ANALYSIS & DESIGN OF ALGORITHMS

May 2012

Paper Code: CSE- 306 - F

Figures in the bracket indicate full marks.

Note : Question No. 1 is compulsory. Attempt any five questions,selecting at least one question from each part.

Q. 1. (a) A symptotic notations. (3)

Ans. Worst case: Running time of an algorithm increases with the size of the input in the limit as the size of the input increases without bound.

Big-theta notation: g(n) is an asymptotically tight bound of $f(n)$.

O-notation: $f(n)$:O($g(n)$) some constant multiple of $g(n)$ is an asymptotic upper bound of $f(n)$, no claim about how tight an upper bound is.

RG PUBLICATION

Q. 1. (b) Difference in Greedy and Dynamic approaches. (3)

Ans. Differences are:

Greedy approaches	Dynamic approaches
1. More efficient	1. Less efficient
2. Optimal solution cannot be guaranteed by a greedy algorithm.	2. Optimal solution can be guaranteed.
3. No efficient solution.	3. DP provides efficient solutions for some problems for which a brute force approach would be very slow.

Q. 1. (c) NP Hard and NP complete problems. (4)

Ans. NP-hard (non-deterministic polynomial-time hard), in computational complexity theory, is a class of problems that are, informally, "at least as hard as the hardest problems in NP". A problem *H* is NP-hard if and only if there is an NP-complete problem L that is polynomial time Turing-reducible to H.

The complexity class NP-complete (abbreviated NP-C or NPC) is a class of decision problems. A decision problem *L* is NP-complete if it is in the set of NP problems and also in the set of NP-hard problems.

Q. 1. (d) Geneal Backtracking method. (3)

Ans. In backtracking the desired solution is expressible as an n-tuple $(x_1,....x_n)$ where the x_i are chosen from some finite set si often the problem to be solved calls for finding one vector that maximizes a criterion function $p(x_1,...x_n)$. Sometimes it seek all vector that satisfy p.

Q. 1. (e) Hamiltonian Cycles with example. (4)

Ans. A Hamiltonian cycle, also called a Hamiltonian circuit, Hamilton cycle, or Hamilton circuit, is a graph cycle (i.e., closed loop) through a graph that visits each node exactly once.

Examples:

(i) a complete graph with more than two vertices is Hamiltonian

(ii) every cycle graph is Hamiltonian

(iii) every tournament has an odd number of Hamiltonian paths

(iv) every platonic solid, considered as a graph, is Hamiltonian

(v) every prism is Hamiltonian

(vi) The Deltoidalhexecontahedron is the only non-hamiltonian Archimedean dual

(vii) Every anti-prism is Hamiltonian

(viii) A maximal planar graph with no separating triangles is Hamiltonian.

Q. 1. (f) Recursive Relations. (3)

Ans. A recursive relation is an equation that recursively defines a sequence, once one or more initial terms are given: each further term of the sequence is defined as a function of the preceding terms.

Section- A

Q. 2. (a) Explain the following : Union and find operations in terms of set and Disjoint set. (10)

Ans. A union-find algorithm is an algorithm that performs two useful operations on such a data structure:

- *Find* : Determine which subset a particular element is in. This can be used for determining if two elements are in the same subset.

- *Union* : Join two subsets into a single subset.

Because it supports these two operations, a disjoint-set data structure is sometimes called a union-find data structure ormerge-find set.

A simple approach to creating a disjoint-set data structure is to create a linked list for each set. The element at the head of each list is chosen as its representative.

Make-Set creates a list of one element. Union appends the two lists, a constant-time operation. The drawback of this implementation is that Find requires Ω (n) or linear time to traverse the list backwards from a given element to the head of the list.

This can be avoided by including in each linked list node a pointer to the head of the list; then Find takes constant time, since this pointer refers directly to the set representative. However, Unionnow has to update each element of the list being appended to make it point to the head of the new combined list, requiring Ω (n) time.

When the length of each list is tracked, the required time can be improved by always appending the smaller list to the longer. Using this weighted-union heuristic, a sequence of m Make-Set,Union, and Find operations on n elements requires $O(m + n\log n)$ time. For asymptotically faster operations, a different data structure is needed.

RG PUBLICATION

Q. 2. (b) Which sorting algorithm is most efficient ? Explain various pros and cons. (10)

Ans. A sorting algorithm is an algorithm that puts elements of a list in a certain order.

Quicksort algorithm is the most efficient sorting algorithm.

Pros: (1) One advantage of parallel quick sort over other parallel sort algorithms is that no synchronization is required. A new thread is started as soon as a sub list is available for it to work on and it does not communicate with other threads. When all threads complete, the sort is done.

(2) All comparisons are being done with a single pivot value, which can be stored in a register.

(3) The list is being traversed sequentially, which produces very good locality of reference and cache behavior for arrays.

Cons:

(1) Auxiliary space used in the average case for implementing recursive function calls is O (log n) and hence proves to be a bit space costly, especially when it comes to large data sets.

(2) Its worst case has a time complexity of $O(n^2)$ which can prove very fatal for large data sets. Competitive sorting algorithms

Q. 3. (a) Explain quick sort algorithm. (10)

Ans. Quicksort, or partition-exchange sort, is a sorting algorithm that, on average, makes O(n log n) comparisons to sort n items. In the worst case, it makes O(n^2) comparisons, though this behaviour is rare. Quicksort is often faster in practice than other O(n log n) algorithms. Additionally, quicksort's sequential and localized memory references work well with a cache. Quicksort can be implemented with an in-place partitioning algorithm, so the entire sort can be done with only O(log n) additional space.

Quicksort is a divide and conquer algorithm. Quicksort first divides a large list into two smaller sub-lists: the low elements and the high elements. Quicksort can then recursively sort the sub-lists.

The steps are:

(i) Pick an element, called a pivot, from the list.

(ii) Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.

RG PUBLICATION

(iii) Recursively sort the sub-list of elements with smaller values and separately the sub-list of elements with greater values.

Q. 3. (b) Explain Strassen's matrix multipliction. (10)

Ans. The Strassen algorithm, named after Volker Strassen, is an algorithm used for matrix multiplication. It is faster than the standard matrix multiplication algorithm and is useful in practice for large matrices, but would be slower than the fastest known algorithm for extremely large matrices.

Given: Two N by N matrices A and B.

Problem: Compute $C = A \times B$

Brute Force :

```
for i:= 1 to N do
for j:=1 to N do
    C[i,j] := 0;
   for k := 1 to N do
  C[i,j] := C[i,j] + A[i,k] * B[k,j]
```

$O(N^3)$ multiplications

Divide and Conquer

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

$P_1 = (A_{11} + A_{22})(B_{11} + B_{22})$

$P_2 = (A_{21} + A_{22})(B_{11})$

$P_3 = (A_{11})(B_{12} - B_{22})$

$P_4 = (A_{22})(B_{21} - B_{11})$

$P_5 = (A_{11} + A_{22})(B_{22})$

$P_6 = (A_{21} - A_{11})(B_{11} + B_{12})$

$P_7 = (A_{12} - A_{22})(B_{21} + B_{22})$

$C_{11} = (P_1 + P_4 - P_5 + P_4 + P_7)$

$C_{12} = (P_3 + P_5)$

$C_{21} = (P_2 + P_4)$

$C_{22} = (P_1 + P_3 - P_2 + P_6)$

RG PUBLICATION

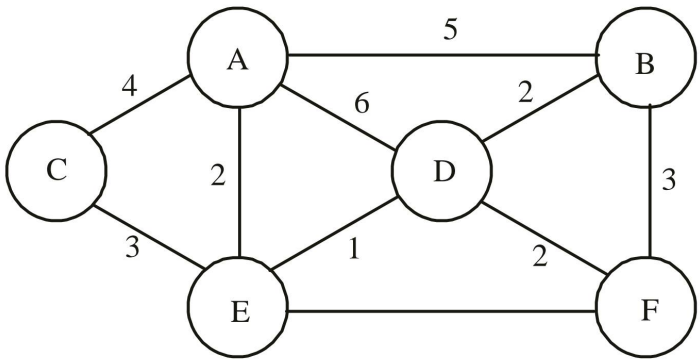
From

$$T(n) = \begin{cases} 7T(n/2) + cn & if\ n > 1 \\ c & if\ n = 1 \end{cases}$$

$T(n) = O(n^{\log 7}) = O(n^{2.81})$.

Section- B

Q. 4. (a) Generate the minimum spanning tree of the following connected graph using Kruskal's algorithm. (10)

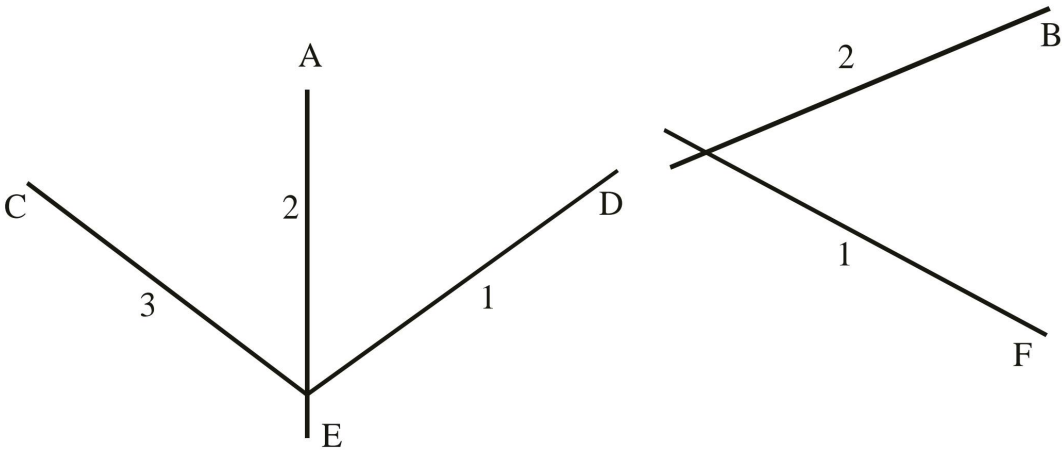


Ans. Minimum Spanning Tree:

Edges	Weight
(E,D)	1
(F,D)	2
(D,B)	2
(E,A)	2
(E,C)	3
(B,F)	3
(A,C)	4
(F,E)	4
(A,B)	5
(A,D)	6

1. *Pick edge E,D:* No cycle is formed, include it.
2. *Pick edge F,,D:* No cycle is formed, include it.
3. *Pick edge D,B:* No cycle is formed, include it.
4. *Pick edge E,A:* No cycle is formed, include it.
5. *Pick edge E,D:* No cycle is formed, include it.
6. *Pick edge E,C:* No cycle is formed, include it.
7. *Pick edge E,D:* No cycle is formed, include it.
8. *Pick edgeB,F:* Cycle is formed, Exclude it.
9. *Pick edgeA,C:* Cycle is formed, Exclude it.
10. *Pick edgeF,E:* Cycle is formed, Exclude it.
11. *Pick edgeA,B:* Cycle is formed, Exclude it.
12. *Pick edgeA,D:* Cycle is formed, Exclude it.

RG PUBLICATION



Q. 4. (b) Explain single source shortest path problem along with the algorithm, example and its complexity. (10)

Ans. The shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weightsof its constituent edges is minimized.

Algorithm :

Dijkstra's algorithm is a graph search algorithmthat solves the single-source shortest path problem for a graph with non-negative edge path costs, producing a shortest path tree. This algorithm is often used in routing and as a subroutine in other graph algorithms.

Let the node at which we are starting be called the initial node. Let the distance of node Y be the distance from the initial node to Y. Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step.

(i) Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.

(ii) Mark all nodes unvisited. Set the initial node as current. Create a set of the unvisited nodes called the *unvisited set* consisting of all the nodes except the initial node.

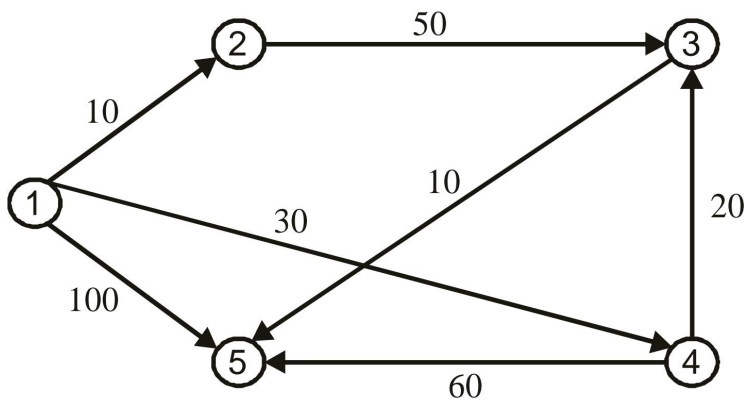
(iii) For the current node, consider all of its unvisited neighbors and calculate their *tentative* distances. For example, if the current node A is marked with a distance of 6, and the edge connecting it with a neighbor B has length 2, then the distance to B (through A) will be 6+2=8. If this distance is less than the previously recorded tentative distance of B, then overwrite that distance. Even though a neighbor has been examined, it is not marked as "visited" at this time, and it remains in the *unvisited set*.

(iv) When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the *unvisited set*. A visited node will never be checked again.

(v) If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the *unvisited set* is infinity (when planning a complete traversal), then stop. The algorithm has finished.

(vi) Select the unvisited node that is marked with the smallest tentative distance, and set it as the new "current node" then go back to step 3.

Example:



Initially:

$S = \{1\}$ $D[2] = 10$ $D[3] = \infty$ $D[4] = 30$ $D[5] = 100$

Iteration 1

Select w = 2, so that S = {1, 2}

$D[3] = \min(\infty, D[2] + C[2, 3]) = 60$

$D[4] = \min(30, D[2] + C[2, 4]) = 30$

$D[5] = \min(100, D[2] + C[2, 5]) = 100$

Iteration 2

Select w = 4, so that S = {1, 2, 4}

$D[3] = \min(60, D[4] + C[4, 3]) = 50$

$D[5] = \min(100, D[4] + C[4, 5]) = 90$

Iteration 3

Select w = 3, so that S = {1, 2, 4, 3}

$D[5] = \min(90, D[3] + C[3, 5]) = 60$

Iteration 4

Select w = 5, so that S = {1, 2, 4, 3, 5}

$D[2] = 10$

$D[3] = 50$

$D[4] = 30$

$D[5] = 60$

RG PUBLICATION

Complexity of Dijkstra's Algorithm : With adjacency matrix representation, the running time is O(n2) By using an adjacency list representation and a partially ordered tree data structure for organizing the set V - S, the complexity can be shown to beO(elog n), where e is the number of edges and n is the number of vertices in the digraph.

Q. 5. Explain the problem of solving optional binary search trees using dynamic programming. (20)

Ans. Optimal BST T must have subtree T' for keys $ki\ldots kj$ which is optimal for those keys

(i) Cut and paste proof: if T' not optimal, improving it will improve T, a contradiction.

(ii) Algorithm for finding optimal tree for sorted, distinct keys $ki\ldots kj$:

(iii) For each possible root kr for $i \leq r \leq j$

(iv) Make optimal subtree for $ki,\ldots,kr-1$

(v) Make optimal subtree for $kr+1,\ldots,kj$

(vi) Select root that gives best total tree.

(vii) Formula: $e(i, j)$ = expected number of comparisons for optimal tree for keys $ki\ldots kj$

$e(i, j) = \{0$, if $i = j + 1 \min i \leq r \leq j\{e(i, r-1) + e(r+1, j) + w(i, j)\}$, if $i \leq j$

(viii) where $w(i, j) = \sum k = ijpi$ is the increase in cost if $ki\ldots kj$ is a subtree of a node.

(ix) Work bottom up and remember solution.

DP : bottom up with table: for all possible contiguous sequences of keys and all possible roots, compute optimal subtrees

```
for size in 1 .. n loop          — All sizes of sequences
fori in 1 .. n - size+1 loop     — All starting points of sequences
j := i + size - 1
for r in i .. j loop             — All roots of sequence ki .. kj
t : = e(i, r -1) + e(r + 1, j) + w(i, j)
if e(i, j) < t then
e(i, j) : = t
root(i, j) : = r
end if
end loop
end loop
end loop
```

Complexity: $\Theta(n^3)$

Section- C

Q. 6. Suggest the solutions to the following problems using backtracking :

(a) 8 - queen

(b) Graph - Coloring. **(20)**

Ans. (a) 8 – queen

Algorithm:

- Start with one queen at the first column first row
- Continue with second queen from the second column first row
- Go up until find a permissible situation
- Continue with next queen

We place the first queen on A1. Note the positions which Q1 is attacking. So the next queen Q2 has to options: B3 or B4. We choose the first one B3. We backtrack to Q3 and try to find admissible place different from C1. Again we need to backtrack. Q2 has no other choice and finally we reach Q1. We place Q1 on A3.

```
intPlaceQueen(int board[8], int row)
If (Can place queen on ith column)
PlaceQueen(newboard, 0)
Else
PlaceQueen(oldboard,oldplace+1)
End
```

Ans. (b) Graph - Coloring : The backtracking search tree starts at non-labeled root. The immediate children of the root represent the available choices regarding the coloring of the first vertex (vertex a in this case) of the graph. However, the nodes of the tree are generated only if necessary. The first (leftmost) child of the root represents the first choice regarding the coloring of vertex a, which is to color the vertex with color 1 — this is indicated by the label assigned to that node. So far this partial coloring does not violate the constraint that adjacent vertices be assigned different colors. So we grow the tree one more level where we try to assign a color to the second vertex (vertex b) of the graph. The leftmost node would represent the coloring of vertex b with color 1. However, such partial coloring where a=1 and b=1 is invalid

since the vertices a and b are adjacent. Also, there is no point of growing the tree further from this node since, no matter what colors we assign to the remaining vertices, we will not get a valid coloring for all of the vertices. So we prune (abandon) this path and consider an alternative path near where we are. We try the next choice for the coloring of vertex b, which is to color it with color 2. The coloring $a = 1$ and $b = 2$ is a valid partial coloring so we continue to the next level of the search tree. When we reach the lowest level and try to color vertex e, we find that it is not possible to color it any of the three colors; therefore, we backtrack to the previous level and try the next choice for the coloring of vertex d. We see that the coloring of vertex d with color 2 leads to a violation (d and b are adjacent and have same color). We continue to d=3 and then down to $e = 1$, which gives the solution, $\{a = 1, b = 2, c = 3, d = 3, e = 1\}$.

Q.7. (a) Differentiate between backtracking and branch and bound. (10)

Ans. Backtracking :

(i) It is used to find all possible solutions available to the problem.

(ii) It traverse tree by DFS(Depth First Search).

(iii) It realizes that it has made a bad choice & undoes the last choice by backing up.

(iv) It searches the state space tree until it found a solution.

(v) It involves feasibility function.

Branch-and-Bound (BB)

(i) It is used to solve optimization problem.

(ii) It may traverse the tree in any manner, DFS or BFS.

(iii) It realizes that it already has a better optimal solution that the pre-solution leads to so it abandons that pre-solution.

(iv) It completely searches the state space tree to get optimal solution.

(v) It involves bounding function.

Q.7. (b) Explain LC branch and bound with example. (10)

Ans. When a node is visited the first time called an E-node, all its children are generated and saved into a data structure called live-node. If the child is not bounded; the structure could be a LC, or Least Cost. Exactly one node is pulled out the live-node list at a time, which becomes an E-node, to be "expanded" next.The LC (least-cost) branch-and-bound strategy uses heuristics to determine the best node for expansion from the current live nodes.

Example: The 15-puzzle

– 15 numbered tiles on a square frame with a capacity for 16 tiles.

– Given an initial arrangement, transform it to the goal arrangement through a series of legal moves.

– Legal move involves moving a tile adjacent to the empty spot E to E.

– Four possible moves in the initial state above: tiles 2, 3, 5, 6.

– Each move creates a new arrangement of tiles, called state of the puzzle.

– Initial and goal states.

– A state is reachable from the initial state iff there is a sequence of legal moves from initial state to this state.

– The state space of an initial state is all the states that can be reached from initial state.

– Search the state space for the goal state and use the path from initial state to goal state as the answer.

– Number of possible arrangements for tiles: 16! ≈ 20.9 × 1012.

– Check whether the goal state is reachable from initial state.

Number the frame positions from 1 to 16*pi* is the frame position containing tile numbered *i*. *p*16 is the position of empty spot. For any state, let *li* be the number of tiles *j* such that $j < i$ and $pj > pi$. For the initial arrangement above, l1 = 0, l4 = 1, and l12 = 6.

Let $x = 1$ if in the initial state, the empty spot is in one of the following positions: 2, 4, 5, 7, 10, 12, 13, 15;otherwise $x = 0$.

Section- D

Q. 8. (a) Show that Clique decision problem is NP Hard. (10)

Ans. Given a graph G, find the largest clique (set of nodes such that all pairs in the set are neighbours).

Decision problem: "Given G and integer k, does G contain a clique of size $\geq k$?"

Proof: We will reduce 3-SAT to Max-Clique. Speciûcally, given a 3-CNF formula F of m clauses over n variables, we construct a graph as follows. First, for each clause c of F we

create one node for every assignment to variables in c that satisûes c. E.g., say we have:

$F = (x_1 \vee x_2 \vee x_4) \wedge (x_3 \vee x_4) \wedge (x_2 \vee x_3) \wedge \ldots$

Then in this case we would create nodes like this:

$(x_1 = 0, x_2 = 0, x_4 = 0)$ $(x_3 = 0, x_4 = 0)$ $(x_2 = 0, x_3 = 0)$ ...

$(x_1 = 0, x_2 = 1, x_4 = 0)$ $(x_3 = 0, x_4 = 1)$ $(x_2 = 0, x_3 = 1)$

$(x_1 = 0, x_2 = 1, x_4 = 1)$ $(x_3 = 1, x_4 = 1)$ $(x_2 = 1, x_3 = 0)$

$(x_1 = 1, x_2 = 0, x_4 = 0)$

$(x_1 = 1, x_2 = 0, x_4 = 1)$

$(x_1 = 1, x_2 = 1, x_4 = 0)$

$(x_1 = 1, x_2 = 1, x_4 = 1)$

We then put an edge between two nodes if the partial assignments are consistent. Notice that the maximum possible clique size is m because there are no edges between any two nodes that correspond to the same clause c. Moreover, if the 3-SAT problem does have a satisfying assignment, then in fact there is an m-clique (just pick some satisfying assignment and take the m nodes consistent with that assignment). So, to prove that this reduction (with $k = m$) is correct we need to show that if there isn't a satisfying assignment to F then the maximum clique in the graph has size $< m$. We can argue this by looking at the contrapositive. Speciûcally, if the graph has an m-clique, then this clique must contain one node per clause c. So, just read off the assignment given in the nodes of the clique: this by construction will satisfy all the clauses. So, we have shown this graph has a clique of size m iff F was satisfiable. Also, our reduction is polynomial time since the graph produced has total size at most quadratic in the size of the formula F ($O(m)$ nodes,$O(m)$ edges). Therefore Clique is NP-complete.

Q. 8. (b) Discuss Node cover decision problem. (10)

Ans. Many NP-complete problems involve sets of vertices in a graph that have some particular property.

The Node Cover decision problem is:

Input: a graph G=(V,E) and a positive integer k.

?: is there some V' V, |V'| = k, where (u, v) E, either u V' or v V' ?

Node Cover is NP-hard by a reduction from 3Sat.

Independent Set: A set of vertices V' is an independent set in a graph G=(V,E) if V' V and: u, v V', (u, v) E.

The Independent Set problem:

Input: a graph *G* and integer *k*

?: Is there an independent set in *G* of size *k* ?

Vertices are independent if they arenot adjacent.

Hardness of Independent Set

We can prove that Independent Set is NP-hard by reducing to itfrom Vertex Cover.To do this reduction, look at thedefinitions of the two problems, anddetermine how to transform one intothe other.

We can notice that if (and only if) V'is a vertex cover for *G*, then V-V' is an independent set for *G*; all edgeshave an endpoint in V', so there areno edges between vertices in V-V'.

Formalized reduction:

Let *G, k* be the input to Vertex Cover,and *G*', *k*' be the input to IndependentSet.

We construct *G*' and *k*' by:

Let G' = G, and let k' = |V|-k

If *G* has a vertex cover of size *k*, thenthere is a set V', |V'| = *k* with:

(*u, v*) E, either *u* V' or v V'

Then (*u, v*) *E*, *u* V-V' or *v* V-V'so *u, v* V-V', (*u, v*) E, and there is anindependent set of size |V|-*k*.

If *G*' = *G* has an independent set of size

|V|-*k*, then there is a V'', |V''|=|V|-*k* where *u, v* V'', (*u, v*) E. If we let V'=V-V'',

thenu,*v* V', (*u, v*) E.

So (*u, v*) *E*, *u* V' or *v* V', and *G* has avertex cover of size *k*.

Q. 9. State and prove Cook's theorem. **(20)**

Ans. Cook's theorem, states that the Boolean satisfiability problem is NP-complete. That is, any problem in NP can be reduced in polynomial time by a deterministic Turing machine to the problem of determining whether a Boolean formula is satisfiable.

Proof-idea:

First show that SAT is in NP.

Then show that every problem in NP is polynomial-time reducible to SAT.

Proof:

It is easy to see that SAT is in NP. Simply non-deterministically choose Boolean values for each variable. Then check whether the expression is true or false for those values.

If it is true, the answer is "yes."

If it is false, the answer is no."

Proof :

Let A be a problem in NP.

Then there is a nondeterministic Turing machine M that decides A in polynomial time. Let the run time of M be at most $n\ k$, for some $k\ \ 0$. When M is run, it can access at most the first $n\ k + 1$ cells.

RG PUBLICATION

ANALYSIS AND DESIGN OF ALGORITHM

May 2013
Paper Code: CSE -306-F

Note: ***Question No.1 is compulsory. Attempt five questions in total selecting one question from eachSections.***

Q.1.(a) What do you mean by an asymptotic notation? Write and explain different types of asymptotic notations with suitable example. (5)

Ans. Asymptotic Notation : Asymptotic notation is a shortest way to represent the time complexity.

Various notations such as Ω, Θ, and O are called asymptotic notations.

Big Oh Notation : The Big oh notation os dewted by 'O'. It is a method of representing the upper bound of algorithm's running time. Using big Oh notation we can give longest amount of time taken by the algorithm to complete.

Definition: Let F(n) and g(n) be two non-negative function. Let n_0 and costant c are two integers such that n_0 denotes some value of input and $n > n_0$. Similarly c is some constant such that $c > 0$. We can write

$$F(n) \le c * g(\mathrm{n})$$

then F(n) is big Oh of g(n). It is also denoted as F(n) $\in$ O(g(n)). In other words F(n) is less than g(n) is g(n) is multiple of some constant c.

Example : Consider function F(n) = 2n + 2 and $g(n) = n^2$. Then we have to find some constant c,so that $F(n) \le c * g(\mathrm{n})$. As F(n) =2n + 2 and $g(n) = n^2$ then we find c for $n = 1$ then

$$\mathrm{F}(n) = 2n + 2$$
$$= 2(1) + 2$$
$$\mathrm{F}(n) = 4$$

and $\mathrm{g}(n) = n^2 = (1)^2 = 1$

$$\text{i.e. } \mathrm{F}(n) > g(n)$$

If $n = 2$ then,

$$\mathrm{F}(n) = 2(2) + 2 = 6$$
$$\mathrm{g}(n) = (2)^2 = 4$$
$$\text{i.e } \mathrm{F}(n) > \mathrm{cg}(n)$$

If $n = 3$, then

$$\mathrm{F}(n) = 2(3) + 2 = 8$$
$$\mathrm{g}(n) = (3)^2 = 9$$
$$\text{i.e } \mathrm{F}(n) < \mathrm{g}(n) \text{ is true}$$

Hence we can conclude that for $n > 2$, we obtain

RG PUBLICATION

$$F(n) < g(n)$$

Thus always upper bound of existing time is obtained by big Oh notation.

Omega Notation : Omega notation is denoted by 'Ω'. This notation is used to represent the lower bound of algorithm's running time. Using omega notation we can denote shortest amount of time taken by algorithm.

Definition : A fanction F(n) is said to be in Ω (g(n)) if F(n) is bounded below by some posifive constant multiple of g(n) such that

$$F(n) \geq c * g(n) \text{ For all } n \geq n_o$$

It is denoted as $F(n) \in \Omega(g(n))$.

Example:

Consider $F(n) = 2n^2 + 5$ and $g(n) = 7n$

Then if $n = O$

$F(n) = 2(0)^2 + 5 = 5$

$g(n) = 7(0) = 0$

i.e $F(n) > g(n)$

But if $n = 1$

$F(n) = 2(1)^2 + 5 = 7$

$g(n) = 7(1) = 7$

i.e $F(n) = g(n)$

If $n = 3$, then

$F(n) = 2(3)^2 + 5 = 23$

$g(n) = 7(3) = 21$

i.e $F(n) > g(n)$

Thus For $n>3$ we fet $F(n) > c * g(n)$

It can be represented as $2n^2 + 5 \in \Omega(n)$

Similarly any $n^3 \in \Omega(n)^2$

Θ Notation : The theta notation is denoted by Θ. By this method the running time is between upper bound and lower bound.

Definition : Let F(n) and $g(n)$ be two non-negative functions. There are two positive constants namely c_1 and c_2 such that

$$c_1 \leq g(n) \; c_2 \leq g(n)$$

Then we can say that

$$F(n) \in \Theta(g(n))$$

Example : If $F(n) = 2n + 8$ and $g(n) = 5n$

Where $n \geq 2$

Similarly $F(n) = 2n + 8$

$g(n) = 7n$

RG PUBLICATION

i.e $5n < 2n + 8 < 7n$ For $n \geq 2$

Here $c_1 = 5$ and $c_2 = 7$ with $n_0 = 2$

The theta notation is more precise with both big Oh and omega notation.

Q.1.(b) Differentiate Fractional and 0-1 Knapsack problem. (5)

Ans. O-1 Knapsack problem : If a thief carries all or none of a good in the store, which cannot be parted or broken then it is called O-1 Knapsack problem.

Fractional Knapsack Problem : If a thief can carry fractional part then it is fractional knapsack problem.

Differences are:

Fractional Knapsack problem	**0-1 Knapsack problem**
Items are divisible.	Items are indivisible.
Solved with a greedy algorithm	Solved with dynamic programming.

Q.1.(c) Define the following associated with algebraic problems : ring, field, indeterminate and extension. (5)

Ans. Ring: A ring is an algebraic system consisting of a set, an identity element, two operations and the inverse operation of the first operation.

Field: A field is an algebraic system consisting of a set, an identity element for each operation, two operations and their respective inverse operations.

Indeterminate: Let F be a field and let x be a symbol not one of those of F, an indeterminate. To any n-tuple (a_0, a_1..... a_{n-1}) of members of F we associate the polynomial in x : $a_0x^0 + a_1x^1 + \ldots\ldots\ldots\ldots\ldots\ldots + a_{n-1}x^{n-1}$.

Extension: If F and E are ûelds, and F $\subseteq$ E, we say that E is an extension of F, and we write either F $\leq$ E or E/F.

Q.1(d) Explain *P*, *NP*, *NP* hard and *NP* Complete problems. Also give the relationship between each of the class. (5)

Ans. P : P is the set of all decision problems solvable by deterministic algo in polynomial time.

NP : NP is the set of all decision problems solvable by non-deterministic algo in polynomial time.

