

Department of MCA

LECTURE NOTE

ON

ANALYSIS AND DESIGN OF ALGORITHMS

(MCA 4th Sem)

COURSE CODE: MCA-209

Prepared by :

Mrs. Sasmita Acharya

Assistant Professor

Department of MCA

VSSUT, Burla.

Prerequisite: Familiarity with Discrete Mathematical Structures, and Data Structures.

UNIT I: (10 Hours)

Algorithms and Complexity: Asymptotic notations, orders, worst-case and average-case, amortized complexity.

Basic Techniques: divide & conquer, dynamic programming, greedy method, backtracking.

UNIT II: (10 Hours)

Branch and bound, randomization.

Data Structures: heaps, search trees, union-find problems.

Applications: sorting & searching, combinatorial problems.

UNIT III: (10 Hours)

Optimization problems, computational geometric problems, string matching.

Graph Algorithms: BFS and DFS, connected components.

UNIT IV: (10 Hours)

Spanning trees, shortest paths, MAX-flow.

NP- completeness, Approximation algorithms.

Text Book:

1. Introduction to Algorithms, 2/e, T.H.Cormen, C.E.Leiserson, R.L.Rivest and C.Stein, PHI Pvt. Ltd. / Pearson Education

Reference Books:

1. Algorithm Design: Foundations, Analysis and Internet examples, M.T.Goodrich and R.Tomassia, John Wiley and sons.

2. Fundamentals of Computer Algorithms, Ellis Horowitz, Satraj Sahnii and Rajasekharam, Galgotia Publications Pvt. Ltd.

Course outcomes:

1. To be able to analyze correctness and the running time of the basic algorithms for those classic problems in various domains and to be able to apply the algorithms and design techniques for advanced data structures.
2. To be able to analyze the complexities of various problems in different domains. and to be able to demonstrate how the algorithms are used in different problem domains.
3. To be able to design efficient algorithms using standard algorithm design techniques and demonstrate a number of standard algorithms for problems in fundamental areas in computer science and engineering such as sorting, searching and problems involving graphs.

Contents

Module I

Algorithms and Complexity.....3

Basic Techniques.....8

Module II

Branch and bound.....14

Data Structures.....18

Sorting & Searching.....26

Module III

Optimization problems.....28

Computational geometric problems.....28

String matching.....33

Graph Algorithms.....37

Module IV

Spanning trees.....43

Max-flow.....46

NP – completeness.....48

ANALYSIS AND DESIGN OF ALGORITHM

Module I

Algorithm:-

Informally an algorithm is any well-defined computational procedure that takes some value or set of values as input and produces some value or set of values as output.

Running time:-

The running time of an algorithm on a particular input is the number of primitive operations or steps executed.

When we look at input sizes large enough to make only the order of growth of the running time relevant we are studying the asymptotic efficiency of algorithms.

Asymptotic notation:-

They are used to describe the asymptotic running time of an algorithm. They are defined in terms of function whose domains are the set of natural numbers $N=\{0,1,2,\dots\}$. Such notations are convenient for describing the worst case running time function which is defined only on integer input sizes. There are 5 notations :-

- (Theta) θ -notation
- (big-oh) O -notation
- (big-omega) Ω -notation
- (small-oh) o -notation
- (small-omega) ω -notation

(Theta) θ -notation:-

This notation asymptotically bounds a function from above and below. For a given function $g(n)$ we denote by $\theta(g(n))$ is given by

$\theta(g(n)) = \{f(n): \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \}$.

For all values of n to the right of n_0 the value of $f(n)$ lies at or above $c_1 g(n)$ or at below $c_2 g(n)$. We say that $g(n)$ is an asymptotically tight bound for $f(n)$ where $c_1 g(n)$ is the lower bound and $c_2 g(n)$ is the upper bound. Definition of $\theta(g(n))$ is required by every member $f(n)$ which is element of $\theta(g(n))$ asymptotically non-negative.

(Big-oh) O -notation:-

This notation used when we have only an asymptotic upper bound. For a given function $g(n)$ we denote by $O(g(n))$ the set of functions ,

$O(g(n)) = \{f(n) : \text{there exist a positive constant } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \}$.

(Big-omega)Ω-notation:-

It provides an asymptotic lower bound. For a given function $g(n)$ we denote by $\Omega(g(n))$ the set of functions

$\Omega(g(n)) = \{f(n) : \text{there exist a positive constant } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \}$.

O and Ω notation are used for average and worst case. Ω notation are used for best case running time.

(small-oh)o-notation:-

The asymptotic upper bound provided by big-oh notation may or may not be asymptotically tight.

Example:- The bound $2n^2 = o(n^2)$ is asymptotically tight but $2n = o(n^2)$ is not.

We use small-oh notation to denote an upper bound that is not asymptotically tight. We define $o(g(n))$ as the set of functions

$o(g(n)) = \{f(n) : \text{for any positive constant } c > 0 \text{ there exist a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0 \}$.

(small-omega)ω-notation:-

We use this notation to denote a lower bound that is not asymptotically tight. We define $\omega(g(n))$ as the set

$\omega(g(n)) = \{f(n) : \text{for any positive constant } c > 0 \text{ there exist a constant } n_0 > 0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \}$.

Order of growth:-

Example:- Arrange the following

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

Answer:- order of growth is

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

Rate of growth:- It refers to the change in the running time of an algorithm as the input size increases.

Amortized analysis:-

Here the time require to perform a sequence of data structure operations is averaged over all the operations performed. Here probability is not involved. The 3 common techniques for amortized analysis are :-

- Aggregate analysis
- Accounting method
- Potential method

General analysis:-

Here an upper bound $T(n)$ on the total cost of a sequence of 'n' operations determined. The average cost for operation is $\frac{T(n)}{n}$.

Example:- Stack operations

2 stack operations are PUSH(s, x) that pus x onto stack s & POP(s) that pops the top most element of stack. Another operation is MULTIPOP(s, k) that simultaneously pop k items from the stack

1. PUSH(s, x)
2. POP(s)
3. MULTIPOP(s, k)
While not STACK_EMPTY(s) and $k \neq 0$ do POP(s)
 $K \leftarrow k-1$

Aggregate analysis:-

Using this we can get a better upper bound that considers the entire sequence of 'n' operations. Although a single multipop operation can be expensive. Any sequence of n push, pop, and multipop operation on an initial empty stack and cost at most $O(n)$ because each object can be pop at most once for each time it is pushed.

For any value of n, any sequence of n push, pop and multipop operations takes a total of $O(n)$ time. So avg. cost of an operation is $\frac{O(n)}{n} = O(1)$. This is equal to the amortized cost of each operation.

Accounting method:-

In this method we assign different charges to different operation with some operations charged more or less than they actually cost. The amount we charge an operation is called amortized cost. When an operation's amortized cost exceeds its actual cost the difference is assigned to specific objects in the data structure as credit.

Example:-

	Stack	Actual cost	Amortized cost
PUSH		1	2
POP		1	0
MULTIPOP	min(k, s)		0

This credit can be used to later on to help pay for operations whose amortized cost is less than their actual cost. If we denote the actual cost of the i^{th} operation by C_i and amortized cost of the operation by \hat{C}_i

$\sum_{i=1}^n \hat{C}_i \geq \sum_{i=1}^n C_i$ we require this for all operations.

The total credit stored in the data structure is the difference between the total amortized cost and the total actual cost.

Total credit = Total amortized cost - Total actual cost

As the amortized cost is greater than or equal to actual cost the total credit associated with the data structure must be non-negative at all times.

Potential method:-

Instead of representing prepaid work as credit stored with specific objects in the data structure this method represents the prepaid work as potential energy or just potential that can be released to pay for future operation.

This potential is associated with the data structure as a whole rather than with specific object within the data structure. We start with an initial data structure ' D_0 ' on which n operations are performed. For each $i=1, 2 \dots n$. Let C_i is the actual cost of the operation & D_i is the data structure that results after applying the i^{th} operation to the data structure D_{i-1} .

A potential function \emptyset maps each data structure D_i to a real number $\emptyset(D_i)$ which is the potential associated with data structure D_i . The amortized cost of the i^{th} operation with respect to potential function is defined by :-

Amortized cost = actual cost + increase in potential

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

The total amortized cost of n operations is

$$\Rightarrow \sum_{i=1}^n \hat{C}_i = \sum_{i=1}^n (C_i + \emptyset(D_i) - \emptyset(D_{i-1}))$$

$$= \sum_{i=1}^n C_i + \emptyset(D_n) - \emptyset(D_0)$$

Example:- Stack operations

Recurrences:-

When an algorithm contains a recursive call to itself its running time can be described by recurrence. A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs. There are 3 methods to solve a recurrence

1. Recursion tree method
2. Substitution method
3. Master theorem

Master theorem:-

It provides a cook book method for solving recurrences of the form is

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

Where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function.

This equation describes the running time of an algorithm that divides a problem of size 'n' into 'a' sub problems each of size n/b . The cost of dividing the problem and combining the results of the sub problems is given by the function $f(n)$. Then $T(n)$ can be bounded asymptotically as follows:

Case-1

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

Case-2

If $f(n) = \theta(n^{\log_b(a)})$, then $T(n) = \theta(n^{\log_b(a)} \lg n)$

Case-3

If $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ for some constant $\epsilon > 0$ & if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and sufficiently large 'n' then $T(n) = \theta(f(n))$

Substitution method:-

It consists of 2 steps:-

- Guess the form of the solution
- Use mathematical induction to find the constant and show that the solution works

Example:

Use the substitution method to show that $T(n) \in O(n)$

$$\begin{array}{l} T(1) = 3 \\ T(n) = 2T(n/2) + 5 \end{array}$$

1. Guess:

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

By definition of Big-O, must find $c > 0$ and $n \geq n_0$

$$0 \leq T(n) \leq cn$$

2. $0 \leq T(k/2) \leq ck/2$

$$\text{Show: } T(k) = 2T(k/2) + 5 \leq ck$$

3. Substitution

$$T(k) = 2T(k/2) + 5 \quad \text{Recurrence definition}$$

$$\leq 2[ck/2] + 5 \quad \text{IH substitution}$$

$$= 2c k/2 + 5$$

$$= ck + 5$$

$$\leq ck \quad \text{Show } T(k) \leq ck$$

Find constant c so the last two lines hold.

$$ck + 5 \leq ck \quad \text{Not possible for } c > 0 \text{ and } k \geq 1$$

$$5 \leq 0 \quad \text{Subtract } ck$$

Fails to satisfy the substitution

Algorithm and design technique:-

There are different techniques to design an algorithm:

1. Divide and conquer
2. Dynamic programming
3. Greedy method
4. Backtracking
5. Branch and bound

1. Divide and conquer:-

Many algorithms are recursive in structure. To solve a given problem they call themselves recursively one or more time to deal with closely related sub problems. These algorithms follow a divide and conquer approach. It involves 3 steps

- Divide
- Conquer
- Combine

Divide: - divide the problem into a number of sub problem.

Conquer: - Conquer the sub problems by solving them recursively.

Combine the solution to the sub problems into the solution for the original problem.

Example: - Merge sort

Let $T(n)$ be the running time of an problem of size n . If the problem size is small for some constant c that is $n \leq c$, the straight forward solution takes constant time $\theta(1)$.

Suppose our division of the problem gives a sub problem each of which is $(1/b)^{\text{th}}$ size. Let $D(n)$ be the time to divide the problem into sub problem and $C(n)$ be the time to combine the solution to the sub problem into the solution to the original sub problem. The recurrence is given by:-

$$T(n) = \{\theta(1) \text{ if } n \leq c$$

$$\{aT(n/b) + D(n) + C(n) \text{ otherwise}$$

Dynamic programming:-

Divide and conquer algorithm partition the problem into independent sub problem. Solve the sub problems recursively and then combine their solution to solve the original sub problem. But the dynamic programming is applicable when the sub problems are not independent that is when sub problems share sub sub problems.

A DPA algorithm solves every sub sub problems just once and saves its answers in a table avoiding the work of re-computation. It applies to optimization problems in which in which a set of choices must be in order to arrive at an optimal solution. The development of DP algorithm can be broken into a sequence of 4 steps.

- Characterize the structure of n optimal solution
- Recursively define the value of an optimal solution
- Compute the value of an optimal solution in a bottom up fashion
- Construct the optimal solution from computed information

Elements of dynamic programming:-

The 2 key ingredients that an optimization problem must have in order for DP to be applicable are

- **Optimal substructure**
A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solution to sub problems
- **Overlapping sub problem**
When a recursive algorithm revisits a same problem over and over again we say that the optimization problem has overlapping sub problems.

Matrix chain multiplication:-

It can be stated as given a chain of 'n' matrices $\langle A_1 A_2 \dots A_n \rangle$ where for $i=1,2, \dots n$, matrix A_i has dimensions $A_i \leftarrow p_{i-1} \times p_i$ fully parenthesize the product $A_1, A_2 \dots A_n$ in way that minimizes the number of scalar multiplication.

Step-1: structure of an optimal parenthesizes

The optimal substructure of this problem is suppose optimal parenthesizes of $A_i, A_{i+1} \dots A_j$ splits the product between A_k and A_{k+1} . Then the parenthesize of the prefix sub chain $A_i, A_{i+1} \dots A_k$ within the optimal parenthesize of $A_i, A_{i+1} \dots A_j$ must be optimal. Also the parenthesize of the sub chain of $A_{k+1}, A_{k+2} \dots A_j$ must be optimal.

