END TERM EXAMINATION [FEB. 2023]
FIFTH SEMESTER [B.TECH]
ALGORITHMS DESIGN AND ANALYSIS [ETCS-301]

Time: 3 Hrs. Max. Marks: 75

Note: Attempt any five questions in all including Q. No. 1 which is compulsory.

Q.1. Attempt all questions: (5 × 5 = 25)

Q.1 (a) Define time complexity and space complexity. Write an algorithm for adding n natural numbers and find the space required by that algorithm.

Q.1 (b) Define Big 'Oh' notation. Formulate the order of growth. Compare the order of growth n! and 2n. Differentiate between Best, average and worst case efficiency.

Q.1 (c) Differentiate divide and conquer and dynamic programming.

Q.1 (d) Explain dynamic programming method of problem solving. What type of problems can be solved by dynamic programming?

Q.1 (e) Determine an LCS of <1,0,0,1,0,1,0,1> and <0,1,0,1,1,0,1,1,0>

Q.2 (a) Discuss the concepts of asymptotic notations and its properties. (4)

Q.2 (b) Analyze the order of growth. (4)

$F(n) = 2n2 + 5$ and $g(n) = 7n$. Use the Ω (g(n)) notation.

Q.2 (c) Evaluate the recurrence relations. (4.5)

(i) . $x(n) = x(n - 1) + 5$ for $n > 1$.

(ii) . $X(n) = x(n/3) + 1$ for $n > 1$, $x(1) = 1$. (Solve for $n = 3k$)

Q.3 (a) Which sorting algorithm is best if the list is already sorted? Why? (4)

Q.3 (b) Prove that the average running time of Quick Sort is O(nlog(n)) where n is the number of elements. (4)

Q.3 (c) What are stable algorithms? Which sorting algorithm is stable? Give one example and explain. (4.5)

Q.4 (a) Implement UNION using linked list representation of disjoint sets. (4)

Q.4 (b) Explain the characteristics of problems that can be solved using dynamic programming. (4)

Q.4 (c) Give a control abstraction for Divide and Conquer method. Explain with an example. (4.5)

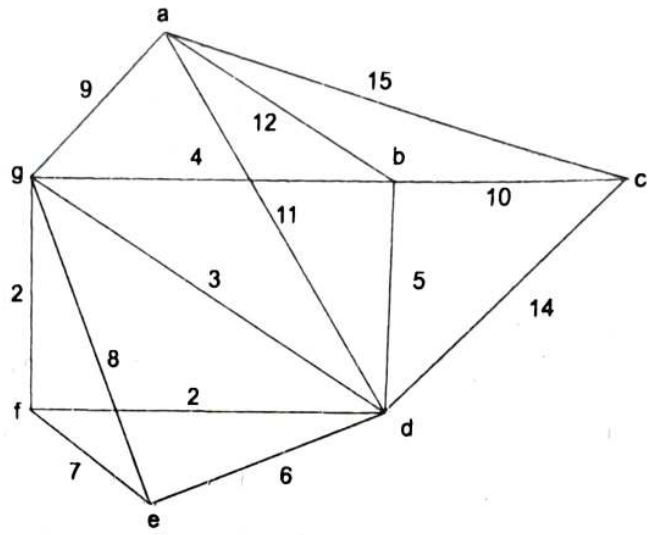
Q.5 (a) Explain the effect of negative weight edges and negative weight cycles on shortest paths. (4)

Q.5 (b) Define strongly connected components. How DFS can be used to find strongly concerted components? (4)

Q.5 (c) Find an optimal paranthesization of a matrix-chain product whose sequence of dimensions is 4x10, 10x3, 3x12, 12x20 (4.5)

Q.6 (a) Write Dijkstra's Single Source Shortest path algorithm. Analyze the complexity. (6)

Q.6 (b) Find minimum spanning tree of the following graph using Prims algorithm and discuss complexity. (6.5)



Q.7 (a) Explain Rabin-karp string matching algorithm, in brief. (6)

Q.7 (b) Find longest common subsequence of following two strings X and Y using any algorithm: (6.5)

X = 'aabdbacdcba'

Y = 'aabddcbac'

Q.8 (a) Differentiate between P, NP, NP-completeness and NP-Hard problems. (4)

Q.8 (b) How a problem is identified as NP complete problem? Give atleast five problems that can be classified as NP complete problems. (4)

Q.8 (c) With examples explain polynomial time reducibility. (4.5)

IMPORTANT QUESTIONS

Q.1. Write note on the optimal binary search tree problems.

Ans. Optimal binary search tree: We are given a sequence $K = \_k_1, k_2, ..., k_n$ of n distinct keys in sorted order (so that $k_1 < k_2 < ... < k_n$), and we wish to build a binary search tree from these keys. For each key k_i, we have a probability pi that a search will be for ki. Some searches may be for values not in K, and so we also have $n + 1$ "dummy keys" $d_0, d_1, d_2, ..., d_n$ representing values not in K. In Particular, d_0 represents all values less than k_1, d_n represents all values greater than k_n and for $i = 1, 2, ..., n - 1$ the dummy key d_i represents all values between k_i and k_{i+1}.

Step 1: The structure of an optimal binary search tree:

To characterize the optimal substructure of optimal binary search trees, we start with observation about subtrees. Consider any subtree of a binary search tree. It must contain keys in a contiguous range $k_i, ..., k_j$, for some $1 \le i \le j \le n$. In addition, a subtree that contains keys $k_i, ..., k_j$ must also have as its leaves the dummy keys $d_{i-1}, ..., d_j$.

If an optimal binary search tree T has a subtree T' containing keys $k_i, ..., k_j$, then this subtree T' must be optimal as well for the subproblem with keys $k_i, ..., k_j$ and dummy keys $d_{i-1}, ..., d_j$. The usual cut-and-paste argument applies. If there were a subtree T'', expected cost is lower than T', then we could cut T' out of T and paste in T'', resulting in a binary search tree of lower expected cost is lower than T' then we could cut T' out of T and paste in T'' resulting in a binary search tree of lower expected cost than T, thus contracting the optimality of T.

Step 2: A recursive solution

We are ready to define the value of an optimal solution recursively. We pick our subproblem domain as finding an optimal binary search tree containing the keys $k_i, ..., k_j$, where $i \ge 1, j \le 1, j \le n$, and $j \ge i - 1$. (It is when $j = i - 1$ that there are no actual keys; we have just the dummy key d_{i-1}) Let us define $e[i, j]$ as the expected cost of searching an optimal binary search tree containing the keys $k_i, ..., k_j$. Ultimately, we wish to compute $e[1, n]$. The easy case occurs when $j = i - 1$. Then we have just the dummy key d_{i-1}. The expected search cost is $e[i, i - 1] = q_{i-1}$. Thus, if k_r is the root of an optimal subtree containing keys $k_i, ..., k_j$ we have $e[i, j] = p_r + (e[i, r - 1] + w(i, r - 1)) + (e[r + 1, j] + w(r + 1, j))$.

Nothing that

$$w(i, j) = w(i, r - 1) + p_r + w(r + 1, j).$$

Step 3: Computing the expected search cost of an optimal binary search tree: At this point, you may have noticed some similarities between our characterizations of optimal binary search trees and matrix-chain multiplication. For both problem domains, our subproblems consist of contiguous index subranges, we store the $e[i, j]$ values in a table $e[1 \_ n + 1, 0 \_ n]$. The first index needs to run to $n + 1$ rather than $n$because in order to have a subtree containing only the dummy key d_n, we will need to compute and store $e[n + 1, n]$. The second index needs to start from 0 because in order to have a subtree containing only the dummy key d_0, we will need to compute and store $e[1, 0]$. We will use only the entries $e[i, j]$ for which $j \ge i - 1$. We also use a table $root[i, j]$, for recording the root of the subtree containing keys $k_i, ..., k_j$. This table uses only the entries for which $1 \le i \le j \le n$.

Algorithms in Pseudocode:

Optimal-BST (p, q, n)

1. for $i \leftarrow 1$ to $n + 1$
2. do $e[i, i - 1] \leftarrow q_{i-1}$
3. $w[i, i - 1] \leftarrow q_{i-1}$
4. for $1 \leftarrow 1$ to n
5. do for $i \leftarrow 1$ to $n - 1 + 1$
6. do $j \leftarrow i + 1 - 1$

7. $e[i, j] \leftarrow \infty$

8. $w[i, j] \leftarrow w[i, j-1] + p_j + q_j$

9. for $r \leftarrow i$ to j

10. do $t \leftarrow e[i, r-1] + e[r+1]$

11. if $t < e[i, j]$

12. Then $e[i, j] \leftarrow t$

13. root $[i, j] \leftarrow t$

14. return e and root

Q.2. What is NP-Completeness? Discuss any five NP-complete problems in detail.

Ans. NP-complete problems are in NP, the set of all decision problems whose solutions can be verified in polynomial time; NP may be equivalently defined as the setof decision problems that can be solved in polynomial time on a non-deterministic Turing machine. A problem p in NP is NP-complete if every other problem in NP can be transformed into p in polynomial time.

NP-complete problems are studied because the ability to quickly verify solutions to a problem (NP) seems to correlate with the ability to quickly solve that problem (P). It is not known whether every problem in NP can be quickly solved-this is called the P versus NP problem. But if any NP-complete problem can be solved quickly, then every problem in NP can, because the definition of an NP-complete problem states that every problem in NP must be quickly reducible to every NP-complete problem (that is, it can be reduced in polynomial time). Because of this, it is often said that NP-complete problems are harder or more difficult than NP problems in general.

A decision problem C is NP-complete if:

1. C "is in NP, and

2. Every problem in NP is reducible to C in polynomial time.

C can be shown to be in NP by demonstrating that a candidate solution to C can be verified in polynomial time.

Note that a problem satisfying condition 2 is said to be NP-hard. whether or not it satisfies condition 1.

A consequence of this definition is that if we had a polynomial time algorithm (on a UTM. or any other Turing-equivalent abstract machine) for C, we could solve all problems in NP in polynomial time.

CLIQUE

We will now use the fact that 3-SAT is NP-complete to prove that a natural graph problem called the Max-Clique problem is NP-complete.

Max-Clique: Given a graph G, find the largest clique (set of nodes such that all pairs in the set are neighbors). Decision problem: "Given G and integer k, does G contain a clique of size ≥ k?"

We will reduce 3-SAT to Max-Clique. Specifically, 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 wecreate one node for every assignment to variables in c that satisfies c. E.g., say we have:

$$F = (x_1 \vee x_2 \vee \bar{x}_4) \wedge (\bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee \bar{x}_3) \wedge \ldots$$

Then in this case we would create nodes likes this:

$(x_1 = 0, x_2 = 0, x_4 = 0)$ $(x_3 = 0, x_4 = 0)$ $(x_2 = 0, x_3 = 0)\ldots$

$(x_1 = 0, x_{2=1,} 1, x_4 = 0)$ $(x_3 = 0, x_4 = 1)$ $(x_2 = 0, x_3 = 1)$

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

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. Specifically, 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, $O(m^2)$ edges). Therefore Max-Clique is NP-complete.