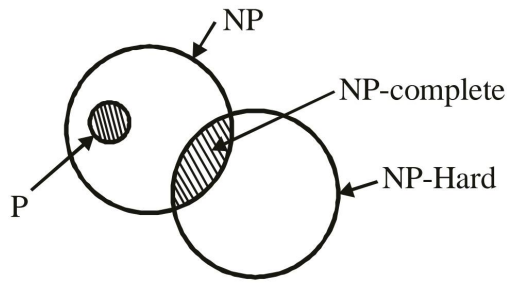
Since deterministic algorithms are just a special case of non-deterministic once, we conclude that $P \subseteq NP$.

NP-Hard : If an NP-hard problem can be solved in polynomial time then all NP-Complete problem can be solved polynomial time.

N-P Complete : A problem that is NP-Complete has the property that it can be solved in polynomial time iff all other NP-Complete problems can also be solved in polynomial time.

All NP-complete problems are NP-hard, but some NP-hard problems are not known to be NP-complete.

Relationship between P, NP, NP-Complete and NP-Hard :



Section – A

Q.2.(a) Write algorithms for Union and Find operations for disjoint sets. (10)

Ans. A union-find algorithm is an algorithm that performs two useful operations on such a data structure:

Find : Determine which subset a particular element is in. This can be used for determining if two elements are in the same subset.

RG PUBLICATION

Union : Join two subsets into a single subset.

procedure Make-Set (x)

1. size[x] $\leftarrow$ 1
2. parent[x] $\leftarrow$ x

end.

procedure UNION(a , b) { with weight balancing }

{a and b are roots of two distinct trees in the forest.}

{Makes the root of the smaller tree a child of the root of the larger tree.}

1. if size[a] < size[b] then a $\leftrightarrow$ b
2. parent[b] $\leftarrow$ a
3. size[a] $\leftarrow$ size[a] + size[b]

end.

function FIND(x) { with path compression }

{Returns the root of the tree that contains node x.}

1. if parent[x] $\neq$ x then
2. parent[x] $\leftarrow$ FIND(parent[x])
3. return parent[x]

end.

Q.2.(b) What is Divide and Conquer strategy? Explain Merge sort algorithm with example. Also give its recurrence relation. (10)

Ans. Divide and Conquer strategy : Divide and conquer is a technique for designing algorithms that consists of decomposing the instance to be solved into a number of smaller subinstance of the same problem, solving successively and independently each of these subinstances, and then combining the subsolutions thus obtained to obtain the solution of the original instance.

Most of the algorithms are recursive in natare. To solve a given problem they call them selves recursively one or more times. All these algorithms follows "Divid and conquer" approach.

The Divide and conquer paradigm involves three steps at each level of recursion :

(1) Divide : In this step whole problem is divided into number of several subproblems .

(2) Conquer : The subproblems by solving them recursively. If the subproblem size are small enough, however, just solve the subproblem in a straight forward manner.

(3) Combine : Finally, the solutions obtained by the subproblems are combined to create solution to the original problem.

Algorithm:

Input: An array a and two indices p and q between which we want to do the sorting.

Output: The array a will be sorted between p and q as a side effect

```
if p < q then
int m <-p+q/2
mergeSort(a,p,m)
mergeSort(a,m+1,q)
merge(a,p,m,q)
```

The pseudo code of the merging routine is as follows:

Algorithm merge(a,p,m,q)

Input: An array a, in which we assume that the halves from p:::m and m+1:::q are each sorted

Output: The array should be sorted between p and q

```
Array tmp of size q p+1 // this array will hold the temporary result
inti<-p
int<-j m+1
int k <-1
while (i<= m or j <=q) do
if (j = q+1 or a[i] <= a[ j]) then
tmp[k]<- a[i]
i<- i+1
else if (i = m+1 or a[i] > a[ j]) then
```

```
tmp[k]<- a[ j]
j<- j +1
k<- k +1
for k = p to q do
a[k] <- tmp[k]
```

Example: For the first call of the function, the data is partitioned into two, one list of 6, 5, 8 and 1, and a second list of 4, 3, 7 and 2. The list (6 5 8 1) is then partitioned into two smaller lists (6 5) and (8 1). The base case has now been reached, and we can sort (6 5) into (5 6) and we can sort (8 1) into (1 8). We can now merge (5 6) and (1 8). We compare 1 and 5, so the first element of the merged sequence is 1. We next compare 5 and 8, so the second element is 5. Next we compare 6 and 8 the third element will be 6, leaving 8 as the last element. The merged result is (1 5 6 8). We now turn our attention to the second half of the original data set. Again we partition (4 3 7 2) into (4 3) and (7 2) . Sorting these we get (3 4) and (2 7) Merging these we get (2 3 4 7). We now have two halves of the data sorted as (1 5 6 8) and (2 3 4 7) . All that remains to be done is to merge the two halves together.

(1 2 3 4 5 6 7 8)

Recurrence relation: $T(n)=2T(n/2) + c$,

RG PUBLICATION

Q.3.(a) State Matrix chain multiplication problem. How to solve this problem with Dynamic programming? Explain. (10)

Ans. Matrix chain multiplication is an optimization problem that can be solved using dynamic programming. Given a sequence of matrices, we want to find the most efficient way to multiply these matrices together. The problem is not actually to *perform* the multiplications, but merely to decide in which order to perform the multiplications.

Dynamic Programming Algorithm : To begin, let us assume that all we really want to know is the minimum cost, or minimum number of arithmetic operations, needed to multiply out the matrices. If we are only multiplying two matrices, there is only one way to multiply them, so the minimum cost is the cost of doing this. In general, we can find the minimum cost using the following recursive algorithm:

- Take the sequence of matrices and separate it into two subsequences.
- Find the minimum cost of multiplying out each subsequence.
- Add these costs together, and add in the cost of multiplying the two result matrices.
- Do this for each possible position at which the sequence of matrices can be split, and take the minimum over all of them.

For example, if we have four matrices *ABCD*, we compute the cost required to find each of (*A*)(*BCD*), (*AB*)(*CD*), and (*ABC*)(*D*), making recursive calls to find the minimum cost to

compute *ABC*, *AB*, *CD*, and *BCD*. We then choose the best one. Better still, this yields not only the minimum cost, but also demonstrates the best way of doing the multiplication: group it the way that yields the lowest total cost, and do the same for each factor.

Unfortunately, if we implement this algorithm we discover that it is just as slow as the naive way of trying all permutations! What went wrong? The answer is that we're doing a lot of redundant work. For example, above we made a recursive call to find the best cost for computing both *ABC* and *AB*. But finding the best cost for computing ABC also requires finding the best cost for *AB*. As the recursion grows deeper, more and more of this type of unnecessary repetition occurs.

One simple solution is called memoization: each time we compute the minimum cost needed to multiply out a specific subsequence, we save it. If we are ever asked to compute it again, we simply give the saved answer, and do not recompute it. Since there are about $n^2/2$ different subsequences, where n is the number of matrices, the space required to do this is reasonable. It can be shown that this simple trick brings the runtime down to $O(n^3)$ from $O(2^n)$, which is more than efficient enough for real applications. This is top-down dynamic programming.

Q.3.(b) What do you understand by best case and worst case behavior of an algorithm? Discuss their significance with the help of suitable example. (10)

Ans. *Worst-Case Analysis* –The maximum amount of time that an algorithm requires to solve a problem of size n.

- This gives an upper bound for the time complexity of an algorithm.
- Normally, we try to find worst-case behavior of an algorithm.

Best-Case Analysis – The minimum amount of time that an algorithm requires to solve a problem of size n.

- The best case behavior of an algorithm is NOT so useful.

The *worst-case complexity* of the algorithm is the function defined by the maximum number of steps taken on any instance of size n. It represents the curve passing through the highest point of each column.

The *best-case complexity* of the algorithm is the function defined by the minimum number of steps taken on any instance of size n. It represents the curve passing through the lowest point of each column.

For example, the best case for a simple linear search on a list occurs when the desired element is the first element of the list.

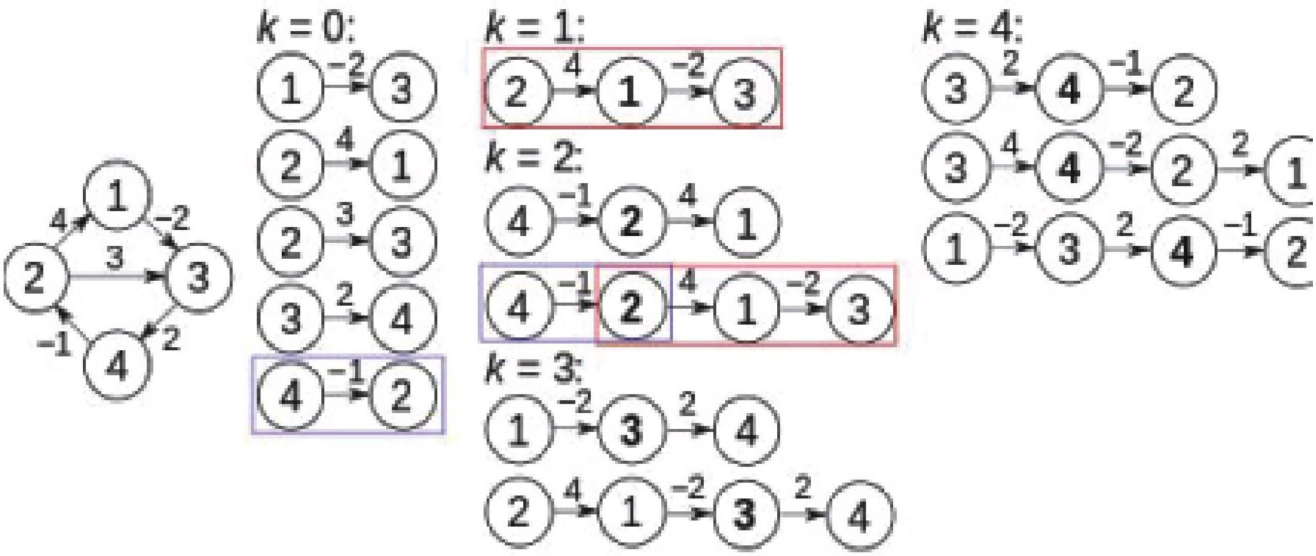
Example: For quicksort, the best case complexity is $O(n \log(n))$ and worst case is $O(n^2)$.

Section – B

Q.4.(a) Define all pair shortest path problem. Discuss solution of this problem based on dynamic programming. Give suitable algorithm and find its computing time. (10)

Ans. The all pairs shortest path problem involves finding the shortest path from each node in the graph to every other node in the graph.

Example:



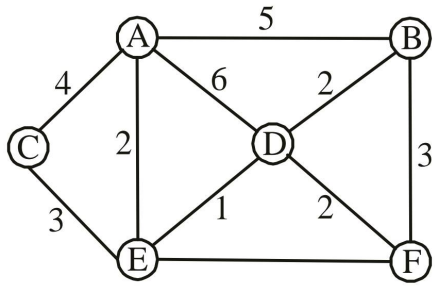
Algorithm:

```
0 Initialise Matrix;
1 for(i=0;i<n;i++) {
2 for(j=0;j<n;j++) {
3 for(l=0;l<n;l++) {
4 Matrix[i][j] += Matrix[i][l] * Matrix[l][j];
```

```
0 Initialise Matrix;
1 for(i=0;i<n;i++)
2 for(j=0;j<n;j++)
3 for(k=0;l<n;l++)
4 if(Matrix[i][k] > Matrix[i][j] + Matrix[j][i])
5 Matrix[i][k] = Matrix[i][j] + Matrix[j][i];
```

Computing time: Let n be |V|, the number of vertices. To find all n^2 of shortest Path(i,j,k) (for all i and j) from those of shortest Path(i,j,k"1) requires $2n^2$ operations. Since we begin with shortest Path(i,j,0) = edge Cost(i,j) and compute the sequence of n matrices shortest Path(i,j,1), shortest Path(i,j,2), …, shortest Path(i,j,n), the total number of operations used is $n \cdot 2n^2 = 2n^3$. Therefore, the complexity of the algorithm is $Q(n^3)$.

Q.4.(b) Generate the minimum spanning tree of the following connected graph using Kruskal's algorithm. (10)

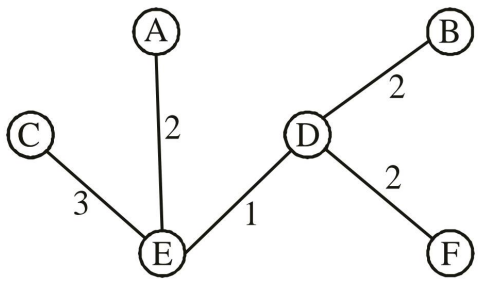


Ans. Minimum Spanning Tree:

Edges	**Weight**
(E,D)	1
(F,D)	2
(D,B)	2
(E,A)	2
(E,C)	3
(B,F)	3
(A,C)	4
(F,E)	4
(A,B)	5
(A,D)	6

RG PUBLICATION

1. *Pick edge E,D:* No cycle is formed, include it.
2. *Pick edge F,,D:* No cycle is formed, include it.
3. *Pick edge D,B:* No cycle is formed, include it.
4. *Pick edge E,A:* No cycle is formed, include it.
5. *Pick edge E,D:* No cycle is formed, include it.
6. *Pick edge E,C:* No cycle is formed, include it.
7. *Pick edge E,D:* No cycle is formed, include it.
8. *Pick edgeB,F:* Cycle is formed, Exclude it.
9. *Pick edgeA,C:* Cycle is formed, Exclude it.
10. *Pick edgeF,E:* Cycle is formed, Exclude it.
11. *Pick edgeA,B:* Cycle is formed, Exclude it.
12. *Pick edgeA,D:* Cycle is formed, Exclude it.



Q.5.(a) Consider a Knapsack instance :

Number of objects $n = 4$

Weights $Wi = (15, 10, 9, 5)$

Profits $Pi = (1, 5, 3, 4)$

Knapsack capacity $m = 8$

Use dynamic approach to find the solution. **(10)**

Ans. Number of objects $n = 4$

Weights $Wi = (15, 10, 9, 5)$

Profits $Pi = (1, 5, 3, 4)$

Knapsack capacity $m = 8$

$x_0 \quad x_1 \quad x_2 \quad x_3$

(P, W) Pairs are {(1,15), (5,10), (3,1), (4,5)}

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

$S_1^0 = \{(1,15)\}$

$S^1 = \{(0,0), (1,15)\}$

$S_1^1 = \{(5,10), (6,25)\}$

$S^2 = \{(0,0), (1,15), (5,10), (6,25)\}$

$S_1^2 = \{(3,9), (4,24), (8,19), (9,34)\}$

$S^3 = \{(0,0), (1,15), (5,10), (6,25), (3,9), (4,24), (8,19), (9,34)\}$

$S_1^3 = \{(4,5), (5,20), (9,15), (10,30), (7,14), (8,29), (12,24), (13,39)\}$

Solution ={ (0,0), (1,15), (5,10), (6,25), (3,9), (4,24), (8,19), (9,34), (4,5) (5,20) (9,15) (10,30), (7,14), (8,29), (12,24), (13,39)}

Q.5.(b) Explain optimal substructure property. Do greedy and dynamic programming strategies differ on this property? If not, where and how do they differ? **(10)**

Ans. Optimal substructure: An optimal solution to the problem contains an optimal solution to sub-problems. Optimal Substructure property makes greedy algorithms look like dynamic programming. However, the two techniques are quite different.

Yes, greedy and dynamic programming strategies differ on this property. Dynamic programming is applicable to problems exhibiting the properties of:

- overlapping sub-problems, and
- Optimal substructure.

Optimal substructure means that you can greedily solve sub problems and combine the

solutions to solve the larger problem. The difference between dynamic programming and greedy algorithms is that the sub problems overlap. That means that by "memoizing" solutions to some sub problems, you can solve other sub problems more quickly.

Section – C

Q.6.(a) Explain Huffman codes to generate the optimal prefix codes . (10)

Ans. Huffman coding is an entropy encoding algorithm used for compression, where the variable-length code table has been derived in a particular way based on the estimated probability of occurrence for each possible value of the source symbol.

Huffman coding uses a specific method for choosing the representation for each symbol, resulting in a prefix code that expresses the most common source symbols using shorter strings of bits than are used for less common source symbols. Huffman was able to design the most efficient compression method of this type: no other mapping of individual source symbols to unique strings of bits will produce a smaller average output size when the actual symbol frequencies agree with those used to create the code. The running time of Huffman's method is fairly efficient; it takes operations to construct it.

Q.6.(b) Explain *LC* branch and bound techique. (10)

Ans. When a node is visited the first time called an E-node, all its children are generated and saved into a data structure called live-node. If the child is not bounded; the structure could be a LC, or Least Cost. Exactly one node is pulled out the live-node list at a time, which becomes an E-node, to be "expanded" next. The LC (least-cost) branch-and-bound strategy uses heuristics to determine the best node for expansion from the current live nodes.

Example: The 15-puzzle

– 15 numbered tiles on a square frame with a capacity for 16 tiles.

– Given an initial arrangement, transform it to the goal arrangement through a series of legal moves.

– Legal move involves moving a tile adjacent to the empty spot E to E.

– Four possible moves in the initial state above: tiles 2, 3, 5, 6.

– Each move creates a new arrangement of tiles, called state of the puzzle.

– Initial and goal states.

– A state is reachable from the initial state iff there is a sequence of legal moves from initial state to this state.

– The state space of an initial state is all the states that can be reached from initial state.

– Search the state space for the goal state and use the path from initial state to goal state as the answer.

– Number of possible arrangements for tiles: 16! H" 20.9 × 1012.

– Check whether the goal state is reachable from initial state.

Number the frame positions from 1 to 16pi is the frame position containing tile numbered i. p16 is the position of empty spot. For any state, let li be the number of tiles j such that j < i and pj > pi. For the initial arrangement above, l1 = 0, l4 = 1, and l12 = 6.

Let x = 1 if in the initial state, the empty spot is in one of the following positions: 2, 4, 5, 7, 10, 12, 13, 15; otherwise x = 0.

Q.7.(a) Explain backtracking. Write an algorithm for 8 queens problem. Also compute its complexcity. (10)

Ans. Backtracking is a refinement of the brute force approach, which systematically searches for a solution to a problem among all available options. It does so by assuming that the solutions are represented by vectors (v_1, ..., v_m) of values and by traversing, in a depth first manner, the domains of the vectors until the solutions are found.

8 queens problem:

- Start with one queen at the first column first row
- Continue with second queen from the second column first row
- Go up until find a permissible situation
- Continue with next queen

We place the first queen on A1. Note the positions which Q1 is attacking. So the next queen Q2 has to options : B3 or B4. We choose the first one B3. We backtrack to Q3 and try to find admissible place different from C1. Again we need to backtrack. Q2 has no other choice and finally we reach Q1. We place Q1 on A3.

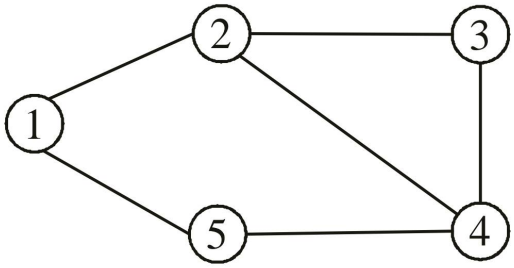
```
intPlaceQueen(int board[8], int row)
If (Can place queen on ith column)
PlaceQueen(newboard, 0)
Else
PlaceQueen(oldboard,oldplace+1)
End
```

Complexity : Placing queens takes constant time/queen, O(n)

Drawing the board takes $O(n^2)$ time

Q.7.(b) What are Hamiltonian cycle? Write an algorithm that finds all Hamiltonian cycles in a graph using backracking. (10)

Ans. Homiltonian Cycle : It is a cycle in which when we start from the initial node, we cover all the nodes present is a graph without repeating any node so finally arrive on initial node.



The hamiltonian cycle of this graph is 1-2-3-4-5-1

Algorithm Hamittonion(k)

```
{
    repeat
        {
          Next value (k),
                        If (x [k] = 0) then return
                        If (k = n) then write (x[1 : n]),
          Hamittonion (k+1),
        }
    until (false);
}
```

RG PUBLICATION

Section – D

Q.8.(a) State and prove Cook's theorem. (10)

Ans. Cook's theorem, states that the Boolean satisfiability problem is NP-complete. That is, any problem in NP can be reduced in polynomial time by a deterministic Turing machine to the problem of determining whether a Boolean formula is satisfiable.

Proof-idea:

First show that SAT is in NP.

Then show that every problem in NP is polynomial-time reducible to SAT.

Proof :

It is easy to see that SAT is in NP. Simply non-deterministically choose Boolean values for each variable. Then check whether the expression is true or false for those values.

If it is true, the answer is "yes."

If it is false, the answer is no."

Proof :

Let A be a problem in NP.

Then there is a nondeterministic Turing machine M that decides A in polynomial time. Let the run time of M be at most n k, for some k 0. When M is run, it can access at most the ûrst n k + 1 cells.

Q.8.(b) What do you mean be *NP*-scheduling problems? Show that the job sequencing with deadline problem is *NP* hard. (10)

Ans. The problem is stated as below.

- There are n jobs to be processed on a machine.
- Each job i has a deadline d_ie" 0 and profit p_ie"0 .
- Pi is earned iff the job is completed by its deadline.
- The job is completed if it is processed on a machine for unit time.
- Only one machine is available for processing jobs.
- Only one job is processed at a time on the machine.
- A feasible solution is a subset of jobs J such that each job is completed by its deadline.
- An optimal solution is a feasible solution with maximum profit value.

A scheduling problem is NP-hard in the ordinary sense if

- partition (or a similar problem) can be reduced to this problem with a polynomial time algorithm and
- there is an algorithm with pseudo polynomial time complexity that solves the scheduling problem.

Q.9.(a) What do you mean by Clique decision problem? (10)

Ans. In computer science, the **clique problem** refers to any of the problems related to finding particular complete subgraphs ("cliques") in a graph, i.e., sets of elements where each pair of elements is connected.

For example, the **maximum clique problem** arises in the following real-world setting. Consider a social network, where the graph's vertices represent people, and the graph's edges represent mutual acquaintance. To find a largest subset of people who all know each other, one can systematically inspect all subsets, a process that is too time-consuming to be practical for social networks comprising more than a few dozen people. Although this brute-force search can be improved by more efficient algorithms, all of these algorithms take exponential time to solve the problem. Therefore, much of the theory about the clique problem is devoted to identifying special types of graph that admit more efficient algorithms, or to establishing the computational difficulty of the general problem in various models of computation. Along with its applications in social networks, the clique problem also has many applications in bioinformatics and computational chemistry.

Clique problems include:

·- finding the·maximum clique (a clique with the largest number of vertices),

- finding a maximum weight clique in a weighted graph,

- listing all maximal cliques (cliques that cannot be enlarged)

- solving the decision problem of testing whether a graph contains a clique larger than a given size.

Q.9.(b) Prove that the class *NP* of languages is closed under union, intersection, concatenation and kleen star. (10)

Ans. It is an open problem whether NP is closed under complement or not. The proofs for the remaining four language operations can go as follows. Assume that $L_1, L_2 \in$ NP. This means that there are nondeterministic deciders M_1 and M_2 such that M_1 decides L_1 in nondeterministic time $O(n^k)$ and M_2 deciders L_2 in nondeterministic time $O(n^l)$. We want to show that

1. there is a nondeterministic poly-time decider M such that $L(M) = L_1 \cap L_2$, and
2. there is a nondeterministic poly-time decider M such that $L(M) = L_1 \cup L_2$, and
3. there is a nondeterministic poly-time decider M such that $L(M) = L_1 \circ L_2$, and
4. there is a nondeterministic poly-time decider M such that $L(M) = L_1^*$

Now we provide the four machines M for the different operations. The constructions are the standard ones, the additional part is the complexity analysis of the running time. Note that we can use the power of nondeterministic choices to make the constructions very simple.

(1) Intersection:

M = "On input w:

(i) Run M_1 on w. If M_1 rejected then reject.

(ii) Else run M_2 on w. If M_2 rejected then reject.

(iii) Else accept."

Clearly, the longest branch in any computation tree on input w of length n is $O(n^{max\{k, l\}})$. So M is a poly-time nondeterministic decider for $L_1 \cap L_2$.

(2) Union:

M = "On input w:

(i) Run M_1 on w. If M_1 accepted then accept.

(ii) Else run M_2 on w. If M_2 accepted then accept.

(iii) Else reject."

Clearly, the longest branch in any computation tree on input w of length n is $O(n^{max\{k, l\}})$. So M is a poly-time nondeterministic decider for $L_1 \cup L_2$. Note that in our case, we do not have the run M_1 and M_2 in parallel, as it was necessary e.g. in the proof that recognizable languages are closed under union. Another possible construction would be to nondeterministically choose either M_1 or M_2 and simulate only the selected machine.

(3) Concatenation:

M = "On input w:

(i) Nondeterministically split w into w_1, w_2 such that $w = w_1w_2$.

(ii) Run M_1 on w_1. If M_1 rejected then reject.

(iii) Else run M_2 on w_2. If M_2 rejected then reject.

(iv) Else accept."

Clearly, the longest branch in any computation tree on input w of length n is still $O(n^{max\{k, l\}})$ because step (i), takes only $O(n)$ steps on e.g. a two tape TM. So M is a poly-time nondeterministic decider for $L_1 \circ L_2$.

(4) Kleen star:

M = "On input w:

(i) If $w = \in$ then accept.

(ii) Nondeterministically select a number m such that $1 \leq m \leq |w|$.

(iii) Nondeterministically split w into m pieces such that $w = w_1 w_2 \ldots w_m$.

(iv) For all i, $1 \leq i \leq m$: run M_1 on w_i. If M_1 rejected then reject.

(v) Else (M_1 accepted all w_i, $1 \leq i \leq m$), accept."

Observe that steps (i) and (ii) take time $O(n)$, because the size of the number m is bounded by n (the length of the input). Step (iii) is also doable in polynomial time (e.g. by nondeterministically inserting m separation symbols # into the input string w). In step (iv) the for loop is run at most n times and every run takes at most $O(n^k)$. So the total running time is $O(n^{k+1})$. This means that M is a poly-time nondeterministic decider for L_1.

RG PUBLICATION

ANALYSIS AND DESIGN OF ALGORITHM

May 2014

Paper Code: CSE -306-F

Figures in the bracket indicate full marks

Note: ***Question No.1 is compulsory. Attempt five questions in total selecting one question from each Sections.***

Q.1.(a) Define algorithm. **(2)**

Ans. The algorithm is defined as a collection of unambiguous instructions occurring in some specific sequence and such an algorithm should produce output for given set of input in finite amount of time.

Q.1.(b) Define Big Theta Notations? **(2)**

Ans. Big Theta Notation (θ) : The lower the upper bound for the function *f* is provided by the big theta notation (θ).

Consider '*g*' be the function from the non negative integer in to the positive real number's Then

$$\theta\ (g) = O(g) \cap \theta(g)$$

i.e., the set of function are both in O(*g*) and Ω(*g*).

The θ(*g*) is the set of function '*f* ' such that for same positive constant C_1 and C_2 and a number exists such that

$$C_1(g(n))\, f(n) \le C_2\ (g(n)) \text{ for all } n,\ n \ge n_0$$

By $f \in \theta(g)$ we mean "*f* is order *g*".

Q.1.(c) Define feasible and optimal solution. **(2)**

Ans. Feasible solution is any element of the feasible region of an optimization problem. The feasible region is the set of all possible solutions of an optimization problem. An optimal solution is one that either minimizes or maximizes the objective function.

Q.1.(d) What do you mean by Amortized Analysis? **(2)**

Ans. Amortized analysis is a method of analyzing algorithms that considers the entire sequence of operations of the program. It allows for the establishment of a worst-case bound for the performance of an algorithm irrespective of the inputs by looking at all of the operations.

Q.1.(e) Define feasible and optimal solution. **(2)**

Ans. Feasible solution is any element of the feasible region of an optimization problem. The feasible region is the set of all possible solutions of an optimization problem. An optimal solution is one that either minimizes or maximizes the objective function.

Q.1.(f) Write any two characteristics of Greedy Algorithm? **(2)**

Ans. Greedy is a strategy that works well on optimization problems with the following characteristics :

(i) *Greedy-choice property* : A global optimum can be arrived at by selecting a local optimum.

(ii) *Optimal substructure :* An optimal solution to the problem contains an optimal solution to subproblems.

Q.1.(g) Define principle of optimality. (2)

Ans. The principle of optimality states that ″in an optimal sequence of decisions or choices, each subsequence must also be optimal.

When it is not possible to apply the principle of optimality it is almost impossible to obtain the solution using the dynamic programming approach.

For example : Finding of shortest path in a given graph uses the principle of optimality.

Q.1.(h) Define a planar graph. (2)

Ans. A planar graph is a graph that can be embedded in the plane, i.e., it can be drawn on the plane in such a way that its edges intersect only at their endpoints. In other words, it can be drawn in such a way that no edges cross each other.

Q.1.(i) What are the searching techniques that are commonly used in Branch-and-Bound method? (2)

Ans. Searching techniques that are commonly used in Branch-and-Bound method are as follows :

(i) FIFO (First In First Out)

(ii) LIFO (Last In First Out)

(iii) LC (Least Cost)

RG PUBLICATION

Q.1.(j) What is approximate solution? (2)

Ans. Approximate solution : A feasible solution with are the value close to the value of an optimal solution is called an approximation solution and algorithm is known as approximation algorithm.

An approximation algorithm for *P* is an algorithm that generate approximate solution for *P*.

A approximate algorithm for a problem is an absolute approximation for solution *P* if very instance *I* of *P* is :

$$| A^*(I) - \hat{6}(I) | \leq n$$

where n is same constant.

where $A^*(I)$ = value of optimal solution.

$\hat{6}(I)$ = value of feasible solution.

Section-A

Q.2.(a) What is an algorithm? Explain the property of an algorithm. (10)

Ans. Algorithm : The algorithm is defined as a collection of unambiguous instructions occurring in some specific sequence and such an algorithm should produce output for given set of input in finite amount of time.

Properties of the algorithm are as follows :

(i) *Finiteness* : An algorithm terminates after a finite numbers of steps.

(ii) *Definiteness* : Each step in algorithm is unambiguous. This means that the action specified by the step cannot be interpreted (explain the meaning of) in multiple ways & can be performed without any confusion.

(iii) *Input* : An algorithm accepts zero or more inputs

(iv) *Output* : It produces at least one output.

(v) *Effectiveness* : It consists of basic instructions that are realizable. This means that the instructions can be performed by using the given inputs in a finite amount of time.

(vi) It is written in simple English.

(vii) Each step of an algorithm is unique and should be self explanatory.

(viii) Should be unambiguous, precise and lucid.

(ix) Should provide the correct solutions.

(x) Should have an end point.

(xi) The output statements should follow input, process instructions.

(xii) The initial statements should be of input statements.

Q.2.(b) What do you mean by time complexity of an algorithm? Explain the notations used to denote the time complexity of an algorithm. (10)

Ans. Time complexity : The time complexity of an algorithm is the amount of computer time required by an algorithm to run to completion.

Various notations used to denote the time complexity of an algorithm are as follows:

(i) Big oh notation : The Big oh notation is denoted by 'O'. It is a method of representing the upper bound of algorithm's running time. Using big oh notation we can give longest amount of time taken by the algorithm to complete.