Step-2: Recursive solution

Let $m[i, j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_i \dots A_j$. The recursive formula is

$$M[i, j] = \begin{cases} 0 & \text{if } i=j \\ \min_{i \leq k \leq j} \{m[i, k] + m[k+1, j] + p_i - 1 p_k p_j\} & \text{if } i < j \end{cases}$$

Step-3: computing the optimal cost

It is done by a tabular bottom approach
optimal parenthesize algorithm

- Pop(s, i, j)
- If (i=j)
Then print A_i
- Else print (
Pop(s, i, s[i, j])
Pop(s, s[i, j]+1, j)
- Print)

Longest common sub-sequences (LCS):-

Given 2 sequences X and Y , we say that a sequence Z is a common sub-sequence of X and Y if Z is a sub-sequence of both X and Y. In LCS problem given 2 sequences X and Y , we wish to find a maximum length common sub-sequence of X and Y.

Step-1: characterizing a longest common sub-sequence

Let $X = \langle x_1, x_2, \dots, x_n \rangle$ and $Y = \langle y_1, y_2, \dots, y_n \rangle$ be sequences. Let $Z = \langle z_1, z_2, \dots, z_n \rangle$ be any LCS of X and Y.

Case-1: If $x_m = y_n$ then $z_k = x_m = y_n$ and z_{k-1} is an LCS of x_{m-1} and y_{n-1} .

Case-2: $x_m \neq y_n$ then $z_k \neq x_m$ and z is an LCS of x_{m-1} and y .

Case-3: $x_m \neq y_n$ then $z_k \neq y_n$ and z is an LCS of x and y_{n-1} .

Step-2: recursive solution

Let $c[i, j]$ be the length of an LCS of sequences x_i, y_j . optimal sub-structure of the LCS problem gives the recursive formula.

$$C[i, j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \end{cases}$$

$$C[i-1, j-1] + 1 \text{ if } i, j > 0 \text{ and } x_i = y_j$$

$$\text{Max}(c[i, j-1], c[i-1, j]) \text{ if } i, j > 0 \text{ and } x_i \neq y_j$$

Step-3: computing the length of an LCS

It takes 2 sub sequences as inputs. It stores the $c[i, j]$ values in a table entries are computed in RMO.

Optimal binary search tree:-

We are given a sequence $k = \{k_1, k_2, \dots\}$ of n distinct keys in sorted order such that $k_1 < k_2 < \dots < k_n$ and we wish to build a binary search tree from these keys . for each key we have a probability p_i that a search will be for k_i . Some searches may be for values not in k , so we have $(n+1)$ dummy keys $\{d_0, \dots, d_n\}$ representing values not in k .

d_0 represents all value less than k_1 . d_n represents all value greater than k_n . For each dummy key d_i we have probability q_i that a search will correspond to d_i .each key k_i is an internal node and each dummy key is a leaf. Every search is either successful (finding some key) or unsuccessful (finding some dummy key).

For given set of probabilities our goal is to construct BST whose expected search cost is smallest such a tree is called as an optimal BST.

Step-1: structure of an optimal BST

The optimal sub structure property is given by if an optimal BST has a sub tree containing keys k_i to k_j then this sub tree must be optimal for the sub problem with keys k_i to k_j and dummy keys d_{i-1} to d_j .

Step-2: Recursive solution

The $e[i, j]$ values gives the expected search cost in optimal BST. The recursive formula is given by

Greedy algorithm:-

A greedy algorithm always makes the choice that looks best at the moment. That is it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution. These algorithms do not always yield optimal solution.

Example:-Minimum spanning tree algorithms

Elements of greedy strategy

1. *Optimal substructure*:- a problem exhibits optimal sub structure if an optimal solution to the problem contains within it optimal solution to sub problems.
2. *Greedy choice property*:- a globally optimal solution can be arrived at by making a locally optimal greedy choice. When we are considering which choice to make we make the choice that looks best in the current problem without considering results from sub problems.

Design of greedy algorithm

- Subset paradigm

The greedy method suggest that one can devise an algorithm that works in stages considering one input at a time. At each decision is made regarding whether a particular input is in an optimal solution. This is done by considering the inputs in order determined by some selection procedure.

If the inclusion of the next input into the partially constructed optimal solution will result in an infeasible solution then this input is not added to the partial solution otherwise it is added. This version of the greedy technique is called the subset paradigm.

Example: knapsack problem

- Ordering paradigm

For problems that do not call for the selection of an optimal subset in the greedy method we make decisions by considering the inputs in some order. Each decision is

made using an optimization criterion that can be computed using decisions already made. This version of greedy method is called ordering paradigm.

Example: single source shortest path problem

Huffman codes:-

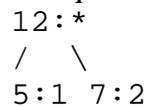
They are widely used and very effective technique for data. We consider the data to be sequence of characters. Huffman greedy algorithm uses a table of the frequencies of occurrence of the characters to build up an optimal way of representing each character as a binary string.

Example:

Let's say you have a set of numbers and their frequency of use and want to create a Huffman encoding for them:

	FREQUENCY	VALUE
5	1	2
	7	3
	10	4
	15	5
	20	6
	45	

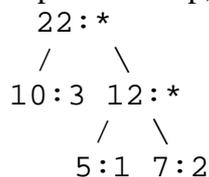
Creating a Huffman tree is simple. Sort this list by frequency and make the two-lowest elements into leaves, creating a parent node with a frequency that is the sum of the two lower element's frequencies:



The two elements are removed from the list and the new parent node, with frequency 12, is inserted into the list by frequency. So now the list, sorted by frequency, is:

- 10:3
- 12:*
- 15:4
- 20:5
- 45:6

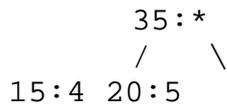
You then repeat the loop, combining the two lowest elements. This results in:



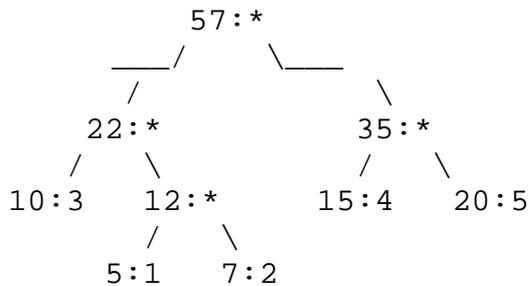
And the list is now:

- 15:4
- 20:5
- 22:*
- 45:6

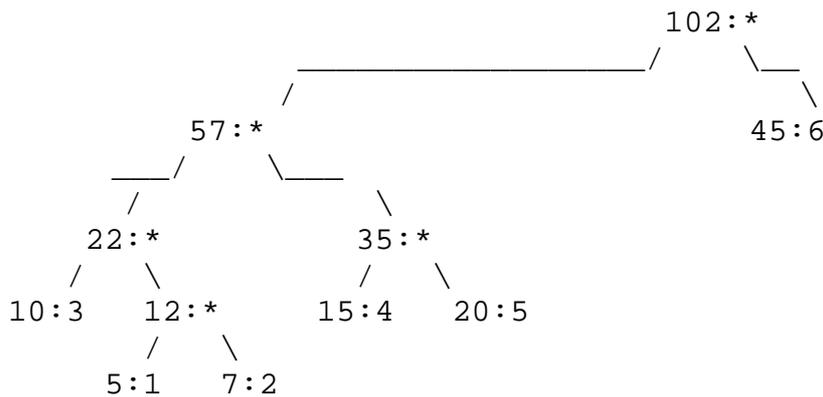
You repeat until there is only one element left in the list.



22: *
35: *
45: 6



45: 6
57: *



Decoding a Huffman encoding is just as easy as you read bits in from your input stream you traverse the tree beginning at the root, taking the left hand path if you read a 0 and the right hand path if you read a 1. When you hit a leaf, you have found the code.

Backtracking:-

Many problem which deal with searching for a set of solutions or which asks for an optimal solution satisfying some constraints can be solved using the backtracking formula. The name backtracking was first coined by D.H. Lehman in 1950s.

Many application of the backtrack method the desired solution is expressible as an n-tuple $(x_1, x_2 \dots x_n)$ where the x_i are chosen from some finite set s_i . Often the problem to be solved calls for finding one vector that maximizes or minimizes a criterion function $p(x_1, x_2 \dots x_n)$.

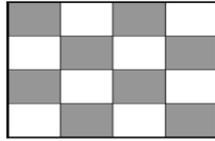
The basic idea of backtracking algorithm is to build up the solution vector one component at a time and to use modified criterion functions to test whether the vector being formed has any chance of success. The major advantage is that if it is realised that the partial vector can

no way lead to an optimal solution then the rest of the test vectors can be ignored completely.

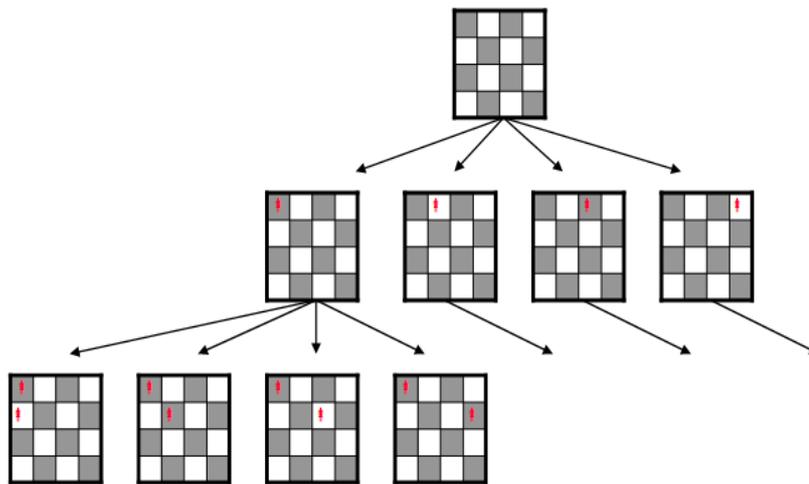
Example: N-Queens Problem

Given an $N \times N$ sized chess board

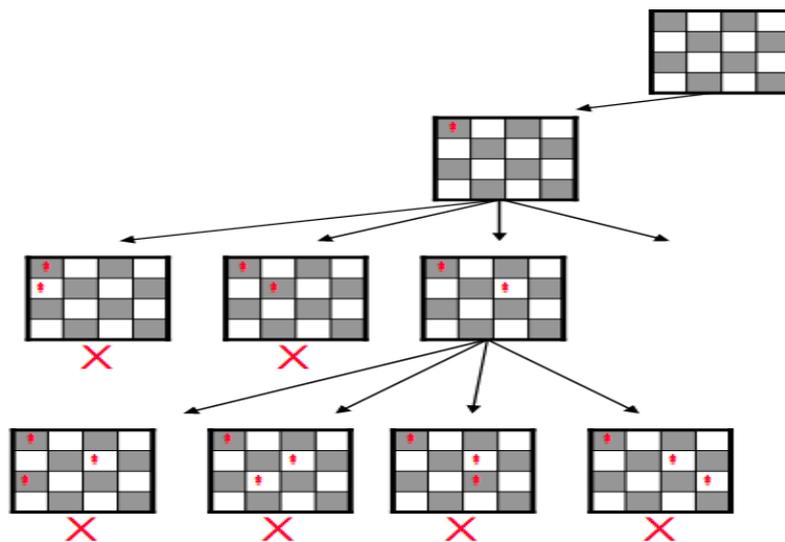
Objective: Place N queens on the board so that no queens are in danger



One option would be to generate a tree of every possible board layout This would be an expensive way to find a solution



Backtracking prunes entire sub trees if their root node is not a viable solution. The algorithm will “backtrack” up the tree to search for other possible solutions



Module- 2

Branch and bound:-

The term branch and bound refers to all state space search method in which all children of the E-node are generated before any other live node can become the E-node. A BFS like state space search will be called FIFO search as the list of live nodes is a FIFO list or queue. A DFS search like state space search is called LIFO search as the list of live nodes is a LIFO list or a stack.

Least-cost search: Here a function is used to select the live node. The node with least cost function value is selected as the live node. Bounding functions are used to help avoid the generations of sub trees that do not contain an answer node.

In both LIFO and FIFO branch & bound the selection rule for the next E-node is rigid and in a sense blind. It does not give any preference to a node that very good chance of getting the search to a answer node quickly.

The search for an answer node can be speeded by using an intelligent ranking function for live nodes. Here the next E-node is selected on the basis of this ranking function. The ideal way to assign ranks to nodes is on the basis of the additional computational effort or cost needed to reach an answer node from the live node.

For any node x the cost could be:

1. The number of nodes in the sub tree x that need to be generated before an answer node is generated.
2. The number of levels the nearest answer node is from x.

Let $\hat{g}(x)$ be an estimate of the additional effort needed to reach an answer node from x. Node x is assigned a rank using a function $\hat{C}()$ such that

$$\hat{C}(x) = f(h(x)) + \hat{g}(x)$$

Where $h(x)$ is the cost of reaching x from the root and $f()$ is any non-decreasing function.

A search strategy that uses such a cost function select the next E-node would always choose the node with least value of $\hat{C}(x)$ as a live node. So such a search strategy is called a LC-search.

Let us consider an example of 15-puzzle. We are defined with the start state and goal state as shown in the figure below.