Independent Set

An Independent Set in a graph is a set of nodes no two of which have an edge. E.g., in a 7-cycle, the largest independent set has size 3, and in the graph coloring problem, the set of nodes colored red is an independent set. The Independent Set problem is: given a graph G and an integer k, does G have an independent set of size $\geq k$?

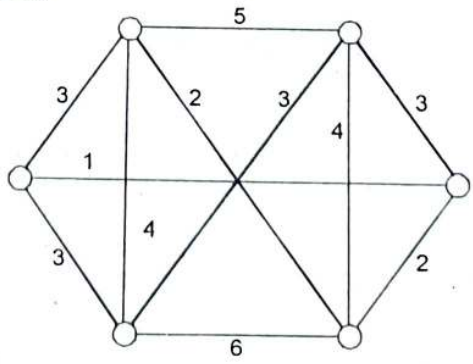
We reduce from Max-Clique. Given an instance (G,k) of the Max-Clique problem, we output the instance (H,k) of the Independent Set problem where H is the complement of G. That is, H has edge (u,v) if G does not have edge (u,v). Then H has an independent set of size k if G has a k-clique.

Vertex Cover

A vertex cover in a graph is a set of nodes such that every edge is incident to at least one of them. For instance, if the graph represents rooms and corridors in a museum, then a vertex cover is a set of rooms we can put security guards in such that every corridor is observed by at least one guard. In this case we want the smallest cover possible. The Vertex Cover problem is: given a graph G and an integer k, does G have a vertex cover of size $\leq$ k?

If *C* is a vertex cover in a graph G with vertex set V , then V – C is an independent set. Also if S is an independent set, then V – S is a vertex cover. So, the reduction from Independent Set to Vertex Cover is very simple: goiven an instance (G,k) for Independent Set, produce the instance (G,n – k) for Vertex Cover, where n = |V|. In other words, to solve the question "is there an independent set of size at least k" just solve the question "is there a vertex cover of size $\leq$ n - k?" So, Vertex Cover is NP-Complete too.

The traveling salesman problem



In the traveling salesman problem (TSP) we are given n vertices 1; :: :; n and all n(n − 1) = 2 distances between them, as well as a budget b. We are asked to find a tour, a cycle that passes through every vertex exactly once, of total cost b or less or to report that no such tour exists. That is, we seek a permutation $T(1),..., T(n)$ of the vertices such that when they are toured in this order the total distance covered is at most b:

$$d_{T(1),T(2)} + d_{T(2),T(3)} + ... + d_{T(n),T(1)} \leq b$$

Notice how we have defined the TSP as a search problem: given an instance, find a tour within the budget (or report that none exists). But why are we expressing the traveling salesman problem in this way, when in reality it is an optimization problem, in which the shortest possible tour is sought? Why dress it up as something else? For a good reason. Our plan in this chapter is to compare and relate problems. The framework of search problems is helpful in this regard, because it encompasses optimization problems like the TSP in addition to true search problems like SAT.

Q.3. Discuss in detail any one algorithm for finding the cost spanning tree.

Ans. Given a connected, undirected graph, a spanning tree of that graph is a subgraph that is a tree and connects all the Vertices together. A single graph can have many different spanning trees. We can also assign a weight to each edge, which is a number representing how unfavorable it is, and use this to assign a weight to a spanning tree by computing the sum of the weights of the edges in that spanning tree. A minimum spanning tree (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 (not necessarily connected) has a minimum spanning forest, which is a union of minimum spanning trees for its connected components.

Possible multiplicity :There may be several minimum spanning trees of the same weight having a minimum number of edges; in particular, if all the edge weights of a given graph are the same, then every spanning tree of that graph is minimum.

If there are n vertices in the graph, then each spanning tree has $n - 1$ edges.

Uniqueness :If each edge has a distinct weight then there will be only one, unique minimum spanning tree. This is true in many realistic situations, such as the telecommunications company example above, where it's unlikely any two paths have exactly the same cost. This generalizes to spanning forests as well.

If the edge weights are not unique, only the (multi-)set of weights in minimum spanning trees is unique, that is the same for all minimum spanning trees.

Proof:

1. Assume the contrary, that there are two different MSTs A and B.

2. Let e_1 be the edge of least weight that is in one of the MSTs and not the other. Without loss of generality, assume e_1 is in A but not in B.

3. As B is a MST, $\{e_1\} \cup$ B must contain a cycle C.

4. Then C has an edge e_2 whose weight is greater than the weight of e_1, since all edges in B with less weight are in A by the choice of e_1 and C must have at least one edge that is not in A because otherwise A would contain a cycle in contradiction with its being an MST.

5. Replacing e_2 with e_1 in B yields a spanning tree with a smaller weight.

6. This contradicts the assumption that B is a MST.

Minimum-cost subgraph :If the weights are positive, then a minimum spanning tree is in fact a minimum-cost subgraph connecting all vertices, since subgraphs containing cycles necessarily have more total weight.

Prim's alogrithm :The algorithm may informally be described as performing the following steps:

1. Initialize a tree with a single vertex, chosen arbitrarily from the graph.

2. Grow the tree by one edge: of the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and transfer it to the tree.

3. Repeat step 2 (until all vertices are in the tree).

In more detail, it may be implemented following the pseudocode below.

1. Associate with each vertex v of the graph a number O[v] (the cheapest cost of a connection to v) and an edge E[v] (the edge providing that cheapest connection). To initialize these values, set all values of C[v] to $+\infty$ (or to any number larger than the maximum edge weight) and set each E[v] to a special flag value indicating that there is no edge connecting to earlier vertices.

2. Initialize an empty forest F and a set Q of vertices that have not yet been included in F (initially, all vertices).

3. Repeat the following steps until Q is empty:

a. Find and remove a vertex v from Q having the minimum possible value of $C[v]$

b. Add v to F and, if $E[v]$ is not the special flag value, also add $E[v]$ to F

c. Loop over the edges vw connecting v to other vertices w. For each such edge, if w still belongs to Q and vw has smaller weight than $C[w]$, perform the following steps:

(i) Set $C[w]$ to the cost of edge vw

(ii) Set E[w] to point to edge vw.

4. Return F

5. The time complexity of Prim's algorithm depends on the data structures used for the graph and for ordering the edges by weight, which can be done using a priority queue. The following table shows the typical choices.

Minimum edge weight data structure	Time complexity (total)
adjacency matrix, searching	$O(\|V\|^2)$
binary heap and adjacency list	$O((\|V\| + \|E\|) \log \|V\|)$ $= O(\|E\|) \log \|V\|)$
Fibonacci heap and adjacency list	$O(\|E\| + \|V\| \log \|V\|)$

Q.4. What is String matching? Explain the Knuth-Morris pratt Algorithm along with its complexity.

Ans. In computer science, string searching algorithms, sometimes called string matching algorithms, are an important class of string algorithms that try to find a place where one or several string (also called patterns) are found within a larger string or text.

Let Σ be an alphabet (finite set). Formally, both the pattern and searched text are vectors of elements of Σ. The Σ may be a usual human alphabet (for example, the letters A through Z in the Latin alphabet). Other applications may use binary alphabet (Σ = {0, 1}) or DNA alphabet (Σ = {A, C, G, T}) in bioinformatics.

In practice, how the string is encoded can affect the feasible string search algorithms. In particular if a variable width encoding is in use then it is slow (time proportional to N) to find the Nth character. This will significantly slow down many of the more advanced search algorithms. A possible solution is to search for the sequence of code units instead, but doing so may produce false matches unless the encoding is specifically designed to avoid it.

Knuth Morris Pratt Algorithm

KNUTH Morcis Prat Algo

KMP: Matcher (T, P)

```
1. n ← length [T]
2. m ← length [P]
3. π ← compute-prefix function (P)
4. q ← 0
5. for i ← 1 to n
6.     do while q > 0  and P[q + 1] ≠ T[i]
7.         do q ← π[q]
8.     if P[q + 1] = T[i]
9.         then q ← q + 1
10.    if q = m
11.        Then print "Pattern occues with shift" i–m
12.            q ← π[q]
```

Compute-prefix-function (*P*)

```
1. m ← length [P]
2. π[i] ← 0
3. K ← 0
4. For q ← 2 to m
5.         do while K > 0 and P[K + 1] ≠ P[q]
6.                 do k ← π[K]
7.         if P[K + 1] = P[q]
8.             then K ← K + 1
9.             π[q] ← K
10. return π.
```

Performance of KMP

The running time of COMPUTE-PREFIX-FUNCTION is $\theta(m)$, using the potential method of amortized analysis. We associate a potential of k with the current state k of the algorithm. This potential has an initial value of 0, by line 3. Line 6 decreases k whenever it is executed, since $\pi[k] < k$. Since $\pi[k] \geq 0$ for all k, however, k can never become negative. The only other line that affects k is line 8, which increases k by at most one during each execution of the for loop body. Since $k < q$ upon entering the for loop, and since q is incremented in each iteration of the for loop body, $k < q$ always holds. We can pay for each execution of the while loop body on line 6 with the corresponding decrease in the potential function, since $\pi[k] < k$. Line 8 increases the potential function by at most one, so that the amortized cost of the loop body on lines 5-9 is O(1). Since the number of outer-loop iterations is $\theta(m)$, and since the final potential function is at least as great as the initial potential function, the total actual worst-case running time of COMPUTE-PREFIX-FUNCTION is $\theta(m)$.

Failure Function of KMP. The main idea of KMP algorithm is to preprocess the pattern string P so as to compute a failure function f' that indicateds the proper shift of P so that, to the largest extent possible, we can ceuse premiously performed comparisons, specifically the failure function $f(j)$ is defined as the length of the longest prefix of P that is suffix of $P[i..j]$. We also use the convention that $f(O) = 0$.

FIRST TERM EXAMINATION [SEPT. 2015]
FIFTH SEMESTER [B.TECH]
ALGORITHM ANALYSIS AND DESIGN
[ETCS–301]

Time : 1.5 hrs. **M.M. : 30**

Note: *1. Attempt three question in total*

2. Q. No.1 is compulsory. Attmept any two more question from the remainings.

Q.1. (a) Define Big Omega (Ω) notation. (5×2)

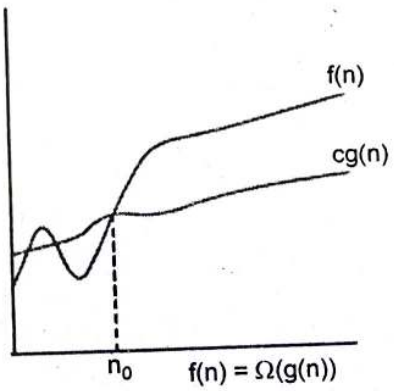
Ans. Ω Notation:

For a given function $g(n)$, we denote by $\Omega(g(n))$ as:

$\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that

$0 \le cg(n) \le f(n)$ for all $n \ge n_0\}$

Ω notation is used for asymptotic lower bounds. For all values n to the right of n_0, the value of the function $f(n)$ is on or above $cg(n)$. Figure shows the Ω notation.