Let $F(n)$ and $g(n)$ be two-negative functions.

Let n_0 and constant c are two integers such that n_0 denotes some value of input and $n > n_0$. Similarly c is some constant such that c > 0. We can write

$$F(n) \leq c * g(n)$$

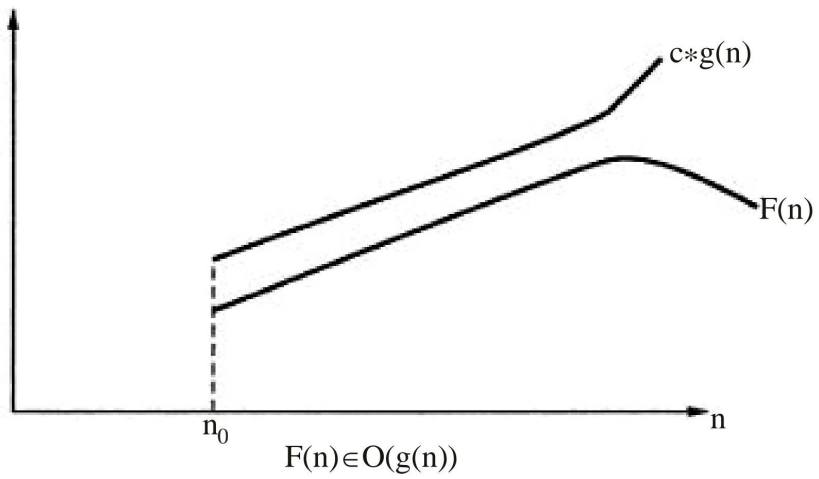


Fig.(1)

Then $F(n)$ is big oh of $g(n)$. It is also denoted as $F(n) \in O\ g(n)$. In other words $F(n)$ is less than $g(n)$ is multiple of some constant c.

(ii) Omega notation : Omega notation is denoted by Ω. This notation is used to represent the lower bound of algorithm's running time. Using omega notation we can denote shortest amount of time taken by algorithm.

A function $F(n)$ is said to be in $\Omega\ g(n)$, if $F(n)$ is bounded below by some positive constant multiple of $g(n)$ such that

$$F(n) \geq c * g(n) \qquad \text{for all } n \geq n_0$$

It is denoted as $F(n) \in \Omega\ g(n)$. Following graph illustrates the curve for Ω notation.

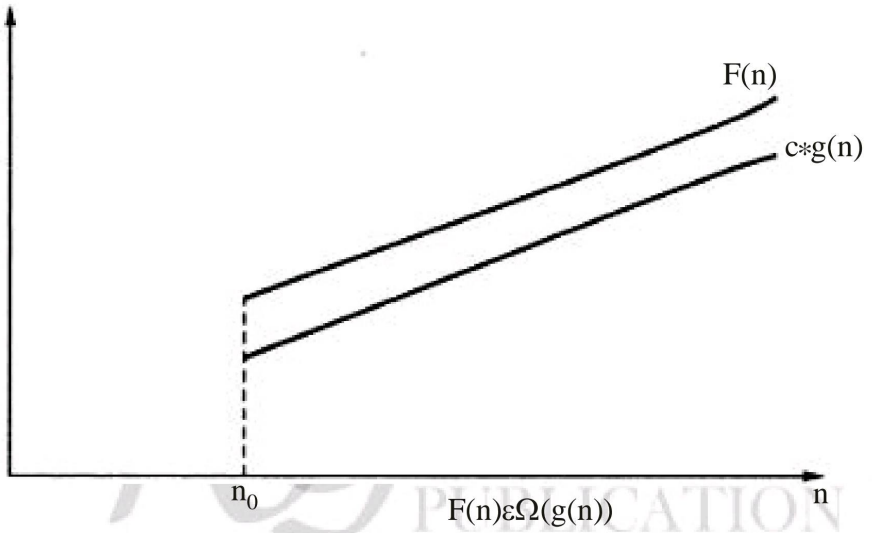
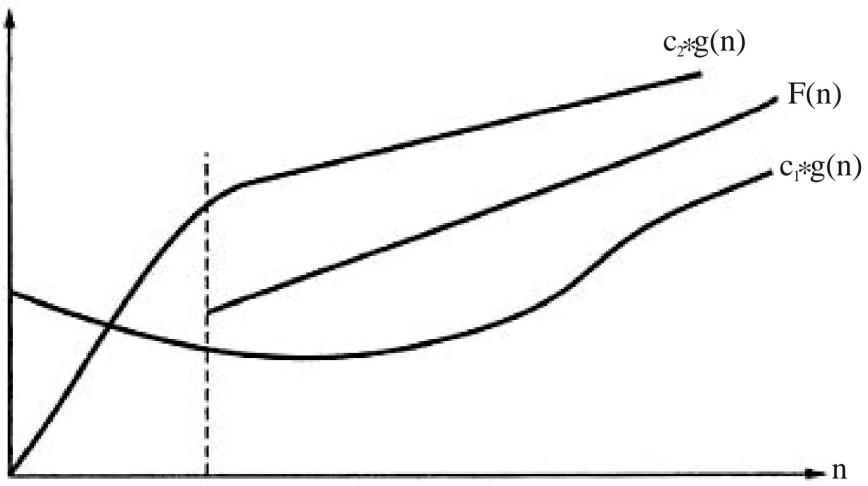


Fig.(2)

(iii) Θ Notation : The theta notation is denoted by Θ. By this method the running time is between upper bound and lower bound.



Theta notation F(n)ε Θ (g(n))

Fig.(3)

Let $F(n)$ and $g(n)$ be two non negative functions. There are two positive constants namely c_1 and c_2 such that

$$c_1 \le g(n) \le c_2 g(n)$$

Then we can say that

$$F(n) \in \Theta g(n)$$

Q.3. Explain the following:

(a) Selection sort.

(b) Disjoint Sets. **(20)**

Ans. (a) Selection sort : Selection sort is a sorting algorithm, specifically an in-place comparison sort. It has O(n^2) time complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity, and it has performance advantages over more complicated algorithms in certain situations, particularly where auxiliary memory is limited.

The algorithm divides the input list into two parts: the sublist of items already sorted, which is built up from left to right at the front (left) of the list, and the sublist of items remaining to be sorted that occupy the rest of the list. Initially, the sorted sublist is empty and the unsorted sublist is the entire input list. The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sublist, exchanging it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right.

Here is an example of this sort algorithm sorting five elements :

64 25 12 22 11 // this is the initial, starting state of the array

11 25 12 22 64 // sorted sublist = {11}

11 12 25 22 64 // sorted sublist = {11, 12}

11 12 22 25 64 // sorted sublist = {11, 12, 22}

11 12 22 25 64 // sorted sublist = {11, 12, 22, 25}

11 12 22 25 64 // sorted sublist = {11, 12, 22, 25, 64}

Ans. (b) Disjoint sets : Two or more sets are said to be disjoint if they have no element in common. For example, {1, 2, 3} and {4, 5, 6} are disjoint sets, while sets {1, 2, 3} and {2, 4, 5, 6} are not. The elements of a set are letters, numbers, or any other objects. For example, the elements of the set {a, b, c} are the letters a, b, and c.

Formally, two sets A and B are disjoint, if their intersection is the empty set, that is if $A \cap B = \Phi$. This definition extends to any collection of sets. A collection of sets is pairwise disjoint or mutually disjoint if any two distinct sets in the collection are disjoint. Formally, let I be an index set, and for each i in I, let A_i be a set. Then, the family of sets $\{A_i, : i \in I\}$ is pairwise disjoint if for any i and j in I with $i \ne j, A_i \cap B_j = \Phi$. For example , the collection of sets {(1), (2), (3), ...} is pairwise disjoint. If $\{A_i\}$ is a pairwise disjoint collection (containing at least two sets), then clearly its intersection is empty.

Section-B

Q.4. What do you mean by dynamic Programming? Explain 0/1 knapsack problem by using dynamic programming. (20)

Ans. Dynamic programming : Dynamic programming is technique for solving problems with overlapping subproblems. In this method each subproblem is solved only once. The result of each subproblem is recorded in a table from which we can obtain a solution to the original problem.

The 0/1 Knapsack Problem : We have a knapsack (bag) with capacity, W = 20. The weights and values of five items are given below.

Table : Kanpsack example

item (i):	1	2	3	4	5
w_i:	3	4	7	8	9
v_i:	4	5	10	11	13

The objective of knapsack problem is to fill the knapsack with items to maximize the total value subject to its capacity. It is known as 0/1 knapsack problem, because we put one item into the knapsack or not (may not include a fraction of an item). The following decisions are taken to optimize the total value.

(i) Highest value, v_5 = 13, add w_5 = 9 to knapsack. Total value is 13 and knapsack remaining capacity is 11 (20 – 9).

(ii) Next higher value, v_4 = 11, add w_4 = 8 to knapsack. Total value is 24 (13 + 11) and knapsack remaining capacity is 3 (11 – 8).

(iii) Next higher value, v_3 = 10, we cannot add w_3 = 7 to knapsack. Because it exceeds the knapsack capacity (9 + 8 + 7 = 24 > 20).

(iv) Next higher value, v_2 = 5, we cannot add w_2 = 4 to knapsack. Because it exceeds the knapsack capacity (9 + 8 + 4 = 21 > 20).

(v) Next higher value, v_1 = 4, add w_1 = 3 to kanpsack. Total value is 28 (13 + 11 + 4) and knapsack remaining capacity is 0 (3 – 3).

(vi) Maximum value is 28 (items included in knapsack : 1, 4, and 5).

The knapsack problem shows both overlapping subproblems and optimal substructure. Say there are n items with weights w_1, ... w_n and their values v_1, ... v_n. In general, the 0-1 knapsack problem can be expressed as follows :

Maximize total value of items included in the knapsack : $v_1 x_1 + v_2 x_2 + ... v_n x_n$

Subject to the knapsack capacity constraint : $w_1x_1 + w_2x_2 + ... + w_nx_n \quad W$

where x_1, x_2, ... x_n take a value either 0 or 1.

The knapsack example given in table is formulated as follows :

Maximize : $4x_1 + 5x_2 + 10x_3 + 11x_4 + 13x_5$

Subject to : $3w_1 + 4w_2 + 7w_3 + 8w_4 + 9w_5 \quad 20.$

Optimal solution is : $x_1 = 1, x_2 = 0, x_3 = 0, x_4 = 1$ and $x_5 = 1$, with a maximum value of 28 (4 + 11 + 13).

To formulate the problem in terms of dynamic programming, we define a subproblem $S[k, u]$, which will be the optimal solution for the first k items, involving up to u weight. $S[k, u]$

is a recursive formula, which will specify how the subproblem can be solved. The real insight of dynamic programming is that we will take this recursive solution and convert it into an iterative solution, just as we did with Fibonacci numbers.

We will formulate this in terms of the solution that includes any of the items 1.... k-1. If we have the optimal solution for a given weight u for these k-1 items, then the optimal solution for the first k items will be described as follows. There are two main cases :

(i) Case : $w_k > u$. As the weight of item k is greater than the weight constraint that we are working with, item k cannot be included in the knapsack.

(ii) Case : w_k u. If item k is included, the value will be equal to the value of item k plus the maximum value of $k - 1$ previous items, subject to a total weight of $u–w_k$, since we have to include w_k without exceeding the weight limit u. This is equivalent to $S[k–1, u - w_k] + v_k$. This leaves us two sub-cases.

(a) If this total is less than the maximum for the first k–1 values, we will not include item k. The new value will be $S[k - 1, u]$.

(b) If it is larger, then we will include item k, and $S[k–1, u–w_k] + v_k$ will be the new value.

Thus, the recursive definition for the optimum is given as follows :

$$S[k,\ u] = \begin{cases} s[K-1, u] & \text{if } w_k > u \\ \max\{S[k-1,u],\ S[k-1,u-w_k]+v_k\} & \text{otherwise} \end{cases}$$

We can store the subproblems in a two-dimensional array, with first subscript being the items that we choose from 1 to n, and the second being the weight that we fill from 0 to W. For any given k, we can find all of the values of $S[k, u]$ for $u = 0$ to W.

Algorithm : knapsack [*w* (1..*n*), *v*(1..*n*), *W*]

```
For u = 0 to W
      S[0, u] = 0
Endfor
For i = 0 to n
      S[i, 0] = 0
Endfor
For i = 0 to W
      For u = 0 to W
                        If [w (i) u] then          // item fits in knapsack
                           If (v[i] + S[i-1, u-w[i]] > S [i-1, u]) then
                                 S[i, u] = v[i] + S [i-1, u-w[i]]
                           Else S[i, u] = S[i-1, u]
                        Else S[i, u] = S[i-1, u]
      Endfor
Endfor
Print S[n][W]          // max value of items included in the knapsack
// Find knapsack items
i = n, k = W
While (i > 0 and k > 0)
```

```
If (S[i] k] ≠ S[i-1][k])
        Print i          //item i is in the knapsack
        k = k – [i].
Endif
i = i – 1
Endwhile
```

Q.5.(a) What is minimum spanning tree? Explain. (10)

Ans. Minimum spanning trees : A minimum spanning tree (MST) or minimum weight spanning tree is then a spanning tree with weight less than or equal to the weight of every other spanning tree. More generally, any undirected graph has a minimum spanning forest, which is a union of minimum spanning trees for its connected components.

Practical applications based on minimal spanning trees include :

– Taxonomy, one of the earliest motivating applications.

– Cluster analysis: clustering points in the plane, single-linkage clustering , graph-theoretic clustering, and clustering gene expression data. Constructing trees for broadcasting in computer networks.

– Image registration and segmentation *e.g.* see minimum spanning tree-based segmentation.

– Curvilinear feature extraction in computer vision.

– Handwriting recognition of mathematical expressions.

– Circuit design: implementing efficient multiple constant multiplications, as used in finite impulse response filters.

– Regionalisation of socio-geographic areas, the grouping of areas into homogeneous, contiguous regions.

– Comparing ecotoxicology data.

– Topological observability in power systems.

– Measuring homogeneity of two-dimensional materials.

– Minimax process control

Method to generate MST: Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a connected weighted undirected graph.

If a graph is empty then we are done immediately. Thus, we assume otherwise. The algorithm starts with a tree consisting of a single vertex, and continuously increases its size one edge at a time, until it spans all vertices.

- *Input* : A non-empty connected weighted graph with vertices V and edges E (the weights can be negative).
- Initialize : $V_{new} = \{x\}$, where x is an arbitrary node (starting point) from V, $E_{new} = \{\}$
- Repeat until $V_{new} = V$:
- 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

Q.5.(b) Explain the job sequencing with deadlines. (10)

Ans. Job sequencing with deadlines : The arrangement of jobs on a single processor with deadline constraints is named as job sequencing with deadlines. We arrange n jobs on a processor in a sequence to obtain maximum profit subject to deadlines.

Given a set of n jobs, each having a "deadline" (an integer) $d[i]$ and a profit $p[i]$ associated with it. For a job i the profit is earned if and only if the job is completed within its deadline. Each job takes one unit of time on a machine (processor) and only one machine is available. We want to maximize the profit. The jobs are arranged in decreasing order of profit in an array.

A feasible solution to this problem is a subset J of the n jobs, such that each of them can be completed within its deadline. The value of feasible solution if $\sum p[i]$ for all $i \in J$. An optimal solution is a feasible solution with maximum value.

For example, let $n = 4$, profit vector $P = \{30, 35, 20, 25\}$, deadline vector $D = \{2, 1, 2, 1\}$.

Assumption : Execution time of each job is one time unit.

The feasible solutions and their values are :

Sol. no	solution	processing sequence	value
1	1, 4	4 1	55
2	1, 3	1 3 or 3 1	50
3	1, 2	2 1	65
4	3, 4	4 3	45
5	2, 3	2 3	55
6	1	1	30
7	2	2	35
8	3	3	20
9	4	4	25

RG PUBLICATION

It is seen that solution no. 3 is optimal, where only jobs 1 and 2 are processed and the profit is maximum. Note that the jobs must be done in the order shown, first 2 and then 1, in order to obey the deadlines.

In order to use the greedy strategy, we have to decide the selection criterion. As a first attempt we can use $\sum_{i \in j} p_i$ as our optimisation measure. Then we select the next job which increases this measure the most, provided that the resulting J is a feasible solution. Thus we should consider the jobs in non-increasing order of profit p. This was the reason that in the above example, the jobs are given in such an order. Considering the jobs in order of decreasing p values, the 1st can be in J because $J = \{1\}$is a feasible solution. Then the 2nd can be in J, because $J = \{1, 2\}$ is a feasible solution, since the deadlines can be met. Now job 3 cannot be in J as $\{1, 2, 3\}$ is not a feasible solution. The largest value of the deadline is 2 and as each included job would take 1 time unit each, we can have at most 2 jobs in J. So we have $J = \{1, 2\}$.

What should be the order of execution of the jobs? They should be executed earliest deadline first, that is, in the non-decreasing order of the deadline d. Thus the jobs $\{1, 2\}$ should be done in the order 2, 1.

The greedy method given above always obtains an optimal solution to the job sequencing problem. We give a high level algorithm for this method as follows :

Algorithm :

1. [J is an output vector of selected jobe]
2. [Job numbers are in non-increasing order of profit, *d* is deadline vector]
3. local *i* ;
4. $J \leftarrow i$; [best job always selected]
5. for $i \leftarrow 2$ to n do
6. if all jobs in J U { i } can be completed by their deadlines then
7. $J \leftarrow J$ U i ;
8. end
9. end

We can keep the jobs in the set J in an array j { } sorted by non-decreasing order of deadlines, which will be easier to perform the feasibility check. On the other hand, we assume that job numbers are given in non-increasing order of profit p, thus their deadlines are stored in d [ ] in that order.

In the C language implementation, a ficititious job no. 0 is used with d [0] = 0 and j [0] = 0 to simplify the insertion of a job i in J in the deadline order. The jobs in J with subscript larger than the new job i are to be moved. Only the jobs after the insertion point for i, say k, need be checked for their deadlines. It is to be noted that the atual values of profit do not enter into consideration in this algorithm.

Section-C

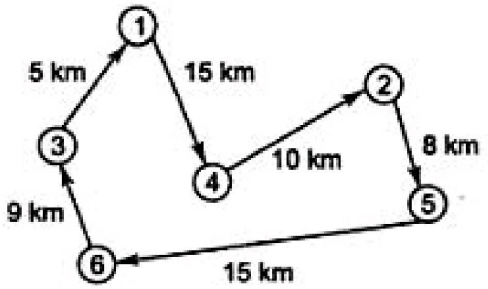
Q.6. What is branch and bound method? Solve the traveling salesperson problem with branch and bound method by taking a suitable example. (20)

Ans. Branch and bound method : Branch-and-bound (B & B) is an algorithmic technique to find the optimal solution by keeping the best solution found so far. If a partial solution cannot improve on the best, it is discarded.

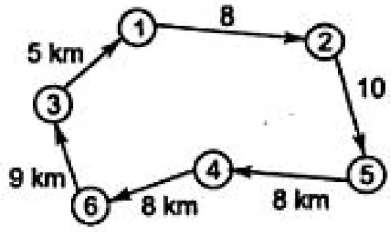
B & B can be applied to an assignment problem by assigning n persons to n jobs so that the total cost of assignment is a minimum. Each person can be assigned exactly one job.

Traveling-Salesman Problem : Travelling Salesman Problem (TSP) requires that we find the shortest path visiting each of a given set of cities once and returning to the starting point. In other words, the problem is to find, in an undirected graph with weights on the edges, a tour the sum of whose edge-weights is a minimum. A tour is a simple cycle that includes all the vertices and is often called a hamiltonian cycle. TSP is a famous example of a problem which can be solved by a B & B.

Problem : Given a complete undirected graph $G = (V, E)$ that has non-negative integer cost $C(u, v)$ associated with each edge (u, v) in E, the problem is to find a hamiltonian cycle (tour) of G with minimum cost.



Total = 62 km



Total = 48 km

A salesperson starts from the city 1 and has to visit six cities (1 through 6) and must come back to the starting city *i.e.*, 1. The first route (left side) $1 \rightarrow 4 \rightarrow 2 \rightarrow 5 \rightarrow 6 \rightarrow 3 \rightarrow 1$ with the total length of 62 km, is a relevant selection but is not the best solution. The second route (right side) $1 \rightarrow 2 \rightarrow 5 \rightarrow 4 \rightarrow 6 \rightarrow 3 \rightarrow 1$ represents to the must better solution as the total distance, 48 km, is less than for the first route.

Suppose C (A) denoted the total cost of the edges in the subset A subset of E i.e.,

$$C(A) = \sum_{u,v in A} C(u,v)$$

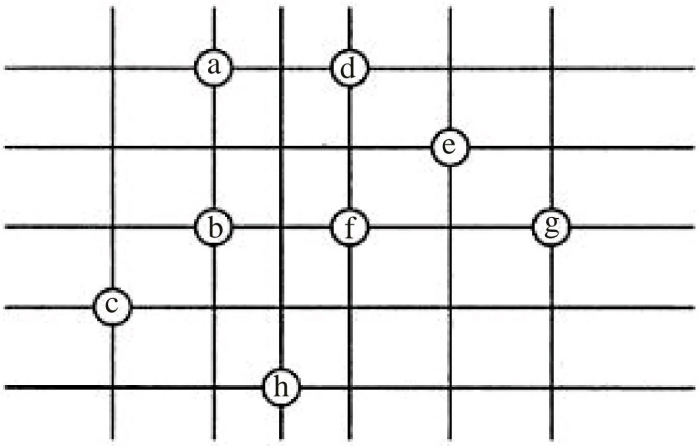
Moreover, the cost function satisfies the triangle inequality, that is, for all vertices u, v, w in v, w have $C(u, w) \leq C(u, v) + C(v, w)$.

Note that the TSP problem is NP-complete even if we require that the cost function satisfies the triangle inequality. This means that it is unlikely that we can find a polynomial time algorithm for TSP.

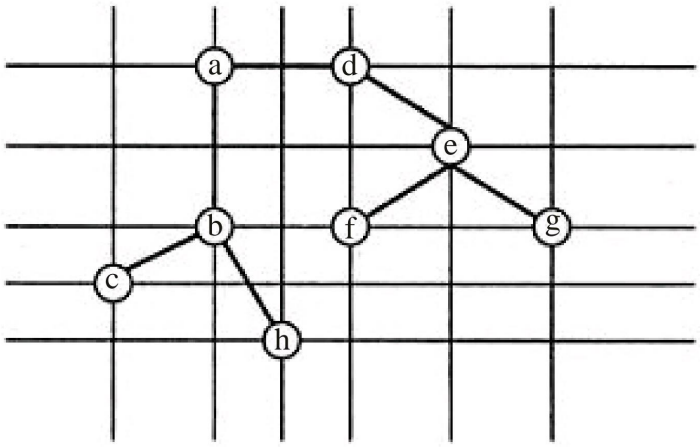
TSP with the triangle-inequality : When the cost function satisfies the triangle-inequality, we can design an approximate algorithm for TSP that returns a tour whose cost is not more than twice the cost of an optimal tour.

Outline of an APPROX-TSP-TOUR : First, compute a MST (minimum spanning tree) whose weight is a lower bound on the length of an optimal TSP tour. Then, use MST to build a tour whose cost is no more than twice that of MST's weight as long as the cost function satisfies triangle inequality.

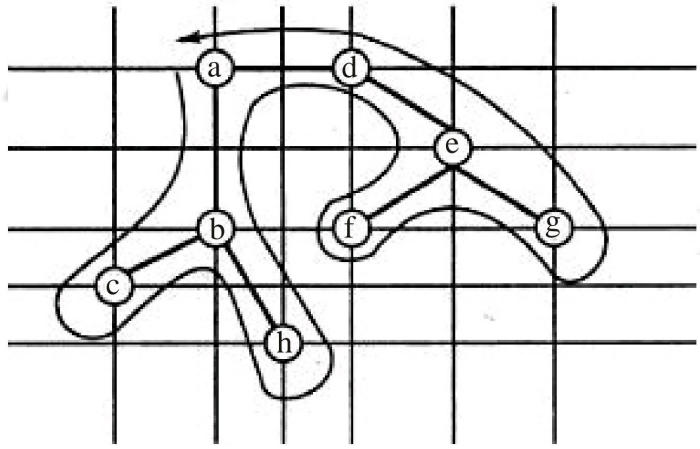
Operation of APPROX-TSP-TOUR : Let root r be a in the following given set of points (graph).



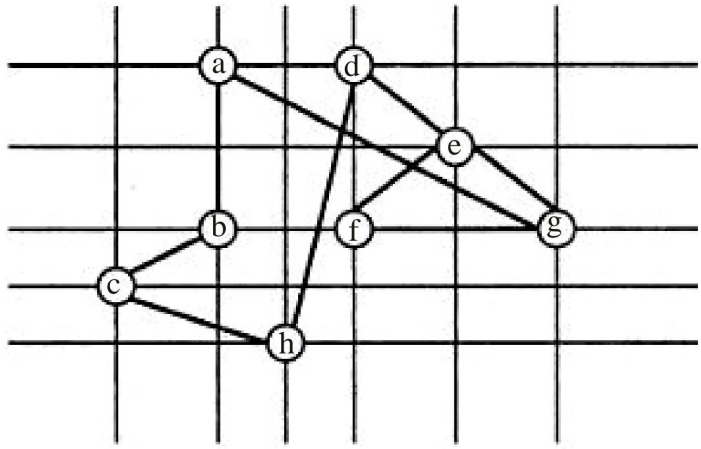
Construct MST from root a using MST-PRIM (G, C, r)



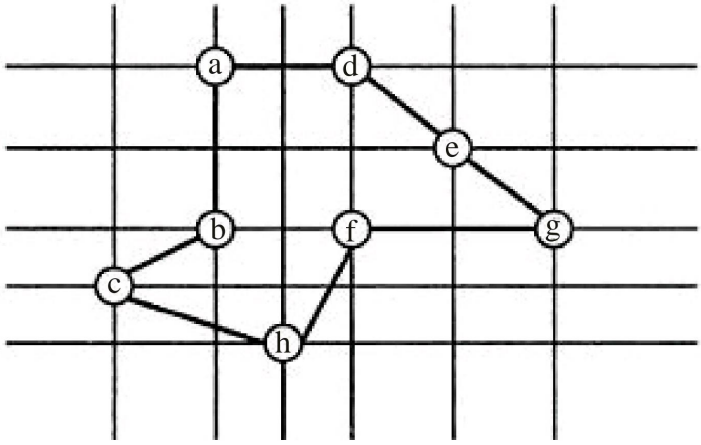
List vertices visited in pre-order walk $L = [a, b, c, d, e, f, g]$



Return hamiltonian cycle



Optimum TSP tour for a given problem (graph) would be



which is about 23 % shorter.

APPROX-TSX-TOUR (G, C)

(i) Select a vertex $r \in V$ [G] to be a root vertex.

(ii) Compute a minimum spanning tree T for G from root r using MST-PRIM (G, C, r).

(iii) Let L be the list of vertices visited in a pre-order tree walk of T.

(iv) Return the hamiltonian cycle H that visits the vertices in the order L.

Theorem : APPROX-TSP-TOUR is a polynomial-time 2-approximation algorithm for TSP with triangle inequality.

Proof 1. We have already shwon that APPROX-TSP-TOUR-time.

2. Let H denote the optimal tour. Observe that a TSP with one edge removed is a spanning tree (not necessarily MST). It implies that the weight of the MST in lower bound on the cost of an optimal tour.

$$C(T) = C(H)$$

A "full" walk, W, traverse every edge of MST, T exactly twice, that is,

$$C(W) = 2C(T)$$

which means

$$C(W) \leq 2C(H)$$

and we have

$$\frac{C(W)}{C(H)} \leq P(n) = 2$$

That is, the cost of walk, $C(W)$, of the solution produced by the algorithm is within a factor of $P(n) = 2$ of the cost $C(H)$ of an optimal solution.

Q.7. What is backtracking? Solve 8 queens problem with back tracking. (20)

Ans. Backtracking : Backtracking is a refinement of the brute force approach, which systematically searches for a solution to a problem among all available options. It does so by assuming that the solutions are represented by vectors (v_1, ..., v_m) of values and by traversing, in a depth first manner, the domains of the vectors until the solutions are found.

Backtracking is used to find all possible solutions available to the problem. It traverse tree by DFS(Depth First Search). It realizes that it has made a bad choice & undoes the last choice by backing up. It searches the state space tree until it found a solution. It involves feasibility function.

8 queens problem :

- Start with one queen at the first column first row
- Continue with second queen from the second column first row
- Go up until find a permissible situation
- Continue with next queen

We place the first queen on A1. Note the positions which Q1 is attacking. So the next queen Q2 has to options : B3 or B4. We choose the first one B3. We backtrack to Q3 and try to find admissible place different from C1. Again we need to backtrack. Q2 has no other choice and finally we reach Q1. We place Q1 on A3.

```
intPlaceQueen(int board[8], int row)
If (Can place queen on ith column)
PlaceQueen(newboard, 0)
Else
PlaceQueen(oldboard,oldplace+1)
End
```

Section-D

Q.8.(a) What are NP Hard and NP Complete problems? (10)

Ans. NP-hard : NP-hard (non-deterministic polynomial-time hard), in computational complexity theory, is a class of problems that are, informally, "at least as hard as the hardest problems in NP". A problem H is NP-hard if and only if there is an NP-complete problem L that is polynomial time Turing-reducible to H. L can be solved in polynomial time by an oracle machine with an oracle for H. Informally, we can think of an algorithm that can call such an oracle machine as a subroutine for solving H, and solves L in polynomial time, if the subroutine call takes only one step to compute. NP-hard problems may be of any type: decision problems, search problems, or optimization problems.

NP-hard problem are at least as hard as NP-Complete problem. Hence there are no polynomial problem algorithms defined.

Example of NP-hard problem is :

(a) Consider the halting problem for determining algorithms.

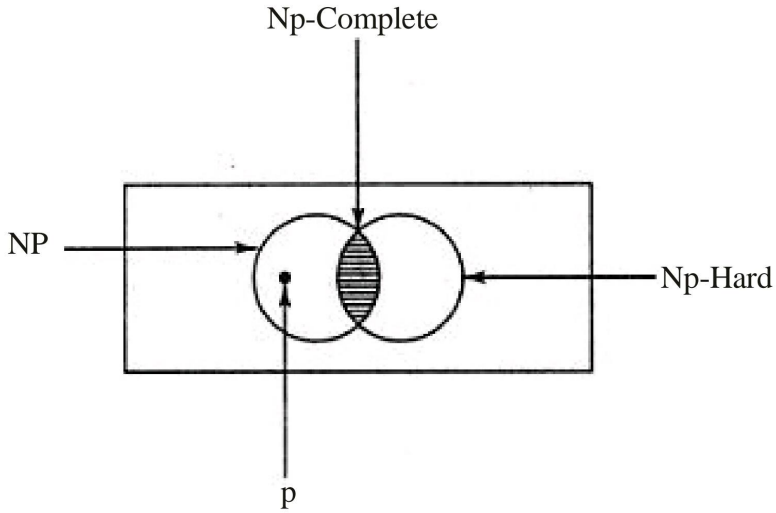
(b) It is well known that the halting problem is undecidable. Hence there exists no algorithm to solve the problem. Therefore it clearly cannot be NP.