1	2	3	4
5	6		8
9	10	7	11

13	14	15	12
----	----	----	----

Start state

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

goal state

Solution:

1	2	3	4
5	6		8
9	10	7	11
13	14	15	12

1	2	3	4
5	6	7	8
9	10		11
13	14	15	12

Step1 step-2

1	2	3	4
5	6	7	8
9	10	11	
13	14	15	12

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Step-3step-4(target state)

Randomization: -

A randomized algorithm is an algorithm that employs a degree of randomness as part of its logic. The algorithm typically uses uniformly bits as an auxiliary input to guide its behaviour, in the hope of achieving good performance in the "average case" over all possible choices of random bits. Formally, the algorithm's performance will be a random variable determined by the random bits; thus either the running time, or the output (or both) are random variables. Example of randomized algorithm is Quicksort.

Quicksort:-

Quicksort is a familiar, commonly used algorithm in which randomness can be useful. Any deterministic version of this algorithm requires $O(n^2)$ time to sort n numbers for some well-defined class of degenerate inputs (such as an already sorted array), with the specific class of inputs that generate this behaviour defined by the protocol for pivot selection. However, if the algorithm selects pivot elements uniformly at random, it has a provably high probability of finishing in $O(n \log n)$ time regardless of the characteristics of the input.

Data Structure:-

Heap Sort:-

If you have values in a heap and remove them one at a time they come out in (reverse) sorted order. Since a heap has worst case complexity of $O(\log(n))$ it can get $O(n\log(n))$ to remove n value that are sorted.

There are a few areas that we want to make this work well:

- how do we form the heap efficiently?
- how can we use the input array to avoid extra memory usage?
- how do we get the result in the normal sorted order?

If we achieve it all then we have a worst case $O(n\log(n))$ sort that does not use extra memory. This is the best theoretically for a comparison sort.

The steps of the heap sort algorithm are:

1. Use data to form a heap
2. remove highest priority item from heap (largest)
3. reform heap with remaining data

You repeat steps 2 & 3 until you finish all the data.

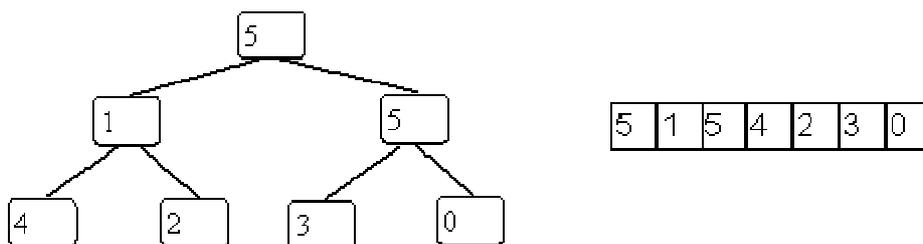
You could do step 1 by inserting the items one at a time into the heap:

- This would be $O(n\log(n))$. Turns out we can do in $O(n)$. This does not change the overall complexity but is more efficient.
- You would have to modify the normal heap implementation to avoid needing a second array.

Instead we will enter all values and make it into a heap in one pass.

As with other heap operations, we first make it a complete binary tree and then fix up so the ordering is correct. We have already seen that there is a relationship between a complete binary tree and an array.

Our standard sorting example becomes:

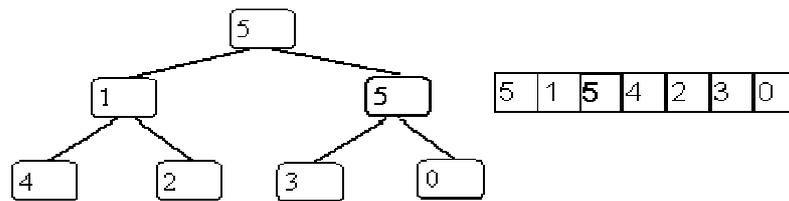


Now we need to get the ordering correct.

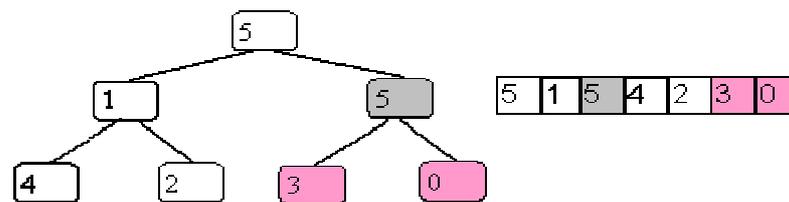
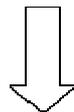
It will work by letting the smaller values percolate down the tree.

To make into a heap you use an algorithm that fixes the lower part of the tree and works its way toward the root:

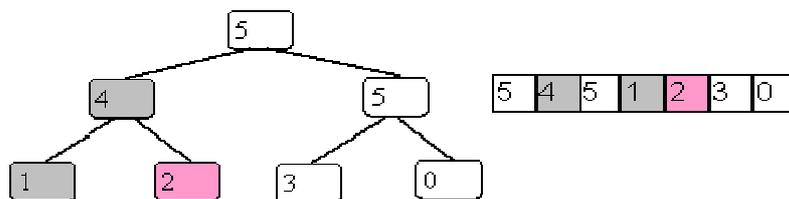
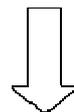
- Go from lowest right parent (non-leaf) and proceed to left. When finish one level go to next starting again from right.
- at each node, percolate down the item to its proper place in this part of the subtree, e.g., subheap. Here is how the example goes:



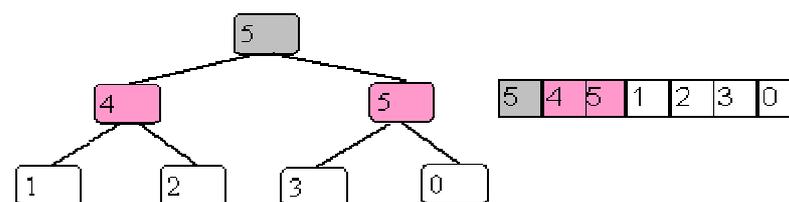
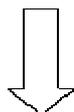
start at 5 (in second level).
It is larger than all values
in subtree so ok.



fix 1: swap with 4 since
largest child



fix 5 (root): not smaller
than children so ok

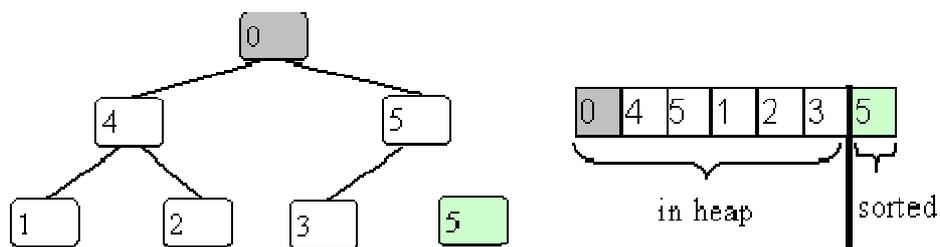


This example has very few swaps. In some cases you have to percolate a value down by swapping it with several children.

The Weiss book has the details to show that this is worst case $O(n)$ complexity. It isn't $O(n \log(n))$ because each step is $\log(\text{subtree height currently considering})$ and most of the nodes root subtrees with a small height. For example, about half the nodes have no children (are leaves).

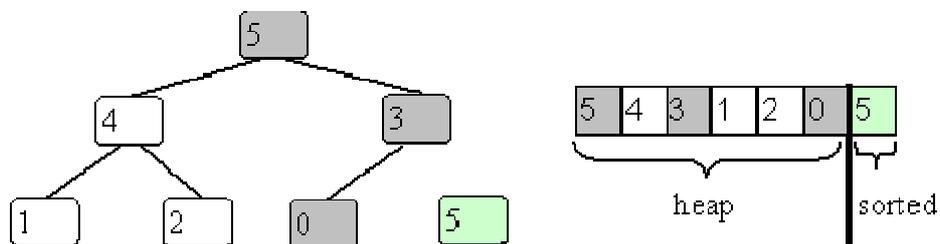
Now that we have a heap, we just remove the items one after another.

The only new twist here is to keep the removed item in the space of the original array. To do this you swap the largest item (at root) with the last item (lower right in heap). In our example this gives:

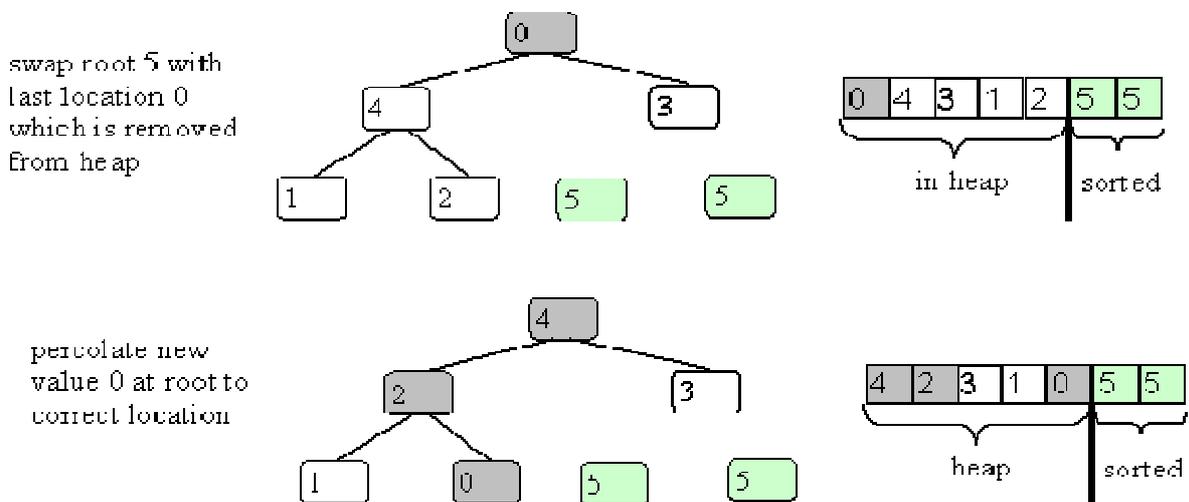


The last value of 5 is no longer in the heap.

Now let the new value at the root percolate down to where it belongs.

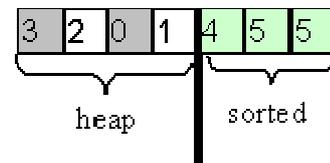
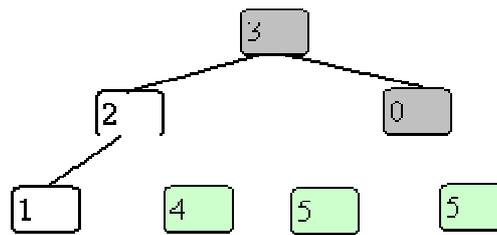


Now repeat with the new root value (just chance it is 5 again):

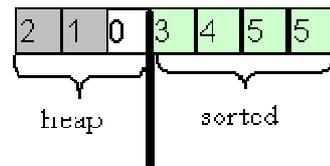
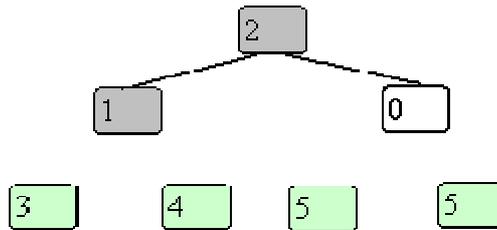


And keep continuing:

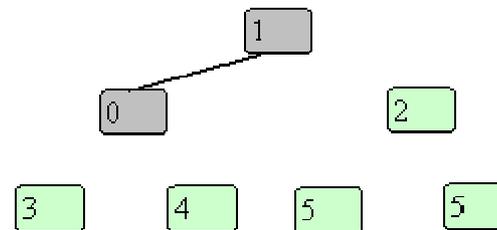
swap root 4 with last location 0 which is removed from heap. Let new root percolate to correct location



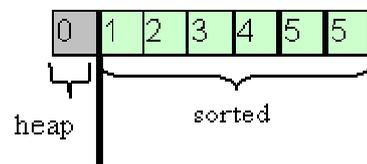
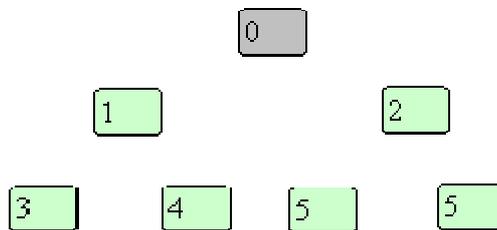
swap root 3 with last location 1 which is removed from heap. Let new root percolate to correct location



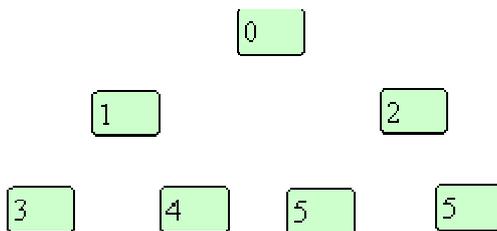
swap root 2 with last location 0 which is removed from heap. Let new root percolate to correct location



swap root 1 with last location 0 which is removed from heap. Let new root percolate to correct location



one value left in heap so done



Heap Complexity:-

The part just shown very similar to removal from a heap which is $O(\log(n))$. You do it $n-1$ times so it is $O(n\log(n))$. The last steps are cheaper but for the reverse reason from the building of the heap, most are $\log(n)$ so it is $O(n\log(n))$ overall for this part. The build part was $O(n)$ so it does not dominate. For the whole heap sort you get $O(n\log(n))$.

There is no extra memory except a few for local temporaries.

Thus, we have finally achieved a comparison sort that uses no extra memory and is $O(n \log(n))$ in the worst case.