Q.1. (b) Define Memorization.

Ans.

→ Apart from bottom-up fashion. dynamic programming can also be implemented using memorization.

Using memorization we implement an algorithm recursively, but we keep track of all of the substitution. If we answer a subproblem that we have seen that, we look up the solution. If we encounter a subgroup ons which we not have seen. Each subsequent time that the subproblem is encountered the value stored in the table is simply looked up and returned.

Memorization offers the efficiency of dynamic programming. It maintains the top down recursive strategy

Q.1. (c) f(n) = $\frac{1}{2}n^2 - 3n$ **find Θ**

Ans.

- We need to find positive constants c_1, c_2, and n_0 such that

$$0 \le c_1 n^2 \le \frac{1}{2}n^2 - 3n \le c_2 n^2 \text{ for all } n \ge n_0$$

- Dividing by n^2, we get

$$0 \le c_1 \le \frac{1}{2} - \frac{3}{n} \le c_2$$

- $c_1 \le \frac{1}{2} - \frac{3}{n}$ holds for $n \ge 10$ and $c_1 = 1/5$
- $\frac{1}{2} - \frac{3}{n} \le c_2$ holds for $n \ge 10$ and $c_2 = 1$
- Thus, if $c_1 = 1/5$, $c_2 = 1$, and $n_0 = 10$, then for all $n \ge n_0$,

$$0 \le c_1 n^2 \le \frac{1}{2} n^2 - 3n \le c_2 n^2 \qquad \text{for all } n \ge n_0$$

Thus we have show that $\frac{1}{2} n^2 - 3n = \Theta(n^2)$

Q.1. (d) Differentiate Dynamic Programming and Divide and conquer approach

Ans.

Divide and conquer Approach	Dynamic Programming
1. The sub problems in divide and conquer approach are more are less independent.	1. The sub problems in dynamic programming are dependent and thus overlap.
2. The subproblems are solved recursively until the instances are small enough to solve easily Thus it does more work than required by repeatedly solving same sub problems.	2. As sub problems are shared, it is solved just once and the solution is stored in table to use for solving higher level sub problems.
3. It may or may not provide an optimal solution.	3. It guarantees an optimal solution.
4. It uses top down approach.	4. It uses bottom-up approach.
5. Binary search algorithm follows divide and conquer approach.	5. Floyd-warshall algorithm uses dynamic programming.

Q.1. (e) Prove following:

(i) $n! = O(n^m)$

(ii) $1^k + 2^k + 3^k \ldots\ldots + n^k = 0\ (n^{k+1})$

Ans. → (i)

$$j(n) = n!$$
$$= n(n-1)(n-2)\ldots 1$$
$$f(n) \le n^m$$

$$\boxed{f(n) = 0(n^m)}$$

where $c = 1$ and $n_0 = 1$

(ii)

$$f(n) = 1^k + 2^k + 3^k + \ldots\ldots + n^k$$
$$\le n^k + n^k + n^k + n^k + \ldots\ldots + n^k$$
$$\le n.\ n^k$$

$$f(n) \leq n^{k+1}$$

$$\boxed{f(n) = 0(n^{k+1})}$$

Q.2. (a) Solve the following recurrence relations: (3×2)

(i) T(n) = $2T\left(\sqrt{n}\right)$ + 1 (using substitution method)

Ans. We guess that the solution is T(n) = O(nlgn) our method is to prove that T(n) ≤ (nlgn) for an appropriate choice of the constant c > 0. We start by assuming that this bound holds for $\sqrt{n}$, that is

$$T(\sqrt{n}) \leq C(\sqrt{n})\lg(\sqrt{n})$$

$$T(n) \leq 2(C\sqrt{n}\lg(\sqrt{n}))+1$$

$$\leq 2C\sqrt{n}\lg n^{1/2}+1$$

$$\leq 2C\left(\sqrt{n}\frac{1}{2}\lg(n)+1\right)$$

$$\leq C\sqrt{n}\lg(n)+1$$

$$\leq C\, n \log n$$

where the last step holds as long as c ≥ 1.

(ii) T(n) = 4T $\left(\lfloor n/2 \rfloor\right)$ + n (using iteration method)

Ans.

$$T(n) = 4T\left(\lfloor n/2 \rfloor\right)+n$$

we iterate it as follows:

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

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

$$= 4\left[4.\left(4.T\left(\left\lfloor \frac{n}{2^3} \right\rfloor\right)+\frac{n}{2^2}\right)+\frac{n}{2}\right]+n$$

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

$$\leq 4^iT\left(\frac{n}{2i}\right)+....+4^2\frac{n}{2^2}+4\frac{n}{2^1}+4^\circ\frac{n}{2^\circ}$$

The series Terminaters when

$$\frac{n}{2^i} = 1 \Rightarrow n = 2^1 \text{ or } i = \log_2 n$$

So,

$$T(n) \leq 4^{\log_2 n}TC1)+...+4^2\frac{n}{2^2}+4\frac{n}{2^1}+\frac{4n}{2^0}$$

$$\leq \left(\frac{4^\circ}{2^\circ}\right)n + \left(\frac{4^1}{2^1}\right)n + \left(\frac{4^2}{2^2}\right)n + \ldots + 4^{\log_2 n}$$

$$\leq n\sum_{i=0}^{h-1}\left(\frac{4}{2}\right)^i + 4^{\log_2 n}$$

$$\leq n\sum_{i=0}^{h-1}(2)^i + 4^{\log_2 n}$$

$$\leq n(n-1) + n^{\log_2 4}$$

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

since, $[1+2+2^2....2^{h-1}]$

$$= r = 2, a = 1 \ \& \frac{n}{2^h} = 1 \Rightarrow h = \log_2 n$$

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

$$= 2^{\log_2 n} - 1 \Rightarrow n^{\log_2 2} - 1$$

$$= (n-1)$$

$$\leq n.(n-1) + n^{\log_2 2}$$

$$\leq n(n-1) + n\ n^{\log_2 2}$$

$$\leq n(n-1) + n^{2(\log_2 2)}$$

$$\leq n(n-1) + n^2$$

$$T(n)\ \Sigma\ n(n-1) + n^2$$

$$T(n) = \theta(h^2)$$

(iii) $T(n) = \begin{cases} 5T(n-3) + O(n^2) & \text{when } n>0 \\ 1 & \text{otherwise} \end{cases}$ **(Using substract and conquer master theorem)**

Ans.

$$T(n) = 5\,T(n-3) + O(n^2)$$

$$a = 5, b = 3$$

$$n^d = n^2$$

$\therefore$

$$d = 2$$

$\therefore$

$$d > 1$$

$$T(n) = O(n^d, a^{nb})$$

$$= O(n^2 . a^{n/3})$$

$$T(n) = O(n^2 . 5^{n/3})$$

Q.2. (*b*) Write Floyed Warshall Algorithm. (4)

Ans. Consider the shortest path problem n which the objective is to find the shortest distance as well as the corresponding path for any given pair of nodes in a distance network. This type of problem can be solved using floyd's algorithm.

This algorithm takes the n its distance matrix [D°] and the initial precedence matrix [P°] as input. Then it performs n iterations (n is the no. of nodes in the distance network) and generates the final distance matrix [D^N] and the final precedence matrix [P^N]. One can find the shortest distance between any two nodes from the first distance matrix [D^N] and can trace the corresponding path from the final precedence matrix [P^N].

Steps of Floyd's Algorithm:

The steps of floyd's algorithm are presented as follows:

Step 1: Set the iteration number k = 0

Step 2: From the initial distance matrix [D°] and the inital distance Precedence [P°] from the distance network.

Step 3: Increment the iteration number by 1 (K = k + 1)

Step 4: Obtain the values of the distance matrix [D^k] for all its cells where i is not equal to j using the following formula.

$$D_{ij}^k = \min\left(D_{ij}^{k-1}, D_{ik}^{k-1} + D_{kj}^{k-1}\right)$$

Step 5: Obtain the values of the precedence matrix [p^k] for all its cells where i is not equal to j using the following formula

$$p_{ij}^k = p_{kj}^{k-1} \text{ if } D_{ij}^k \text{ is not equal to } D_{ig}^{k-1}$$

$$\neq p_{ij}^{k-1} \text{ otherwise.}$$

Step 6. If $k = n$ go to step 7, otherwise $k = k + 1$ and go to step 4.

Step 7. For each source destination nodes combination, as required in relality, find the shortest distance from the final distance matrix [D^N] and trace the corresponding shortest path, from the final precedence matrix [p^N].

Floyd Warshall's Algorithm:

Floyd Warshall (w)

1. $n \leftarrow$ rows [w]
2. $D^{[0]} \leftarrow w$
3. for $k \leftarrow 1$ to n do
4. for $i \leftarrow 1$ to n do
5. for $j \leftarrow 1$ to n do
6. $D_{ij}^{(k)} \leftarrow \left(D_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)}\right)$
7. return $D(n)$.

OR

Q.2. (b) Explain Strassen matrix multiplication with example.

Ans. Strassens Matrix Multiplication : By using divide-and-conquer technique, the overall complexity for multiplying two square matrices is reduced. This happens by decreasing the total number of multiplications performed at the expense of a slight increase in the number of additions.

For providing optimality in multiplication of matrics an algorithm was published by *V.* Strassen in 1969. which gives an overview how one can find the product C of two 2×2 dimension matrices A and B with just seven mutliplications as opposed to eight required by the brute-force algorithm.

The overall procedure can be explained as below:

$$C = \begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix}; A = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix}; B = \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$C = A' * B$$

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} x_1 + x_4 - x_5 + x_7 & x_3 + x_5 \\ x_2 + x_4 & x_1 + x_3 - x_2 + x_6 \end{bmatrix}$$

where

$$x_1 = (a_{00} + a_{11}) * (b_{00} + b_{11})$$
$$x_2 = (a_{10} + a_{11}) * b_{00}$$
$$x_3 = a_{00} * (b_{01} - b_{11})$$
$$x_4 = a_{11} * (b_{10} - b_{00})$$
$$x_5 = (a_{00} + a_{01}) * b_{11}$$
$$x_6 = (a_{10} + a_{00}) * (b_{00} + b_{01})$$
$$x_7 = (a_{01} + a_{11}) * (b_{10} + b_{11})$$

Thus, in order to multiply two 2 × 2 dimension matrices Strassen's formula used seven multiplications and eighteen additions/subtractions, whereas brute force algorithm requires eight multiplications and four additions. The utility of Strassen's formula is shown by its asymptotic superiority when order n of matrix reaches infinity.