NP-Complete : The complexity class NP-complete is a class of decision problems. A decision problem *L* is NP-complete if it is in the set of NP problems and also in the set of NP-hard problems. The abbreviation NP refers to "nondeterministic polynomial time."

A decision problem *D* is said to be NP-complete if

– It belongs to class NP.

– Every problem in NP is polynomially reducible to D.



From above diagram, there are two classes of problems NP-Hard and NP-complete. A problem is NP-complete has the property that it can be solved in polynomial time if and only if all over NP-complete problems can be solved in the polynomial time.

Example of NP-Complete problem :

– Let us prove that the Hamiltonian circuit problem is polynomially reducible to the decision version of the travelling salesman problem.

– It can be stated as the problem to determine whether there exists a Hamiltonian circuit in a complete graph with positive integer weights whose length is not greater than given positive integer m.

– We can map a graph G of a given instance of the Hamiltonian circuit problem to a complete weighted graph G representing an instance of the travelling salesman problem by assigning 1 as weight to each edge in G and adding edge of weight 2 between any pair of not adjacent vertices in G.

– As the upper bound m on the Hamiltonian circuit length, we can take $m = n$, where n is the number of vertices in G and G'. Therefore this transformation can be done in polynomial time. So problem is NP-complete.

The decision version of problems are NP-complete whereas the P optimization versions of such difficult problems fall in the class of NP-Hard Problems.

Q.8.(b) Explain Cook's theorem. (10)

Ans. Cook's theorem, states that the Boolean satisfiability problem is NP-complete. That is, any problem in NP can be reduced in polynomial time by a deterministic Turing machine to the problem of determining whether a Boolean formula is satisfiable.

Proof-idea :

(i) First show that SAT is in NP.

(ii) Then show that every problem in NP is polynomial-time reducible to SAT.

Proof (i) :

It is easy to see that SAT is in NP. Simply non-deterministically choose Boolean values for each variable. Then check whether the expression is true or false for those values.

If it is true, the answer is "yes."

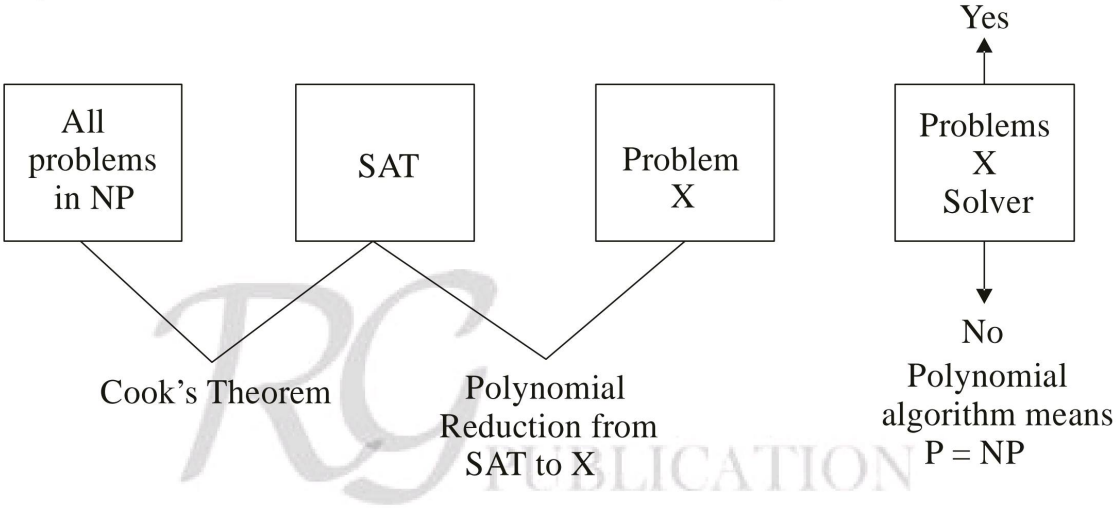
If it is false, the answer is no."

Proof (ii) :

Let A be a problem in NP.

Then there is a nondeterministic Turing machine M that decides A in polynomial time. Let the run time of M be at most n k, for some k 0. When M is run, it can access at most the ûrst n k + 1 cells.

Example : Cook's theorem proved SATISFIABILITY was hard by using a polynomial time reduction translating each problem in NP into an instance of SAT. Since a polynomial time algorithm of SAT would imply a polynomial time algorithm for everything in NP, SAT is NP hard. Since we can guess a solution to SAT, it is in NP and thus NP-complete.



The proof of Cook's Theorem,while quite clever, was certainly difficult and complicated. We had to show that all problems in NP could be reduced to SAT to make sure we did not miss a hard one. But now that we have a known NP-complete problem in SAT. For any other problem, we can prove it NP-hard of two polynomial time reductions can be done in polynomial time, all we need show is that SAT, that is, any instance of SAT can be translated to an instance of X in polynomial time.

Q.9. Explain the following:

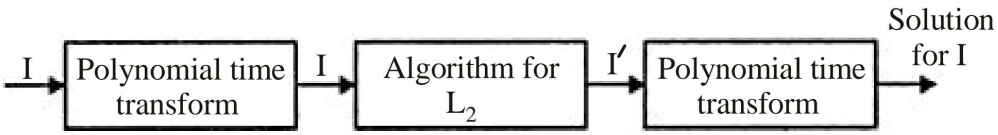
(a) NP hard Graph.

(b) NP scheduling problems. **(20)**

Ans. (a) NP-Hard Graph : The strategy we adopt to show that a problem L_2 is NP hard is :

(i) Pick a problem L_1 already known to be NP-hard.

(ii) Show how to obtain (in polynomial deterministic time) an instance I' of L_2 from any instance I of L_1 such that from the solution of I' we can determine (in polynomial deterministic time) the solution to instance I of L_1.



(iii) Conclude from step (ii) that $L_1 \alpha L_2$.

(iv) Conclude from steps (i) and (iii) and the transitivity of α that L_2 is NP-hard.

Various **NP-hard decision problems** are as follows :

(a) *Node Cover Decision Problem (NCDP) :* In it we are given a graph G and an integer k. We are required to determine whether G has a node cover of size at most k.

(b) *Chromatic Number Decision Problem (CNDP) :* A coloring of a graph $G = (V, E)$ is function $f : v \rightarrow \{1, 2, ... k\}$ defind for all $i \in v$. If $(u, v) \in E$, then $f(u) \neq f(v)$. The chromatic number decision problem is to determine whether G has a coloring for a given k.

(c) *Directed Hamiltonian Cycle (DHC) :* A directed Hamiltonian cycle in a directed graph $G = (V, E)$ is a directed cycle of length $n = | V |$. So, the cycles goes through every vertex exactly once and then return to start is vertex. The DHC problem is to determine whether G has a directed Hamiltonian cycle.

(d) *Clique Decision Problem (CDP) :* The clique decision problem shows that CNF satisfiability $\propto$ CDP. Using this result, the transitivity of $\propto$, and the knowledge that satisfiability CNF satisfiability, we can readily establish that satisfiability $\propto$ CDP. Hence, CDP is NP-hard. Since, CDP $\in$ NP, CDP is also NP-complete.

Q.9.(b) NP scheduling problems. (10)

Ans. NP Scheduling Problems : These problem requires us to decide whether a given multiset, $A = \{a_1, a_2, ..., a_n\}$ of n positive integers has a partition P such that $\Sigma_{i \in P} a_i = \Sigma_{i \notin P} a_i$. We can show this problem is NP hard by first showing the sum of subsets problem to be NP hard. Recall that in the sum of subsets problem we have to determine whether $A = \{a_1, a_2, ..., a_n\}$ has a subset S that sums to a given integer M.

Various **scheduling problems** are as follows :

(i) Scheduling identical processors : Let P_i, $1 \leq i \leq m$, be m identical processors (or machines). The P_i could, for example, be line printers in a computer output room. Let J_i, $1 \leq i \leq n$, be n jobs. Job J_i requires t_i processing time. A schedule S is an assignment of jobs to processors. For each job J_i, S specifices the time intervals and the processor (s) on which this job is to be processed. A job cannot be processed by more than one processor at any given time. Let f_i be the time at which the processing of job J_i is completed. The mean finish time (MFT) of schedule S is

$$\text{MFT(S)} = \frac{1}{n} \sum_{1 \leq i \leq n} f_i$$

Lel w_i be a weight associated with each job J_i. The weighted mean finish time (WMFT) of schedule S is

$$\text{WMFT(S)} = \sum_{1 \leq i \leq n} w_i f_i$$

Let T_i be the time at which P_i finishes processing all jobs (or job segments) assigned to it. The finish time (FT) of S is

$$FT(S) = \max_{1 \le i \le m} \{T_i\}$$

Schedule S is a non-preemptive schedule if and only if each job J_i is processed continuously from start to finish on the same processor. In a preemptive schedule each job need not be processed continuously to completion on one processor.

(ii) Flow shop scheduling : When $m = 2$, minimum finish time schedules can be obtained in O(n log n) time if n jobs are to be scheduled. When $m = 3$, obtaining minimum finish time schedules (whether preemptive or nonpreemptive) is NP hard. For the case of nonpreemptive schedules this is easy to see. We prove the result for preemptive schedules. The proof we give is also valid for the nonpreemptive case. However, a much simpler proof exists for the nonpreemptive case.

(iii) Job shop scheduling : A job shop, like a flow shop, has m different processors. The n jobs to be scheduled require the completion of several tasks. The time of the jth task of job J_i is $t_{k,i,j}$. Task j is to be performed on processor P_k. The tasks for any job J_i are to be carried out in the order 1, 2, 3,....., and so on. Task j cannot begin until task j–1 (if $j > 1$) has been completed. Note that it is possible for a job to have many tasks that are to be preformed on the same processor. In a nonpreemptive schedule a task once begun is processed without interruption until it is completed. The definitions of FT (S) and MFT(S) extend to this problem in a natural way. Obtaining either a minimum finish time preemptive schedule or a minimum finish time non preemptive schedule is NP hard even when $m = 2$. The proof for the nonpreemptive case is very simple (use partition). We present the proof for the preemtive case. This proof will also be valid for the non preemptive case but will not be the simplest proof for this case.

ANALYSIS & DESIGN OF ALGORITHMS

May 2015
Paper Code: CSE-306-F

Note: ***Attempt five questions in all, selecting one question from each section. Q. No. 1 is compulsory.***

Q.1.(a) Which function grows faster e^n or 2^n ? Justify your answer.

Ans. e^n will grow faster.

Reason : e =2.718

as we know, $(2.718)^n > 2^n$

Q.1.(b) Analyze the various cases of Binary Search complexity.

Ans. Analysis of Binary Search :

Best case - O (1) comparisons

In the best case, the item X is the middle in the array A.

A constant number of comparisons (actually just 1) are required.

Worst case - O (log n) comparisons

In the worst case, the item X does not exist in the array A at all.

Through each recursion or iteration of Binary Search, the size of the admissible range is halved. This halving can be done ceiling(lg n) times. Thus, ceiling(lg n) comparisons are required.

Average case - O (log n) comparisons

To find the average case, take the sum over all elements of the product of number of comparisons required to find each element and the probability of searching for that element.

To simplify the analysis, assume that no item which is not in A will be searched for, and that the probabilities of searching for each element are uniform.

Q.1.(c) Using big-O notation state time and space complexity of quick sort.

Ans. Quicksort Time complexity :

Worst Case = $O(n^2)$

Average Case = O(n*log(n))

Best Case = O(n*log(n))

Quicksort Space Complexity : The space complexity is O(n) but if the code is optimized(if code uses iterative bottom up approach) space complexity can be reduced to O(log(n))

Q.1.(d) What is the difference between greedy and dynamic approach ?

Ans. Differences are:

Greedy approaches	Dynamic approaches
1. More efficient	1. Less efficient
2. Optimal solution cannot be guaranteed by a greedy algorithm.	2. Optimal solution can be guaranteed.
3. No efficient solution.	3. DP provides efficient solutions for some problems for which a brute force approach would be very slow.

Q.1.(e) Can the master method be applied to solve recurrence.

$$T(n) = 4T\left(\frac{n}{2}\right) + n^2 \log n$$ **? why or why not ?**

Ans. $$T(n) = 4T\left(\frac{n}{2}\right) + n^2 \log n$$

If has the proper form : $a = 4$, $b = 2$, $f(n) = n^2 \log n$

$$n^{\log_b a} = n^{\log_2 4} = n^2$$

It might seem that case 3 should apply, since $f(n) = n^2 \log n$ is asymptotically larger than $n^{\log_b a} = n^2$. The problem is that it is not polynomially larger. The ratio $f(n)/n^{\log_b a} = \frac{n^2 \log n}{n^2} = \log n$ is asymptotically less thanb n^{ε} for carry any positive constant ε. The recurrence falls into the gap between case 2 and 3 case. So we can not determine. Hence the master method can not apply.

Q.1.(f) What are three properties of NP-Complete problem ?

Ans. Properties of NP -complete Problems :

(i) No polynomial time algorithm has been found for any NP-Complete problem.

(ii) It has not been established that polynomial time algorithms for these problems are not in existence.

(iii) If a polynomial time algorithm is found for any NP complete problems, then there will be polynomial time algorithms for all such problems

(iv) It can be proved that if no polynomial time algorithm exists for any such problem, then definitely for any other such problem, it will not exist.

Q.1.(g) Explain Branch and Bound technique.

Ans. Branch and bound is a systematic method for solving optimization problems. B&B is a rather general optimization technique that applies where the greedy method and dynamic programming fail. However, it is much slower. Indeed, it often leads to exponential time complexities in the worst case. On the other hand, if applied carefully, it can lead to algorithms that run reasonably fast on average.

B&B Technique -

1. The first idea of B&B is to develop "a predictor" of the likelihood (in a loose sense) of a node in the solution tree that it will lead to an optimal solution. This predictor is quantitative.

2. Which node to expand next: B&B chooses the live node with the best predictor value

3.B&B simply expands that node (i.e., generate all its children)

4.The predictor value of each newly generated node is computed, just expanded node is now designated as a dead node, and the newly generated nodes are designated as live nodes

5.Termination criterion : When the best node chosen for expansion turn out to be a final leaf (i.e., at level n), that when the algorithm terminates, and that node corresponds to the optimal solution.

Q.1.(h) Discuss Hamiltonian cycles with examples.

Ans. A Hamiltonian cycle, also called a Hamiltonian circuit, Hamilton cycle, or Hamilton circuit, is a graph cycle (i.e., closed loop) through a graph that visits each node exactly once.

Examples:

(i) a complete graph with more than two vertices is Hamiltonian
(ii) every cycle graph is Hamiltonian
(iii) every tournament has an odd number of Hamiltonian paths
(iv) every platonic solid, considered as a graph, is Hamiltonian
(v) every prism is Hamiltonian
(vi) The Deltoidalhexecontahedron is the only non-hamiltonian Archimedean dual
(vii) Every anti-prism is Hamiltonian
(viii) A maximal planar graph with no separating triangles is Hamiltonian.

Section - A

Q.2.(a) Explain strassen's matrix multiplication with example.

Ans. The Strassen algorithm, named after Volker Strassen, is an algorithm used for matrix multiplication. It is faster than the standard matrix multiplication algorithm and is useful in practice for large matrices, but would be slower than the fastest known algorithm for extremely large matrices.

Given: Two N by N matrices A and B.

Problem: Compute $C = A \times B$

Brute Force :

```
for i:= 1 to N do
for j:=1 to N do
C[i,j] := 0;
for k := 1 to N do
C[i,j] := C[i,j] + A[i,k] * B[k,j]
```

$O(N^3)$ multiplications

Divide and Conquer

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

$P_1 = (A_{11} + A_{22})(B_{11} + B_{22})$
$P_2 = (A_{21} + A_{22})(B_{11})$
$P_3 = (A_{11})(B_{12} - B_{22})$
$P_4 = (A_{22})(B_{21} - B_{11})$
$P_5 = (A_{11} + A_{22})(B_{22})$
$P_6 = (A_{21} - A_{11})(B_{11} + B_{12})$
$P_7 = (A_{12} - A_{22})(B_{21} + B_{22})$
$C_{11} = (P_1 + P_4 - P_5 + P_4 + P_7)$
$C_{12} = (P_3 + P_5)$
$C_{21} = (P_2 + P_4)$
$C_{22} = (P_1 + P_3 - P_2 + P_6)$

From

$$T(n) = \begin{cases} 7T(n/2) + cn & if\ n > 1 \\ c & if\ n = 1 \end{cases}$$

$T(n) = O(n^{\log 7}) = O(n^{2.81})$.

Q.2.(b) Write algorithms for Union & Find operations for disjoint sets.

Ans. A union-find algorithm is an algorithm that performs two useful operations on such a data structure:

Find : Determine which subset a particular element is in. This can be used for determining if two elements are in the same subset.

Union : Join two subsets into a single subset.

```
procedure Make-Set (x)
1. size[x]    ← 1
2. parent[x]    ← x
   end.
    procedure UNION(a , b) { with weight balancing }
   {a and b are roots of two distinct trees in the forest.}
  {Makes the root of the smaller tree a child of the root of the larger tree.}
1. if size[a] < size[b] then a    ↔ b
2. parent[b]    ← a
3. size[a]    ← size[a] + size[b]
   end.
    function FIND(x) { with path compression }
    {Returns the root of the tree that contains node x.}
1. if parent[x] ≠ x then
2. parent[x]    ← FIND(parent[x])
3. return parent[x]
   end.
```

Q.3.(a) What is Merge Sort ? Write a recursive algorithm for same and show that its running time is $O(n \log n)$.

Ans. Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merg() function is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See following C implementation for details.

Algorithm : Merge sort (A, P, q, r)

(1) $n_1 \leftarrow q - p + 1$

(2) $n_2 \leftarrow r - q$

RG PUBLICATION

(3) Create Array $L\{1, 2 \ldots\ldots\ldots n_{1+1}\}$
$R\{1, 2 \ldots\ldots\ldots n_{2+1}\}$
(4) for $i \rightarrow 1$ to n_1.
(5) do $L[i] \leftarrow A[P + i - 1)$
(6) for $j \leftarrow$ *to* n_2
(7) do
$R[j] = A\ (q + j)$
(8) $L[n_1 + 1] \leftarrow \infty$
(9) $R[n_2 + 1] \leftarrow \infty$
(10) $i \leftarrow 1$
(11) $j \leftarrow 1$
(12) $k \leftarrow p$ to r
(13) do
if $L[i] > R\ [j\ ]$
(14) then $A[k] \leftarrow L[i]$
(15) $i \leftarrow i + 1$
(16) else
$A\ [K] \leftarrow R[j\ ]$
(17) $j \leftarrow j + 1$

Analyzing Merge sort : We assume that 'n' at a power of 2 i.e. each divide step decide the Merge sort (A, p, r)

If $p < r$ $\rightarrow 0\ (1)$
then
$q \leftarrow (p + r)/2$ $\rightarrow 0\ (1)$
Merge sort (A, p, r) $\rightarrow 0\ (n/2)$
Merge sort (A, $q + 1$, r) $\rightarrow 0\ (n/2)$
Merge sort (A, p, q, r) $\rightarrow 0\ (n)$

Problem into two subProblem of size exactly $n/2$.

When $n = 1$ Base case occurs

When $n \geq 2$ we break down running time.

The recurrence for merge sort running time as

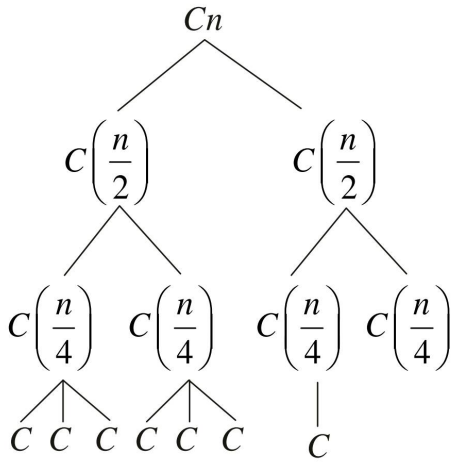
$$T_{(n)} = \begin{cases} \theta(1) & \text{if } n = 1 \\ 2T\left(\dfrac{n}{2}\right) + \theta(n) & \text{if } n > 1 \end{cases}$$

We can write as

$$T_{(n)} = \begin{cases} C & \text{if } n = 1 \\ 2T\left(\dfrac{n}{2}\right) + Cn & \text{if } n > 1 \end{cases}$$

RG PUBLICATION

Recursion Tree :



$$T_n = C_n\,[\log(n) + 1]$$
$$= Cn \log n + Cn$$
$$= n \log n$$

Q.3.(b) What is Divide and Conquer strategy? Explain Merge sort algorithm with example. Also give its recurrence relation.

Ans. Divide and Conquer strategy : Divide and conquer is a technique for designing algorithms that consists of decomposing the instance to be solved into a number of smaller subinstance of the same problem, solving successively and independently each of these subinstances, and then combining the subsolutions thus obtained to obtain the solution of the original instance.

Most of the algorithms are recursive in natare. To solve a given problem they call them selves recursively one or more times. All these algorithms follows "Divide and conquer" approach.

The Divide and conquer paradigm involves three steps at each level of recursion :

(1) Divide : In this step whole problem is divided into number of several subproblems .

(2) Conquer : The subproblems by solving them recursively. If the subproblem size are small enough, however, just solve the subproblem in a straight forward manner.

(3) Combine : Finally, the solutions obtained by the subproblems are combined to create solution to the original problem.

Algorithm:

Input: An array a and two indices p and q between which we want to do the sorting.

Output: The array a will be sorted between p and q as a side effect

```
if p < q then
int m <-p+q/2
mergeSort(a,p,m)
mergeSort(a,m+1,q)
merge(a,p,m,q)
```

The pseudo code of the merging routine is as follows:

Algorithm merge(a,p,m,q)

Input: An array a, in which we assume that the halves from p:::m and m+1:::q are each sorted

Output: The array should be sorted between p and q

```
Array tmp of size q p+1 // this array will hold the temporary result
inti<-p
int<-j m+1
int k <-1
while (i<= m or j <=q) do
if (j = q+1 or a[i] <= a[ j]) then
tmp[k]<- a[i]
i<- i+1
else if (i = m+1 or a[i] > a[ j]) then
tmp[k]<- a[ j]
j<- j +1
k<- k +1
for k = p to q do
a[k] <- tmp[k]
```

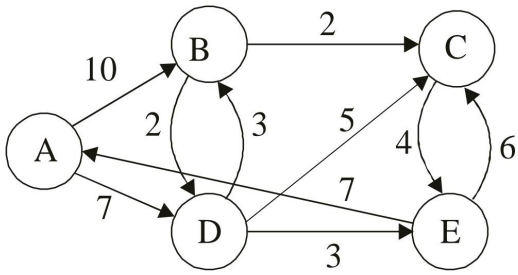
Example: For the first call of the function, the data is partitioned into two, one list of 6, 5, 8 and 1, and a second list of 4, 3, 7 and 2. The list (6 5 8 1) is then partitioned into two smaller lists (6 5) and (8 1). The base case has now been reached, and we can sort (6 5) into (5 6) and we can sort (8 1) into (1 8). We can now merge (5 6) and (1 8). We compare 1 and 5, so the first element of the merged sequence is 1. We next compare 5 and 8, so the second element is 5. Next we compare 6 and 8 the third element will be 6, leaving 8 as the last element. The merged result is (1 5 6 8). We now turn our attention to the second half of the original data set. Again we partition (4 3 7 2) into (4 3) and (7 2) . Sorting these we get (3 4) and (2 7) Merging these we get (2 3 4 7). We now have two halves of the data sorted as (1 5 6 8) and (2 3 4 7) . All that remains to be done is to merge the two halves together.

(1 2 3 4 5 6 7 8)

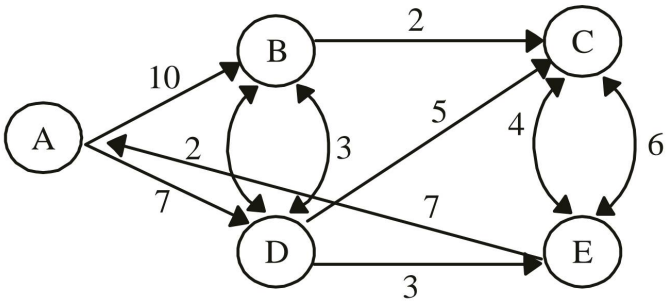
Recurrence relation: T(n)=2T(n/2) + c,

Section -B

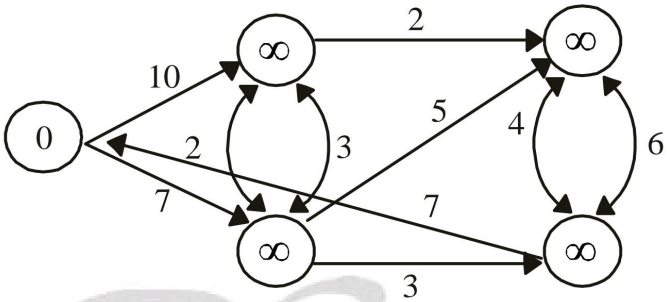
Q.4.(a) Use Dijkstra algorithm to find single source shortest path for following graph taking vertex '*A*' as the source.



Ans.



A	B	C	D	E
0	∞	∞	∞	∞

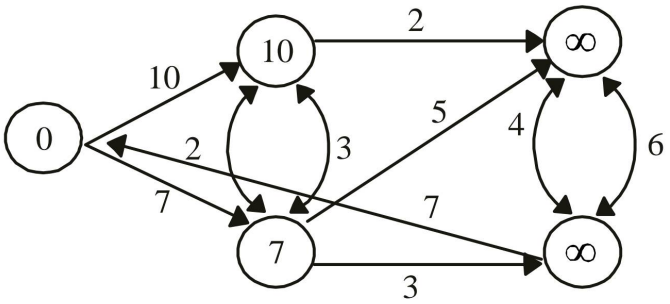


Node A is selected

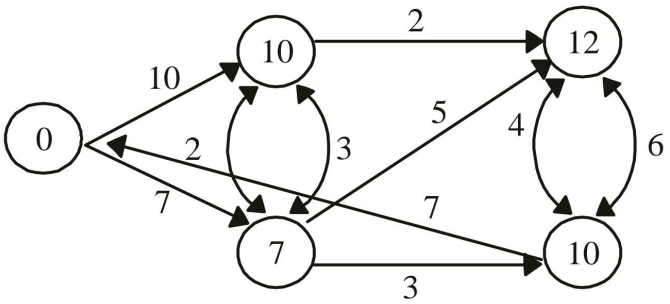
A	B	C	D	E
$\boxed{0}$	10	∞	7	∞

RG PUBLICATION

Now, select next node with min. value. i.e. next selected node is D

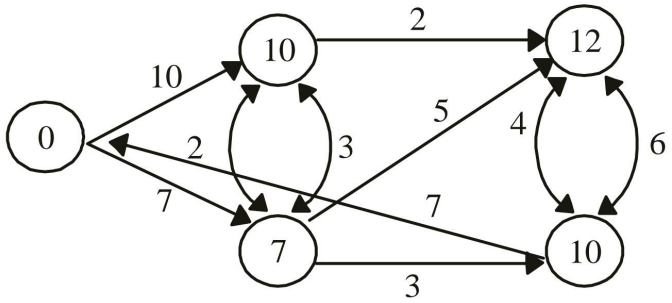


Relaxing edges from node D.

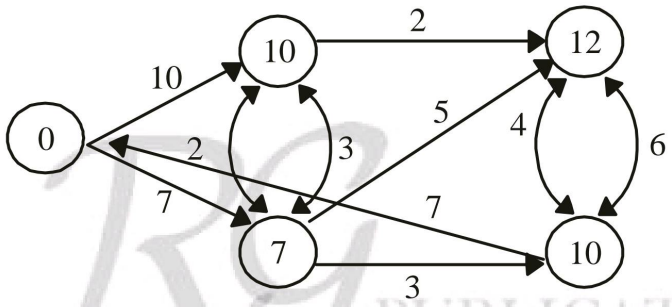


A	B	C	D	E
0	10	12	7	10

Next selected node is E



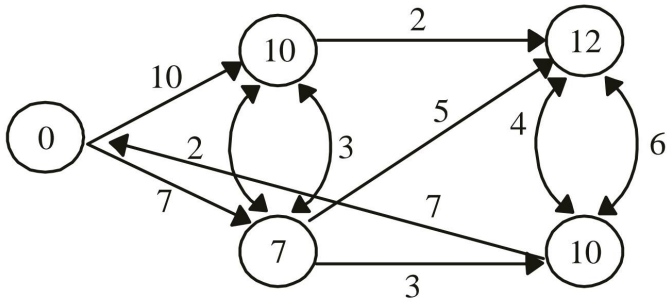
Next selected node is C



RG PUBLICATION

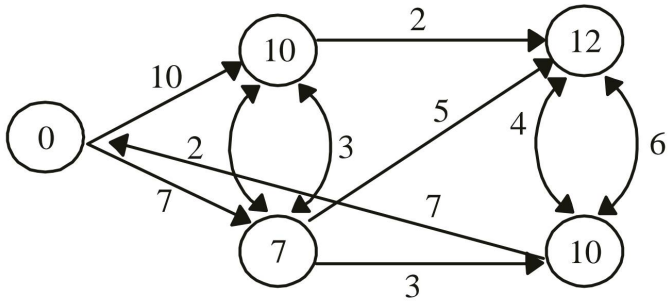
A	B	C	D	E
0	10	12	7	10

Next selected node is B



Next selected node is 12

A	B	C	D	E
0	10	12	7	10



Q.4.(b) Write Kruskal's algorithm for finding minimum spanning tree of an undirected graph.

Ans. Kruskal's algorithm is a greedy algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. 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.

This algorithm is directly based on the MST(minimum spanning tree) property.

MST-KRUSKAL(G, w)
1. A ? Ø
2. for each vertex v V[G]
3. do MAKE-SET(v)
4. sort the edges of E into nondecreasing order by weight w
5. for each edge (u, v) E, taken in nondecreasing order by weight
6. do if FIND-SET(u) ? FIND-SET(v)
7. then A ? A {(u, v)}
8. UNION (u, v)
9. return A

RG PUBLICATION

Q.5.(a) Explain Traveling Salesman Problem.

Ans. Traveling Salesman Problem (TSP) : Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

Note the difference between Hamiltonian Cycle and TSP. The Hamiltonian cycle problem is to find if there exist a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.

This is one of the most known problems ,and is often called as a difficult problem.Let's number the cities from 1 to n ,and let city 1 be the city-base of the salesman.Also let's assume that c(i,j) is the visiting cost from i to j.There can be c(i,j)<>c(j,i).Apparently all the possible solutions are (n-1)!.Someone could probably determine them systematically,find the cost for each and everyone of these solutions and finally keep the one with the minimum cost.These requires at least (n-1)! steps.

If for example there were 21 cities the steps required are (n-1)!=(21-1)!=20! steps.If every step required a msec we would need about 770 centuries of calculations.Apparently,the exhausting examination of all possible solutions is out of the question.Since we are not aware of any other quick algorithm that finds a best solution we will use a heuristic algorithm. According

to this algorithm whenever the salesman is in town he chooses as his next city, the city j for which the c(i,j) cost,is the minimum among all c(i,k) costs, where k are the pointers of the city the salesman has not visited yet.There is also a simple rule just in case more than one cities give the minimum cost,for example in such a case the city with the smaller k will be chosen.This is a greedy algorithm which selects in every step the cheapest visit and does not care whether this will lead to a wrong result or not.