In many cases people still use quick sort because it uses no extra memory and is usually $O(n \log(n))$. Quick sort runs faster than heap sort in practice. The worst case of $O(n^2)$ is not seen in practice.

Search Tree:-

Search tree is a tree data structure used for locating specific values from within a set. In order for a tree to function as a search tree, the key for each node must be greater than any keys in subtrees on the left and less than any keys in subtrees on the right.

The advantage of search trees is their efficient search time given the tree is reasonably balanced, which is to say the leaves at either end are of comparable depths. Various search-tree data structures exist, several of which also allow efficient insertion and deletion of elements, which operations then have to maintain tree balance.

Optimal substructure of a shortest path:-

Shortest path algorithm rely on the property that a shortest path between two vertices contain other shortest path with in it.

- Dijkstra's algorithm
- Floyd-warshall algorithm

Dijkstra's algorithm:-

Dijkstra's algorithm is very similar to Prim's algorithm for minimum spanning tree. Like Prim's MST, we generate a *SPT (shortest path tree)* with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has minimum distance from source.

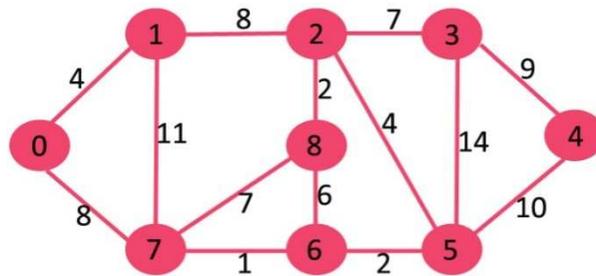
Below are the detailed steps used in Dijkstra's algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

Algorithm:-

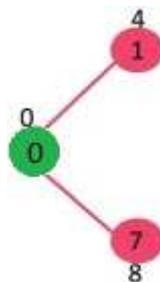
- 1) Create a set *sptSet* (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.
- 2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.
- 3) While *sptSet* doesn't include all vertices
 -a) Pick a vertex *u* which is not there in *sptSet* and has minimum distance value.
 -b) Include *u* to *sptSet*.

....c) Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

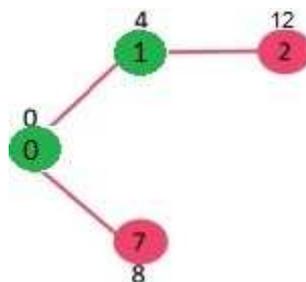
Let us understand with the following example:



The set *sptSet* is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum distance value. The vertex 0 is picked, include it in *sptSet*. So *sptSet* becomes {0}. After including 0 to *sptSet*, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green color.

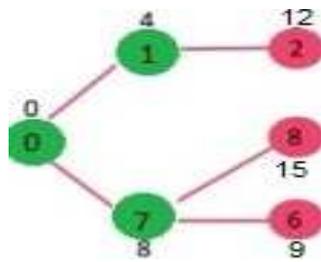


Pick the vertex with minimum distance value and not already included in SPT (not in *sptSet*). The vertex 1 is picked and added to *sptSet*. So *sptSet* now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.

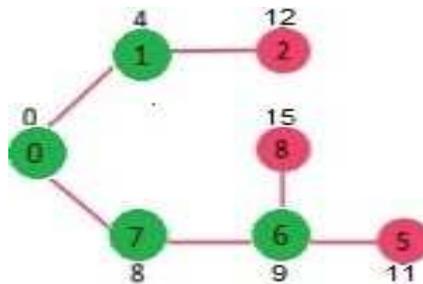


Pick the vertex with minimum distance value and not already included in SPT (not in *sptSet*). Vertex 7 is picked. So *sptSet* now becomes {0, 1, 7}. Update the distance values of

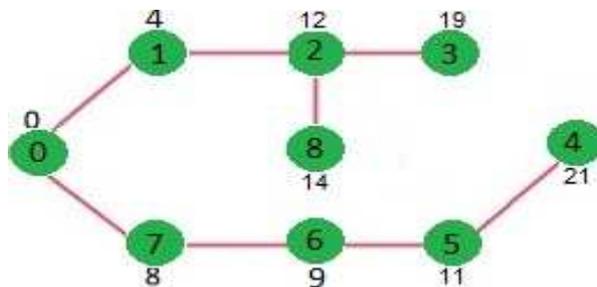
adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).



Pick the vertex with minimum distance value and not already included in SPT (not in *sptSet*). Vertex 6 is picked. So *sptSet* now becomes {0, 1, 7, 6}. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



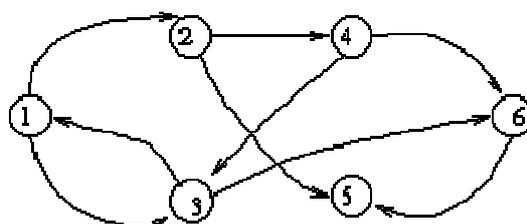
We repeat the above steps until *sptSet* doesn't include all vertices of given graph. Finally, we get the following Shortest Path Tree (SPT).



Floyd-warshall algorithm:-

This algorithm simply applies the rule *n* times, each time considering a new vertex through which possible paths may go. At the end, all paths have been discovered.

Let's look at an example of this algorithm. Consider the following graph:



So we have $V = \{ 1, 2, 3, 4, 5, 6 \}$ and $E = \{ (1, 2), (1, 3), (2, 4), (2, 5), (3, 1), (3, 6), (4, 6), (4, 3), (6, 5) \}$. Here is the adjacency matrix and corresponding $t^{(0)}$:

down = "from"
across = "to"

adjacency matrix for G:

	1	2	3	4	5	6		1	2	3	4	5	6
	1	0	1	1	0	0		1	1	1	1	0	0
	2	0	0	0	1	1		2	0	1	0	1	1
$t^{(0)}$:	3	1	0	0	0	0	1	3	1	0	1	0	0
	4	0	0	1	0	0	1	4	0	0	1	1	0
	5	0	0	0	0	0	0	5	0	0	0	0	1
	6	0	0	0	0	1	0	6	0	0	0	0	1

Now let's look at what happens as we let k go from 1 to 6:

$k = 1$

add (3,2); go from 3 through 1 to 2

	1	2	3	4	5	6
	1	1	1	1	0	0
	2	0	1	0	1	1
$t^{(1)}$ =	3	1	1	1	0	0
	4	0	0	1	1	0
	5	0	0	0	0	1
	6	0	0	0	0	1

$k = 2$

add (1,4); go from 1 through 2 to 4

add (1,5); go from 1 through 2 to 5

add (3,4); go from 3 through 2 to 4

add (3,5); go from 3 through 2 to 5

	1	2	3	4	5	6
	1	1	1	1	1	1
	2	0	1	0	1	1
$t^{(2)}$ =	3	1	1	1	1	1
	4	0	0	1	1	0
	5	0	0	0	0	1
	6	0	0	0	0	1

$k = 3$

add (1,6); go from 1 through 3 to 6

add (4,1); go from 4 through 3 to 1

add (4,2); go from 4 through 3 to 2

add (4,5); go from 4 through 3 to 5

	1	2	3	4	5	6
	1	1	1	1	1	1
	2	0	1	0	1	1
$t^{(3)}$ =	3	1	1	1	1	1
	4	1	1	1	1	1
	5	0	0	0	0	1
	6	0	0	0	0	1

k = 4

add (2,1); go from 2 through 4 to 1

add (2,3); go from 2 through 4 to 3

add (2,6); go from 2 through 4 to 6

```
      1 2 3 4 5 6
      1 1 1 1 1 1
      2 1 1 1 1 1
t(4) = 3 1 1 1 1 1
      4 1 1 1 1 1
      5 0 0 0 1 0
      6 0 0 0 1 1
```

k = 5

```
      1 2 3 4 5 6
      1 1 1 1 1 1
      2 1 1 1 1 1
t(5) = 3 1 1 1 1 1
      4 1 1 1 1 1
      5 0 0 0 1 0
      6 0 0 0 1 1
```

k = 6

```
      1 2 3 4 5 6
      1 1 1 1 1 1
      2 1 1 1 1 1
t(6) = 3 1 1 1 1 1
      4 1 1 1 1 1
      5 0 0 0 1 0
      6 0 0 0 1 1
```

At the end, the transitive closure is a graph with a complete subgraph (a *clique*) involving vertices 1, 2, 3, and 4. You can get to 5 from everywhere, but you can get nowhere from 5. You can get to 6 from everywhere except for 5, and from 6 only to 5. **Analysis** This algorithm has three nested loops containing a $\Theta(1)$ core, so it takes $\Theta(n^3)$ time.

What about storage? It might seem with all these matrices we would need $\Theta(n^3)$ storage; however, note that at any point in the algorithm, we only need the last two matrices computed, so we can re-use the storage from the other matrices, bringing the storage complexity down to $\Theta(n^2)$.

Sorting & Searching:-

Sorting:-

Several algorithms are presented, including insertion sort, shell sort, and quicksort. Sorting by insertion is the simplest method, and doesn't require any additional storage. Shell sort is a simple modification that improves performance significantly. Probably the most efficient and popular method is quicksort, and is the method of choice for large arrays

Insertion Sort:-

One of the simplest methods to sort an array is an insertion sort. An example of an insertion sort occurs in everyday life while playing cards. To sort the cards in your hand you

extract a card, shift the remaining cards, and then insert the extracted card in the correct place. This process is repeated until all the cards are in the correct sequence. Both average and worst-case time is $O(n^2)$.

Shell Sort:-

Shell sort, developed by Donald L. Shell, is a non-stable in-place sort. Shell sort improves on the efficiency of insertion sort by quickly shifting values to their destination. Average sort time is $O(n^{1.25})$, while worst-case time is $O(n^{1.5})$.

Quicksort:-

Although the shell sort algorithm is significantly better than insertion sort, there is still room for improvement. One of the most popular sorting algorithms is quicksort. Quicksort executes in $O(n \lg n)$ on average, and $O(n^2)$ in the worst-case. However, with proper precautions, worst-case behaviour is very unlikely. Quicksort is a non-stable sort. It is not an in-place sort as stack space is required.

Searching:-

Hash Tables:-

Hash tables are a simple and effective method to implement dictionaries. Average time to search for an element is $O(1)$, while worst-case time is $O(n)$.

Binary Search Trees:-

In the Introduction, we used the binary search algorithm to find data stored in an array. This method is very effective, as each iteration reduced the number of items to search by one-half. However, since data was stored in an array, insertions and deletions were not efficient. Binary search trees store data in nodes that are linked in a tree-like fashion. For randomly inserted data, search time is $O(\lg n)$. Worst-case behaviour occurs when ordered data is inserted. In this case the search time is $O(n)$.

Module III

Optimization Problem:-

An optimization problem is the problem of finding the *best* solution from all feasible solutions. Optimization problems can be divided into two categories depending on whether the variables are continuous or discrete. An optimization problem with discrete variables is known as a combinatorial optimization problem.

Combinatorial Optimization Problem:-

Formally, a combinatorial optimization problem \mathcal{A} is a quadruple (I, f, m, g) , where

- I is a set of instances;
- given an instance $x \in I$, $f(x)$ is the set of feasible solutions;
- Given an instance x and a feasible solution y of x , $m(x, y)$ denotes the measure of y , which is usually a positive real.
- g is the goal function, and is either **min** or **max**.

The goal is then to find for some instance x an *optimal solution*, that is, a feasible solution y with

$$m(x, y) = g\{m(x, y') \mid y' \in f(x)\}.$$

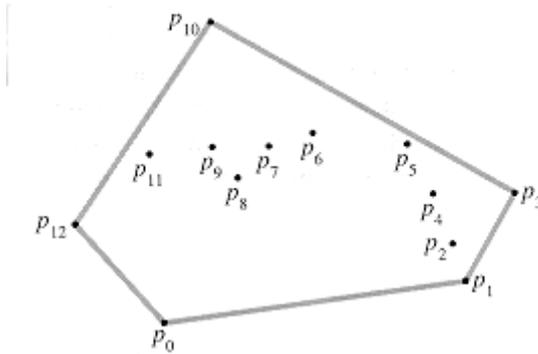
For each combinatorial optimization problem, there is a corresponding decision problem that asks whether there is a feasible solution for some particular measure m_0 . For example, if there is a graph G which contains vertices u and v , an optimization problem might be "find a path from u to v that uses the fewest edges". This problem might have an answer of, say, 4. A corresponding decision problem would be "is there a path from u to v that uses 10 or fewer edges?" This problem can be answered with a simple 'yes' or 'no'.

In the field of approximation algorithms, algorithms are designed to find near-optimal solutions to hard problems. The usual decision version is then an inadequate definition of the problem since it only specifies acceptable solutions. Even though we could introduce suitable decision problems, the problem is more naturally characterized as an optimization problem.

Computational Geometric Problems:-

This is the branch of computer science that studies algorithms for solving geometric problems. It has applications in computer graphics, robotics, VLSI design, computer-aided design and statistics. The input to a computational geometric problem is a description of a set of geometric objects such as a set of points, a set of line segments or vertices of a polygon in clockwise order. The output is a response to a query about an object such as whether any of the lines intersect or a new geometric object such as a convex hull (small enclosure).

convex polygon) of a set of points. Each input object is represented as a set of points $\{P_1, P_2, \dots\}$ where $P_i = \{x_i, y_i\}$ and $x_i, y_i \in \mathbb{R}$, $\mathbb{R} =$ set of real number.