Let us consider two matrices A and b, $n \times n$ dimension, where n is a power of two. It can be observed that we can obtain four dimension submatrices from A, B and their product C. It can easily be verified by treating submatrices as number to get the correct product.

For example:

$$\left[\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array}\right] = \left[\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right] * \left[\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array}\right]$$

C_{11} can be computed as $A_{10} * B_{01} + A_{11} * B_{11}$ or as $x_1 + x_3 - x_2 + x_3$, where x_1, x_3, x_2 and x_6 can be found by using strassen's formula, with the numbers replaced by the corresponding submatrices.

We can have Strassen's algorithm for matrix multiplication, if the seven multiplication of $n/2 \times n/2$ matrices are computed recursively by the same method.

OR

Q.3. (a) Explain Quicksort algorithm and explain worst case time complexity of the algorithm.

Ans. Quick sort Algorithm **(5×2)**

QUICKSHORT (A, P, R)

1. if $p < r$
2. then $q \leftarrow$ PARTITION (A, P, R)
3. QUICKSORT $(A, P, q-1)$
4. QUICKSORT $(A, q+1, r)$

PARTITION (A, P, r)

1. $x \leftarrow A[x]$
2. $i \leftarrow p = 1$
3. for $j \leftarrow p$ to $r - 1$
4. do if $A[j] \leq x$
5. then $i \leftarrow i + 1$
6. exchange $A[i] \leftarrow A[j]$
7. exchange $A[i + 1] \leftrightarrow A[x]$
8. return $i + 1$

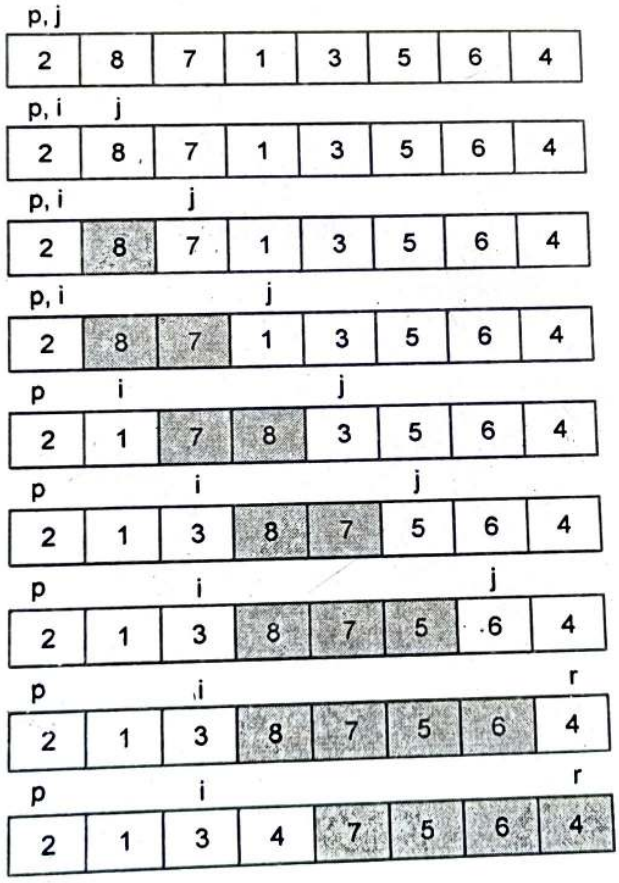
Figure shows example of partitioning on an array where $x = A[r]$ is a pivot element, we consider the following loop invariant for the quick sort algorithm to show its correctness.

Loop invariant: 1. If $p \leq K < i$ then, $A[k] \leq x$

2. If $i + 1 \leq K \leq j - 1$, then $A[K] > x$

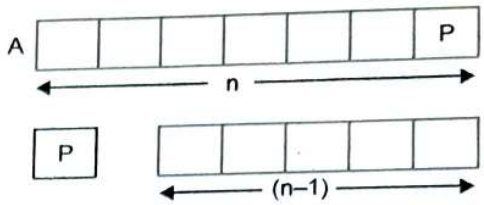
3. If $k = r$, then $A[k] = x$

Figure shows the loop invariant is tree for initialization, maintenance and termination cases.



Performance of Quick sort: The running time of quick sort depends on whether the partitioning is balanced or unbalanced. If the partitioning is balanced, the algorithm runs asymptotically as fast as merge sort and if the partitioning is unbalanced it runs asymptotically as slow as inserction sort.

Worst Case: Worst case occures when the array is divided in two unbalanced sub arrays where one subarray is empty.



Thus $T(n) = T(n-1) + T(0) + q(n)$
$= T(n-1) + q(n)$
$= 0(n^2)$

Q.3. (*b*) Sort the following numbers using Quicksort algorithm:
12 34 25 40 19 10 30 8

Ans.

I i P.j r

(*a*)

12	34	25	40	19	10	30	8

P j r

(*b*) *i*

12	34	25	40	19	10	30	8

P *j* *r*

(*c*) *i*

12	34	25	40	19	10	30	8

: continues for all as for every *j* value is less than *r*
A[*i* +1] ↔ A[*r*]

p.j *r*

II. (*a*) *i*

8	34	25	40	19	10	30	12

P.i *j* *r*

(*b*)

8	24	25	40	19	10	30	12

Pi *j* *r*

(*c*)

8	34	25	40	19	10	30	12

Pi *j* *r*

(*d*)

8	34	25	40	19	10	30	12

P.i *j* *r*

(*e*)

8	34	25	40	19	10	30	12

P,i *j* *r*

(*f*)

8	34	25	40	19	10	30	12

p. *i* *j* *r*

(*g*)

8	10	25	40	19	34	30	12

(*h*)

P *j*							*r*
8	10	12	40	19	34	30	25

continues till: $j = 2$ and next exchange results in

8	10	12	25	19	34	30	40

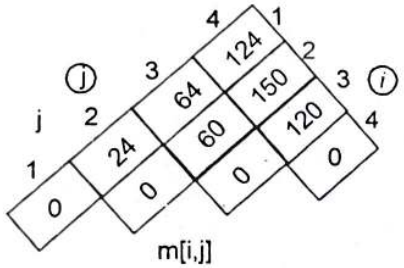
and finally the array becomes

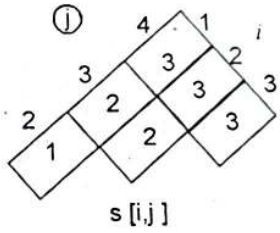
8	10	12	19	25	30	34	40

Q.4. (*a*) Find the optimal parenthesization of a matrix chain product whose sequence of dimensions are <2,3,4,5,6>. (5×2)

Ans.

Matrix	dimesnsion	P0 = 2
A_1	2×3	P1 = 3
A_2	3×4	P2 = 4
A_3	4×5	P3 = 5
A_4	5×6	P4 = 6





Way of paranthesization

$s[1,4] = 3$

(A1 A2 A3) (A4)

$s[1,3] = 2$

Optimal paranthesization $\boxed{(((A1\ A2)A3)A4)}$

Q.4. (*b*) Determine LCS of X = < B,D,C,A,B,A> and Y = <A,B,C,B,D,A,B>

Ans4 (b) . Refer Q.No.5(b) of End Term Exam 2016 (Page No.: 23-2016)

SECOND TERM EXAMINATION [NOV. 2015]
FIFTH SEMESTER [B.TECH]
ALGORITHM ANALYSIS AND DESIGN
[ETCS–302]

Time : 1.5 hrs. **M.M. : 30**

Note: *Attempt three question in total Q. No.1 is compulsory. Attempt any two more questions from the remaining.*

Q.1. (a) Define Matroid . **(5×2)**

Ans. Refer Q.No.1(h) End Term Exam 2016 (Page No.: 15-2016)

Q.1. (b) What are string matching problems ?

Ans. Refer Q.No.4 Important Questions (Page No.: 7)

Q.1. (c) Explain prefix function used in Knuth Morris Pratt algorithm using suitable example.

Ans. The perfix function π for a pattern encapsulates knowledge about how the pattern matches against shifts to itself. Give the pattern $P[1...m]$, the prefix function for the pattern P is the function $\pi : (1, 2, ..., m) \rightarrow \{0, 1, ..., m-1\}$ such that,

$\pi[q] = \max \{K : K < q \text{ and } P_K \supset P_q\}$

We call $\pi[q]$ as the length of the longest prefix of p that is a proper suffix *of* P_q.

The figure shows the prefix function.

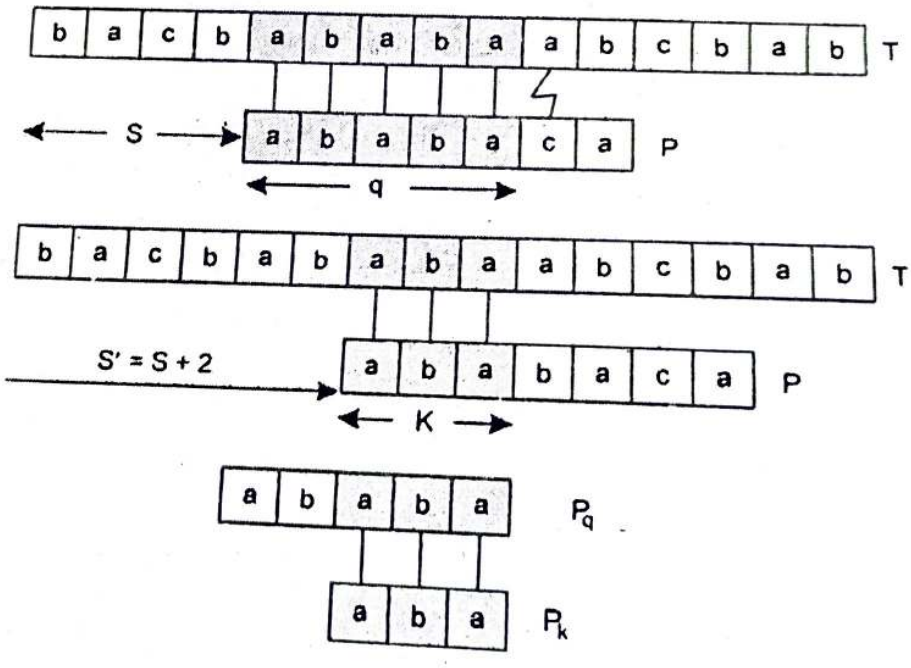


Figure shows the pattern P matches with text T so that $q = 5$. The valid shift occured at S = 4. The figure shows the pattern will not match text T when the shift $S' = S + 1$, but its valid when $S' = S + 2$. When the pattern Pq is compared with P_K, we find the array π can be represented as $\pi[5] = 3$.

The significance of prefix function is that it avoids testing useless shifts in the naive pattern matching algorithm or avoids precomputation of δ for a string matching automation.

Q.1. (*d*) Write down the worst case complexity of naive, Rabin-Karp, KMP and string matching using finite automata algorithms.