Input : Number of cities n and array of costs c(i,j) i,j=1,..n (We begin from city number 1)

Output:Vector of cities and total cost.

```
(* starting values *)
C=0
cost=0
visits=0
e=1 (*e=pointer of the visited city)
(* determination of round and cost)
for r=1 to n-1 do
choose of pointer j with
minimum=c(e,j)=min{c(e,k);visits(k)=0 and k=1,..,n}
cost=cost+minimum
e = j
C(r) = j
end r-loop
C(n)=1
cost=cost+c(e,1)
```

RG PUBLICATION

Q.5.(b) Set *n* = 7; (p1, p2 ....., p7) = (3, 5, 20, 18, 1, 6, 30) and (d1, d2,....., d7) = (1, 3, 4, 3, 2, 1, 2).

What is the solution generated by job sequencing algorithm for given problem?

Ans. $n = 7$

P_1	3	d_1	1
P_2	5	d_2	3
P_3	20	d_3	4
P_4	18	d_4	3
P_5	1	d_5	2
P_6	6	d_6	1
P_7	30	d_7	2

Sort the task in descending order of profit.

i	7	3	4	6	2	1	5
p_i	30	20	18	6	5	3	1
d_i	20	4	3	1	3	1	2

Size of array is 7
initialized array with 0

1	2	3	4	5	6	7
0	0	0	0	0	0	0

Assign task 7 to slot 2

1	2	3	4	5	6	7
0	30	0	0	0	0	0

Profit = 30

Assign task 3 to slot 4

1	2	3	4	5	6	7
0	30	0	20	0	0	0

Profit = 50

Assign task 4 to slot 3

1	2	3	4	5	6	7
0	30	18	20	0	0	0

Profit = 68

Assign task 2 to slot 3 but reject task 2

1	2	3	4	5	6	7
6	30	18	20	0	0	0

Profit = 74

Task 1 is also rejected

1	2	3	4	5	6	7
6	30	18	20	0	0	0

Profit = 74

Task 5 is also rejected

1	2	3	4	5	6	7
6	30	18	20	0	0	0

Profit = 74

Optimal task sequence is

$P_7 \rightarrow P_3 \rightarrow P_4 \rightarrow P_6$ with Max. Profit = 74

Section - C

Q.6.(a) Discuss how 8-Queen's Problem is solved through Backtracking.

Ans. Algorithm:

- Start with one queen at the first column first row
- Continue with second queen from the second column first row
- Go up until find a permissible situation
- Continue with next queen

We place the first queen on A1. Note the positions which Q1 is attacking. So the next queen Q2 has to options: B3 or B4. We choose the first one B3. We backtrack to Q3 and try to find admissible place different from C1. Again we need to backtrack. Q2 has no other choice and finally we reach Q1. We place Q1 on A3.

```
intPlaceQueen(int board[8], int row)
```

```
If (Can place queen on ith column)
PlaceQueen(newboard, 0)
Else
PlaceQueen(oldboard,oldplace+1)
End
```

Q.6.(b) Explain LC Branch-&-Bound with example.

Ans. When a node is visited the first time called an E-node, all its children are generated and saved into a data structure called live-node. If the child is not bounded; the structure could be a LC, or Least Cost. Exactly one node is pulled out the live-node list at a time, which becomes an E-node, to be "expanded" next. The LC (least-cost) branch-and-bound strategy uses heuristics to determine the best node for expansion from the current live nodes.

Example: The 15-puzzle

– 15 numbered tiles on a square frame with a capacity for 16 tiles.

– Given an initial arrangement, transform it to the goal arrangement through a series of legal moves.

– Legal move involves moving a tile adjacent to the empty spot E to E.

– Four possible moves in the initial state above: tiles 2, 3, 5, 6.

– Each move creates a new arrangement of tiles, called state of the puzzle.

– Initial and goal states.

– A state is reachable from the initial state if there is a sequence of legal moves from initial state to this state.

– The state space of an initial state is all the states that can be reached from initial state.

– Search the state space for the goal state and use the path from initial state to goal state as the answer.

– Number of possible arrangements for tiles: 16! H" 20.9 × 1012.

– Check whether the goal state is reachable from initial state.

Number the frame positions from 1 to 16pi is the frame position containing tile numbered i. p16 is the position of empty spot. For any state, let li be the number of tiles j such that j < i and pj > pi. For the initial arrangement above, l1 = 0, l4 = 1, and l12 = 6.

Let x = 1 if in the initial state, the empty spot is in one of the following positions: 2, 4, 5, 7, 10, 12, 13, 15; otherwise x = 0.

Q.7.(a) Write Backtracking algorithm to find chromatic number of a given graph.

Ans. Chromatic number of a graph 'G' is the smallest number of color needed to color the vertices of G. So, that no two adjacent vertices share the same color.

The chromatic number of a graph 'G' is denoted by $x(G)$.

Algorithm :

```
Algorithm m coloring
    (int k)
{
    do
        {
        x[k] = get node color (k) ;
        if (x [k] = = 0) return ;
```

```
        if (k = = n )
        write (x [1 ..... n]) ;
    else
        m coloring (k + 1) ;
        }
        while (true) ;
    }
    int getnode color (int k) {
    || x [1], x [2] ..... x[k – 1]
    || x [k] ∈ [0 ..... m]
        do {
        x [k] = (x [k] + 1) mode (m + 1)
            if (x [k] = = 0)
        return ;
    for ( j = 1 ; j < = n ; j + +) {
    if (G [k] [ j ] ≠ 0 & &
                x[k] = = x [ j ])
        break ;
    }
        if ( j = n + 1)
        return ; }
        while (true) ;
    }
```

RG PUBLICATION

Q.7.(b) What O/1 knapsack problem ? Slove this problem using Branch & Bound method taking suitable example.

Ans. O-1 Knapsack problem : If a thief carries all or none of a good in the store, which cannot be parted or broken then it is called O-1 Knapsack problem.

Branch & Bound is very powerful tool to solve optimal problem when you have to maximise something when given some constraints.

Branch & Bound Problems solve 0/1 knapsack problem Statement :

Given 'n' items with benefits $b_1, b_2 \ldots\ldots b_n$ & weights $w_1, w_2 \ldots. w_n$ & max. capacity of knapsack M. Find the items that should be chosen that maximize benefit.

$0 \rightarrow$ not choose the items

$1 \rightarrow$ choose the items.

Process : Let us have $x_1, x_2 \ldots\ldots x_n$

$x_i \in \{0, 1\}$

We need to maximize

$$\sum_{i=1}^{n} b_i \times i \quad , \quad \text{constraint} \quad \sum_{i=1}^{n} w_i \times i \leq m$$

(Maximize)

as we have to make 'n' choices

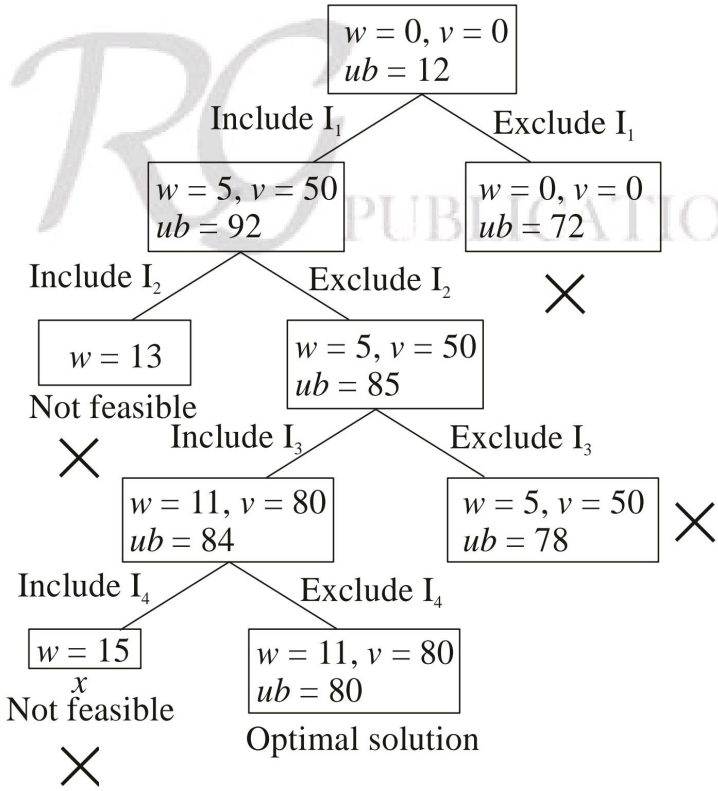
2^n possibilities are there

$x_1, x_2, \ldots\ldots\ldots\ldots x_n$
$0|1\ 0|1 \ldots\ldots\ldots\ldots 0|1$
$2 \times 2 \times \ldots\ldots\ldots\ldots \times 2 = 2^n$

0/1 knapsack Problem : Example using $m = 12$ Branch & Bound

Items (I)	Weigth(w_i)	Value (v_i)	$\frac{v_i}{w_i}$
I_1	5	50	10
I_2	8	48	6
I_3	6	30	5
I_4	4	16	4

$ub = v + (w - w)\ (v_{i+1}|w_{i+1})$
Arrange the table according to its descending values of per weight Ratio.
Now start with root node, the upper bound for the root can be computed as
$ub = 0 + (12 - 0) * 10 = 120$
$v = 0,\ w = 0,\ v_1|w_1 = 10$



Section - D

Q.8.(a) Giving suitable example prove that traveling Salesperson Problem is NP-complete.

Ans. TSP is NP-complete : Instance: A ûnite set of cities $\{c_1; c_2; : : : ; c_n\}$, a positive integer distance $d(i, j)$ between each pair (c_i, c_j), and an integer B.

Traveling salesman is NP complete since a permutation of the cities is a certificate that can be verified.To prove that traveling salesman is NP-complete we describe a reduction from hamiltonian cycle.

Given an instance of hamiltonian cycle *g* (with n vertices), create an instance of traveling salesman.

- For each vertex *v*, create a city *cv*;
- If there is an edge (*u,v*), then let the distance from *cu* to *cv* be 1;
- Otherwise let the distance be 2.

Claim: G has a Hamiltonian Cycle if and only if there is a route of distance at most *n*.

Proof: If *G* contains a cycle, then the cycle forms a route through the cities of distance *n*. If there is a route of distance *n* through the *n* cities,then clearly the distance between each pair of cities along the route is 1. Thus each pair of cities along the route is adjacent in G and the route is a hamiltonian cycle.

Q.8.(b) Show that Clique decision problem is NP-hard.

Ans. Given a graph G, find the largest clique (set of nodes such that all pairs in the set are neighbours).

Decision problem: "Given G and integer k, does G contain a clique of size $\geq k$?"

Proof: We will reduce 3-SAT to Max-Clique. Speciûcally, given a 3-CNF formula F of m clauses over n variables, we construct a graph as follows. First, for each clause c of F we create one node for every assignment to variables in c that satisûes c. E.g., say we have:

$$F = (x_1 \vee x_2 \vee x_4) \wedge (x_3 \vee x_4) \wedge (x_2 \vee x_3) \wedge \ldots$$

Then in this case we would create nodes like this:

$(x_1 = 0, x_2 = 0, x_4 = 0)$ $(x_3 = 0, x_4 = 0)$ $(x_2 = 0, x_3 = 0)$ ...
$(x_1 = 0, x_2 = 1, x_4 = 0)$ $(x_3 = 0, x_4 = 1)$ $(x_2 = 0, x_3 = 1)$
$(x_1 = 0, x_2 = 1, x_4 = 1)$ $(x_3 = 1, x_4 = 1)$ $(x_2 = 1, x_3 = 0)$
$(x_1 = 1, x_2 = 0, x_4 = 0)$
$(x_1 = 1, x_2 = 0, x_4 = 1)$
$(x_1 = 1, x_2 = 1, x_4 = 0)$
$(x_1 = 1, x_2 = 1, x_4 = 1)$

We then put an edge between two nodes if the partial assignments are consistent. Notice that the maximum possible clique size is m because there are no edges between any two nodes that correspond to the same clause c. Moreover, if the 3-SAT problem does have a satisfying assignment, then in fact there is an m-clique (just pick some satisfying assignment and take the m nodes consistent with that assignment). So, to prove that this reduction (with $k = m$) is correct we need to show that if there isn't a satisfying assignment to *F* then the maximum clique in the graph has size $< m$. We can argue this by looking at the contrapositive. Speciûcally, if the graph has an *m*-clique, then this clique must contain one node per clause *c*. So, just read off the assignment given in the nodes of the clique: this by construction will satisfy all the clauses. So, we have shown this graph has a clique of size *m* iff *F* was satisfiable. Also, our reduction is polynomial time since the graph produced has total size at most quadratic in the size of the formula *F* ($O(m)$ nodes,$O(m)$ edges). Therefore Clique is NP-complete.

Q.9. Write short notes on :

(a) Difference between deterministic and non deterministic algorithms.

(b) NP-hard and NP complete problems

(c) Cook's theorem.

Ans.(a) Difference between deterministic and non deterministic algorithms : A *deterministic algorithm* is an algorithm which, given a particular input, will always produce the same output, with the underlying machine always passing through the same sequence of states. Examples : Skewed matrix products

Non-deterministic algorithm is an algorithm that can exhibit different behaviours on different runs, as opposed to a deterministic algorithm. There are several ways an algorithm may behave differently from run to run. A concurrent algorithm can perform differently on different runs due to a race condition. A probabilistic algorithm's behaviour depends on a random number generator. An algorithm that solves a problem in non-deterministic polynomial time can run in polynomial time or exponential time depending on the choices it makes during execution.

Example : Merge sort

- split the collection in two approximately equal parts
- sort the two halves with merge sort (i.e. recursively)
- merge the results

Items can only be uniquely sorted if the sorting criterion chosen always defines a total order; e.g. student numbers are expected to be unique, but if we sort exams by name and two students happen to have the same name, the order in which their exams get sorted is left undefined. In such cases, merge sort will always arrive at one of the possible valid orderings, but which one is left unspecified - hence it is nondeterministic.

RG PUBLICATION

Ans. (b)NP-hard : NP-hard (non-deterministic polynomial-time hard), in computational complexity theory, is a class of problems that are, informally, "at least as hard as the hardest problems in NP". A problem H is NP-hard if and only if there is an NP-complete problem L that is polynomial time Turing-reducible to H. L can be solved in polynomial time by an oracle machine with an oracle for H. Informally, we can think of an algorithm that can call such an oracle machine as a subroutine for solving H, and solves L in polynomial time, if the subroutine call takes only one step to compute. NP-hard problems may be of any type: decision problems, search problems, or optimization problems.

NP-hard problem are at least as hard as NP-Complete problem. Hence there are no polynomial problem algorithms defined.

Example of NP-hard problem is :

(a) Consider the halting problem for determining algorithms.

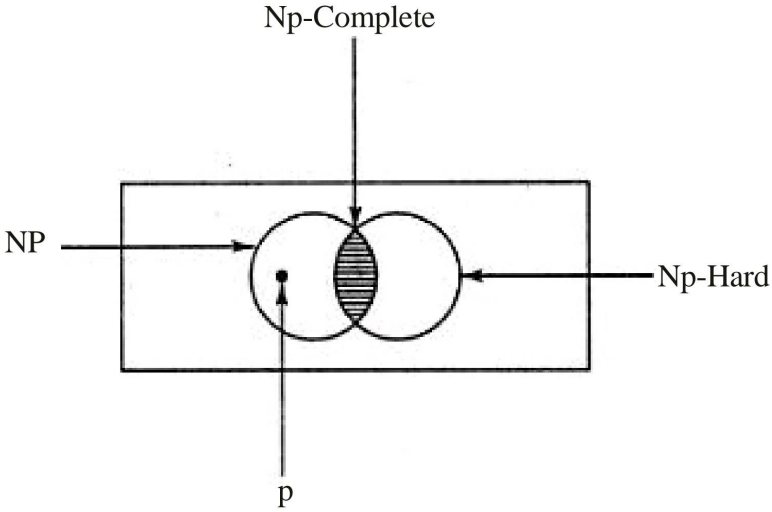
(b) It is well known that the halting problem is undecidable. Hence there exists no algorithm to solve the problem. Therefore it clearly cannot be NP.

NP-Complete : The complexity class NP-complete is a class of decision problems. A decision problem *L* is NP-complete if it is in the set of NP problems and also in the set of NP-hard problems. The abbreviation NP refers to "nondeterministic polynomial time."

A decision problem *D* is said to be NP-complete if

– It belongs to class NP.

– Every problem in NP is polynomially reducible to D.



From above diagram, there are two classes of problems NP-Hard and NP-complete. A problem is NP-complete has the property that it can be solved in polynomial time if and only if all over NP-complete problems can be solved in the polynomial time.

Example of NP-Complete problem :

– Let us prove that the Hamiltonian circuit problem is polynomially reducible to the decision version of the traveling salesman problem.

– It can be stated as the problem to determine whether there exists a Hamiltonian circuit in a complete graph with positive integer weights whose length is not greater than given positive integer *m*.

– We can map a graph *G* of a given instance of the Hamiltonian circuit problem to a complete weighted graph *G* representing an instance of the traveling salesman problem by assigning 1 as weight to each edge in *G* and adding edge of weight 2 between any pair of not adjacent vertices in *G*.

– As the upper bound *m* on the Hamiltonian circuit length, we can take $m = n$, where *n* is the number of vertices in *G* and G'. Therefore this transformation can be done in polynomial time. So problem is NP-complete.

The decision version of problems are NP-complete whereas the *P* optimization versions of such difficult problems fall in the class of NP-Hard Problems.

Ans.(c)Cook's theorem : Refer Q.8(b) of paper May 2014.

ANALYSIS AND DESIGN OF ALGORITHMS

May - 2018
Paper Code:-CSE-306-F

Note : ***Attempt five questions in all, selecting one question from each Section. Question No.* 1 *is compulsory. All questions carry equal marks.***

Q.1. Write short note on the following : (4×5=20)

(a) Asymptotic notation

(b) Greedy algorithms

(c) Dominance rule

(d) P & NP Class

Ans. (a) Asymptotic Notation : Asymptotic notation is a shortest way to represent the time complexity.

Various notations such as Ω, Θ, and O are called asymptotic notations.

Big Oh Notation : The Big oh notation is denoted by 'O'. It is a method of representing the upper bound of algorithm's running time. Using big Oh notation we can give longest amount of time taken by the algorithm to complete.

Definition: Let F(n) and g(n) be two non-negative function. Let n_0 and constant c are two integers such that n_0 denotes some value of input and $n > n_0$. Similarly c is some constant such that $c > 0$. We can write

$$F(n) \le c * g(\mathrm{n})$$

then F(n) is big Oh of g(n). It is also denoted as $F(n) \in O(g(n))$. In other words F(n) is less than g(n) is g(n) is multiple of some constant c.

Example : Consider function F(n) = $2n + 2$ and $g(n) = n^2$. Then we have to find some constant c,so that $F(n) \le c * g(\mathrm{n})$. As F(n) =$2n + 2$ and $g(n) = n^2$ then we find c for $n = 1$ then

$$F(n) = 2n + 2$$
$$= 2(1) + 2$$
$$F(n) = 4$$

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

$$\text{i.e. } F(n) > g(n)$$

If $n = 2$ then,

$$F(n) = 2(2) + 2 = 6$$
$$g(n) = (2)^2 = 4$$
$$\text{i.e } F(n) > cg(n)$$

If $n = 3$, then

$$F(n) = 2(3) + 2 = 8$$
$$g(n) = (3)^2 = 9$$
$$\text{i.e } F(n) < g(n) \text{ is true}$$

Hence we can conclude that for $n > 2$, we obtain

$$F(n) < g(n)$$

Thus always upper bound of existing time is obtained by big Oh notation.

Omega Notation : Omega notation is denoted by 'Ω'. This notation is used to represent the lower bound of algorithms running time. Using omega notation we can denote shortest amount of time taken by algorithm.

Definition : A function F(n) is said to be in Ω (g(n)) if F(n) is bounded below by some positive constant multiple of g(n) such that

$$F(n) \geq c * g(n) \text{ For all } n \geq n_o$$

It is denoted as F(n) ∈ Ω (g(n)).

Example:

Consider F(n) = $2n^2 + 5$ and g(n) = 7n

Then if n = O

F(n) = $2(0)^2 + 5 = 5$

g(n) = 7(0) = 0

i.e F(n) > g(n)

But if n = 1

F(n) = $2(1)^2 + 5 = 7$

g(n) = 7(1) = 7

i.e F(n) = g(n)

If n = 3, then

F(n) = $2(3)^2 + 5 = 23$

g(n) = 7(3) = 21

i.e F(n) > g(n)

Thus For n>3 we fet F(n) > c * g(n)

It can be represented as $2n^2 + 5 \in \Omega(n)$

Similarly any $n^3 \in \Omega\ (n)^2$

Θ Notation : The theta notation is denoted by Θ. By this method the running time is between upper bound and lower bound.

Definition : Let F(n) and g(n) be two non-negative functions. There are two positive constants namely c_1 and c_2 such that

$$c_1 \leq g(n)\ c_2 \leq g(n)$$

Then we can say that

$$F(n) \in \Theta\ (g(n))$$

Example : If F(n) = $2n + 8$ and g(n) = $5n$

Where $n \geq 2$

Similarly F(n) = $2n + 8$

g(n) = $7n$

i.e $5n < 2n + 8 < 7n$ For $n \geq 2$

Here $c_1 = 5$ and $c_2 = 7$ with $n_0 = 2$

The theta notation is more precise with both big Oh and omega notation.

Ans. (b) Greedy algorithms : An algorithm is designed to achieve optimum solution for a given problem. In greedy algorithm approach, decisions are made from the given solution domain. As being greedy, the closest solution that seems to provide an optimum solution is chosen.

Greedy algorithms try to find a localized optimum solution, which may eventually lead to glAn algorithm is designed to achieve optimum solution for a given problem. In greedy algorithm

approach, decisions are made from the given solution domain. As being greedy, the closest solution that seems to provide an optimum solution is chosen.

Greedy algorithms try to find a localized optimum solution, which may eventually lead to globally optimized solutions. However, generally greedy algorithms do not provide globally optimized solutions.

Counting Coins : This problem is to count to a desired value by choosing the least possible coins and the greedy approach forces the algorithm to pick the largest possible coin. If we are provided coins of ¹ 1, 2, 5 and 10 and we are asked to count ¹ 18 then the greedy procedure will be :

(1) Select one ¹ 10 coin, the remaining count is 8

(2) Then select one ¹ 5 coin, the remaining count is 3

(3) Then select one ¹ 2 coin, the remaining count is 1

(4) And finally, the selection of one ¹ 1 coins solves the problem

Though, it seems to be working fine, for this count we need to pick only 4 coins. But if we slightly change the problem then the same approach may not be able to produce the same optimum result.

For the currency system, where we have coins of 1, 7, 10 value, counting coins for value 18 will be absolutely optimum but for count like 15, it may use more coins than necessary. For example, the greedy approach will use 10 + 1 + 1 + 1 + 1 + 1, total 6 coins. Whereas the same problem could be solved by using only 3 coins (7 + 7 + 1)

Hence, we may conclude that the greedy approach picks an immediate optimized solution and may fail where global optimization is a major concern.

Examples : Most networking algorithms use the greedy approach. Here is a list of few of them :

PUBLICATION

(1) Travelling Salesman Problem

(2) Prim's Minimal Spanning Tree Algorithm

(3) Kruskal's Minimal Spanning Tree Algorithm

(4) Dijkstra's Minimal Spanning Tree Algorithm

(5) Graph - Map Coloring

(6) Graph - Vertex Cover

(7) Knapsack Problem

(8) Job Scheduling Problem

There are lots of similar problems that uses the greedy approach to find an optimum solution.

Ans.(c)Dominance rule or Purging rule : If S^{i+1} contains (P_j, W_j) and (P_k, W_k); these two pairs such that :

$P_j \leq P_k$ and $W_j \geq W_k$, then (P_j, W_j) can be eliminated. This purging rule is also called as dominance rule. In purging rule basically the dominated tuples gets purged. In short, remove the pair with less profit and more weight.

Ans.(d)P Class : The class P consists of those problems that are solvable in polynomial time, i.e. these problems can be solved in time $O(n^k)$ in worst-case, where k is constant.

These problems are called *tractable*, while others are called *intractable* or superpolynomial.

Formally, an algorithm is polynomial time algorithm, if there exists a polynomial $p(n)$ such that the algorithm can solve any instance of size n in a time $O(p(n))$.

Problem requiring $\Omega(n^{50})$ time to solve are essentially intractable for large n. Most known polynomial time algorithm run in time $O(n^k)$ for fairly low value of k.

The advantages in considering the class of polynomial-time algorithms is that all reasonable deterministic single processor model of computation can be simulated on each other with at most a polynomial slow-d

NP-Class : The class NP consists of those problems that are verifiable in polynomial time. NP is the class of decision problems for which it is easy to check the correctness of a claimed answer, with the aid of a little extra information. Hence, we aren't asking for a way to find a solution, but only to verify that an alleged solution really is correct.

Every problem in this class can be solved in exponential time using exhaustive search.

Section – A

Q.2.(a) What are disjoint sets. Also write Union and find algorithm for disjoint sets. (10)

Ans. Disjoint sets : Two or more sets are said to be disjoint if they have no element in common. For example, {1, 2, 3} and {4, 5, 6} are disjoint sets, while sets {1, 2, 3} and {2, 4, 5, 6} are not. The elements of a set are letters, numbers, or any other objects. For example, the elements of the set {a, b, c} are the letters a, b, and c.

PUBLICATION

Formally, two sets A and B are disjoint, if their intersection is the empty set, that is if $A \cap B = \Phi$. This definition extends to any collection of sets. A collection of sets is pairwise disjoint or mutually disjoint if any two distinct sets in the collection are disjoint. Formally, let I be an index set, and for each i in I, let A_i be a set. Then, the family of sets $\{A_i, : i \in I\}$ is pairwise disjoint if for any i and j in I with $i \neq j, A_i \cap B_j = \Phi$. For example , the collection of sets {(1), (2), (3), ...} is pairwise disjoint. If $\{A_i\}$ is a pairwise disjoint collection (containing at least two sets), then clearly its intersection is empty.

A union-find algorithm is an algorithm that performs two useful operations on such a data structure:

Find : Determine which subset a particular element is in. This can be used for determining if two elements are in the same subset.

Union : Join two subsets into a single subset.

```
procedure Make-Set (x)
1. size[x]   ← 1
2. parent[x]   ← x
end.
procedure UNION(a , b) { with weight balancing }
{a and b are roots of two distinct trees in the forest.}
```

{Makes the root of the smaller tree a child of the root of the larger tree.}
1. if size[a] < size[b] then a $\leftrightarrow$ b
2. parent[b] $\leftarrow$ a
3. size[a] $\leftarrow$ size[a] + size[b]
end.
function FIND(x) { with path compression }
{Returns the root of the tree that contains node x.}
1. if parent[x] $\neq$ x then
2. parent[x] $\leftarrow$ FIND(parent[x])
3. return parent[x]
end.

Q.2.(b) Explain time & space complexity. (10)

Ans. *Time complexity* is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm. "Time" can mean the number of memory accesses performed, the number of comparisons between integers, the number of times some inner loop is executed, or some other natural unit related to the amount of real time the algorithm will take. We try to keep this idea of time separate from "wall clock" time, since many factors unrelated to the algorithm itself can affect the real time (like the language used, type of computing hardware, proficiency of the programmer, optimization in the compiler, etc.). It turns out that, if we chose the units wisely, all of the other stuff doesn't matter and we can get an independent measure of the efficiency of the algorithm.

Space complexity is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm. We often speak of "extra" memory needed, not counting the memory needed to store the input itself. Again, we use natural (but fixed-length) units to measure this. We can use bytes, but it's easier to use, say, number of integers used, number of fixed-sized structures, etc. In the end, the function we come up with will be independent of the actual number of bytes needed to represent the unit. Space complexity is sometimes ignored because the space used is minimal and/or obvious, but sometimes it becomes as important an issue as time.

For example, we might say "this algorithm takes n^2 time," where n is the number of items in the input. Or we might say "this algorithm takes constant extra space," because the amount of extra memory needed doesn't vary with the number of items processed.

Q.3.(a) Explain Quick sort algorithm in detail. Analyse its complexity also.(10)

Ans. Quick Sort : Quick sort is a sorting algorithm that uses the divide and conquer strategy. In this method division is dynamically carried out. The three steps of quick sort are as follows :

(1) *Divide* : Split the array into two sub arrays that each element in the left sub array is less than or equal the middle element and each element in the right sub array is greater than the middle element. The splitting of the array into two sub arrays is based on pivot element. All the elements that are less than pivot should be in left sub array and all the elements that are more than pivot should be in right sub array.

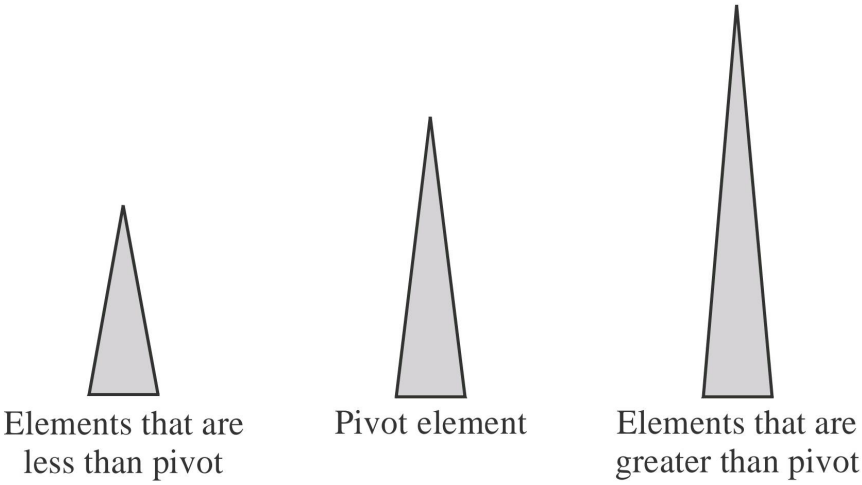
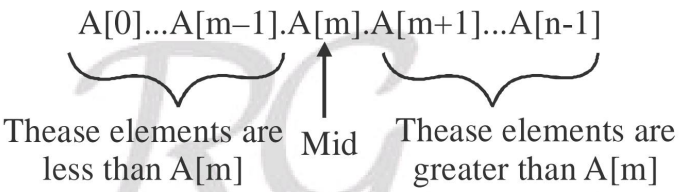


Fig. : (1)

(2) *Conquer* : Recursively sort the two sub arrays.

(3) *Combine* : Combine all the sorted elements in a group to form a list of sorted elements.

In merge sort the division of array is based on the positions of array elements, but in quick sort this division is based on actual value of the element. Consider an array A[*i*] where *i* is ranging from 0 to $n - 1$ then we can formulize the division of array elements as



Algorithm : The quick sort algorighm is performed using following two important function- Quick and partition. Let us see them :

Algorithm Quick (A[0...n – 1], low, high)
// Problem Description : This algorithm performs sorting of
// the elements given in Array A [0...n – 1]
// Input : An array A[0...n –1] in which unsorted elements are
// given. The low indicates the leftmost element in the list
// and high indicates the rightmost element in the list
// Output: Creates a sub array which is sorted in ascending
// order
if (low<high) then
// split the array into two sub arrays
m ← partition (A[low...high]) // m is mid of the array
Quick (A[low...m – 1])
Quick (A[mid + 1...high])

In above algorithm call to partition algorithm is given. The partition performs arrangement of the elements in ascending order. The recursive quick routine is for dividing the list in two sub lists.