Line Segment Properties:-

Cross Product:-

Computing cross products is at the heart of our line-segment methods. Consider vectors p_1 and p_2 , shown in Figure (a). The cross product $p_1 \times p_2$ can be interpreted as the signed area of the parallelogram formed by the points $(0, 0)$, p_1 , p_2 , and $p_1 + p_2 = (x_1 + x_2, y_1 + y_2)$. An equivalent, but more useful, definition gives the cross product as the determinant of

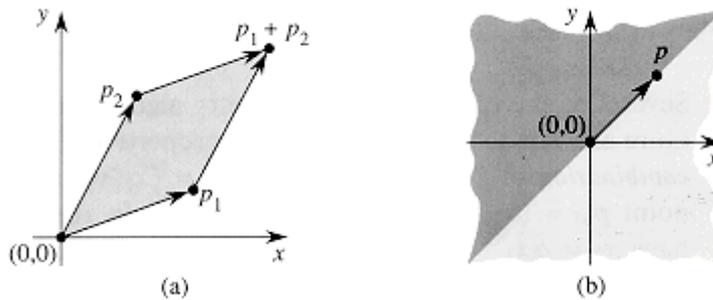


Figure 35.1 (a) The cross product of vectors p_1 and p_2 is the signed area of the parallelogram. (b) The lightly shaded region contains vectors that are clockwise from p . The darkly shaded region contains vectors that are counterclockwise from p .

a matrix:¹

$$\begin{aligned}
 p_1 \times p_2 &= \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} \\
 &= x_1 y_2 - x_2 y_1 \\
 &= -p_2 \times p_1 .
 \end{aligned}$$

¹ Actually, the cross product is a three-dimensional concept. It is a vector that is perpendicular to both p_1 and p_2 according to the "right-hand rule" and whose magnitude is

$|x_1y_2 - x_2y_1|$. In this chapter, however, it will prove convenient to treat the cross product simply as the value $x_1y_2 - x_2y_1$.

If $p_1 \times p_2$ is positive, then p_1 is clockwise from p_2 with respect to the origin $(0, 0)$; if this cross product is negative, then p_1 is counter clockwise from p_2 . Figure (b) shows the clockwise and counter clockwise regions relative to a vector p . A boundary condition arises if the cross product is zero; in this case, the vectors are collinear, pointing in either the same or opposite directions.

To determine whether a directed segment $\overrightarrow{p_0p_1}$ is clockwise from a directed segment $\overrightarrow{p_0p_2}$ with respect to their common endpoint p_0 , we simply translate to use p_0 as the origin. That is, we let $p_1 - p_0$ denote the vector $p'_1 = (x'_1, y'_1)$, where $x'_1 = x_1 - x_0$ and $y'_1 = y_1 - y_0$, and we define $p_2 - p_0$ similarly. We then compute the cross product

$$(p_1 - p_0) \times (p_2 - p_0) = (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0).$$

If this cross product is positive, then $\overrightarrow{p_0p_1}$ is clockwise from $\overrightarrow{p_0p_2}$; if negative, it is counter clockwise.

Determining whether consecutive segments turn left or right:-

Two consecutive line segments $\overrightarrow{p_0p_1}$ and $\overrightarrow{p_1p_2}$ turn left or right at point p_1 . Equivalently, we want a method to determine which way a given angle $\angle p_0p_1p_2$ turns. Cross products allow us to answer this question without computing the angle. As shown in Figure 35.2, we simply check whether directed segment $\overrightarrow{p_0p_2}$ is clockwise or counter clockwise relative to directed segment $\overrightarrow{p_0p_1}$. To do this, we compute the cross product $(p_2 - p_0) \times (p_1 - p_0)$. If the sign of this cross product is negative, then $\overrightarrow{p_0p_2}$ is counter clockwise with respect to $\overrightarrow{p_0p_1}$, and thus we make a left turn at P_1 . A positive cross product indicates a clockwise orientation and a right turn. A cross product of 0 means that points p_0, p_1 , and p_2 are collinear.

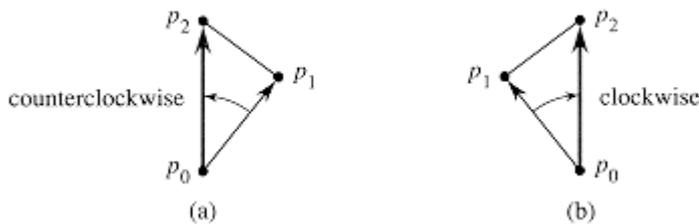


Figure 35.2 Using the cross product to determine how consecutive line segments $\overrightarrow{p_0p_1}$ and $\overrightarrow{p_1p_2}$ turn at point p_1 . We check whether the directed segment $\overrightarrow{p_0p_2}$ is clockwise or counter clockwise relative to the directed segment $\overrightarrow{p_0p_1}$. (a) If counter clockwise, the points make a left turn. (b) If clockwise, they make a right turn.

Determining whether two line segments intersect:-

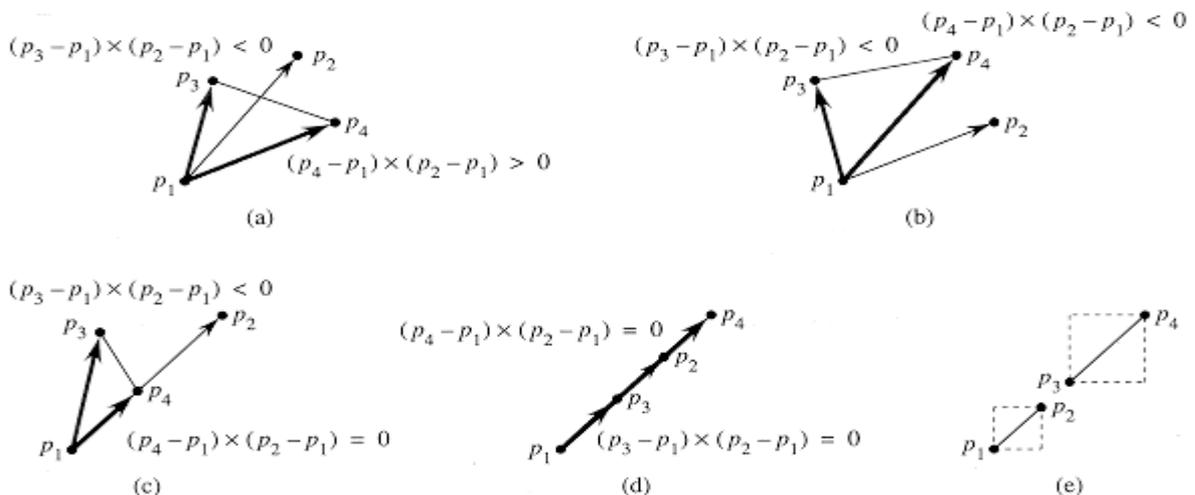
We use a two-stage process to determine whether two line segments intersect. The first stage is quick rejection: the line segments cannot intersect if their bounding boxes do not intersect. The bounding box of a geometric figure is the smallest rectangle that contains the figure and whose segments are parallel to the x-axis and y-axis. The bounding box of line segment $\overline{p_1p_2}$ is represented by the rectangle (\hat{p}_1, \hat{p}_2) with lower left point $\hat{p}_1 = (\hat{x}_1, \hat{y}_1)$ and upper right point $\hat{p}_2 = (\hat{x}_2, \hat{y}_2)$, where $\hat{x}_1 = \min(x_1, x_2)$, $\hat{y}_1 = \min(y_1, y_2)$, $\hat{x}_2 = \max(x_1, x_2)$, and $\hat{y}_2 = \max(y_1, y_2)$

where (\hat{p}_1, \hat{p}_2) and (\hat{p}_3, \hat{p}_4) . Two rectangles, represented by lower left and upper right points (\hat{p}_1, \hat{p}_2) and (\hat{p}_3, \hat{p}_4) , intersect if and only if the conjunction

$$(\hat{x}_2 \geq \hat{x}_3) \wedge (\hat{x}_4 \geq \hat{x}_1) \wedge (\hat{y}_2 \geq \hat{y}_3) \wedge (\hat{y}_4 \geq \hat{y}_1)$$

is true. The rectangles must intersect in both dimensions. The first two comparisons above determine whether the rectangles intersect in x; the second two comparisons determine whether the rectangles intersect in y.

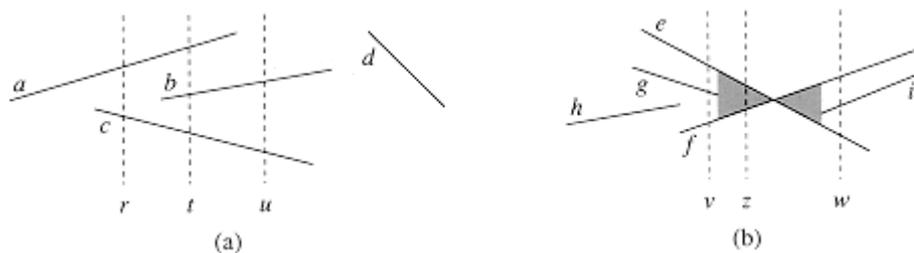
The second stage in determining whether two line segments intersect decides whether each segment "straddles" the line containing the other. A segment $\overline{p_1p_2}$ straddles a line if point p_1 lies on one side of the line and point p_2 lies on the other side. If p_1 or p_2 lies on the line, then we say that the segment straddles the line. Two line segments intersect if and only if they pass the quick rejection test and each segment straddles the line containing the other.



Ordering segments:-

Since we assume that there are no vertical segments, any input segment that intersects a given vertical sweep line intersects it at a single point. We can thus order the segments that intersect a vertical sweep line according to the y-coordinates of the points of intersection.

To be more precise, consider two nonintersecting segments s_1 and s_2 these segments are comparable at x if the vertical sweep line with x -coordinate x intersects both of them. We say that s_1 is above s_2 at x , written $s_1 >_x s_2$, if s_1 and s_2 are comparable at x and the intersection of s_1 with the sweep line at x is higher than the intersection of s_2 with the same sweep line. In Figure (a), for example, the relationships $a >_r c$, $a >_t b$, $b >_t c$, $a >_u c$, and $b >_u c$. Segment d is not comparable with any other segment.



For any given x , the relation " $>_x$ " is a total order on segments that intersect the sweep line at X . The order may differ for differing values of x , however, as segments enter and leave the ordering. A segment enters the ordering when its left endpoint is encountered by the sweep, and it leaves the ordering when its right endpoint is encountered.

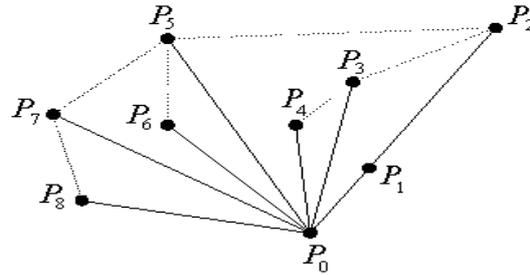
Their positions in the total order are reversed. Sweep lines v and w are to the left and right, respectively, of the point of intersection of segments e and f , and we have $e >_v f$ and $f >_w e$. Note that because we assume that no three segments intersect at the same point, there must be some vertical sweep line x for which intersecting segments e and f are consecutive in the total order $>_x$. Any sweep line that passes through the shaded region of Figure (b), such as z , has e and f consecutive in its total order.

Graham's scan:-

Graham's scan is a method of computing the convex hull of a finite set of points in the plane with time complexity $O(n \log n)$. It is named after Ronald Graham, who published the original algorithm in 1972.^[1] The algorithm finds all vertices of the convex hull ordered along its boundary.

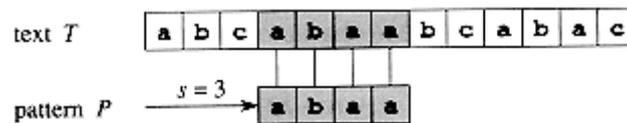
The algorithm proceeds by considering each of the points in the sorted array in sequence. For each point, it is determined whether moving from the two previously considered points to this point is a "left turn" or a "right turn". If it is a "right turn", this means that the second-to-last point is not part of the convex hull and should be removed from consideration. This process is continued for as long as the set of the last three points is a "right turn". As soon as a "left turn" is encountered, the algorithm moves on to the next point in the sorted array. (If at any stage the three points are collinear, one may opt either to

discard or to report it, since in some applications it is required to find all points on the boundary of the convex hull.)



String Matching:-

String matching algorithm used to search for particular pattern in string sequences. The string matching problem can be treated as assume that the text array and pattern is array of length $m \leq n$.



Naive Algorithm:-

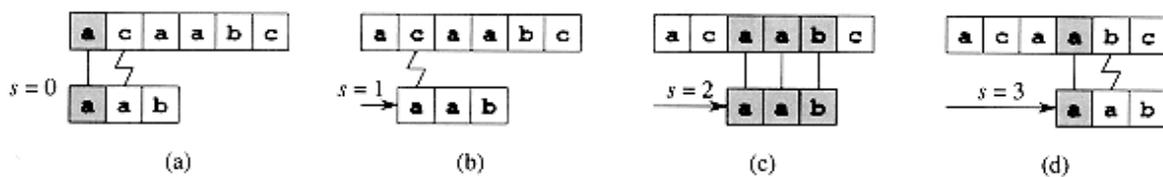
suppose $n = \text{length}(T)$, $m = \text{length}(P)$;

for shift $s=0$ through $n-m$ do

 if $(P[1..m] = T[s+1 .. s+m])$ then // actually a for-loop runs here

 print shift s ;

End algorithm.