Ans. Naive

O(mn)

where m is the length of pattern and n is the length of string when in worst case length of pattern m and length of text n becomes equal algorithm runs in quadratic time

Rabin karp.

Same as Naive = 0(mn) (quadratic in worst case)

KMP:

Running time for prefix function

calculation = $\theta(m)$

and for KMP matcher = $0(n)$

Hence total time: $\Theta(m+n)$

Finite Automata

Running time = $0(n)$

n = length of Text.

Q.1. (*e*) Differentiate local and global optima.

Ans. Differentiate between global and local optima: When an algorithm finds a solution to a linear optimization model it is the definitive best solution and we say it is the global optimum. A globally optimal solution has an objective value that is as good or better than all other feasible solutions.

A locally optimum solution is the one for which no better feasible solution can be found in the immediate neighbourhood of the given solution. Additional local optimal points may exist some distance away from the current solution.

Q.2. (*a*) Find an optimal Huffman code for the following set of frequencies:

A:45 b:15 c:5 d:25 e:10 **(5)**

Ans. Optimal Huffman code.

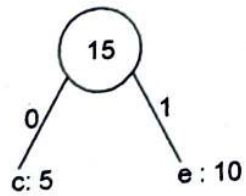
A:45 b: 15 C: 5 d:25 e: 10

(*a*) Huffman codes are widely used and popular technique for compressing data; savings upto 90% are typical, depending on the characters of the data being compressed.

Huffman's greedy algorithm uses table of the frequencies of occurence of the characters to build an optimal way of representing each character as a binary string

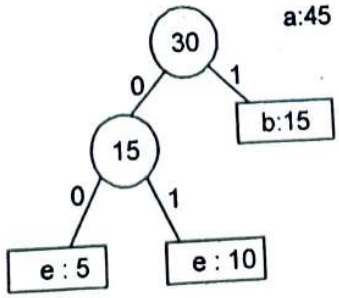
(*a*) c: 5 e: 10 *b* : 15 *d*: 25 *a*: 45

(*b*)

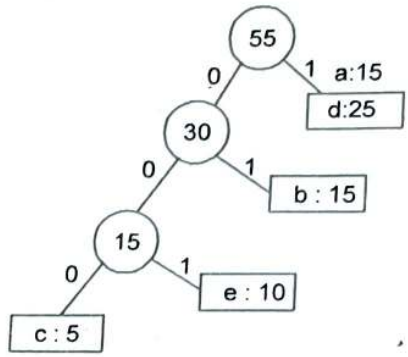


b : 15 *d*: 25 *a*: 45

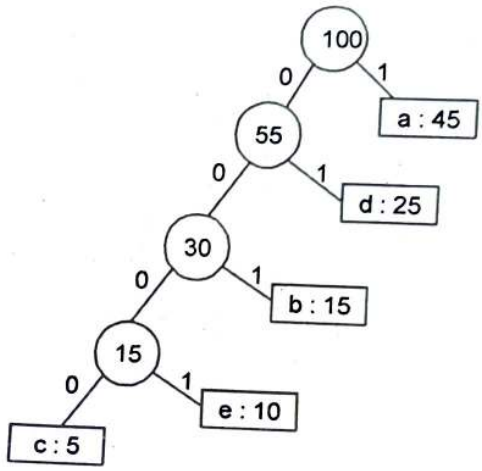
(*c*) d:25



(*d*)



(*e*)



Hence the final coding for each of the given characters is

$$\left.\begin{array}{r} a = 1 \\ b = 001 \\ c = 0000 \\ d = 01 \\ e = 0001 \end{array}\right\} \text{optimal Huffman code}$$

Q.2. (*b*) Differentiate fixed length code and variable length code using frequencies mentioned in part (*a*) of the question. (2)

Ans. Fixed length code: Here we have 5 characters, hence we need 3 bits to represent 5 characters

$a = 000$
$b = 001$
$c = 010$
$d = 011$
$e = 100$

Total bits required

$= (3\times45+3\times15 + 3\times5 + 3\times25 + 3\times10)$
$= 135 + 45 + 15 + 75 + 30$
$= 300$ bits

Variable length code: In this we give frequent characters short codes and infrequent characters long codes. This would require:

$$\left.\begin{array}{l} a = 1 \\ b = 001 \\ c = 0000 \\ d = 01 \\ e = 0001 \end{array}\right] \text{Variable length code}$$

$(45\times1 + 3\times15 + 4\times5 + 2\times25 + 4\times 10)$

$= 45 + 45 + 20 + 50 + 40$

$= 200$ bits

Q.2. (c) Differentiate Prim's and Kruskal's algorithm. (3)

Ans. Kruskul's Algorithm:

1. It is an Algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph.
2. Kruskal is where we order the nodes from smallest to largest and pick accordingly.
3. Kruskal allows both new-new nodes and old-old nodes to get connected.
4. Kruskal's algorithm builds a minimum spanning tree by adding one edge at a timc.The next line is always the shortest only if it does not create a cycle.
5. Kruskal's require us to sort the edge weight's first.

Prim's Algorithm:

1. It is the Algorithm that finds a minimum spanning tree for a connected weighted unidirected graph.
2. In Prim's algorithm we select an arbitrary node then correct the ones nearest to it.
3. Prim's always joins a new vertex to old vertex.
4. Prim's builds a minimum spanning tree by adding one vertex at a time. The next vertex to be added is always the one nearest to a vertex already on a graph.
5. In Prim's algorithm we select the shortest edge when executing the algorithm.

Q.3. (a) Write Rabin-Karp string matching algorithm. Consider working module q = 11, how many spurious hits does the Rabin-Karp matcher algorithm finds in the text T=314159265348 when looking for the pattern P=26. (6)

Ans. A string search algorithm which compares a string's hash values, rather than the strings themselves. For efficiency. the hash value of the next position in the text is easily computed from the hash value of the current position.

How Rabin-Karp works:

Let characters in both array T and P be digits in radix-S notation. (S = (0,1,...9)

Let p be the value of the characters in P.

Choose a prime number q such that fits within a computer word to speed computations.

compute (p mod q)

– The value of p mod q is what we will be using to find all matches of the pattern P ın T.

Compute (T(s+1,....s+m) mod q) for s = 0...n-m

Test against P only those sequences in T having the same (mod q) value

(T[s+1,..,s+m) mod q) can be incrementally computed by subtracting the high-order digit, shifting, adding the low-order bit, all in modulo q arithmetic.

Algorithm:

RABIN-KARP–MATCHER (T,P, d,q)

1. n ← length [*T*]
2. *m* ← length [*P*]
3. *h* ← d^{m-1} mod *q*
4. *p* ← 0
5. t_0 ← 0
6. for *i* ← 1 to m Preprocessing.
7. do *p* ← (*dp* + *p[i]*) *mod q*
8. t_0 ← (dt_0 + T[*i*]) mod *q*
9. for s ← 0 to *n* - m Matching.
10. do if *p* = t_s
11. then if *p* [1 to *m*] = *T* [*s* + 1 to *s* + *m*]
12. then print "Pattern occurs with shift" s
13. if s <*n* - *m*
14. then t_{s+1} ← (*d* (t_s – *T* [*s* + 1]*h*] + *T* [*s* + *m* +1]) mod q

A Rabin-Karp example

- Given T = 31415926535 and P = 26
- We choose q = 11
- p mod q = 26 mod 11 = 4

3	1	4	1	5	9	2	6	5	3	5

31 mod 11 = 9 not equal to 4

3	1	4	1	5	9	2	6	5	3	5

14 mod 11 = 3 not equal to 4

3	1	4	1	5	9	2	6	5	3	5

41 mod 11 = 8 not equal to 4

3	1	4	1	5	9	2	6	5	3	5

15 mod 11 = 4 equal to 4-> spurious hit

3	1	4	1	5	9	2	6	5	3	5

59 mod 11 = 4 equal to 4-> spurious hit

3	1	4	1	5	9	2	6	5	3	5

92 mod 11 = 4 equal to 4-> an spurious hit

3	1	4	1	5	9	2	6	5	3	5

26 mod 11 = 4 equal to 4 → an exact match!!

3	1	4	1	5	9	2	6	5	3	5

65 mod 11 = 10 not equal to 4

3	1	4	1	5	9	2	6	5	3	5

53 mod 11 = 9 not euqal to 4

3	1	4	1	5	9	2	6	5	3	5

35 mod 11 = 2 not equal to 4

As we can see, when a match is found, further testing is done to insure that a match has indeed been found.

Total spurious hits = 4

Q.3. (*b*) Define the complexity classes: P,NP and NPC. (4)

Ans. Define P and NP class of problems: Informally the class **P** is the class of decision problems solvable by some algorithm within a number of steps bounded by some fixed polynomial in the length of the input. Turing was not concerned with the efficiency of his machines, but rather his concern was whether they can simulate arbitrary algorithms given sufficient time. However it turns out Turing machines can generally simulate more efficient computer models (for example machines equipped with many tapes or an unbounded random access memory) by at most squaring or cubing the computation time. Thus **P** is a robust class, and has equivalent definitions over a large class of computer models. Here we follow standard practice and define the class **P** in terms of Turing machines.

Formally the elements of the class **P** are languages. Let Σ be a finite alphabet (that is, a finite nonempty set) with at least two elements, and let Σ^* be the set of finite strings over Σ. Then a language over Σ is a subset L of Σ^*. Each Turing machine M has an associated input alphabet Σ. For each string w in Σ^* there is a computation associated with M with input w. We say that M accepts w if this computation terminates in the accepting state.

Note that M fails to accept w either if this computation ends in the rejecting state, or if the computation fails to terminate. The language accepted by M, denoted $L(M)$, has associated alphabet Σ and is defined by:

$$L(M) = \{w \in \Sigma^* \mid \text{M accepts } w\}$$

P = $\{L \mid L = L(M)$ for some Turing machine M which runs in polynomial time$\}$

The notation **NP** stands for "nondeterministic polynomial time", since originally **NP** was defined in terms of nondeterministic machines (that is, machines that have more than one possible move from a given configuration). However now it is customary to give an equivalent definition using the notion of a checking relation, which is simply a binary relation $R \subseteq \Sigma^* \times {\Sigma_1}^*$ for some finite alphabets Σ and Σ_1. We associate with each such relation R a language L_R over $\Sigma \cup \Sigma_1 \cup \{\#\}$ defined by

$$L_R = \{w\#y \mid R(w,y)\}$$

where the symbol # is not in Σ. We say that R is polynomial-time iff $L_R \in$ **P**. Now we define the class **NP** of languages by the condition that a language L over Σ is in **NP** iff there is $k \in N$ and a polynomial-time checking relation R such that for all $w \in \Sigma^*$, $w \in L \Leftrightarrow \exists y(|y| \leq |w|_k$ and R(w, y)) where $|w|$ and $|y|$ denote the lengths of w and y, respectively.