Analysis :

Best case (split in the middle) : If the array is always partitioned at the mid, then it brings the best case efficiency of an algorithm.

The recurrence relation for quick sort for obtaining best case time complexity is,

$$C(n) = \underbrace{C(n/2)}_{\text{Time required to sort left sub array}} + \underbrace{C\left(\frac{n}{2}\right)}_{\text{Time required to sort right sub array}} + \underbrace{n}_{\text{Time required for partitioning the sub array}}$$

and $C(1) = 0$

Method 1 : Using master theorem : We will solve equation 1 using master theorem. The master theorem is

If $f(n) \in \Theta(n^d)$ then

(1) $T(n) = \Theta(n^d)$ if $a < b^d$

(2) $T(n) = \Theta(n^d \log n)$ if $a = b^d$

(3) $T(n) = \Theta(n^{\log_b a})$ if $a > b^d$

We get,

$$C(n) = 2\,C(n/2) + n$$

Here $f(n) \in n^1$ $\therefore d = 1$

Now, a = 2 and b = 2.

As from case 2 we get $a = b^d$ i.e. $2 = 2^1$, we get

T(n) i.e. $C(n) = \Theta(n^d \log n)$

$\therefore$ $C(n) = \Theta(n \log n)$

Thus,

Best case time complexity of quick sort is $\Theta(n \log_2 n)$.

Worst case (sorted array) : The worst case for quick sort occurs when the pivot is a minimum or maximum of all the elements in the list. This can be graphically represented as

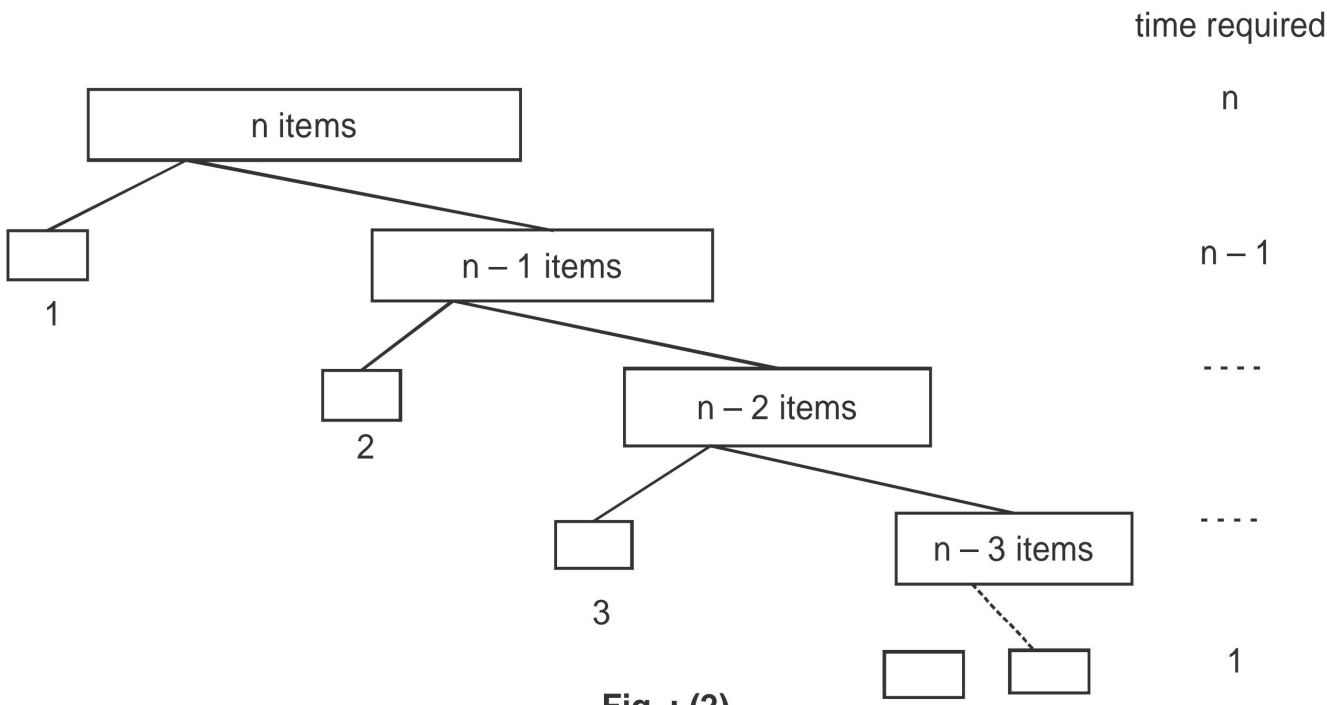


Fig. : (2)

We can write it as

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

or $$C(n) = n + (n-1) + (n-2) + \ldots + 2 + 1$$

But as we know

$$1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2} = \frac{1}{2}n^2.$$

$$\therefore \quad C(n) = \Theta(n^2).$$

The time complexity of worst case of quick sort is $\Theta(n^2)$.

$n*C_{avg}(n) - (n-1)*C_{avg}(n-1) = 2C_{avg}(n-1) + (2n-1)$

$n*C_{avg}(n) = (n+1)*C_{avg}(n-1) + (2n-1) < (n+1)*C_{avg}(n-1) + 2n$

If we assume

$$(n+1)*C_{avg}(n-1) + 2n = (n+1)*C_{avg}(n-1) + C'n$$

Then, $(n+1)*C_{avg}(n-1) < (n+1)*C_{avg}(n-1) + C'n$

Divide this equation by n (n + 1) then

$$n*C_{avg}(n) < (n+1)*C_{avg}(n-1)\ C'n$$

$$\frac{C_{avg}(n)}{(n+1)} < \frac{C_{avg}(n-1)}{n} + \frac{C'}{(n+1)}$$

If we assume

$$A(n) = \frac{C_{avg}(n)}{(n+1)} \text{ then}$$

$$A(n) < A(n-1) + \frac{C'}{(n+1)}$$

$$A(n-1) < A(n-2) + \frac{C'}{n}$$

$$A(n-2) < A(n-3) + \frac{C'}{n-1}$$

...

...

...

$$A(1) < A(0) + \frac{C'}{2}$$

Adding and cancelling these equations we get,

$$A(n) < C'*\left[\frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} + \frac{1}{n+1}\right]$$

$$\therefore C_{avg}(n) = (n+1)*A(n)$$

$$i.e. (n+1)*A(n) < C''(n+1)\log n$$

$$\therefore C_{avg}(n) = \Theta(n \log n)$$

Hence average case time complexity of quick sort is $\Theta(n \log n)$.

Average case (random array) : Let, $C_{avg}(n)$ denotes the average time of quick sort (A[1 ... n]).

The recurrence relation for random input array is

$$C(n) = C(0) + C(n-1) + n$$

or $$C(n) = C(1) + C(n-2) + n$$

or $$C(n) = C(2) + C(n-3) + n$$

or $$C(n) = C(3) + C(n-4) + n$$

...

...

...

$$C(n) = C(n-1) + C(0) + n$$

The average value of C(n) is sum of all the possible values divided by n.

$$\therefore \quad C_{avg}(n) = \frac{2}{n}(C(1)+C(2)+...+C(n-1))+n$$

Multiplying both the sides by n we get,

$$n^* C_{avg}(n) = 2 (C(1) + C(2) + ... + C(n-1)) + n^2$$

Now we put n = n – 1 then,

$$(n-1)^* C_{avg}(n-1) = 2 [C_{avg}(1) + C_{avg}(2) + ... + C_{avg}(n-2)] + (n-1)^* (n-1)$$

$$n^*C_{avg}(n) - (n-1)^*C_{avg}(n-1) = 2[C_{avg}(1)+ C_{avg}(2)+ ... + C_{avg}(n-2)] + (n-1)^*(n-1)$$

Time complexity of quick sort

Best case	Average case	Worst case
$\Theta(n \log_2 n)$	$\Theta(n \log_2 n)$	$\Theta(n^2)$

Q.3.(b)State Matrix chain multiplication problem. How to solve this problem with Dynamic programming? Explain. (10)

Ans. Matrix chain multiplication is an optimization problem that can be solved using dynamic programming. Given a sequence of matrices, we want to find the most efficient way to multiply these matrices together. The problem is not actually to *perform* the multiplications, but merely to decide in which order to perform the multiplications.

Dynamic Programming Algorithm : To begin, let us assume that all we really want to know is the minimum cost, or minimum number of arithmetic operations, needed to multiply out the matrices. If we are only multiplying two matrices, there is only one way to multiply them, so the minimum cost is the cost of doing this. In general, we can find the minimum cost using the following recursive algorithm:

- Take the sequence of matrices and separate it into two subsequences.
- Find the minimum cost of multiplying out each subsequence.
- Add these costs together, and add in the cost of multiplying the two result matrices.
- Do this for each possible position at which the sequence of matrices can be split, and take the minimum over all of them.

For example, if we have four matrices *ABCD*, we compute the cost required to find each of (*A*)(*BCD*), (*AB*)(*CD*), and (*ABC*)(*D*), making recursive calls to find the minimum cost to compute *ABC*, *AB*, *CD*, and *BCD*. We then choose the best one. Better still, this yields not only the minimum cost, but also demonstrates the best way of doing the multiplication: group it the way that yields the lowest total cost and do the same for each factor.

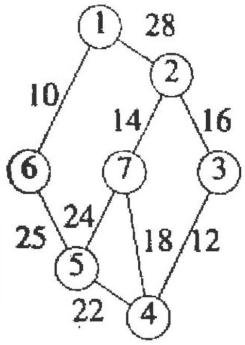
Unfortunately, if we implement this algorithm we discover that it is just as slow as the naive way of trying all permutations! What went wrong? The answer is that we're doing a lot of redundant work. For example, above we made a recursive call to find the best cost for computing both *ABC* and *AB*. But finding the best cost for computing ABC also requires finding the best

cost for *AB*. As the recursion grows deeper, more and more of this type of unnecessary repetition occurs.

One simple solution is called memoization: each time we compute the minimum cost needed to multiply out a specific subsequence, we save it. If we are ever asked to compute it again, we simply give the saved answer, and do not recompute it. Since there are about $n^2/2$ different subsequences, where n is the number of matrices, the space required to do this is reasonable. It can be shown that this simple trick brings the runtime down to $O(n^3)$ from $O(2^n)$, which is more than efficient enough for real applications. This is top-down dynamic programming.

Section – B

Q.4. Explain the concept of minimum spanning trees. Solve the following graph using prime's algorithm. Also analyse its complexity. (20)



Ans. Minimum spanning trees : A minimum spanning tree (MST) or minimum weight spanning tree is then a spanning tree with weight less than or equal to the weight of every other spanning tree. More generally, any undirected graph has a minimum spanning forest, which is a union of minimum spanning trees for its connected components.

Practical applications based on minimal spanning trees include :

– Taxonomy, one of the earliest motivating applications.

– Cluster analysis: clustering points in the plane, single-linkage clustering , graph-theoretic clustering, and clustering gene expression data. Constructing trees for broadcasting in computer networks.

– Image registration and segmentation *e.g.* see minimum spanning tree-based segmentation.

– Curvilinear feature extraction in computer vision.

– Handwriting recognition of mathematical expressions.

– Circuit design: implementing efficient multiple constant multiplications, as used in finite impulse response filters.

– Regionalisation of socio-geographic areas, the grouping of areas into homogeneous, contiguous regions.

– Comparing ecotoxicology data.

– Topological observability in power systems.

– Measuring homogeneity of two-dimensional materials.

– Minimax process control

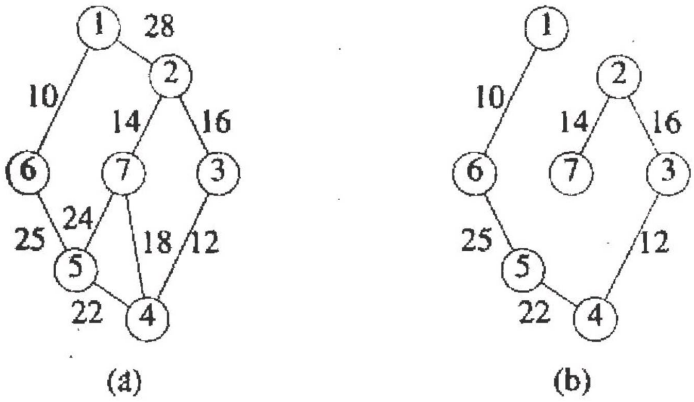


Fig.(1) : Graph and its minimum cost spanning tree

Let us obtain a pseudocode algorithm to find a minimum-cost spanning tree using prim's method. The algorithm will start with a tree that includes only a minimum-cost edge of G. Then, edges are added to this tree one by one. The next edge (i, j) to be added is such that i is a vertex already included in the tree, j is a vertex not yet included, and the cost of (i, j) cost[i, j], is minimum among all edges (k, l) such that vertex k is in the tree and vertex l is not in the tree. To determine this edge (i, j) efficiently, we associate with each vertex j not yet included in the tree a value near [j]. The value near [j] is a vertex in the tree such that cost[j, near[j]] is minimum among all choices for near [j]. We define near [j] = 0 for all vertices j that are already in the tree. The next edge to include is defined by the vertex j such that near[j] $\neq$ 0 (j not already in the tree) and cost[j, near[j]] is minimum.

In function Prim, line 9 selects a minimum-cost edge. Lines 10 to 15 initialize the variables so as to represent a tree comprising only the edge (k, l). In the for loop of line 16 the remainder of the spanning tree is built up edge by edge. Lines 18 and 19 select (j, near[j]) as the next edge to include. Lines 23 to 25 update near[ ].

The time required by algorithm Prim is $O(n^2)$, where n is the number of vertices in the graph G. To see this, note that line 9 takes $O(|E|)$ time and line 10 takes $\Theta(1)$ time. To see this, note that line 9 takes O(|E|) time and line 10 takes $\Theta(1)$ time. The for loop of line 12 takes $\Theta(n)$ time. Lines 18 and 19 and the for loop of line 23 require $O(n)$ time. So, each iteration of the for loop of line 16 takes $O(n)$ time. The total time for the for loop of line 16 is therefore $O(n^2)$. Hence, Prim runs in $O(n^2)$ time.

If we store the nodes not yet included in the tree as a red-black tree, lines 18 and 19 take $O(\log n)$ time. Note that a red-black tree supports the following operations in $O(\log n)$ time: insert, delete (an arbitrary element), find-min, and search (for an arbitrary element). The overall frequency is $O(|E|)$. Updating in lines 24 and 25 also takes $O(\log n)$ time (since an update can be done using a delete and an insertion into the red-black tree). Thus the overall run time is $O(n + |E|) \log n)$.

The algorithm can be speeded a bit by making the observation that a minimum-cost spanning tree includes for each vertex v a minimum-cost edge incident to v. To see this, suppose

t is a minimum-cost spanning tree for $G = (V, E)$. Let v be any vertex in t. Let (v, w) be an edge with minimum cost among all edges incident to v. Assume that $(v, w) \notin E(t)$ and cost$[v, w] <$ cost$[v, x]$ for all edges $(v, x) \in E(t)$. The inclusion of (v, w) into t creates a unique cycle. This cycle must include an edge (v, x), $x \neq w$. Removing (v, x) from $E(t) \cup \{(v, w)\}$ breaks this cycle without disconnecting the graph $(V, E(t) \cup \{v, w)\})$. Hence, $(V, E(t) \cup \{(v, w)\} - \{(v, x)\})$ is also a spanning tree. Since $cost[v, w] < cost[v, x]$, this spanning tree has lowe cost than t. This contradicts the assumption that t is a minimum-cost spanning tree of G. So, t includes minimum-cost edges as stated above.

From this observation it follows that we can start the algorithm with a tree consisting of any arbitrary vertex and no edge. Then edges can be added one by one. The changes needed are to lines 9 to 17. These lines can be replaced by the lines

```
9'          mincost : = 0;
10'         for i : = 2 to n do near[i] : = 1;
11'                 // Vertex 1 is initially in t.
12'         near[1] : = 0;
13'-16'     for i : = 1 to n – 1 do
17'         { // Find n – 1 edges for t.
```

Q.5. Explain Optimal Binary search tree which includes following problem : n = 4 and (a_1, a_2, a_3, a_4) = (do, if, int, while) with profits P (1 : 4) = (3, 3, 1, 1) in case of successful search & loss q (0 : 4) = (2, 3, 1, 1, 1) in case of unsuccessful search. Initially [W (i, i) = q (i)], c (i, i) = 0 and r (i, i) = 0 where 0 < i < = 4. Also write its algorithm and analyse its complexity. (20)

Ans. Using the observation $w(i, j) = p(j) + q(j) + w(i, j - 1)$, we get

$$
\begin{aligned}
w(0, 1) &= p(1) + q(1) + w(0, 0) = 8 \\
c(0, 1) &= w(0, 1) + \min\{c(0, 0) + c(1, 1)\} = 8 \\
r(0, 1) &= 1 \\
w(1, 2) &= p(2) + q(2) + w(1, 1) = 7 \\
c(1, 2) &= w(1, 2) + \min\{c(1, 1) + c(2, 2)\} = 7 \\
r(0, 2) &= 2 \\
w(2, 3) &= p(3) + q(3) + w(2, 2) = 3 \\
c(2, 3) &= w(2, 3) + \min\{c(2, 2) + c(3, 3)\} = 3 \\
r(2, 3) &= 3 \\
w(3, 4) &= p(4) + q(4) + w(3, 3) = 3 \\
c(3, 4) &= w(3, 4) + \min\{c(3, 3) + c(4, 4)\} = 3 \\
r(3, 4) &= 4
\end{aligned}
$$

Knowing $w(i, i + 1)$ and $c(i, i + 1)$, $0 \leq i < 4$, we can use to compute $w(i, i + 2)$, and $r(i, i + 2)$, $0 \leq i < 3$. This process can be repeated until $w(0, 4)$, $c(0, 4)$, and $r(0, 4)$ are obtained. The table of Fig.(1) shows the results of this computation. The box in row i and column j shows the values of $w(j, j + i)$, $c(j, j + i)$ and $r(j, j + i)$ respectively.

	0	1	2	3	4
0	$w_{00} = 2$ $c_{00} = 0$ $r_{00} = 0$	$w_{11} = 3$ $c_{11} = 0$ $r_{11} = 0$	$w_{22} = 1$ $c_{22} = 0$ $r_{22} = 0$	$w_{33} = 1$ $c_{33} = 0$ $r_{33} = 0$	$w_{44} = 1$ $c_{44} = 0$ $r_{44} = 0$
1	$w_{01} = 8$ $c_{01} = 8$ $r_{01} = 1$	$w_{12} = 7$ $c_{12} = 7$ $r_{12} = 2$	$w_{23} = 3$ $c_{23} = 3$ $r_{23} = 3$	$w_{34} = 3$ $c_{34} = 3$ $r_{34} = 4$	
2	$w_{02} = 12$ $c_{02} = 19$ $r_{02} = 1$	$w_{13} = 9$ $c_{13} = 12$ $r_{13} = 2$	$w_{24} = 5$ $c_{24} = 8$ $r_{24} = 3$		
3	$w_{03} = 14$ $c_{03} = 25$ $r_{03} = 2$	$w_{14} = 11$ $c_{14} = 19$ $r_{14} = 2$			
4	$w_{04} = 16$ $c_{04} = 32$ $r_{04} = 2$				

Fig.(1) : Computation of c(0,4), w(0,4), and r(0,4)

The computation is carried out by row from row 0 to row 4. From the table we see that $c(0, 4) = 32$ is the minimum cost of a binary search tree for (a_1, a_2, a_3, a_4). The root of tree t_{04} is a_2. Hence, the left subtree is t_{01} and the right subtree t_{24}. Tree t_{01} has root a_1 and subtrees t_{00} and t_{11}. Tree t_{24} has root a_3; its left subtree is t_{22} and its right subtree t_{34}. Thus, with the data in the table it is possible to reconstruct t_{04}. Fig.(2) shows t_{04}.

RG PUBLICATION

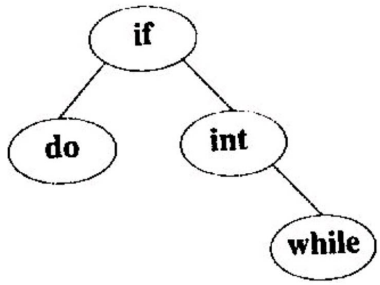


Fig.(2) : Optimal search tree

Complexity : Let us examine the *complexity* of this procedure to evaluate the c's and r's. The evaluation procedure described in the above problem requires us to compute $c(i, j)$ for $(j - i) = 1, 2, ..., n$ in that order. When $j - i = m$, there are $n - m + 1$ $c(i, j)$'s to compute. The computation of each of these $c(i, j)$'s requires us to find the minimum of m quantities. Hence, each such $c(i, j)$ can be computed in time $O(m)$. The total time for all $c(i, j)$'s with $j - i = m$ is therefore $O(nm - m^2)$. The total time to evaluate all the $c(i, j)$'s and $r(i, j)$'s is therefore

$$\sum_{1 \le m \le n} (nm - m^2) = O(n^3)$$

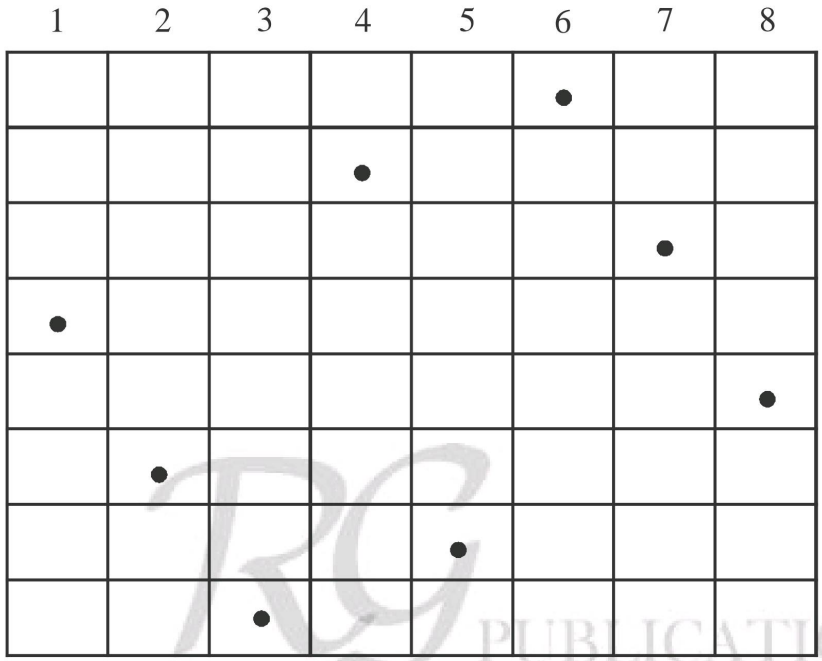
Section – C

Q.6. Explain 8-queens method, graph coloring and Hamiltonian cycle. (20)

Ans. 8-Queen's problem : Consider a chessboard of order 8 × 8. The problem is to place 8 queens on this board such that no two queens can attack other. That means no two queens can be placed on the same row, column or diagonal.

The solution to 8-queens problem can be obtained using backtracking method.

The solution can be given as below :



RG PUBLICATION

Graph coloring : The backtracking search tree starts at non-labeled root. The immediate children of the root represent the available choices regarding the coloring of the first vertex (vertex a in this case) of the graph. However, the nodes of the tree are generated only if necessary. The first (leftmost) child of the root represents the first choice regarding the coloring of vertex a, which is to color the vertex with color 1 — this is indicated by the label assigned to that node. So far this partial coloring does not violate the constraint that adjacent vertices be assigned different colors. So we grow the tree one more level where we try to assign a color to the second vertex (vertex b) of the graph. The leftmost node would represent the coloring of vertex b with color 1. However, such partial coloring where a=1 and b=1 is invalid since the vertices a and b are adjacent. Also, there is no point of growing the tree further from this node since, no matter what colors we assign to the remaining vertices, we will not get a valid coloring for all of the vertices. So we prune (abandon) this path and consider an alternative path near where we are. We try the next choice for the coloring of vertex b, which is to color it with color 2. The coloring $a = 1$ and $b = 2$ is a valid partial coloring so we continue to the next level of the search tree. When we reach the lowest level and try to color vertex e, we find that it is not possible to color it any of the three colors; therefore, we backtrack to the previous level and try the next choice for the coloring of vertex d. We see that the coloring of vertex d with color 2 leads to a violation (d and b are adjacent and have same color). We continue to d=3 and then down to $e = 1$, which gives the solution, $\{a = 1, b = 2, c = 3, d = 3, e = 1\}$.

Hamiltonian cycle : This is a problem in which graph G is accepted as input and it is asked to find a simple cycle in G that visits each vertex of G exactly once and returns to its starting vertex. Such a cycle is called an *Hamiltonian cycle*.

Theorem : HAMILTONIAN CYCLE is in NP.

Proof : Let A be some non-deterministic algorithm to which graph G is given as input. The vertices of graph are numbered from 1 to N. We have to call the algorithm recursively in order to get the sequence S. This sequence will have all the vertices without getting repeated. The vertex from which the sequence starts must be ended at the end. This check on the sequence S must be made in polynomial time *n*. Now if there is a Hamiltonian cycle in the graph then algorithm will output "yes". Similarly if we get output of algorithm as "yes" then we could guess the cycle in G with every vertex appearing exactly once and the first visited vertex getting visited at the last. That means A non-deterministically accepts the language HAMILTONIAN CYCLE. It is therefore proved that HAMILTONIAN CYCLE is in NP.

Q.7. Solve the following problem by using least cost Branch & Bound method : Knapsack instance n = 4, p (1 : 4) = (10, 10, 12, 18) and weight w (1 : 4) = (2, 4, 6, 9) & max. capacity m = 15. (20)

Ans. The search begins with the root as the E-node. For this code, node 1 of Fig.(1), we have $\hat{c}(1) = -38$ and $u(1) = -32$.

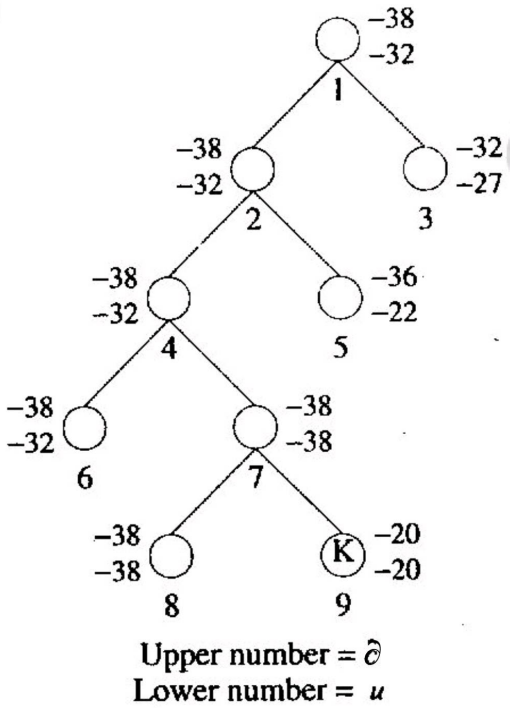


Fig.(1) : LC branch-and-bound tree

The computation of $u(1)$ and $\hat{c}(1)$ is done as follows. The bound $u(1)$ has a value UBound (0, 0, 0, 15). UBound scans through the objects from left to right starting from *j*; it adds these objects into the knapsack until the first object that doesn't fit is encountered. At this time, the negation of the total profit of all the objects in the Knapsack plus *cw* is returned. In Function

UBound, c and b start with a value of zero. For $i = 1$, 2, and 3, c gets incremented by 2, 4, and 6, respectively. The variable b also gets decremented by 10, 10, and 12, respectively. When $i = 4$, the test $(c + w[i] \leq m)$ fails and hence the value returned is –32. Function Bound is similar to UBound, except that it also considers a fraction of the first object that doesn't fit the knapsack. For example, in computing $\hat{c}(1)$, the first object, that doesn't fit is 4 whose weight is 9. The total weight of the objects 1, 2, and 3 is 12. So, Bound considers a fraction $\frac{3}{9}$ of the object 4 and hence returns $-32 - \frac{3}{9} * 18 = -38$.

Since node 1 is not a solution node, *LCBB* sets ans = 0 and upper = –32 (ans being a variable to store intermediate answer nodes). The E-node is expanded and its two children, nodes 2 and 3, generated). The cost $\hat{c}(2) = -38$, $\hat{c}(3) = -32$, $u(2) = -32$, and $u(3) = -27$. Both nodes are put onto the list of live nodes. Node 2 is the next E-node. It is expanded and nodes 4 and 5 generated. Both nodes get added to the list of live nodes. Node 4 is the live node with least $\hat{c}$ value and becomes the next E-node. Nodes 6 and 7 are generated. Assuming node 6 is generated first, it is added to the list of live nodes. Next, node 7 joins this list and upper is updated to –38. The next E-node will be one of nodes 6 and 7. Let us assume it is node 7. Its two children are nodes 8 and 9. Node 8 is a solution node. Then upper is updated to –38 and node 8 is put onto the live nodes list. Node 9 has $\hat{c}(9) >$ upper and is killed immediately. Nodes 6 and 8 are two live nodes with least $\hat{c}$. Regardless of which becomes the next E-node, $\hat{c}(E) \geq$ upper and the search terminates with node 8 the answer node. At this time, the value –38 together with the path 8, 7, 4, 2, 1 is printed out and the algorithm terminates. From the path one cannot figure out the assignment of values to the x_i's such that $\sum p_i x_i$ = upper. Hence, a proper implementation of LCBB has to keep additional information from which the values of the x_i's can be extracted. One way is to associate with each node a one bit field, tag. The sequence of tag bits from the answer node to the root give the x_i values. Thus, we have tag(2) = tag(4) = tag(6) = tag(8) = 1 and tag(3) = tag(5) = tag(7) = tag(9) = 0. The tag sequence for the path 8, 7, 4, 2, 1 is 1 0 1 1 and so $x_4 = 1$, $x_3 = 1$, $x_3 = 0$, $x_2 = 1$, and $x_1 = 1$.

Section – D

Q.8. Write short note on : (20)

(a) Satisfiability problem

(b) Clique decision problem

(c) Reducibility.

Ans. (a) Satisfiability problem :

(1) CNF - SAT problem : This problem is based on Boolean formula. The Boolean formula has various Boolean operations such as OR(+), AND (.) and NOT. There are some notations such as $\rightarrow$ (means implies) and $\leftrightarrow$ (means if and only if).

A Boolean formula is in Conjuctive Normal Form (CNF) if it is formed as collection of subexpressions. These subexpressions are called clauses.

For example : $(\bar{a} + b + d + \bar{g})\,(c + \bar{e})\,(\bar{b} + d + \bar{f} + h)\,(a + c + e + \bar{h})$

This formula evaluates to 1 if b, c, d are 1.

The CNF-SAT is a problem which takes Boolean formula in CNF form and checks whether any assignment is there to Boolean values so that formula evaluates to 1.

Theorem : CNF-SAT is in NP complete.

Proof : Let *S* be the Boolean formula for which we can construct a simple non-deterministic alogirithm which can guess the values of variables in Boolean formula and then evaluates each clause of *S*. If all the clauses evaluate S to 1 then S is satisfied. Thus CNF-SAT is in NP-complete.