Complexity: $O((n-m+1)m)$

A special note: we allow $O(k+1)$ type notation in order to avoid $O(0)$ term, rather, we want to have $O(1)$ (constant time) in such a boundary situation.

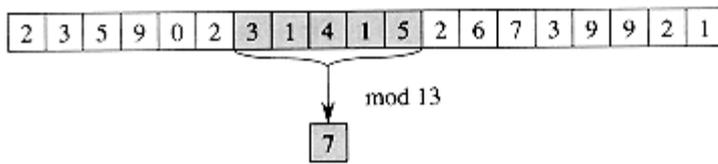
Rabin-Karp Algorithm:-

Consider a character as a number in a radix system, e.g., English alphabet as in radix-26.

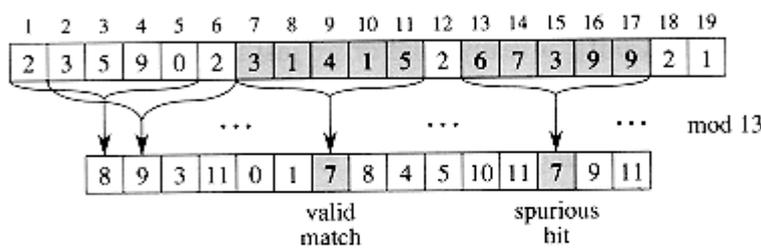
Pick up each m -length "number" starting from shift=0 through $(n-m)$.

General formula: $t_{s+1} = d (t_s - d^{m-1} T[s+1]) + T[s+m+1]$, in radix-d, where t_s is the corresponding number for the substring $T[s..(s+m)]$. Note, m is the size of P .

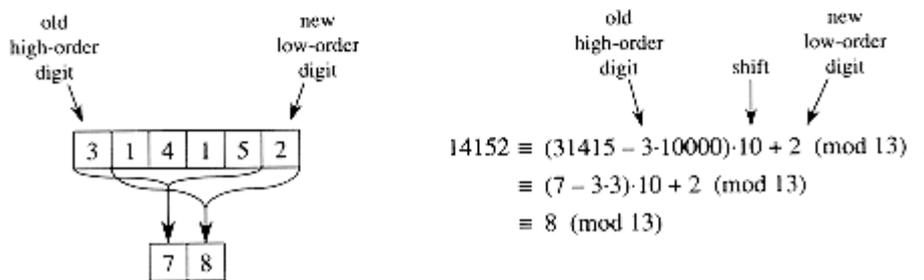
The first-pass scheme: (1) pre-process for $(n-m)$ numbers on T and 1 for P , (2) compare the number for P with those computed on Input: Text string T , Pattern string to search for P , radix to be used $d (= |\Sigma|$, for alphabet Σ), a prime q



(a)



(b)



(c)

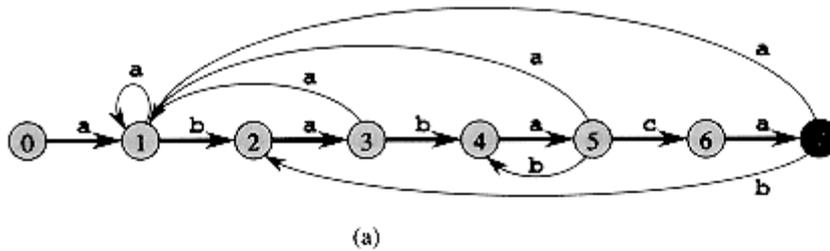
However, if the translated numbers are large (i.e., m is large), then even the number matching could be $O(m)$. In that case, the complexity for the worst case scenario is when every shift is successful ("valid shift"), e.g., $T=a^n$ and $P=a^m$. For that case, the complexity is $O(nm)$ as before.

But actually, for c hits, $O((n-m+1) + cm) = O(n+m)$, for a small c , as is expected in the real life.

String-matching automata:-

There is a string-matching automaton for every pattern P ; this automaton must be constructed from the pattern in a pre-processing step before it can be used to search the text string. Figure 34.6 illustrates this construction for the pattern $P = ababaca$. From now on, we

shall assume that P is a given fixed pattern string; for brevity, we shall not indicate the dependence upon P in our notation.



state	input			P
	a	b	c	
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

i	—	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	—	a	b	a	b	a	b	a	c	a	b	a
state $\phi(T_i)$	0	1	2	3	4	5	4	5	6	7	2	3

(b)

(c)

Knuth-Morris-Pratt Algorithm:-

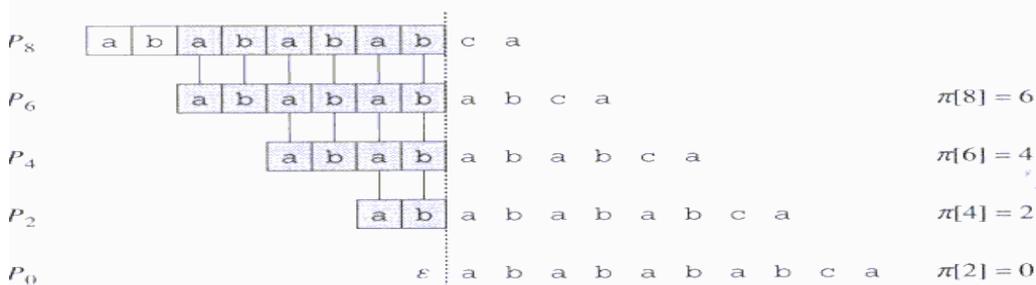
The Knuth–Morris–Pratt string searching algorithm (or KMP algorithm) searches for occurrences of a "word" W within a main "text string" S by employing the observation that when a mismatch occurs, the word itself embodies sufficient information to determine where the next match could begin, thus bypassing re-examination of previously matched characters. Thus, $P=ababababca$, when $S=P6=ababab$, largest K is $abab$, or $\pi(6)=4$.

An array $\pi[1..m]$ is first developed for the whole set for S , $\pi[1]$ through $\pi[10]$ above.

2.4 The Knuth-Morris-Pratt algorithm

i	1	2	3	4	5	6	7	8	9	10
$P[i]$	a	b	a	b	a	b	a	b	c	a
$\pi[i]$	0	0	1	2	3	4	5	6	0	1

(a)



(b)

The array P_i actually holds a chain for transitions, e.g., $P_i[8] = 6$, $P_i[6]=4$, ..., always ending with 0.

Algorithm KMP-Matcher(T, P)

$n = \text{length}[T]$; $m = \text{length}[P]$;

$P_i = \text{Compute-Prefix-Function}(P)$;

$q = 0$; // how much of P has matched so far, or could match possibly

for $i=1$ through n do

 while ($q > 0$ && $P[q+1] \neq T[i]$) do

$q = P_i[q]$; // follow the P_i -chain, to find next smaller available symmetry,
until 0

 if ($P[q+1] = T[i]$) then

$q = q+1$;

 if ($q = m$) then

 print valid shift as $(i-m)$;

$q = P_i[q]$; // old matched part is preserved, & reused in the next iteration

 end if;

end for;

End algorithm.

Algorithm Compute-Prefix-Function (P)

$m = \text{length}[P]$;

$P_i[1] = 0$;

$k = 0$;

for $q=2$ through m do

 while ($k > 0$ && $P[k+1] \neq P[q]$) do // loop breaks with $k=0$ or next if succeeding

$k = P_i[k]$;

 if ($P[k+1] = P[q]$) then // check if the next pointed character extends previously identified symmetry

$k = k+1$;

Pi[q] = k; // k=0 or the next character matched

return Pi;

End algorithm.

Complexity of second algorithm Compute-Prefix-Function: $O(m)$, by amortized analysis (on an average).

Complexity of the first, KMP-Matcher: $O(n)$, by amortized analysis.

In reality the inner while loop runs only a few times as the symmetry may not be so prevalent. Without any symmetry the transition quickly jumps to $q=0$, e.g., $P=acgt$, every P_i value is 0.

Graph Algorithms – BFS and DFS:-

Breadth-first search (BFS):-

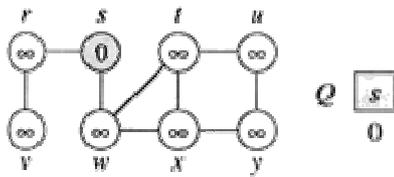
Breadth-first search (BFS) is a strategy for searching in a graph when search is limited to essentially two operations: (a) visit and inspect a node of a graph; (b) gain access to visit the nodes that neighbour the currently visited node. The BFS begins at a root node and inspects all the neighbouring nodes. Then for each of those neighbour nodes in turn, it inspects their neighbour nodes which were unvisited, and so on.

Algorithm:-

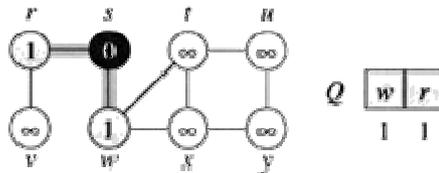
- The algorithm uses a queue data structure to store intermediate results as it traverses the graph, as follows:
- Enqueue the root node
- Dequeue a node and examine it
 - If the element sought is found in this node, quit the search and return a result.
 - Otherwise enqueue any successors (the direct child nodes) that have not yet been discovered.
- If the queue is empty, every node on the graph has been examined – quit the search and return "not found".
- If the queue is not empty, repeat from Step 2.

Example: The following figure (from CLRS) illustrates the progress of breadth-first search on the undirected sample graph.

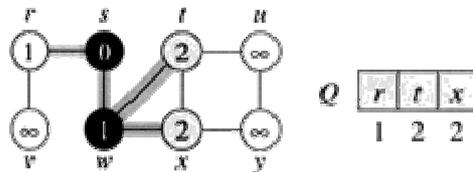
a. After initialization (paint every vertex white, set $d[u]$ to infinity for each vertex u , and set the parent of every vertex to be NIL), the source vertex is discovered in line 5. Lines 8-9 initialize Q to contain just the source vertex s .



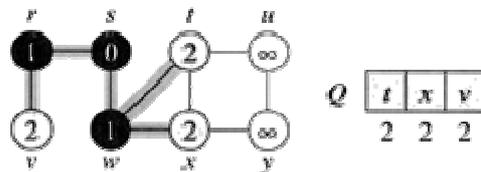
b. The algorithm discovers all vertices 1 edge from s i.e., discovered all vertices (w and r) at level 1.



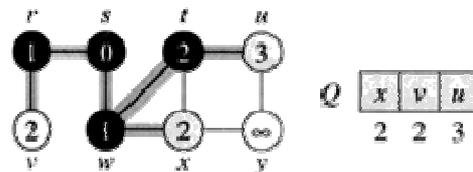
c.



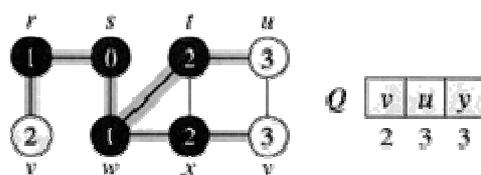
d. The algorithm discovers all vertices 2 edges from s i.e., discovered all vertices (t , x , and v) at level 2.



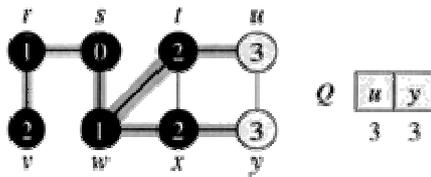
e.



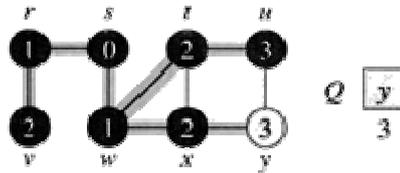
f.



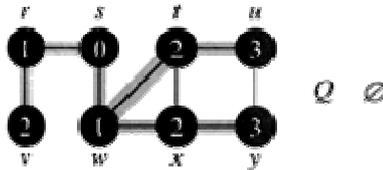
g. The algorithm discovers all vertices 3 edges from s i.e., discovered all vertices (u and y) at level 3.



h.



i. The algorithm terminates when every vertex has been fully explored.



Depth-first search(DFS):-

Depth-first search, or DFS, is a way to traverse the graph. Initially it allows visiting vertices of the graph only, but there are hundreds of algorithms for graphs, which are based on DFS. Therefore, understanding the principles of depth-first search is quite important to move ahead into the graph theory. The principle of the algorithm is quite simple: to go forward (in depth) while there is such possibility, otherwise to backtrack.

Algorithm:-

In DFS, each vertex has three possible colors representing its state:

-  white: vertex is unvisited;
-  gray: vertex is in progress;
-  black: DFS has finished processing the vertex.

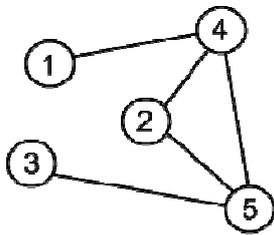
NB. For most algorithms boolean classification unvisited / visited is quite enough, but we show general case here.

Initially all vertices are white (unvisited). DFS starts in arbitrary vertex and runs as follows:

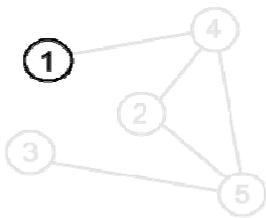
1. Mark vertex u as gray (visited).
2. For each edge (u, v), where u is white, run depth-first search for u recursively.

3. Mark vertex u as black and backtrack to the parent.

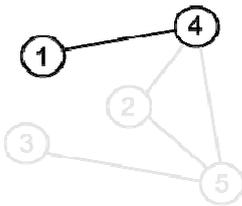
Example. Traverse a graph shown below, using DFS. Start from a vertex with number 1.