A problem is NP-complete if it is both NP-hard and an element of NP (or 'NP-easy'). NPcomplete problems are the hardest problems in NP. If anyone finds a polynomial-time algorithm for even one NP-complete problem, then that would imply a polynomial-time algorithm for every NP-complete problem. Literally thousands of problems have been shown to be NP-complete, so a polynomial-time algorithm for one of them seems incredibly unlikely.

It is not immediately clear that any decision problems are NP-hard or NP-complete. NP-hardness is already a lot to demand of a problem; insisting that the problem also

have a nondeterministic polynomial-time algorithm seems almost completely unreasonable.

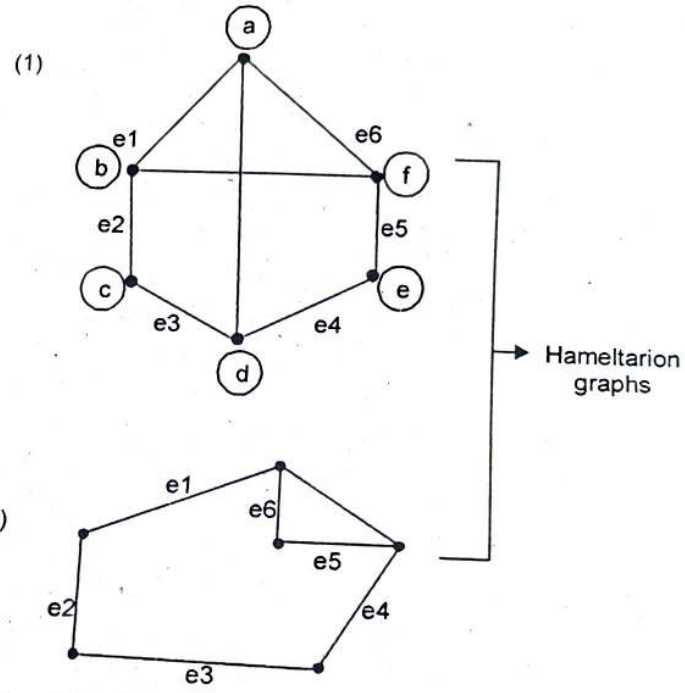
Q.4. Attempt (any two) (5×2)

Q.4. (a) Explain Hamiltonian cycle problem (with suitable example).

Ans. (a) Hamiltonian cycle problem: A Hamiltonian path is a path that visits each vertex exactly once. A graph that contains a Hamiltonian path is called a **traceable** graph. A group is **Hamiltonian-connected** if for every pair of vertices there is a Hamiltonian path between the two vertices.

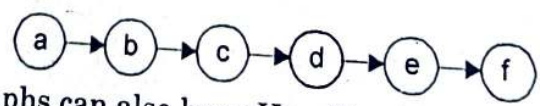
A Hamiltonian cycle. Hamiltonian circuit vertex tour or graph cycle is a cycle that visits each vertex exactly once (except for the vertex that is both the start and end, which is visited twice). A graph that contains a Hamiltonian cycle is called a Hamiltonian graph.

A Hamiltonian decomposition is an edge decomposition of a graph into Hamiltonian circuits. Examples of Hamiltonian graphs.



Hamiltonia cycle of fig (*i*)

If the last edge of a Hamiltonian cycle is dropped, we get a Hamiltonian path.
Hamiltonian path of fig (*i*)



Non Hamiltonian graphs can also have Hamiltonian paths.

Q.4. (b) Find the optimal schedule for the following task with given weight (penalties) and deadlines:

	1	2	3	4	5	6
d_i	2	2	1	3	3	1
w_i	20	15	10	5	1	25

Ans. Task Scheduling Problems: The problem of scheduling unit time tasks with deadlines and penalties for a signle processor has the following inputs:

(*a*) a set s = $\{a_1, a_2 \ldots a_n\}$ of n unit time tasks.

(*b*) a set of n intger deadlines, $d_1, d_2, \ldots d_n$ such that each d_i satisfies $1 \leq d_i \leq n$ and task as is supposed to finish by time d_i.

(*c*) a set of n integers or penalties $w_1, w_2 \ldots w_n$, such that we incur a penalty of w_i if task a_i is not finished by time d_i and we incur no penalty if a task funishes by its deadline.

Given deadlines and penalties are

	1	2	3	4	5	6
d_i	2	2	1	3	3	1
w_i	20	15	10	5	1	25

Step 1. Arrange tasks in decreasing order of penalties

	1	2	3	4	5	6
d_i	1	2	2	1	3	3
w_i	25	20	15	10	5	1

Step 2: Pairwise comparison.

Hence tasks as a_1, a_2, and a_5 are accepted and a_3, a_4 *and* a_6 are rejected because they can't be completed by their deadlines

The final schedule (optimal) is

$$\boxed{< a_1, a_2, a_5, a_3, a_4, a_6 >}$$

and total penalty incurred

$$= w_3 + w_4 + w_6$$
$$= 15 + 10 + 1 = 26$$

Q.4. (*c*) String Matching with finite automata. (with suitable example)

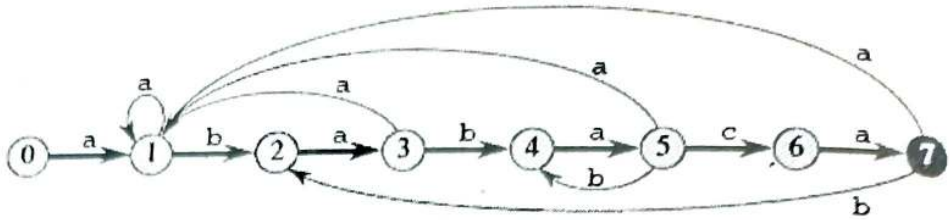
Ans. String matching algorithm using finite automata: There is a string-matching automaton for every pattern P; this automaton must be constructed from the pattern in a pre-processing step before it can be used to search the text string. Figure below illustrates this construction for the pattern $P = ababaca$.

We shall assume that P is a given fixed pattern string; for brevity, we shall not indicate the dependence upon P in our notation. In order to specify the string-matching automaton corresponding to a given pattern $P[1...m]$, we first define an auxiliary function σ, called the suffix function corresponding to P. The function σ is a mapping from Σ to $\{0, 1,...,m\}$ such that $\sigma(x)$ is the length of the longest prefix of P that is a suffix of x : $\sigma(x) = \max \{k : P_k \supset x\}$.

The suffix function σ is well defined since the empty string $P_0 = \varepsilon$ is a suffix of every string. As examples, for the pattern $P = ab$, we have $\sigma(\varepsilon) = 0, \sigma(ccaca) = 1$, and $\sigma(ccab) = 2$. For a pattern P of length m, we have $\sigma(x) = m$ if and only if $P \supset x$. It follows from the definition of the suffix function that if $x \supset y$, then $\sigma(x) \leq \sigma(y)$.

We define the string-matching automaton that corresponds to a given pattern $P[1...m]$ as follows.

- The state set Q is $\{0, 1,...,m\}$. The start state q_0 is state 0, and state m is the only accepting state.
- The transition function σ is defined by the following equation, for any state q and character a : $\delta(q, a) = \sigma(P_q a)$.



(a)

State	Input a	b	c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

(b)

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

(c)

To clarify the operation of a string-matching automaton, we now give a simple efficient program for simulating the behaviour of such an automaton (represented by it transition function δ) in finding occurrences of a pattern P of length m in an input tex $T[1...n]$.

FINITE-AUTOMATION-MATCHER (T, δ, m)

1. $n \leftarrow$ length $[T]$
2. $q \leftarrow 0$
3. for $i \leftarrow 1$ to n
4. do $q \leftarrow d(q, T[i])$
5. if $q = m$
6. then print "Pattern occurs shift" $i - m$

As for any string-matching automaton for a pattern of length m, the state set Q is $\{0$ $1,...,m\}$, the start state is 0, and the only accepting state is state m. The simple loo structure of FINITE-AUTOMATON-MATCHER implies that its matching time on text string of length n is $\theta(n)$.

Q.4. (d) Proof of correctness of Bellman-Ford algorithm.

Ans. Proof of Correctness of Bellman Ford Algorithm: Bellman Ford Algorith that computes shortest paths from a signle source vertex to all of the other vertices in weighted digraph.

The correctness of algorithm can be shown by induction.

Lemma: After i repetition of for loop:

- It distance (u) is not infinity, it is equal to the length of some path from s to u
- If there is a path from s to u with at most i edges, then Distance (u) is at most th length of the shortest path from s to u with at most i edges.

Proof: For the base case of inductions consider [$i = 0$] and the moment before for loop is executed for the first time.

Then, for the source vertex.

Source distance = 0 which is correct for other vertices 4, u. distance = infinity. Which is also correct because there no path from source to u with 0 edges

For the inductive case, we first prove the first part. Consider a moment when a vertex distance is updated by

V distance = u. distances + uv wight

By inductive assumption u. distance is the length of the path from source to u. Then u. distance + uv. weight is the length of the path from source to v that follows the path from source to u and then goes to *v*.

For the second part, consider the shortest path from source to u with atmost i edges let v be the last. Vertex before u on this path. Then, the Part of the path from source to v is the shortest path from source to v with at most $i - 1$ edges. By inductive assumption v distance after $i - 1$ iterations is at most the length of this path. Therefore, uv. weight + v. distance is at most the length of the path from s to u. In the ith iteration, u. distance gets compared with uv. weight + v. distance. and is set equal to it if uv. weight +v. distance was smaller. Therefore after i iteration u. distance is at most the length of the shortest path gets from source to u that uses autmost i edges.

If there are no negative-weight cycles. Then every shortest path visits each vertex at most once. So at step 3 no futher improvement can be made conversely suppose no improvement can be made. Then for any cycle with vertices v[0] ...v[$k-1$]

v (i) distance < = v[($i - 1$) mod K]. distance + v($i - 1$) mod k] weight

Summing around the cycle, the [i] distance terms and the v [(i -1) [mod k]] distance terms cancel leaving

O ≤ sum from 1 to k of v ($i - 1$ (mod k)] v [i] weight

i.e. every cycle has non-negative weight.