(2) A 3SAT problem : A 3SAT problem is a problem which takes a Boolean formula S in CNF form with each clause having exactly three literals and check whether S is satisfied or not. [Note that CNF means each literal is ORed to form a clause, and each clause is ANDed to form Boolean formula S].

Following formula is an instance of 3SAT problem :

$$(\overline{a}+b+\overline{g})(c+\overline{e}+f)(\overline{b}+d+\overline{f})(a+e+\overline{h})$$

Theorem : 3SAT is in NP complete.

Proof : Let S be the Boolean formula having 3 literals in each clause for which we can construct a simple non-deterministic algorithm which can guess an assignment of Boolean values to S. If the S evaulated as 1 then S is satisfied. Thus we can prove that 3SAT is in NP-complete.

Ans.(b)Clique Decision problem (CDP) : The Conjective Normal Form (CNF) α clique decision problem and using transtivity of reducibility and above result we conclude that satisfiability reduces to conjective normal form-satisfiability. Hence satisfiability reduces to clique decision problem and clique decision problem is NP hard. Since clique decision problem belongs to NP, so clique decision problem is also NP complete.

Ans.(c) Reducibility : To prove whether particular problem is NP complete or not we use polynomial time reducibility. That means if

$$A \xrightarrow{Poly} B \text{ and } B \xrightarrow{Poly} C \text{ then } A \xrightarrow{Poly} C.$$

The reduction is an important task in NP completeness proofs. This can be illustrated by Fig.(1).

Various types of reductions are :

Local replacement : In this reduction A → B by dividing input to A in the form of components and then these components can be converted to components of B.

Component design : In this reduction A → B by building special component for input B that enforce propertied required by A.

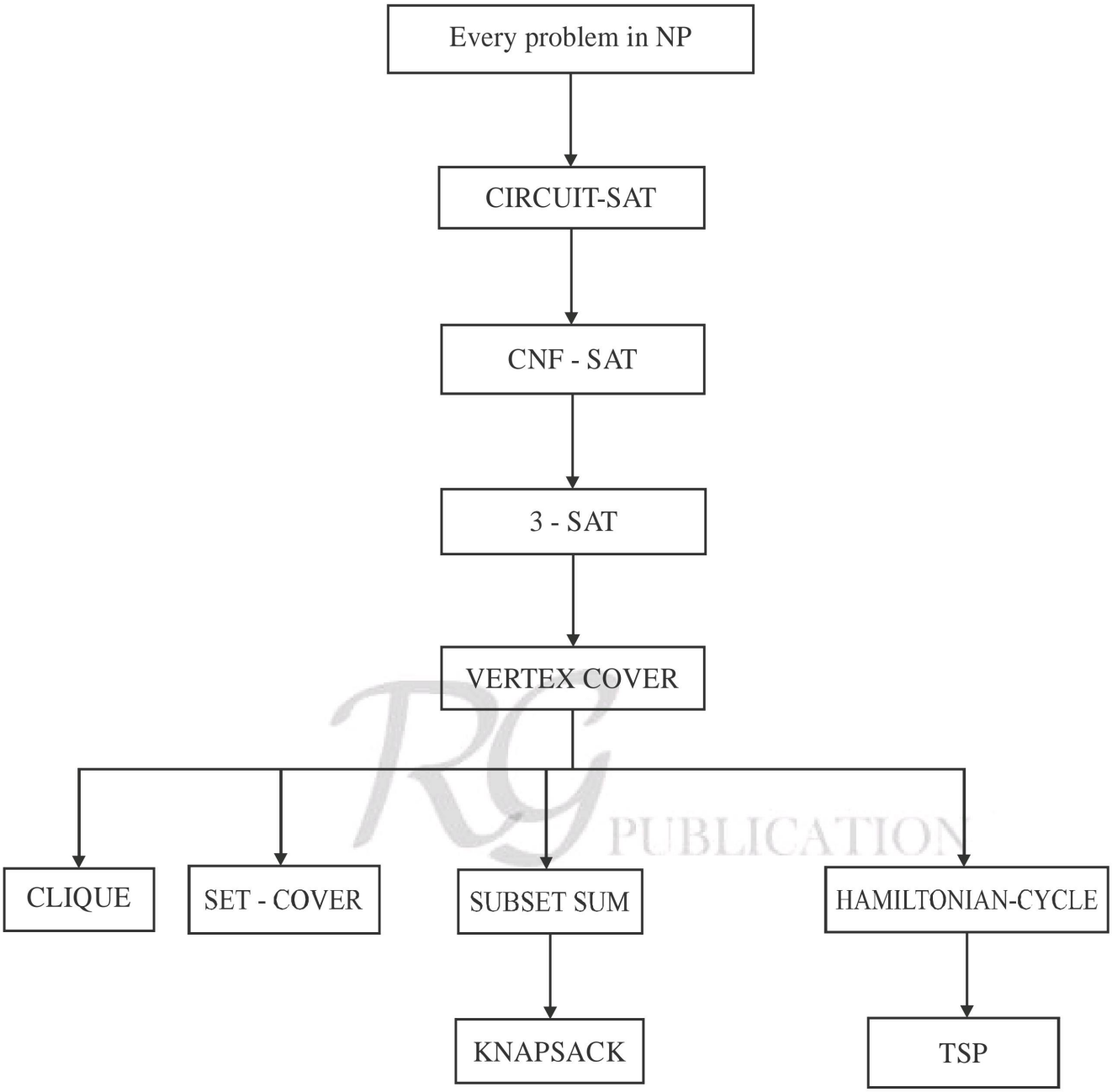


Fig.(1) : Reduction in NP competeness

Q.9.(a) State & Prove Cook's theorem. (10)

Ans. Cook's theorem, states that the Boolean satisfiability problem is NP-complete. That is, any problem in NP can be reduced in polynomial time by a deterministic Turing machine to the problem of determining whether a Boolean formula is satisfiable.

Proof-idea :

(i) First show that SAT is in NP.

(ii) Then show that every problem in NP is polynomial-time reducible to SAT.

Proof (i) :

It is easy to see that SAT is in NP. Simply non-deterministically choose Boolean values for each variable. Then check whether the expression is true or false for those values.

If it is true, the answer is "yes."

If it is false, the answer is no."

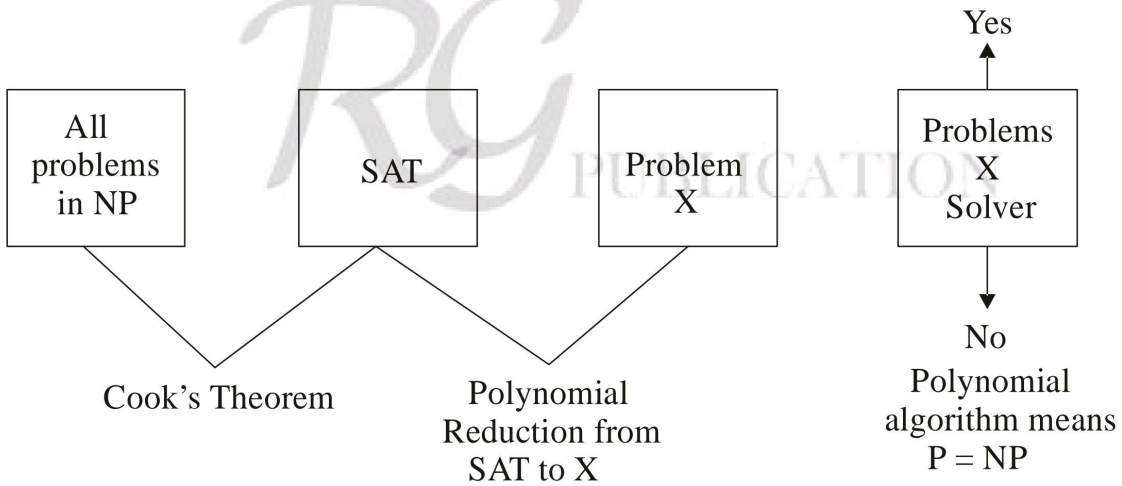
Proof (ii) :

Let A be a problem in NP.

Then there is a nondeterministic Turing machine M that decides A in polynomial time. Let the run time of M be at most n k, for some k 0. When M is run, it can access at most the ûrst n k + 1 cells.

Example : Cook's theorem proved SATISFIABILITY was hard by using a polynomial time reduction translating each problem in NP into an instance of SAT. Since a polynomial time algorithm of SAT would imply a polynomial time algorithm for everything in NP, SAT is NP hard. Since we can guess a solution to SAT, it is in NP and thus NP-complete.

The proof of Cook's Theorem,while quite clever, was certainly difficult and complicated. We had to show that all problems in NP could be reduced to SAT to make sure we did not miss a hard one. But now that we have a known NP-complete problem in SAT. For any other problem, we can prove it NP-hard of two polynomial time reductions can be done in polynomial time, all we need show is that SAT, that is, any instance of SAT can be translated to an instance of X in polynomial time.



Q.9.(b) Show that Job sequencing with deadline is NP hard problem. (10)

Ans. The problem is stated as below.

- There are n jobs to be processed on a machine.
- Each job i has a deadline d_ie" 0 and profit p_ie"0 .
- Pi is earned iff the job is completed by its deadline.
- The job is completed if it is processed on a machine for unit time.
- Only one machine is available for processing jobs.
- Only one job is processed at a time on the machine.
- A feasible solution is a subset of jobs J such that each job is completed by its deadline.
- An optimal solution is a feasible solution with maximum profit value.

A scheduling problem is NP-hard in the ordinary sense if

- partition (or a similar problem) can be reduced to this problem with a polynomial time algorithm and
- there is an algorithm with pseudo polynomial time complexity that solves the scheduling problem.

RG PUBLICATION

DESIGN AND ANALYSIS OF ALGORITHMS

Mar - 2021
Paper Code:PEC-CSE-311-G

Note : ***Attempt five questions in all, selecting one question from each Section. Question No.* 1 *is compulsory. All questions carry equal marks.***

Q.1.(a) What is algorithm ? Explain characteristics of algorithms. (3)

Ans. Algorithm : An algorithm is defined as the sequence of instructions written in simple english that are required to get the desired results. Algorithm is the pseudocode of the problem. It helps to develop the fundamental logic of a problem that leads to a solution.

Characteristics of an algorithm are as follows :

(i) Each step of an algorithm must be precisely defined.

(ii) An algorithm must contain blocks that will help to solve problems more efficiently and logically.

(iii) It should accepts a set of inputs and produce a defined output.

(iv) It should be independent from a computer programming language.

Q.1.(b) What is the time complexity of Merge sort and selection sort ? (3)

Ans. Time complexity is the amount of time taken by an algorithm to run, as a function of the length of the input.

Time complexity of merge Soft : Complexity of merge sort is O (n* log n) in all the 3 ases (worst, average and best) as merge sort always divides the array in two halves and takes linear time to merge two halves.

Time complexity of selection sort : Time complexity of selection sort is $O(n^2)$ in all the 3 cases (worst, average and best) as selection sort required two nested for loops to complete itself.

Q.1.(c) Explain P and NP class. (3)

Ans. P-class : The class P consists of those problems that are solvable in polynomial time, i.e., these problems can be solved in time $O(n^k)$ in worst case, where K is constant.

These problems are tractable.

The advantages in considering the class of polynomial-time algorithms is that all reasonable deterministic single processor model of computation can be simulated on each other with at most a polynomial slow-d.

e.g. – Fractional knapsack, MST and Sorting.

NP–class : In terms of solving a NP class problem, the run time would not be polynomial. It would be something like O(n !) or something much larger. However, this class of problems can be given a specific solution, and checking the solution would have a polynomial run-time.

e.g. – Traveling Salesmen, Graph colouring, etc.

Q.1.(d) Explain Divide and Conquer technique. (3)

Ans. Divide and conquer Technique : Divide and conquer is a recursive problem-solving approach which break a problem into smaller sub-problems, recursively solve the sub-problems, and finally combines the solutions to the sub-problems to solve the original problem. This method usually allows us to reduce the time complexity to a large extent.

Phases of Divide and Conquer Technique : It consists of three phases :

(i) Divide : Dividing the problem into two or more than two sub-problems that are similar to the original problem but smaller in size.

(ii) Conquer : Solve the sub-problems recursively.

(iii) Combine : Combine these solutions to sub-problems to create a solution to the original problems.

Examples of Divide and Conquer Technique : Merge Sort, Binary Search, Quick sort etc.

Q.1.(e) Explain Greedy algorithm. (3)

Ans. Greedy algorithm : A greedy algorithm, as the name suggests, always makes the choice that seems to be the best at that moment. This means that it makes a locally-optimal choice in the hope that this choice will lead to a globally-optimal solution.

How do you decide which choice is optimal?

Assume that you have an objective function that needs to be optimized (either maximized or minimized) at a given point. A Greedy algorithm makes greedy choices at each step to ensure that the objective function is optimized. The Greedy algorithm has only one shot to compute the optimal solution so that it never goes back and reverses the decision.

Greedy algorithms have some advantages and disadvantages:

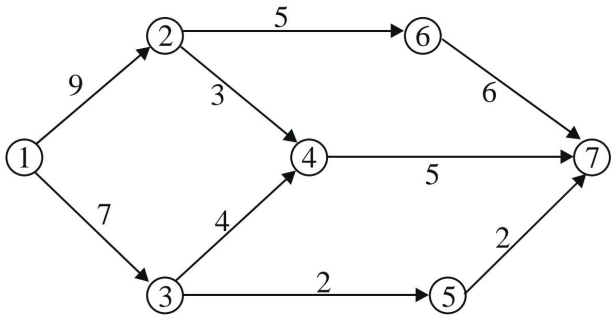
1. It is quite easy to come up with a greedy algorithm (or even multiple greedy algorithms) for a problem.

2. Analyzing the run time for greedy algorithms will generally be much easier than for other techniques (like Divide and conquer). For the Divide and conquer technique, it is not clear whether the technique is fast or slow. This is because at each level of recursion the size of gets smaller and the number of sub-problems increases.

3. The difficult part is that for greedy algorithms you have to work much harder to understand correctness issues. Even with the correct algorithm, it is hard to prove why it is correct. Proving that a greedy algorithm is correct is more of an art than a science. It involves a lot of creativity.

Q.1.(f) What is multistage graph ? (3)

Ans. Multistage Graph : A multistage graph is a directed graph in which the nodes can be divided into a set of stages such that all edges are from a stage to next stage only. In other words there is no edge between vertices of same stage and from a vertex of current stage to previous stage.



Q.1.(g) Write the applications of Branch and Bound problem. (3)

Ans. Branch and Bound is one of the problem solving technique. It is used for solving the optimization problems and minimization problems.

It is used to solve NP-hard problems such as :

(i) Traveling salesmen problem
(ii) 0/1 knapsack Problem
(iii) Integar programming
(iv) Non-linear programming.

Section – A

Q.2.(a) What is Stack ? Explain basic operations of stack and write algorithm of insert and delete. (8)

Ans. A stack is an ordered list in which all insertions and deletions are made at one end, called the top.

The operations of a stack imply that if the elements A, B, C, D, and E are inserted into a stack, in that order, then the first element to be removed (deleted) must be E. Equivalently we say that the last element to be inserted into the stack is the first to be removed. For this reason stacks are sometimes referred to as Last In First Out (LIFO) lists. The operations of a queue require that the first element that is inserted into the queue is the first one to be removed. Thus queues are known as First In First Out (FIFO) lists. See Fig.(1) for examples of a stack and a queue each containing the same five elements inserted in the same order. Note that the data object queue as defined here need not correspond to the concept of queue that is studies in quening theory.

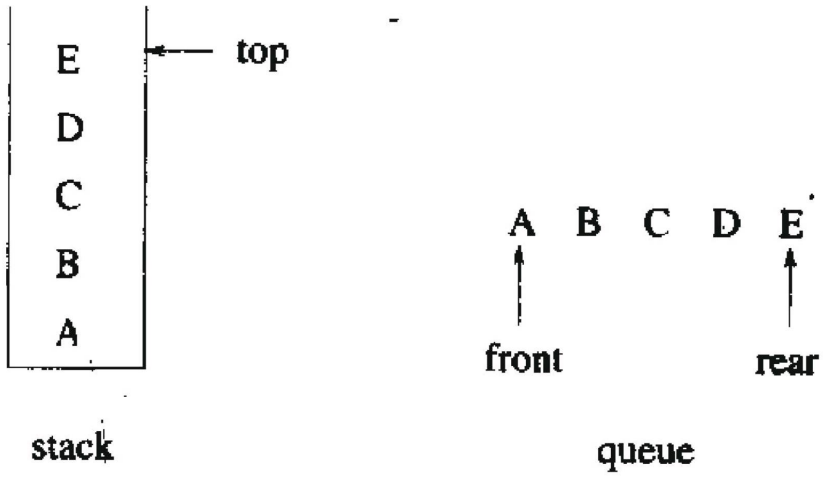


Fig.(1) : Example of a stack and queue

The simplest way to represent a stack is by using a one-dimensional array, say stack [0 : n – 1], where n is the maximum number of allowable entires. The first or bottom element in the stack is stored at stack[0], the second at stack[1], and the ith at stack[i – 1]. Associated with the array is a variable, typically called top, which points to the top element in the stack. To test whether the stack is empty, we ask "if (top < 0)". If not, the topmost element is at stack[top]. Checking whether the stack is full can be done by asking "if (top ≥ n – 1)". Two more substantial operations are inserting and deleting elements. The corresponding algorithms are Add and Delete.

```
Algorithm Add(item)
// Push an element onto the stack. Return true if successful;
// else return false. item is used as an input.
{
    if (top ≥ n – 1) then
    {
        write ("Stack is full!"); return false;
    }
    else
    {
        top = top + 1; stack[top] = item; return true;
    }
}
```

```
Algorithm Delete(item)
// Pop the top element from the stack. Return true if successful
// else return false. item is used as an output.
{
    if (top < 0) then
    {
        write ("Stack is empty!"); return false;
    }
    else
    {
        item = stack[top]; top = top – 1; return true;
    }
}
```

RG PUBLICATION

Q.2.(b) Define the time complexity. Explain asymptotic notation. (7)

Ans. Time complexity : The time complexity of an algorithm is the amount of computer time required by an algorithm to run to completion.

Asymptotic Notation : Asymptotic notation is a shortest way to represent the time complexity.

Various notations such as Ω, Θ, and O are called asymptotic notations.

Big Oh Notation : The Big oh notation is denoted by 'O'. It is a method of representing the upper bound of algorithm's running time. Using big Oh notation we can give longest amount of time taken by the algorithm to complete.

Definition: Let F(n) and g(n) be two non-negative function. Let n_0 and constant c are two integers such that n_0 denotes some value of input and $n > n_0$. Similarly c is some constant such that $c > 0$. We can write

$$F(n) \le c * g(\mathrm{n})$$

then F(n) is big Oh of g(n). It is also denoted as F(n) $\in$ O(g(n)). In other words F(n) is less than g(n) is g(n) is multiple of some constant c.

Example : Consider function F(n) = $2n + 2$ and $g(n) = n^2$. Then we have to find some constant c,so that $F(n) \le c * g(\mathrm{n})$. As F(n) =$2n + 2$ and $g(n) = n^2$ then we find c for $n = 1$ then

$$F(n) = 2n + 2$$
$$= 2(1) + 2$$
$$F(n) = 4$$

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

i.e. F(n) > $g(n)$

If $n = 2$ then,

$$F(n) = 2(2) + 2 = 6$$
$$g(n) = (2)^2 = 4$$

i.e F(n) > cg(n)

If $n = 3$, then

$$F(n) = 2(3) + 2 = 8$$
$$g(n) = (3)^2 = 9$$

i.e F(n) < g(n) is true

Hence we can conclude that for $n > 2$, we obtain

$$F(n) < g(n)$$

Thus always upper bound of existing time is obtained by big Oh notation.

Omega Notation : Omega notation is denoted by 'Ω'. This notation is used to represent the lower bound of algorithms running time. Using omega notation we can denote shortest amount of time taken by algorithm.

Definition : A function F(n) is said to be in Ω (g(n)) if F(n) is bounded below by some positive constant multiple of g(n) such that

$$F(n) \ge c * g(n) \text{ For all } n \ge n_o$$

It is denoted as F(n) $\in$ Ω ($g(n)$).

Example:

Consider F(n) = $2n^2 + 5$ and g(n) = 7n

Then if n = O

$F(n) = 2(0)^2 + 5 = 5$

$g(n) = 7(0) = 0$

i.e F(n) > g(n)

But if $n = 1$

$F(n) = 2(1)^2 + 5 = 7$

$g(n) = 7(1) = 7$

i.e F(n) = g(n)

If $n = 3$, then

$F(n) = 2(3)^2 + 5 = 23$

$g(n) = 7(3) = 21$

i.e $F(n) > g(n)$

Thus For $n>3$ we fet $F(n) > c * g(n)$

It can be represented as $2n^2 + 5 \in \Omega(n)$

Similarly any $n^3 \in \Omega (n)^2$

Θ Notation : The theta notation is denoted by Θ. By this method the running time is between upper bound and lower bound.

Definition : Let $F(n)$ and $g(n)$ be two non-negative functions. There are two positive constants namely c_1 and c_2 such that

$$c_1 \leq g(n)\ c_2 \leq g(n)$$

Then we can say that

$$F(n) \in \Theta\ (g(n))$$

Example : If $F(n) = 2n + 8$ and $g(n) = 5n$

Where $n \geq 2$

Similarly $F(n) = 2n + 8$

$g(n) = 7n$

i.e $5n < 2n + 8 < 7n$ For $n \geq 2$

Here $c_1 = 5$ and $c_2 = 7$ with $n_0 = 2$

The theta notation is more precise with both big Oh and omega notation.

Q.3.(a) Explain the procedure of Quicksort with an example. Also analyze it in best, average and worst case. (10)

Ans. Quick sort : Quick sort is a divide-and-conquer algorithm. It works by selecting a pivot element from the array and partioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively. This can be done in-place, requiring small additional amount of memory to perform the sorting.

Quick sort is a comparison sort.

It rearranges the array elements in such a way that all elements smaller than the picked pivot element move to left side of pivot, and all greater elements move to right side.

Worst case : When the array is in descending order it has to be arranged/sorted in ascending order.

Complexity : $O(n^2)$.

Average case : Complexity : O(n log n)

Best case : Best case is when the array is already sorted.

Complexity : θ(n log n)

Quick sort Algorithm :

(1) if a [low] < a[R]
 then R = R – 1

(2) if a [low] > a[R]
 then swap a[low], a[R]

(3) if a [low] < a[L]
 then, L = L + 1

(4) if a [low] < a[L]
 then swap a[Low], a[L]

Low → Lower index of array
L → a[0]
R → a[n – 1] upper band

Pivot → $\boxed{\begin{array}{c} Low \\ L \\ R \end{array}}$

Quick Sort : Unsorted array

a =

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
7	19	6	9	22	21	4
Low L						R

a[L] = a[Low] = a[0], a[R] = a[6]
a[Low] > a[R]
then swap a[Low], a[R]

a =

4	19	6	9	22	21	7

a[L] = a[0], a[Low] = a[R] = a[6]
a[Low] > a[L] then, L = L + 1

a =

4	19	6	9	22	21	7

a[L] = a[1], a[Low] = a[R] = a[6]
a[Low] < a[L}, swap a[Low], a[L]

a =

4	7	6	9	22	21	19

a[L] = a[1] = a[Low], a[R] = a[6]
a[Low] < a[R] the, R = R – 1

a =

4	7	6	9	22	21	19

a[Low] = a[L] = a[1], 0[R] = a[5]
a[Low] < a[R] then, R = R – 1

a =

4	7	6	9	22	21	19

a[Low] = a[L] = a[1], a[R] = a[4]
a[Low] < a[R] then, R = R – 1

a =

4	7	6	9	22	21	19

a[Low] = a[L] = a[1], a[R] = a[3]
a[Low] < a[R] then, R = R – 1

RG PUBLICATION

a = | 4 | 7 | 6 | 9 | 22 | 21 | 19 |

a[Low] = a[L] = a[1], a[R] = a[2]
a[Low] > a[R], swap a[Low], a[R]

a = | 4 | 6 | 7 | 9 | 22 | 21 | 19 |

a[L] = a[1], a[Low] = a[R] = a[2]
a[Low] > a[L] then L = L + 1

a = | 4 | 6 | 7 | 9 | 22 | 21 | 19 |

a[Low] = a[L] = a[R] = a[2]
a[2] = 7
pivot element = 7

Sorted array :

a = | 4 | 6 | 7 | 9 | 19 | 21 | 22 |

After finding pivot element, same process applied on left subarray and right subarray until we get the sorted array. In each pass we get a pivot element and pivot element placed in its proper position hence proved that quick sort is recursive in nature.

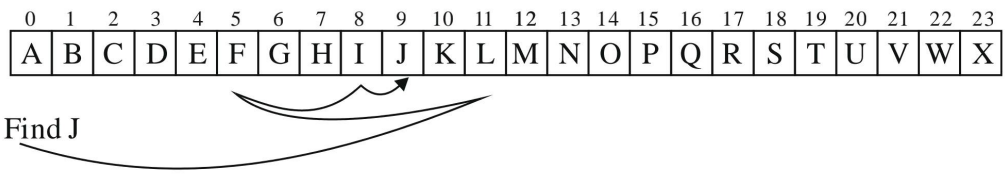
Q.3.(b) Explain Binary Search with example. What is the complexity of binary search ? (5)

Ans. Binary Search : Search a sorted array by repreatedly dividing the search interval in half. Begin with an interval convering the whole array. If the value of the search key is less than the item in the middle of the internal, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

It cuts down your search to half as soon as you find middle of a sorted list.

Binary search performs ordering comparisons.

Binary search to find the element "J" in a given sorted list from A–X.



Algorithm : BINARY (DATA, UB, LB, ITEM, LOC)

Here Data is a sorted array with lower Bound LB and upper bound UB and ITEM is a given item of information. The variable BEG, END and MID denote, the beginning, end and middle location respectively of a segment of elements of DATA. This algorithm finds the location loc of ITEM is DATA or sets LOC = NULL.

(1) Set BEG = LB, END = UB and MID = INT (BEG + END)

(2) Repeat Step 3 & 4

while BEG ≤ END and DATA[MID]≠ ITEM

(3) If ITEM < DATA[MID], then :
Set END = MID – 1
ElSe :
Set BEG = MID + 1

(4) Set MID = INT(BEG + END)/2}

(5) If DATA[MID] = ITEM then :
Set Loc = MID
ElSe :
Set Loc = NULL

(6) EXIT.

Section – B

Q.4.(a) Explain 0/1 Knapsack. (10)

Solve using 0/1 knapsack with capacity 20:

Objects	OBJ1	OBJ2	OBJ3
Profit	25	24	15
Weight	18	15	10

Ans. 0/1 Knapsack : Knapsack is basically means bag. A bag of given capacity.

The Knapsack problem where we have to pack the knapsack with maximum value in such a manner that the total weight of the items should not be greater than the capacity of the knapsack.

Knapsack problem is further divided :

(i) Fractional knapsack

(ii) 0/1 Knapsack

0/1 Knapsack problem : In this item cannot be broken which means their should take the item as a whole or should leave it. That's why it is called 0/1 Knapsack problem.

– Each item is taken or not taken

– Cannot take a fractional amount of an item taken or take an item more than once.

Solving the Given problem using Greedy.

Approach :

Objects	Profit	Weight
Obj 1	25	18
Obj 2	24	15
Obj 3	15	10

As we know, the capacity of the knapsack is 20. So we have to full the knapsack with maximum weight of 20 and with that item which have maximum profit also.

Since Obj 1 has the maximum profit among other objects. Therefore, we will pick Obj 1 firstly.

Total capacity = 20

Object 1 weight = 18

Remaining capcity = 20 – 18 = 2

Profit of the knapsack =25

Since, Obj 2 & Obj 3 has more weight than the remaining knapsack capacity. Therefore, they will not be picked by the knapsacker.

Now, knapsack has only one item in it i.e., object 1 having maximum profit of 25 and weight of 18.

Q.4.(b) Explain Greedy algorithm. Write its applications. (5)

Ans. Greedy Algorithm : It is a simple, intuitive algorithm that is used in optimization problems. The algorithm makes the optimal choice at each step as it attempts to find the overall optimal way to solve the entire problem.

Greedy Algorithm are quite successfull in some problems, such as Huffman encoding which is used to compress data.

However, in many problems, a greedy strategy does not produce an optimal solution because they do not consider all the data.

Application of Greedy Algorithm :

(i) Greedy approach is used in DijKstra's Algorithm to find the shortest path between a & b.

(ii) It is used in Huffman coding.

(iii) Prim's Minimal spanning Tree.

(iv) Travelling salesman Problem.

(v) Graph Map colouring.

(vi) Knapsack problem.

(vii) Job Scheduling problem.

RG PUBLICATION

Q.5.(a) Define Dynamic Programming. Explain Travelling Sales man problem by taking suitable example. (10)

Ans. Dynamic Programming : Dynamic programming is also used is optimization problem, like divide and conquer method dynamic programing solve problem by combining the solution of subproblems.

DP Algo solves each sub-problems just our and these saves its answer is a table there by avoiding the work of re-computing the answer array time.

Dynamic programing approach : (i) Characterize the structure of an optimal solution.

(ii) Precursively define the value of an optimal solution.

(iii) Compute the value of an optimal solution.

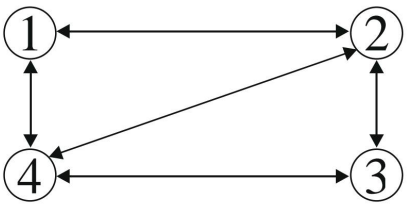
(iv) Construct an optimal solution from the compiled info.

Application of Δ P : (i) Matrix chain multiplication.

(ii) Longest common sub by uses.

(iii) Travelling salesman problem.

Travelling salesman problem :



$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 5 & 7 & 8 \\ 2 & 2 & 0 & 3 & 5 \\ 3 & 8 & 9 & 0 & 10 \\ 4 & 12 & 13 & 11 & 0 \end{array}$$

Calculate g (1,{2,3,4}) = ?

