, p_2, p_3) = (25, 24, 15)$

$(w_1, w_2, w_3) = (18, 15, 10)$

Place object 1 with $p_1 = 25$ and $w_1 = 18$

Remaining capacity $= 20 - 18 = 2$

Place $2/15$ th of object 2.

Remaining capacity $= 0$

Total profit $= 25 + \frac{2}{15} \times 24$ (8)

$= 25 + \frac{16}{5}$ (5)

$= \frac{125 + 16}{5} = \frac{141}{5} = 28.2$

Place object 2 with $p_2 = 24$ and $w_2 = 15$

Remaining capacity = 5

Place 1/2 of object 3

Total profit $= 24 + \frac{15}{2}$

$= 24 + 7.5$

$= 31.5$

↓

optimal solution

In greedy method, compute p_i/w_i

Object 1 = $25/18 = 1.4$

Object 2 = $24/15 = 1.6$

Object 3 = $15/10 = 1.5$

Arrange in non-increasing order, (decreasing)

Object 2
Object 3
Object 1

↓ Selection order for optimal profit

Algorithm knapsack-greedy (m, n)

// Purpose: To obtain optimal solution for knapsack problem using greedy technique.

// Inputs: n → number of objects

m → capacity of knapsack

Weights and profits of 'n' objects

// Output: Optimal profit

```
for i ← 1 to n
        x[i] ← 0
end for
U ← m
for i ← 1 to n
        if (w[i] > U)
                break
        x[i] ← 1
        U ← U - w[i]
end for
if (i <= n)
        x[i] ← U / w[i]
end if
```