END TERM EXAMINATION [DEC. 2015]
FIFTH SEMESTER (B.TECH)
ALGORITHM ANALYSIS AND DESIGN
[ETCS–301]

Time : 3 hrs. M.M. 75

Note: *1. Attempt any five questions including Q.no.1 which is compulsory.*

Q.1. (a) Define θ, O notations and explain. (5×5 = 25)

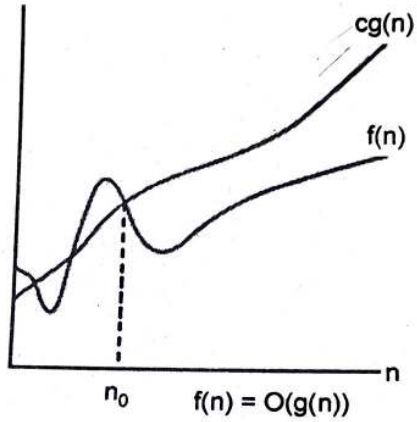
Ans. O Notation:

For a given function $g(n)$, we denote by $O(g(n))$ as :

$O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that

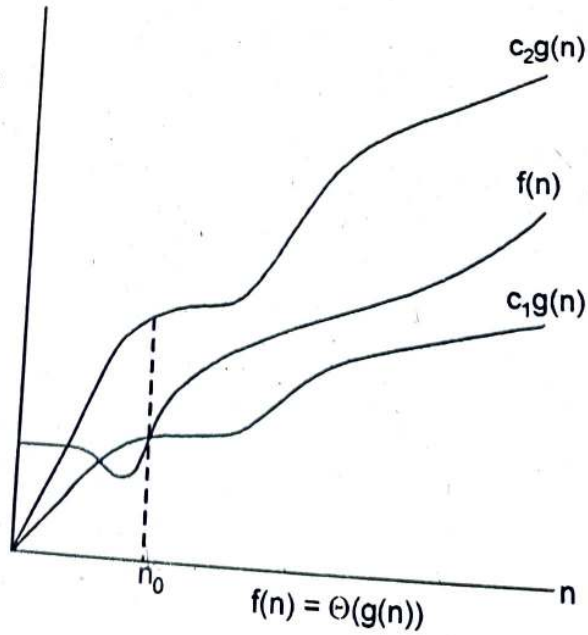
$0 \le f(n) \le cg(n)$ for all $n \ge n_0\}$

O notation is used for asymptotic upper bounds. For all values n to the right of n_0, the value of the function $f(n)$ is on or below $g(n)$. Figure shows the O notation.



θ Notation:

For a given function $g(n)$, we denote by $\theta(g(n))$ the set of functions as: $\theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1 g(n) \le f(n) \le c2g(n)$ for all $n \ge n_0\}$



A function $f(n)$ belongs to the set $\theta(g(n))$ if there exist positive constants c_1 and c_2 such that it can be "sandwiched" between $c_1 g(n)$ and $c_2 g(n)$, for sufficiently large n. Because $\theta(g(n))$ is a set, we could write "$f(n) \in \theta(g(n))$" to indicate that $f(n)$ is a member of $\theta(g(n))$. When $f(n) = \theta(g(n))$, we say that $g(n)$ is an asymptotically tight bound for $f(n)$. Figure shows the θ notation.

Q.1. (b) Prove $3n^2 + 2n^2 = 0\ (n^2)$; $3^n \neq 0\ (2^n)$

Ans.

Proof

when $n \geq 1$

$$f(n) = 3n^2 + 2n^2$$
$$= 5n^2$$
$$g(n) = 3n^2 + 2n^2$$
$$= 5n^2$$
$$f(n) \leq c(g(n) \text{ for all } n \geq n_0 = 1.$$

so, $f(n) = 0\ (n^2)$

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

$g(n) = 2^n$

$f(n) \not\leq g(n)$ for all $n \geq 1$

Hence

$f(n) \neq 0\ (g(n))$

or $(3^n) \neq 0(2^n)$

Q.1. (c) Write an algorithm for merge sort. Find its worst case, best case and average case complexity.

Ans. The ***merge sort*** algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows.

- **Divide:** Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each.
- **Conquer:** Sort the two subsequences recursively using merge sort.
- **Combine:** Merge the two sorted subsequences to produce the sorted answer.

```
MERGE (A, p, q, r)
1   n1 ← q − p + 1
2   n2 ← r − q
3   create arrays L [1 to n1 + 1] and R [1 to n2 + 1] ]
4   for i ← 1 to n1
5           do L[i] ← A [p + i − 1]
6   for j ← 1 to n2
7           do R [j] ← A [q + j]
8   L [n1 + 1] ← ∞
9   R [n2 + 1] ← ∞
10  i ← 1
11  j ← 1
12  for k ← p to r
13          do if L[i] ≤ R [j]
14                  then A [k] ← R [j]
```

```
15          i ← i + 1
16      else A [k] ← R [j]
17          j ← j +1
```

If $p \geq r$, the subarray has at most one element and is therfore sorted. Otherwise, the divide step simply computes an index q that partitions $A[p$ to $r]$ into two subarrays: $A[p$ to $q]$, containing $[n/2]$ elements, and $A[q + 1$ to $r]$, containing $[n/2]$ elements.

```
MERGE-SORT (A,P,r)
1. if p < r
2. then q ← [(p + r)/2]
3.    MERGE-SORT (A,p,q)
4.    MERGE-SORT (A, q + 1, r)
5.    MERGE (A,p,q,r)
```

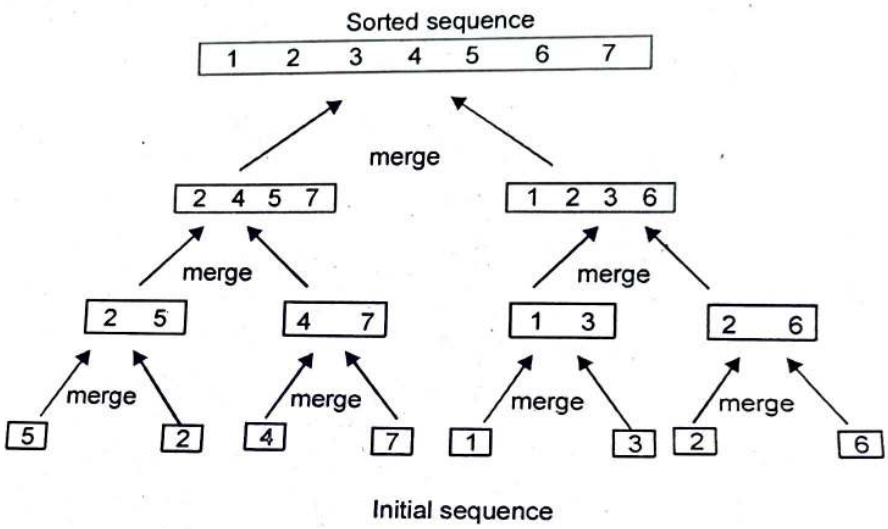


Fig.: The operation of merge sort on the array $A = 5, 2, 4, 7, 1, 3, 2, 6$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top

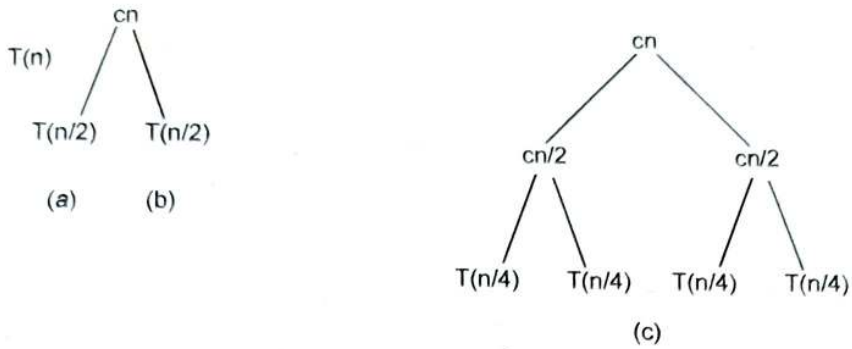
Analysis of merge sort: Merge sort on just one element takes constant time. When we have $n > 1$ elements, we break down the running time as follows.

- **Divide:** The divide step just computes the middle of the subarray, which takes constant time. Thus, $D(n) = \odot\ (1)$.
- **Conquer:** We recursively solve two subproblems, each of size $n/2$, which contributes $2T(n/2)$ to the running time.
- **Combine:** We have already noted that the MERGE procedure on an n-element subarray takes time $\Theta\ (n)$, so $C(n) = \Theta\ (n)$.

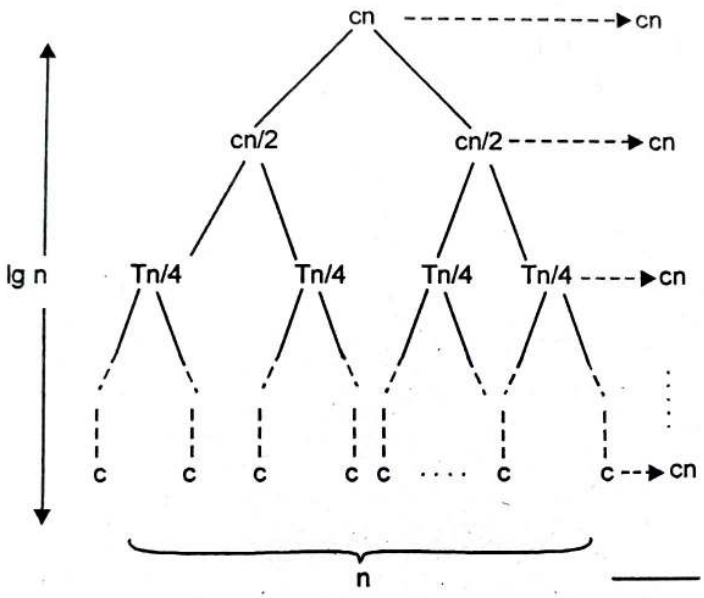
When we add the functions $D(n)$ and $C(n)$ for the merge sort analysis, we are adding a function that is $\Theta\ (n)$ and a function that is $\Theta\ (1)$. This sum is a linear function of n, that is, $\Theta\ (n)$. Adding it to the $2T(n/2)$ term from the "conquer" step gives the recurrence for the worst-case running time $T(n)$ of merge sort:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases}$$

We can solve the recurrence. For convenience, we assume that n is an exact power of 2. Part (a) of the figure shows $T(n)$, which in part (b) has been expanded into an equivalent tree representing the recurrence. Part (c) shows this process carried one step further by expanding $T(n/2)$.



Part (d) shows the resulting tree.



(d) Total: $cn \lg n + cn$

Fig. The construction of a recursion tree for the recurrence $T(n) = 2T(n/2) + cn$.

To compute the total cost represented by the recurrence (2.2), we simply add up the costs of all the levels. There are $\lg n + 1$ levels, each costing cn, for a total cost of $cn(\lg n + 1) = cn \lg n + cn$. Ignoring the low-order term and the constant c gives the desired result of $\Theta(n \lg n)$.

Q.1. (d) Explain 0-1 Knapsack problem and discuss its solution.

Ans. The ***0–1 knapsack problem*** is posed as follows. A thief robbing a store finds n items; the ith item is worth vi dollars and weighs wi pounds, where vi and wi are integers. He wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack for some integer W. Which items should he take? (This is called the 0–1 knapsack problem because each item must either be taken or left behind; the thief cannot take a fractional amount of an item or take an item more than once.)