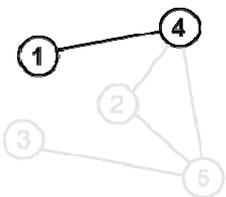
Source graph.



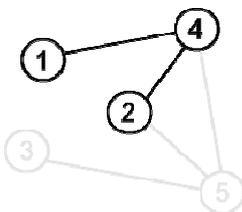
Mark a vertex 1 as grey.



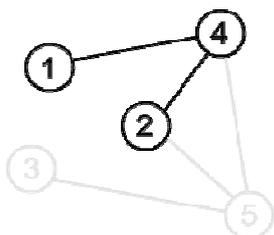
There is an edge $(1, 4)$ and a vertex 4 is unvisited. Go there.



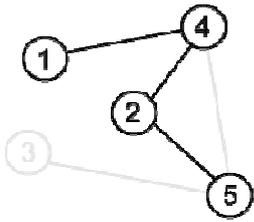
Mark the vertex 4 as gray.



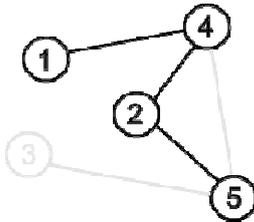
There is an edge $(4, 2)$ and vertex a 2 is unvisited. Go there.



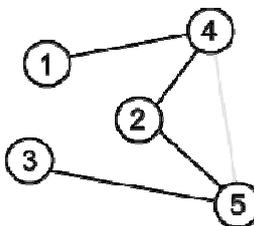
Mark the vertex 2 as gray.



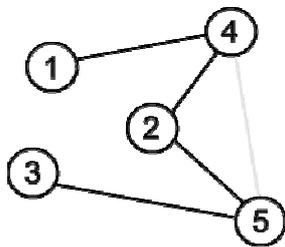
There is an edge (2, 5) and a vertex 5 is unvisited. Go there.



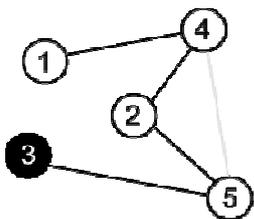
Mark the vertex 5 as gray.



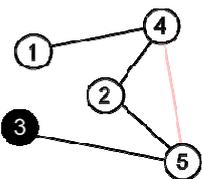
There is an edge (5, 3) and a vertex 3 is unvisited. Go there.



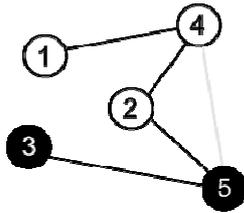
Mark the vertex 3 as gray.



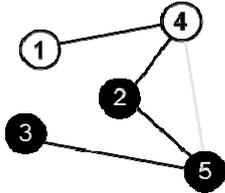
There are no ways to go from the vertex 3. Mark it as black and backtrack to the vertex 5.



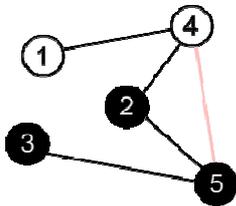
There is an edge (5, 4), but the vertex 4 is gray.



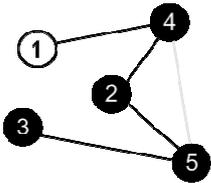
There are no ways to go from the vertex 5. Mark it as black and backtrack to the vertex 2.



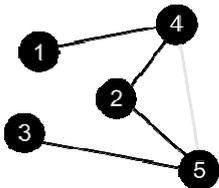
There are no more edges, adjacent to vertex 2. Mark it as black and backtrack to the vertex 4.



There is an edge (4, 5), but the vertex 5 is black.



There are no more edges, adjacent to the vertex 4. Mark it as black and backtrack to the vertex 1.



There are no more edges, adjacent to the vertex 1. Mark it as black. DFS is over.

As you can see from the example, DFS doesn't go through all edges. The vertices and edges, which depth-first search has visited is a tree. This tree contains all vertices of the graph (if it is connected) and is called graph spanning tree. This tree exactly corresponds to the recursive calls of DFS.

If a graph is disconnected, DFS won't visit all of its vertices. For details, see finding connected components algorithm.

Module IV

Spanning tree:-

A spanning tree for a graph G is a sub-graph of G which is a tree that includes every vertex of G . A spanning tree of a graph G is a “maximal” tree contained in the graph G . When you have a spanning tree T for a graph G , you cannot add another edge of G to T without producing a circuit.

Example:

Consider the following graph, G , representing pairs of people (A, B, C, D and E) who are acquainted with each other.

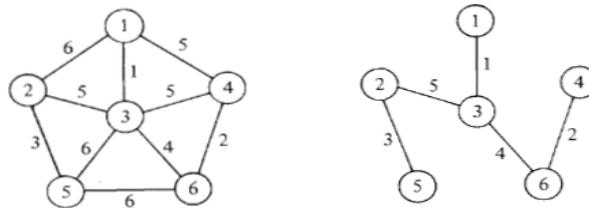
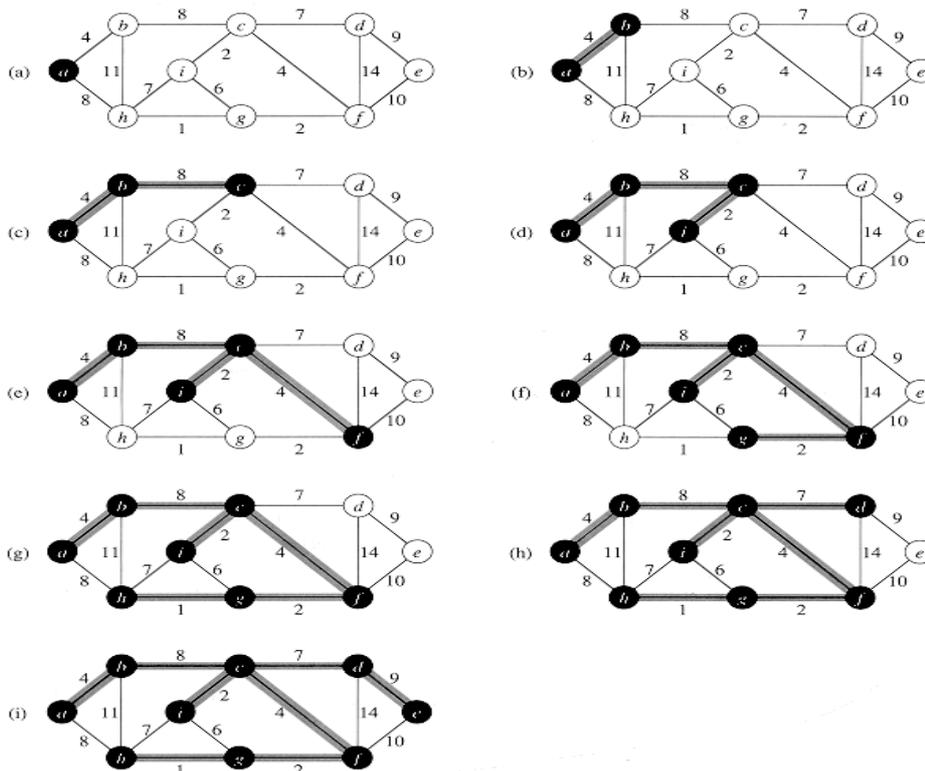


Fig. 7.4. A graph and spanning tree.

We wish to install the minimum number of phone lines so that communication between these people is maintained. As an adviser, you need to find a spanning tree T for G .

Kruskal's Algorithm:-

Find the minimal spanning tree for the following connected weighted graph G . The starting point of Kruskal's Algorithm is to make an “edge” list, in which the edges are listed in order of increasing weights.



Kruskal's Algorithm for finding minimum spanning trees for weighted graphs (Epp's version) is then:

Input: G a connected weighted graph with n vertices.

Algorithm Body: (Build a sub-graph T of G to consist of all of the vertices of G with edges added in order of increasing weight. At each stage, let m be the number of edges of T .)

1. Initialise T to have all of the vertices of G and no edges.

2. Let E be the set of all edges of G and let $m = 0$. (pre-condition: G is connected.)

3. While ($m < n - 1$)

a. Find an edge e in E of least weight.

b. Delete e from E .

c. If addition of e to the edge set of T does not produce a circuit then add e to the edge set of T and set $m = m + 1$

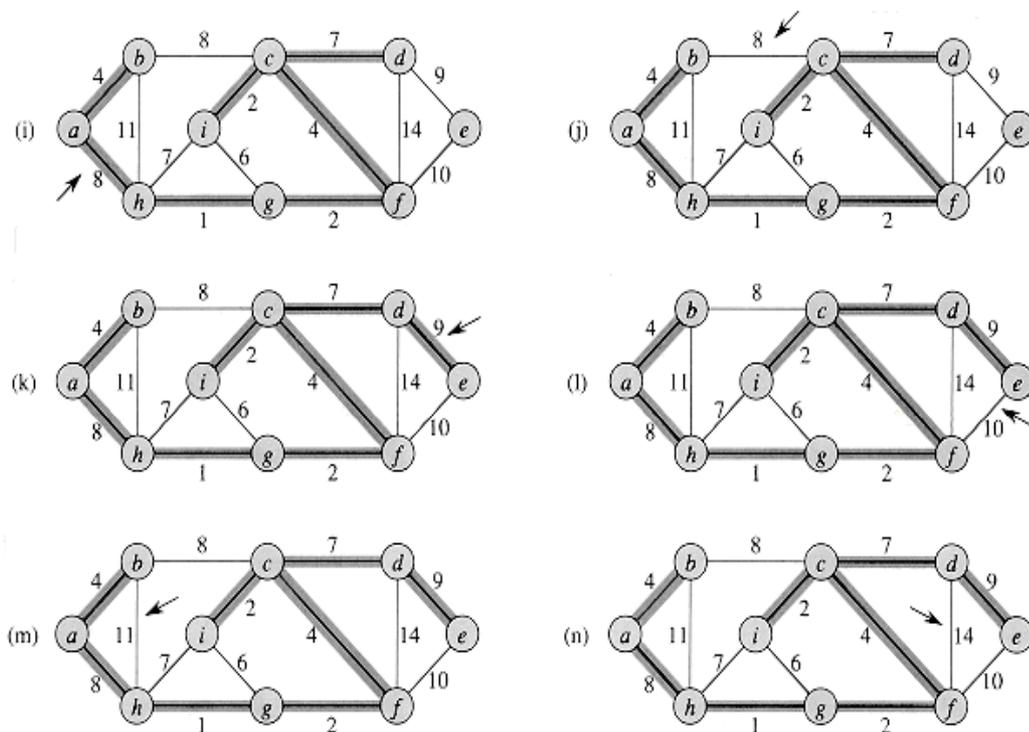
End While (post-condition: T is a minimum spanning tree for G .)

Output: T (a graph)

End Algorithm

Prim's algorithm:-

Prim's algorithm is an algorithm that finds a minimum spanning tree for connected weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. Prim's algorithm is an example of greedy. The only spanning tree of the empty graph (with an empty vertex set) is again the empty graph. The following description assumes that this special case is handled separately. The algorithm continuously increases the size of a tree, one edge at a time, starting with a tree consisting of a single vertex, until it spans all vertices.



- Input: A non-empty connected weighted graph with vertices V and edges E (the weights can be negative).
- Initialize: $V_{\text{new}} = \{x\}$, where x is an arbitrary node (starting point) from V , $E_{\text{new}} = \{\}$
- Repeat until $V_{\text{new}} = V$:
 - Choose an edge (u, v) with minimal weight such that u is in V_{new} and v is not (if there are multiple edges with the same weight, any of them may be picked)
 - Add v to V_{new} , and (u, v) to E_{new}
- Output: V_{new} and E_{new} describe a minimal spanning tree

Dijkstra's Algorithm:-

Dijkstra's algorithm (named after its discover, E.W. Dijkstra) solves the problem of finding the shortest path from a point in a graph (the *source*) to a destination. It turns out that one can find the shortest paths from a given source to all points in a graph in the same time; hence this problem is sometimes called the single-source shortest paths problem.

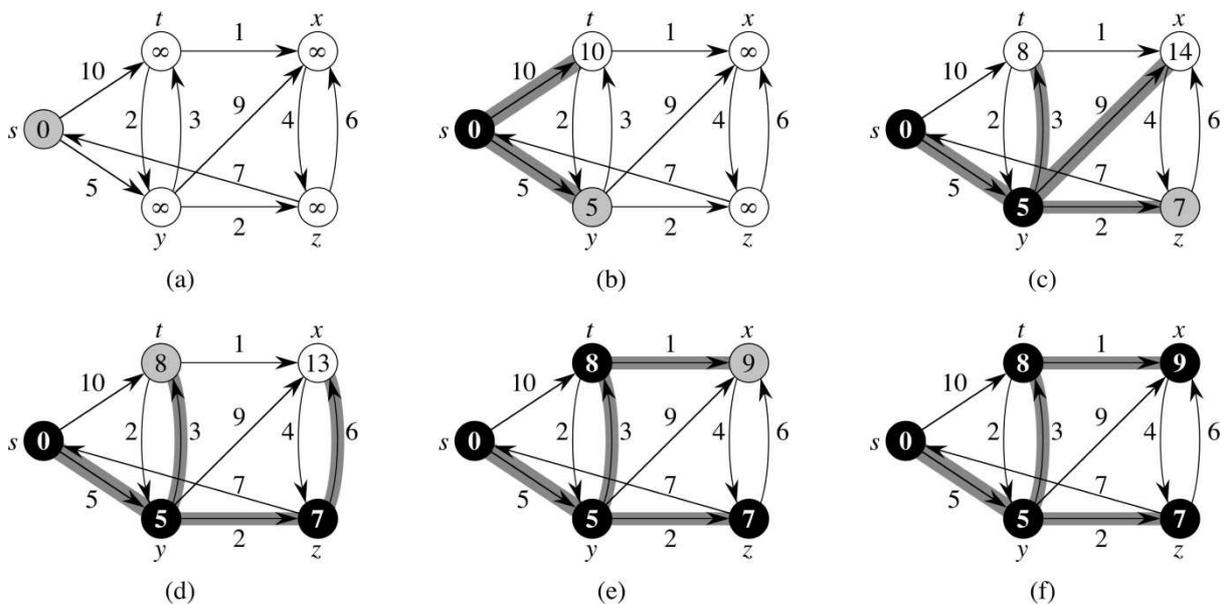
The somewhat unexpected result that all the paths can be found as easily as one further demonstrates the value of reading the literature on algorithms!