Length of optimal sales person tour :

$$g(1, v-\{1\}) = \min_{2\le k\le n}\{c_{1k} + g(k, v-\{1,k\})\} \quad ...(1)$$

$$g(i, s) = \min_{j\in s}\{c_{ij} + g(j, S-\{j\})\} \quad ...(2)$$

From Equation (2)

Q = 1

$$g(i, \varphi) = c_{i1},\ 1 \le i \le n$$

$$g(2, \varphi) = C_{21} = 2$$

$$g(3, \varphi) = C_{31} = 8$$

$$g(4, \varphi) = C_{41} = 12$$

$$g(2, \{3\}) = \min_{3\in\{3\}}\{C_{23} + g(3,\{3\}-\{3\})\}$$

$$= \min_{3\in\{3\}}\{C_{23} + g(3,\varphi)\}$$

$$= C_{23} + g(3, \varphi)$$

$$= C_{23} + C_{31}$$

$$= 3 + 8 = 11$$

$$g(2, \{4\}) = \min_{4\in\{4\}}\{C_{24} + g(4,\{4\}-\{4\})\}$$

$$= \min_{4\in\{4\}}\{C_{24} + g(4,\varphi)\}$$

$$= C_{24} + g(4, \varphi)$$

$$= C_{24} + C_{41}$$

$$= 5 + 12 = 17$$

$$g(3, \{2\}) = C_{32} + g(2, \varphi)$$

$$= 9 + 2 = 11$$

$$g(3, \{4\}) = C_{34} + g(4, \varphi)$$

$$= C_{34} + C_{41}$$

$$= 10 + 12 = 22$$

$$g(4, \{2\}) = C_{42} + g(2, \varphi)$$

$$= C_{42} + C_{21}$$

$$= 13 + 2 = 15$$

$$g(4, \{3\}) = C_{43} + g(3, \varphi)$$

$$= C_{43} + C_{31}$$

$$= 11 + 8 = 19$$

KG PUBLICATION

$$g(2, \{3, 4\}) = \min\{C_{23} + g(3,\{3,4\}-\{3\})\}, C_{24} + g(4,\{3,4\}-\{4\})\}$$
$$= \min\{C_{23} + g(3, \{4\}), C_{24} + g(4, \{3\})\}$$
$$= \min\{3 + 22 + 5 + 19\}$$
$$= \min\{25, 24\}$$
$$g(2, \{3, 4\}) = 24$$
$$g(3, \{2, 4\}) = \min\{C_{32} + g(2, \{2, 4\} - \{2\}), C_{34} + g(4, \{2, 4\} - \{4\})\}$$
$$= \min\{C_{32} + g\{2, 4\}), C_{34} + g(4, \{2\})\}$$
$$= \min\{9 + 17, 10 + 15\}$$
$$= \min\{26, 25\}$$
$$g(3, \{2, 4\}) = 25$$
$$g(4, \{2, 3\}) = \min\{C_{42} + g(2, \{2, 3\} - \{2\}), C_{43} + g(3, \{2, 3\} - \{3\})\}$$
$$= \min\{C_{42} + g(2, \{3\}), C_{43} + g(3, \{2\})$$
$$= \min\{13 + 11, 11 + 11\}$$
$$= \min\{24, 22\}$$
$$g(4, \{2, 3\}) = 22$$
$$g(1, \{2, 3, 4\}) = \min\{C_{12} + g(2,\{3, 4\}), C_{13} + g(3,\{2, 4\}), C_{14} + g(4, \{2, 3\})\}$$
$$= \min\{29, 32, 30\} = 29$$
$$\text{Optimal tour length} = 29$$

Q.5.(b) Write a short note on fractional knapsack problem. (5)

Ans. Fractions of items can be taken rather than having to make binary (0-1) choices for each item.

Fractional Knapsack Problem can be solvable by greedy strategy whereas 0 - 1 problem is not.

Fractional knapsack problem : given a set of items each with the weight and the value determines a subset of items to include a collection so that the total weight is less than or equal to given limit and the total value is as large as possible

Fractional knapsack in this case item can be broken into smaller pieces hence the thief can select fraction of items

knapsack problem is a combinatorial optimization problem it appears as sub problem it may more complex mathematical models of real world one general approach to different problems is to identify the most restrictive constraint no other way to solve problem

Application of knapsack problem

(1) finding the least wasteful way to cut raw material

(2) portfolio optimization

(3) cutting stock problem

Problem scenario : A thief is robbing a store and can carry a maximal weight of W into his knapsack there are 'N' items available in the store and weight of ith items is wi is profit is pi then the thief will only select the item in such a way in which the thief will get maximum profit Hence objective of the thief is maximize the profit.

Fractional Knapsack : In this case, items can be broken into smaller pieces, hence the thief can select fractions of items.

According to the problem statement,

- There are **n** items in the store
- Weight of **i**th item wi > 0wi > 0
- Profit for **i**th item pi > 0pi > 0 and
- Capacity of the Knapsack is **W**

The thief may take only a fraction x_i of **i**th item.

$$0 \leq xi \leq 10 \leq xi \leq 1$$

The **i**th item contributes the weight xi.wixi.wi to the total weight in the knapsack and profit xi.pixi.pi to the total profit.

Hence, the objective of this algorithm is to

$$\text{maximize } \Sigma n=1n(xi.pi)\text{maximize } \Sigma n=1n(xi.pi)$$

subject to constraint,

$$\Sigma n=1n(xi.wi) \leq \Sigma W \Sigma n=1n(xi.wi) \leq W$$

It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit.

Thus, an optimal solution can be obtained by

$$\Sigma n=1n(xi.wi)=W\Sigma n=1n(xi.wi)=W$$

Section – C

RG PUBLICATION

Q.6.What is backtracking? Solve N-queens problem with back tracking. (15)

Ans. Backtracking : Backtracking is a refinement of the brute force approach, which systematically searches for a solution to a problem among all available options. It does so by assuming that the solutions are represented by vectors (v_1, ..., v_m) of values and by traversing, in a depth first manner, the domains of the vectors until the solutions are found.

Backtracking is used to find all possible solutions available to the problem. It traverse tree by DFS(Depth First Search). It realizes that it has made a bad choice & undoes the last choice by backing up. It searches the state space tree until it found a solution. It involves feasibility function.

N-Queens Problem : N - Queens problem is to place n - queens in such a manner on an n x n chessboard that no queens attack each other by being in the same row, column or diagonal.

It can be seen that for n =1, the problem has a trivial solution, and no solution exists for n =2 and n =3. So first we will consider the 4 queens problem and then generate it to n - queens problem.

Given a 4 x 4 chessboard and number the rows and column of the chessboard 1 through 4.

	1	2	3	4
1				
2				
3				
4				

4×4 chessboard

Since, we have to place 4 queens such as q_1 q_2 q_3 and q_4 on the chessboard, such that no two queens attack each other. In such a conditional each queen must be placed on a different row, i.e., we put queen "i" on row "i."

Now, we place queen q_1 in the very first acceptable position (1, 1). Next, we put queen q_2 so that both these queens do not attack each other. We find that if we place q_2 in column 1 and 2, then the dead end is encountered. Thus the first acceptable position for q_2 in column 3, i.e. (2, 3) but then no position is left for placing queen 'q_3' safely. So we backtrack one step and place the queen 'q_2' in (2, 4), the next best possible solution. Then we obtain the position for placing 'q_3' which is (3, 2). But later this position also leads to a dead end, and no place is found where 'q_4' can be placed safely. Then we have to backtrack till 'q_1' and place it to (1, 2) and then all other queens are placed safely by moving q_2 to (2, 4), q_3 to (3, 1) and q_4 to (4, 3). That is, we get the solution (2, 4, 1, 3). This is one possible solution for the 4-queens problem. For another possible solution, the whole method is repeated for all partial solutions. The other solutions for 4 - queens problems is (3, 1, 4, 2) i.e.

	1	2	3	4
1			q_1	
2	q_2			
3				q_3
4		q_4		

The implicit tree for 4 - queen problem for a solution (2, 4, 1, 3) is as follows:

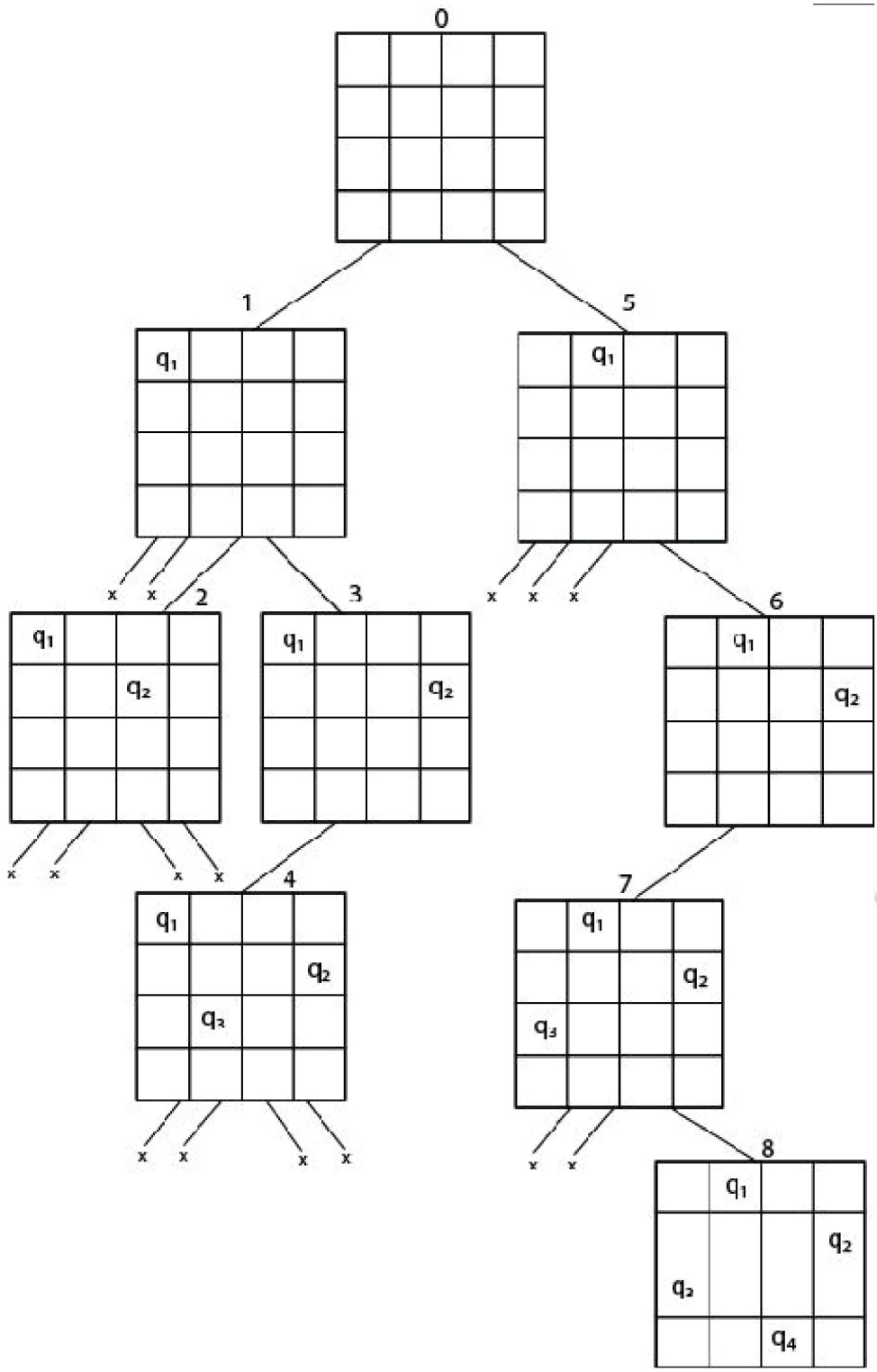
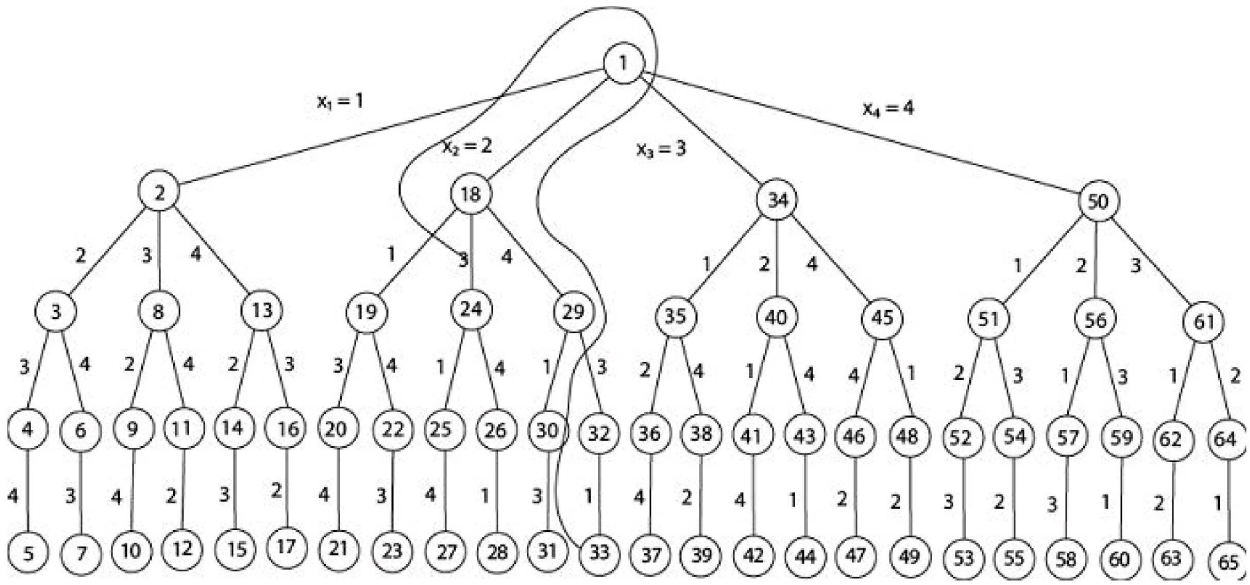


Fig shows the complete state space for 4 - queens problem. But we can use backtracking method to generate the necessary node and stop if the next node violates the rule, i.e., if two queens are attacking.

It can be seen that all the solutions to the 4 queens problem can be represented as 4 - tuples (x_1, x_2, x_3, x_4) where x_i represents the column on which queen "q_i" is placed.



4 - Queens solution space with nodes numbered in DFS

One possible solution for 8 queens problem is shown in fig:

	1	2	3	4	5	6	7	8
1				q_1				
2						q_2		
3								q_3
4		q_4						
5							q_5	
6	q_6							
7			q_7					
8					q_8			

1. Thus, the solution **for** 8 -queen problem **for** (4, 6, 8, 2, 7, 1, 3, 5).
2. If two queens are placed at position (i, j) and (k, l).
3. Then they are on same diagonal only **if** (i - j) = k - l or i + j = k + l.
4. The first equation implies that j - l = i - k.
5. The second equation implies that j - l = k - i.
6. Therefore, two queens lie on the duplicate diagonal **if** and only **if** |j-l|=|i-k|

Place (k, i) returns a Boolean value that is true if the kth queen can be placed in column i. It tests both whether i is distinct from all previous costs x_1, x_2,....x_{k-1} and whether there is no other queen on the same diagonal.

Using place, we give a precise solution to then n- queens problem.

```
Place (k, i)
  {
   For j    !  1 to k - 1
    do if (x [j] = i)
     or (Abs x [j]) - i) = (Abs (j - k))
   then return false;
    return true;
  }
```

Place (k, i) return true if a queen can be placed in the kth row and ith column otherwise return is false.

x [] is a global array whose final k - 1 values have been set. Abs (r) returns the absolute value of r.

```
N - Queens (k, n)
{
  For i ← 1 to n
      do if Place (k, i) then
  {
    x [k]    ←  i;
    if (k ==n) then
      write (x [1....n));
    else
    N - Queens (k + 1, n);
  }
}
```

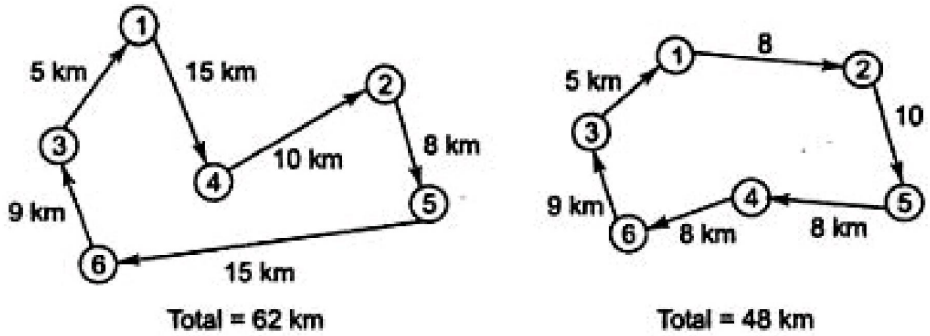
Q.7. What is branch and bound method? Solve the traveling salesperson problem with branch and bound method by taking a suitable example. (15)

Ans. Branch and bound method : Branch-and-bound (B & B) is an algorithmic technique to find the optimal solution by keeping the best solution found so far. If a partial solution cannot improve on the best, it is discarded.

B & B can be applied to an assignment problem by assigning *n* persons to *n* jobs so that the total cost of assignment is a minimum. Each person can be assigned exactly one job.

Traveling-Salesman Problem : Travelling Salesman Problem (TSP) requires that we find the shortest path visiting each of a given set of cities once and returning to the starting point. In other words, the problem is to find, in an undirected graph with weights on the edges, a tour the sum of whose edge-weights is a minimum. A tour is a simple cycle that includes all the vertices and is often called a hamiltonian cycle. TSP is a famous example of a problem which can be solved by a B & B.

Problem : Given a complete undirected graph $G = (V, E)$ that has non-negative integer cost $C(u, v)$ associated with each edge (u, v) in E, the problem is to find a hamiltonian cycle (tour) of G with minimum cost.



A salesperson starts from the city 1 and has to visit six cities (1 through 6) and must come back to the starting city *i.e.*, 1. The first route (left side) $1 \rightarrow 4 \rightarrow 2 \rightarrow 5 \rightarrow 6 \rightarrow 3 \rightarrow 1$ with the total length of 62 km, is a relevant selection but is not the best solution. The second route (right side) $1 \rightarrow 2 \rightarrow 5 \rightarrow 4 \rightarrow 6 \rightarrow 3 \rightarrow 1$ represents to the must better solution as the total distance, 48 km, is less than for the first route.

Suppose $C(A)$ denoted the total cost of the edges in the subset A subset of E i.e.,

$$C(A) = \sum_{u,v in A} C(u,v)$$

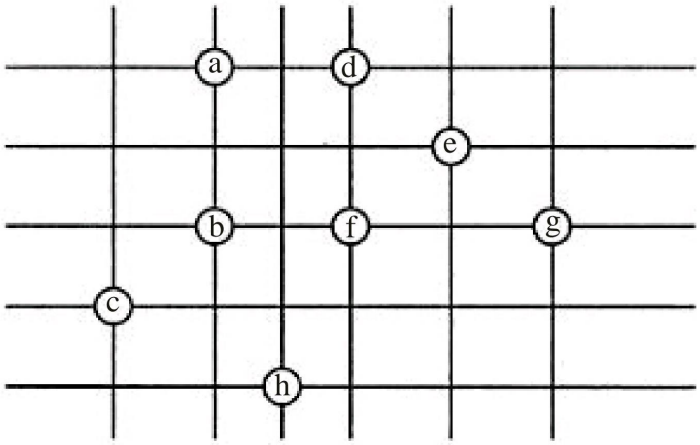
Moreover, the cost function satisfies the triangle inequality, that is, for all vertices u, v, w in v, w have $C(u, w) \leq C(u, v) + C(v, w)$.

Note that the TSP problem is NP-complete even if we require that the cost function satisfies the triangle inequality. This means that it is unlikely that we can find a polynomial time algorithm for TSP.

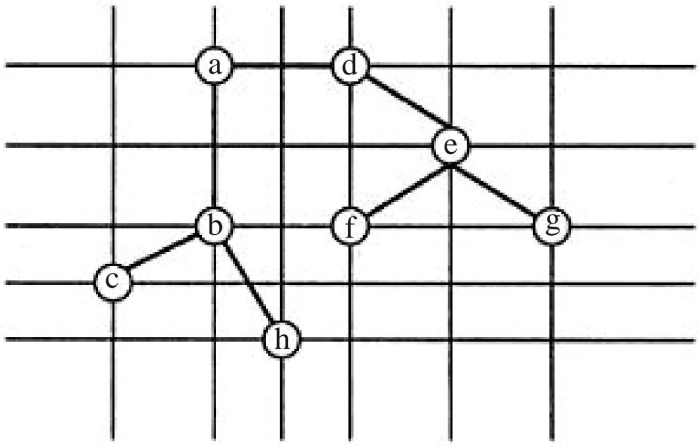
TSP with the triangle-inequality : When the cost function satisfies the triangle-inequality, we can design an approximate algorithm for TSP that returns a tour whose cost is not more than twice the cost of an optimal tour.

Outline of an APPROX-TSP-TOUR : First, compute a MST (minimum spanning tree) whose weight is a lower bound on the length of an optimal TSP tour. Then, use MST to build a tour whose cost is no more than twice that of MST's weight as long as the cost function satisfies triangle inequality.

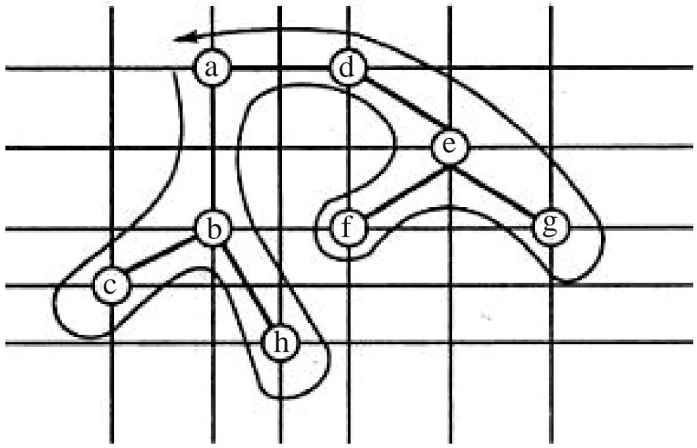
Operation of APPROX-TSP-TOUR : Let root *r* be a in the following given set of points (graph).



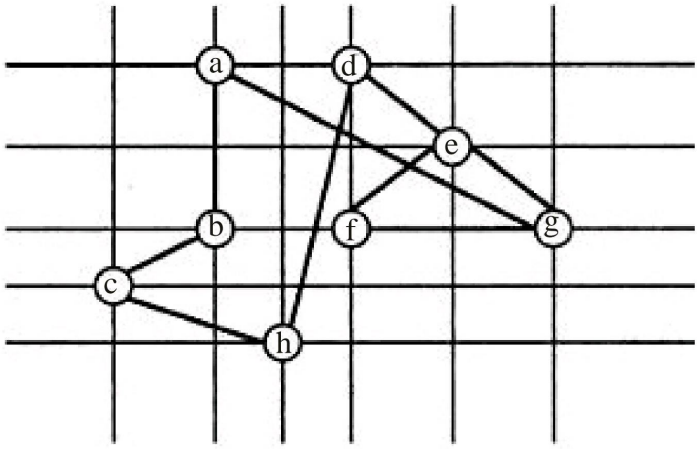
Construct MST from root a using MST-PRIM (*G*, *C*, *r*)



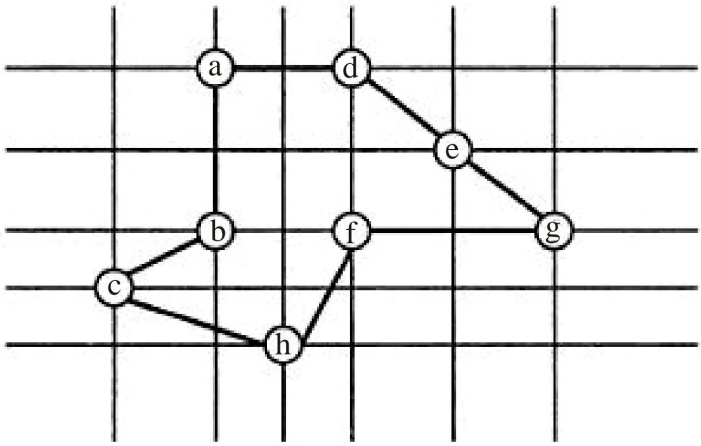
List vertices visited in pre-order walk $L = [a, b, c, d, e, f, g]$



Return hamiltonian cycle



Optimum TSP tour for a given problem (graph) would be



which is about 23 % shorter.

APPROX-TSX-TOUR (G, C)

(i) Select a vertex $r \in V$ [G] to be a root vertex.

(ii) Compute a minimum spanning tree T for G from root r using MST-PRIM (G, C, r).

(iii) Let L be the list of vertices visited in a pre-order tree walk of T.

(iv) Return the hamiltonian cycle H that visits the vertices in the order L.

Theorem : APPROX-TSP-TOUR is a polynomial-time 2-approximation algorithm for TSP with triangle inequality.

Proof 1. We have already shwon that APPROX-TSP-TOUR-time.

2. Let H denote the optimal tour. Observe that a TSP with one edge removed is a spanning tree (not necessarily MST). It implies that the weight of the MST in lower bound on the cost of an optimal tour.

$$C(T) = C(H)$$

A "full" walk, W, traverse every edge of MST, T exactly twice, that is,

$$C(W) = 2C(T)$$

which means

$$C(W) \le 2C(H)$$

and we have

$$\frac{C(W)}{C(H)} \le P(n) = 2$$

That is, the cost of walk, $C(W)$, of the solution produced by the algorithm is within a factor of $P(n) = 2$ of the cost $C(H)$ of an optimal solution.

Section – D

Q.8.(a) What is the relationship among P, NP and NP complete problems ? Show with the help of a diagram. (7)

Ans. P problem:P problem solved and verified in Polynomial Time the set of problems that can be solved in polynomial time using deterministic algorithm

Or

If a problem can be solved in Polynomial Time

NP Problem : NP stands for Non deterministic polynomial time.The set of problems for which "yes" answer can be verified in polynomial time

NP problems is divided into two classes :

NP-complete :A problem that is NP complete can be solved in PT iff all other NPC problems can also be solved in PT

NP hard : If an NP hard problem can be solved in PT then all NP complete problems can be solved in PT

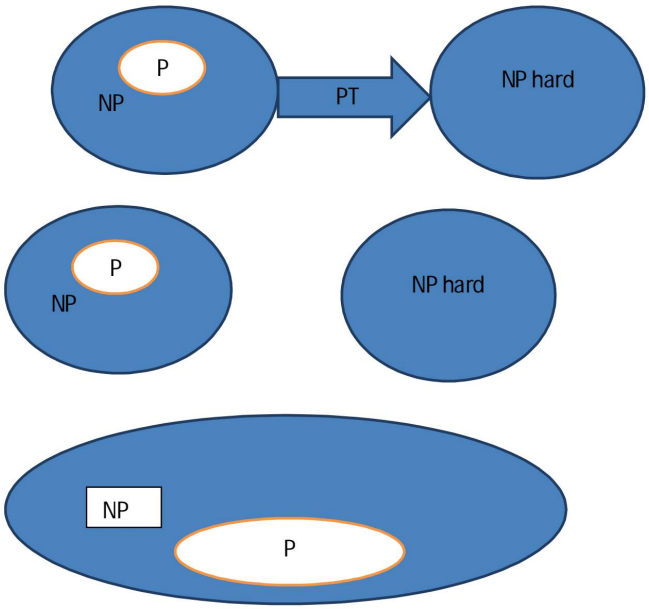
All NPC problems are NP hard but some NP-hard problems are known not to be NP-complete

NPC is a subset of NP hard

P is a subset of NP problem

The set of problems that can not solved in polynomial time if we are given a solution for a problem we may be able to check whether the solution is correct very efficiently

RG PUBLICATION



NPC(NP complete): intersection of NP &NPhard is called NPC

NPC problems: no efficient algorithm for aan NP complete problems has ever been found but nobody has been able to prove that such as algorithm does not exist

NPC :Decision Problem

NP hard : Optimization Problem

Examples

(1) Factorization

(2) Polynomial identities Testing

(3) Graph Coloring Problem

NP COMPLETE

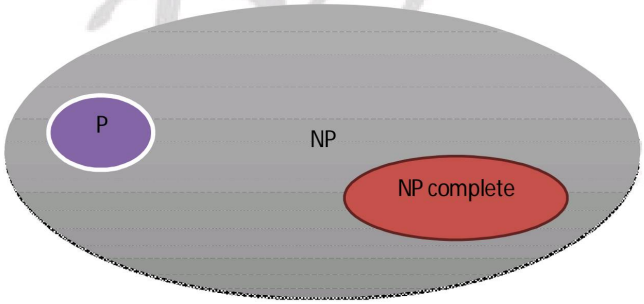
Condition

(1) If it is in NP

(2) If it can be solved in polynomial time

Then any problem in NP can be solved in polynomial time

NP problem considered as hard problem



P=NP ?means not solvable till

P=NP used in information security then online security attack

But everything become more efficient

Transportation problem,DNA structure problem solved easily

P $\neq$ NP means uou can prove that there are some problem that can never be

Q.8.(b) Differentiate between NP hard and NP complete problem. (8)

Ans. NP-Hard Problem : A Problem X is NP-Hard if there is an NP-Complete problem Y, such that Y is reducible to X in polynomial time. NP-Hard problems are as hard as NP-Complete problems. NP-Hard Probem need not be in NP class.

NP-Complete Problem : A problem X is NP-Complete if there is an NP problem Y, such that Y is reducible to X in polynomial time. NP-Complete problems are as hard as NP

problems. A problem is NP-Complete if it is a part of both NP and NP-Hard Problem. A non-deterministic Turing machine can solve NP-Complete problem in polynomial time.

NP-Hard	NP-Complete
NP-Hard problems(say X) can be solved if and only if there is a NP-Complete problem(say Y) that can be reducible into X in polynomial time.	NP-Complete problems can be solved by a non-deterministic Algorithm/Turing Machine in polynomial time.
To solve this problem, do not have to be in NP.	To solve this problem, it must be both NP and NP-hard problems.
Do not have to be a Decision problem.	It is exclusively a Decision problem.
Example : Halting problem, Vertex cover problem, Circuit-satisfiability problem, etc.	Example : Determine whether a graph has a Hamiltonian cycle, Determine whether a Boolean formula is satisfiable or not, etc.

Q.9. Explain NP hard and NP completeness of SAT problem. (15)

Ans. The satisfiability problem : A formula φ is *satisfiable* if there exists an assignment of values to its variables that makes φ true.

For example, formula

$$(\neg x \vee y) \wedge$$
$$(\neg y \vee z) \wedge$$
$$(x \vee \neg z \wedge y)$$

is satisfiable. Choose

$$x = \text{false}$$
$$y = \text{true}$$
$$z = \text{true}$$

The satisfiability problem, usually called SAT, is the following language.

$$\text{SAT} = \{ \varphi \mid \varphi \text{ is a satisfiable clausal formula} \}.$$

Thought of as a computational problem, the input to SAT is a clausal formula ö and the problem is to determine whether ö is satisfiable.

SAT is NP-complete : Cook and Levin independently proved that SAT is NP-complete.

Proof. There are two parts to the proof because there are two parts to the definition of NP-completeness.

First, you must show that SAT is in NP.

Then you must show that, for every problem X in NP, X d”$_p$ SAT.

The first part is by far the easiest. The satisfiablity problem can be expressed as a test for existence.

φ is satisfiable $\Leftrightarrow$ there exists an assignment A of truth values to the variables in φ so that assignment A makes φ true.

So a nondeterministic algorithm for SAT goes like this.

Evidence for φ. An assignment of truth values to the variables that occur in φ.

The evidence is accepted if : the assignment makes φ true.

The second part ($X \leq_p$ SAT for every X in NP) is considerably more difficult.

First, the Cook theorem tells us that, if the conjecture that P $\neq$ NP is correct, then SAT is not in P.It also makes it unnecessary to do more generic reductions in proofs of NP-completeness. Suppose that you want to show that some language, W, is NP-complete.

First, of course, you show that W is in NP. So you find a polynomial-time nondeterministic algorithm for W. Then you must show that $X <_p W$ for every X in NP.But that can be done by showing that SAT $\leq_p W$.

Then, since $X \leq_p$ SAT and SAT $\leq_p W$, $X \leq_p W$, for every X in NP.

RG PUBLICATION