This problem is related to the spanning tree one. The graph representing all the paths from one vertex to all the others must be a spanning tree - it must include all vertices. There will also be no cycles as a cycle would define more than one path from the selected vertex to at least one other vertex. For a graph,

- $G = (V, E)$ where
- V is a set of vertices and
 - E is a set of edges.

The other data structures needed are:

- d array of best estimates of shortest path to each vertex
- pi an array of predecessors for each vertex



The basic mode of operation is:

1. Initialise d and p_i ,
2. Set S to empty,
3. While there are still vertices in $V-S$,
 - Sort the vertices in $V-S$ according to the current best estimate of their distance from the source,
 - Add u , the closest vertex in $V-S$, to S ,
 - Relax all the vertices still in $V-S$ connected to u

Maximum flow:-

We can also interpret a directed graph as a flow network and use it to answer questions about material flows. Consider a material flowing through a system from a source where the material is produced to a sink where it is consumed. The source produces the material at some study rate and the sink consumes it at the same rate.

The flow of the material at any point in the system is the rate at which the material moves. Flow networks can be used to model liquids flowing through pipes parts through assembly lines, current through electrical network, information through communication networks.

Flow conservation property:-The rate at which a material enters a vertex must equal to the rate at which it leaves the vertex. This is called as the flow conservation property.

Maximum flow problem:- Here we wish to compute the greatest rate at which material can be shipped from the source to the sink without violating any capacity constraint.

A flow network $G = (V, E)$ is a directed graph in which each edge $(u, v) \in E$ has a non-negative capacity $c(u, v) \geq 0$.

Definition of flow: Let $G(V, E)$ be a flow network with a capacity function C . Let 's' be the source of the network and 't' be the sink.

A flow in G is a real valued function $f: v \times v \rightarrow \mathbb{R}$ where \mathbb{R} is a set of real number that satisfies the following 3 property.

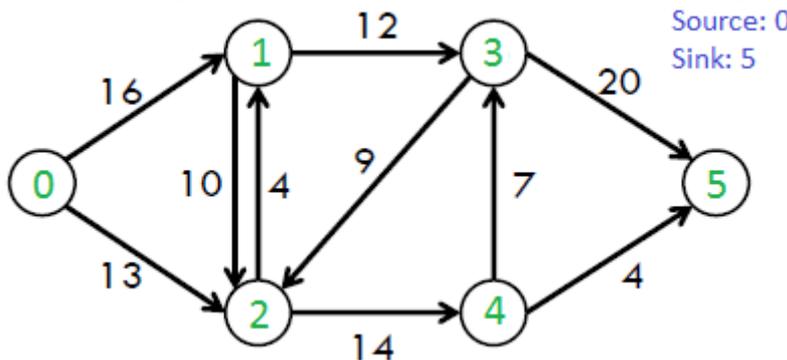
1. *Capacity constraint property:-*It says that the flow from one vertex to another must not exceed the given capacity.
For all $u, v \in V$, we require $f(u, v) \leq c(u, v)$
2. *Skew symmetry property:-* it says that the flow from a vertex u to a vertex v is the negative of the flow in the reverse direction.
For all $u, v \in V$, we require $f(u, v) = -f(v, u)$
3. *Flow conservation property:-* It says that the total flow out of a vertex other then the source or sink is zero.
For all $u \in V - \{s, t\}$, we require $\sum_{v \in V} f(u, v) = 0$

The Ford-Fulkerson method;-

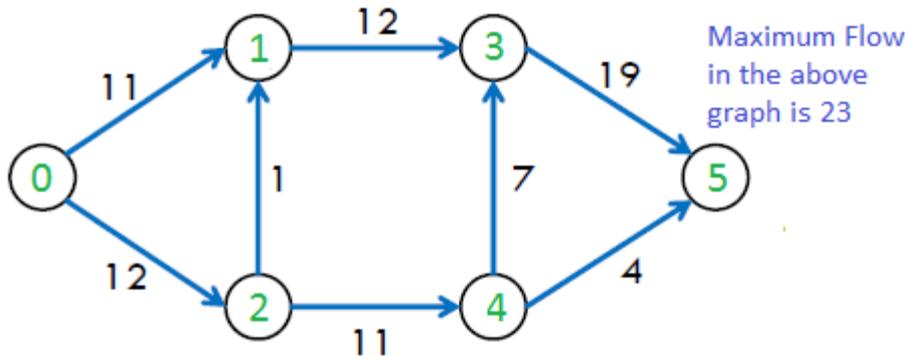
Given a graph which represents a flow network where every edge has a capacity. Also given two vertices *source* 's' and *sink* 't' in the graph, find the maximum possible flow from s to t with following constraints:

- a) Flow on an edge doesn't exceed the given capacity of the edge.
- b) Incoming flow is equal to outgoing flow for every vertex except s and t.

For example, consider the following graph from CLRS book.



The maximum possible flow in the above graph is 23.



Ford-Fulkerson Algorithm:-

The following is simple idea of Ford-Fulkerson algorithm:

- 1) Start with initial flow as 0.
- 2) While there is a augmenting path from source to sink.
Add this path-flow to flow.
- 3) Return flow.

Augmenting path:-

Given a flow network $G=(V, E)$ and a flow 'f' and an augmenting path p is a simple path from s to t in the residual network G_f . Each edge (u, v) on an augmenting path

admits some additional positive flow from u to v without violating the capacity constraints on the edge. The residual capacity of the path p is given by:

$$C_f(p) = \min\{C_f(u, v) : (u, v) \text{ is on } p\}$$

Cuts of flow network:-

Definition: A cut (S, T) of a flow network $G=(V, E)$ is a partition V into S and $T=V-S$, such that $s \in S$ and $t \in T$. The capacity of the cut (S, T) is $c(S, T)$.

NP-complete (polynomial time algorithm):-

They are algorithms which on inputs of size n have a worst case running time of $O(n^k)$ for some constant 'k'.

Example: Quick sort running time $=O(n^2)$ so its algorithm is called NP-complete algorithm.

There are three classes of problem:-

- P-class
- NP-class
- NPC-class

P-class:

The class P consist of those problems that are solvable in polynomial time that is in time $O(n^k)$ for some constant 'k' where n is input size.

Example: Quick sort running time $=O(n^2)$

NP-class:

The class NP consist of those problems that are verifiable in polynomial time that is given a certificate of a solution use could verify that the certificate is correct in time polynomial in the size of the input to the problem.

Example: Hamilton cycle

NPC-class:

The class NP-complete consist of those problems that are in NP and are as hard as any problem in NP. Any NP-complete problem can not be solved in polynomial time.

Polynomial time reduction algorithm:-

Suppose there is a decision problem 'A' which we like to solve in polynomial time. Suppose there is a different decision problem 'B' that we already know how to solve in polynomial time. Procedure that transforms any instance ' α ' of A into some instance ' β ' of B should have the following characteristics:

1. The transformation take polynomial time
2. The answers are the same that is the answer of α is yes if and only if answer for β is also yes

This procedure is called as a polynomial time reduction algorithm.

Steps

1. Given an instance α of problem use polynomial time reduction algorithm to transform it to an instance β of problem B.
2. Run the polynomial time decision algorithm for B on the instance β
3. Use the answer for β as the answer for α .

A finite NP-complete problem:-

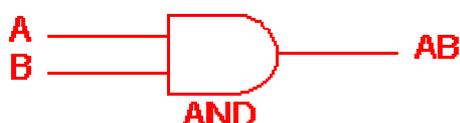
Because the technique of reduction relies on having a problem already known to be NP-complete in order to prove a different problem NP-complete we need a fast NP-complete problem. For the problem we will use is the circuit satisfiability problem in which we are given a Boolean combinational circuit composed of AND, OR & NOT gates and we wish to know whether there is any set of Boolean inputs to this circuit that causes its output to be one.

The circuit satisfiability problem is given Boolean combinational circuit composed of AND, OR, NOT gates is it satisfiable. This problem arises in the area of computer added hardware optimization.

Example: if a sub circuit always produces 0 then that sub circuit can be replaced by a simpler sub circuit that omits all logic gates and provides the constant value 0 as its output.

The three basic logic gates that we use in this problem are:

AND gate



2 Input AND gate		
A	B	A.B
0	0	0
0	1	0
1	0	0
1	1	1

This gate's output is 1, if all its inputs are 1 and output is 0 otherwise.

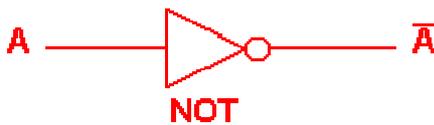
OR gate



2 Input OR gate		
A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

This gate's output is 1, if any of its input is 1 and output is 0 otherwise.

NOT gate



NOT gate	
A	\bar{A}
0	1
1	0

It takes a single binary input either 0 or 1 and produces a binary output whose value is opposite to that of the input value.

A Boolean combinational circuit consist of one or more Boolean combinational elements interconnected by wires. A wire connects the output of one element to the input of another. The number of element inputs fed by wire is called the fan-out of the wire. A one output Boolean combinational circuit is satisfiable if it has a satisfying assignment that is a truth assignment that causes the output of the circuit to be 1.

3-CNF (conjunctive normal form):-

A Boolean formula is in CNF if it is expressed as an AND of clauses each of which is the OR of one or more literals. A Boolean formula is in 3-CNF is each clause has exactly 3 distinct literals.

Example:

$$(x_1 \cdot \neg x_4 \cdot \neg x_2) \cdot (x_3 \cdot x_2 \cdot x_4) \cdot (\neg x_1 \cdot \neg x_3 \cdot \neg x_4)$$

In 3-CNF satisfiability we are asked whether a given Boolean formula in 3-CNF is satisfiable or not.

Approximation algorithms :-

Many problems of practical significance are NP-complete but are too important to abandon nearly because obtaining an optimal solution is intractable. If a problem is NP-complete it is unlikely to find a polynomial time algorithm for solving it exactly.

There are 3 approaches to getting around NP-completeness:

- I. If the actual inputs are small an algorithm with exponential running time may be perfectly satisfactory.
- II. We may be able to isolate important special cases that are solvable in polynomial time.
- III. It may be possible to find near optimal solutions in polynomial time either in the worst case or an average.

An algorithm that returns near optimal solution is called an approximation algorithm.

Performance ratio for approximation algorithms:

Suppose we are working on an optimization problem in which each potential solution has a positive cost and we wish to find a near optimal solution. Depending on the problem an optimal solution may be defined as one with minimum possible cost. That is a problem may be either a minimization or maximization problem.

An algorithm for a problem has an approximation ratio of $\rho(n)$ if for any input of size n the cost 'C' of a solution produced by the algorithm is within a factor of $\rho(n)$ of the first C^* of an optimal solution.

$$\text{Max} ((C/C^*) \cdot (C^*/C)) \leq \rho(n)$$

We called such an algorithm that achieves an approximation ratio of $\rho(n)$ approximation algorithm. For a maximization problem optimal solution is maximum that is $0 < C \leq C^*$. For a minimization problem $0 < C^* \leq C$. So the approximation ratio is never less than 1 since

$$(C/C^*) < 1 \Rightarrow (C^*/C) > 1$$

Approximation scheme:

An approximation scheme for an optimization problem is an approximation algorithm that takes as input not only an instance of the problem but also a value $\epsilon > 0$, such that for any fixed ϵ the scheme is a $1+\epsilon$ approximation algorithm.

This scheme as polynomial time approximation scheme if for any fixed $\epsilon > 0$ the scheme runs in time polynomial in the size of its input that is 'n'

$$O(n^{2/\epsilon}) \text{ where } \epsilon > 0$$

Vertex cover problem:-

A vertex-cover of an undirected graph $G=(V, E)$ is a subset of V subset of V such that if edge (u, v) is an edge of G then either u in V or v in V (or both).

The vertex cover problem is to find a vertex-cover of maximum size in a given undirected graph. This optimal vertex-cover is the optimization version of an NP-complete problem but it is not too hard to find a vertex-cover that is near optimal.

APPROX-VERTEX COVER (G: Graph):-

1. $c \leftarrow \{ \}$
2. $E' \leftarrow E[G]$
3. while E' is not empty do
4. Let (u, v) be an arbitrary edge of E'
5. $c \leftarrow c \cup \{u, v\}$
6. Remove from E' every edge incident on either u or v
7. return c

Example