Formal description: Given two n-tuples of positive numbers

$<v_1, v_2, \ldots, v_n>$ and $<w_1, w_2 \ldots, w_n>$, and $w > 0$, we wish to determine the subset T of {1,2,....,n} (of files of store) that

$$\text{maximizes} \sum_{i \in T} v_i,$$

$$\text{subject to} \sum_{i \in T} w_i, \leq W .$$

Dynamic-Programming Solution to the 0-1 Knapsack Problem

Let i be the highest numbered item in an optimal solution S for W pounds. Then S' = S – (i) is an optimal solution for W-w_i pounds and the value to the solution S is V_i plus the value of subproblem.

We can express this fact in the following formula define C[i, W] to be the solution for items 1,2....i and maximum weight w. Then

$$c[i.w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ c[i\text{-}1,w] & \text{if } w_i \geq 0 \\ \text{Max}\,[v_i + c\,[i\text{-}1, w\text{–}w_i]c[i\text{-}1,w]] & \text{if} > 0 \text{ and} \geq w_i \end{cases}$$

This says that the values of the solution to i items either include i^{th} item in which case it is v_i plus a subproblem solution for (i-1) items and the weight excluding W_i, or does not include i^{th} item in which case it is a subproblem's solution for (i -1) items and the same weight. That if the thief picks item i thief takes v_i value and thief can choose from items w-w_i and get C [i -1, w,-w_i] additional value. On other hand if theif decide not to take item i, thief choose from item 1,2....i –1 upto the weight limit w, and get c[i-1,w] value. The better of these two choices should be made.

Although the 0-1 knaspack problem the above formula for c is similar to LCS formula boundary values are 0, and other values are computed from the input and "earlier" values of C. So the 0-1 knapsack algorithm is like the LCS-length algorithm given in CLR for finding a longest common subsequunce of two sequences.

The algorithm takes as input the maximum weight W. *the number of items* n, and the two sequences $v = <v_1, v_2, \ldots v_n>$ and $w = <w_1, w_2, \ldots, w_n>$. it stores the c[$i$,$j$] values in the table. that is a two dimensional array, c [0...n,0...w] whose entries are computed in a row-major order. That is, the first row of C is filled in from left to right, then the second row, and so on. At the end 0. the computance $c[n,w]$ contains the maximum value that can be picked into the knapsack.

Dynamic-0-1 Knapsack (v, w, n, W)

For $w = 0$ to W
DO $c[0,w] = 0$
FOR $i = 1$ to n
DO c [i,0] = 0
FOR $w = 1$ TO W
DO IFf $w_i \leq w$
THEN IF $v_i + c[i\text{-}1, w\text{-}wi]$
THEN $c[i,w]=v_i + c\,[i\text{ -}1, w\text{-}w_i]$
ELSE $c(i,w) = c[i\text{-}1,w]$
EISE $c(i,w) = c[i - 1,w]$

The set of items to take can be deduced from the table, starting at $c[n. w]$ and tracing backwards where the optimal values came from. If $c[i, w] = c[i\text{-}1, w]$ item i is

part of the solution, and we are continue tracing with $c[i\text{-}1, w]$. Otherwise item i is part of the solution, and we continue tracing with $c[i\text{-}1, w\text{-}W]$.

Q.1. (e) Differentiate between the functioning of Dijkristra and Bellman ford algorithm.

Ans. Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u. In the following implementation, we use a min-priority queue Q of vertices, keyed by their d values.

DIJKSTRA (G, w, s)

1. INITIALIZE-SINGLE-SOURCE (G, s)
2. $S \leftarrow \theta$
3. $Q \leftarrow V[G]$
4. **While** $Q \neq \phi$
5. do $u \leftarrow$ EXTEACT-MIN(Q)
6. $S \leftarrow S \cup \{u\}$
7. for each vertex $u \in Adj[u]$
8. do RELAC (u, v, w)

Bellman-Ford algorithm: The Bellman-Ford algorithm solves the single-source shortest-paths problem in the general case in which edge weights may be negative. Given a weighted, directed graph $G = (V, E)$ with source s and weight function $w : E \rightarrow R$, the Bellman-Ford algorithm returns a boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source. If there is such a cycle, the algorithm indicates that no solution exists. If there is no such cycle, the algorithm produces the shortest paths and their weights.

The algorithm uses relaxation, progressively decreasing an estimate $d[v]$ on the weight of a shortest path from the source s to each vertex $v \in$ V until it achieves the actual shortest-path weight $\delta(s, v)$. The algorithm returns TRUE if and only if the graph contains no negative-weight cycles that are reachable from the source.

BELLMAN-FORD (G, w, s)

1. INITIALIZE-SINGLE-SOURCE (G, s)
2. for $i \leftarrow 1$ to $|V[G]| - 1$
3. do for each edge $(u, v) \in E[G]$
4. do RELAX(u, v, w)
5. for each edge $(u, v) \in E[G]$
6. do if $d[v] > d[u] + w(u, v)$
7. then return FALSE
8. **return** TRUE

Q.2. (a) What are recurrence relations? Solve following using recurrence relation. $x(n) = x(n-1)$ for $n > 0$ and $x(0) = 0$. **(6.25)**

Ans. A recurrence relation is an equation that relates a general term in the sequence, say a_n, with one of more preceding terms. The recurrence relation is said to be obeyed by the sequence. The following is a general recurrence relation.

$$a_n = f(a_{n-1}, a_{n-2}, \ldots, a_{n-m}) \; n \geq m \qquad (1)$$

Here, the term a_n is a function of m immediately preceding terms of the sequences. IF m is a small positive integer, a_n depends on a finite amount of histery [GK90]. If $n = m$, the value of a_n depends on all the prior terms in the sequence and is called a recurrence *with full history.*

The following are examples of some general recurrence relations.

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \ldots + c_m a_{n-m}$$
$$a_n = c\, a_{n-1} + f(n) f(n) \text{ is a function of } n$$
$$a_{n,m} = a_{n-1,m} + a_{n-1,m-1}$$
$$a_n = c\, a_{n-1,m} + a_{n-1,m-1}$$
$$a_n = a_0 a_{n-1} + a_1 a_{n-2} + \ldots + a_{n-1} a_0$$

Solution of rec relation:

The recurrence relation we solve is given below.

$$T(n) = T(n-1) + c \qquad n > 1$$

Here, c is a small positive constant.

The recurrence corresponds to an algorithm that makes one pass over each one of the n elements. It takes c time to examine an element.

Termination Condition:

In terms of an equation, we can say the following. T(0) = 0.

Instantiations:

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

is the recurrence relation we are going to use again and again. Assume n is an integer, n ≥ 1. For example, if we want to instantiate the recurrence for an argument value of n-1, we get

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

Note that the argument to T on the right hand side is one less than the argument on the left hand side. Thus, if we instantiate the formula for an argument k, we get

$$T(k) = T(k-1) + c$$

Or, if we instantiate it for an argument value of 2, wet get T(2) = T(1) + c

The solution to the recurrence relation follows:

$$\begin{aligned} T(n) &= \boxed{T(n-1)+c} \\ &= (T(n-2)+c)+c \\ &= \boxed{T(n-2)+2c} \\ &= (T(n-3)+c)+2c \\ &= \boxed{T(n-c)+3c} \\ &\vdots \\ &= \boxed{T(n-k)+kc} \\ &\vdots \\ &= T(n-(n-1)+(n-1)c \\ &= T(n-n)+(n)c \\ &= T(0)+nc \\ &= 0+nc \\ &= nc \end{aligned}$$

We know $T(1) = 0$ from the terminating condition of the recurrence. Therefore, simplification produces

$T(n) = nc$ as the solution.

Or $T(n) = O(n)$..

Q.2. (b) State and prove master theorem. Solve any example using master method.

Ans. The master method provides a "cookbook" method for solving recurrences of the form (6.25)

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

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function. The master method requires memorization of three cases.

The recurrence describes the running time of an algorithm that divides a problem of size n into a subproblems, each of size n/b, where a and b are positive constants. The a subproblems are solved recursively, each in time $T(n/b)$. The cost of dividing the problem and combining the results of the subproblems is described by the function $f(n)$. For example, the recurrence arising from the MERGE-SORT procedure has $a = 2$, $b = 2$, and $f(n) = \Theta(n)$.

The master method depends on the following theorem.

The master theorem

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

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

where we interpret n/b to mean either n/b or n/b. Then $T(n)$ can be bounded asymptotically as follows.

There are 3 cases:

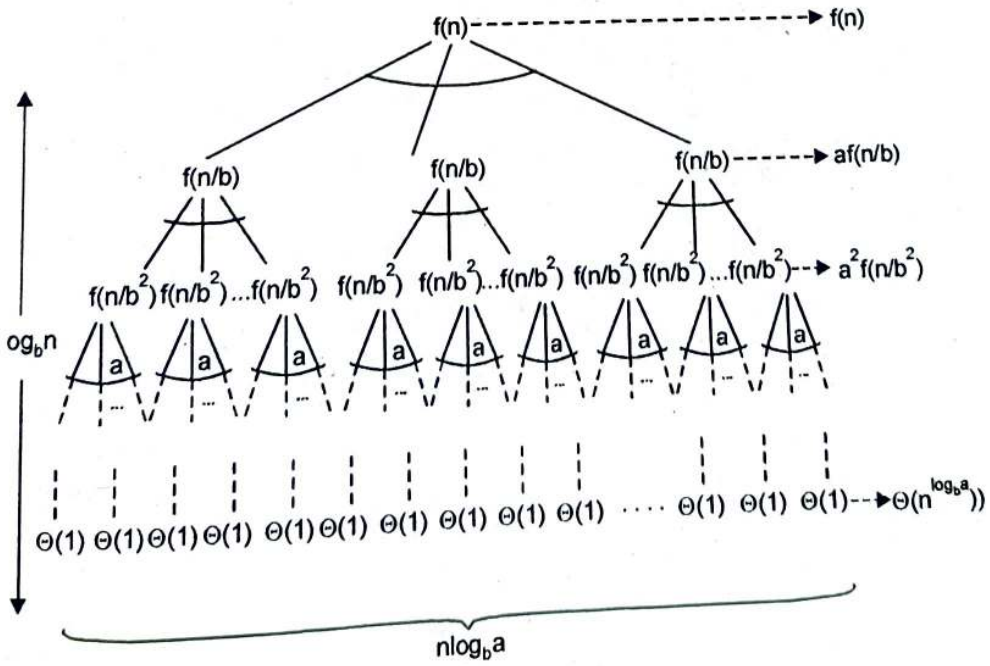
1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta\left(n^{\log_b a} \lg n\right)$

3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ with ϵ, and $f(n)$ satisfies the regularity condition, then $T(n) = Q(f(n))$. Regularity condition: $af(n/b) < cf(n)$ for some constant $c < 1$ and all sufficiently large n.

How does this work?

Master method is mainly derived from recurrence tree method. If we draw recurrence tree of $T(n) = aT(n/b) + f(n)$, we can see that the work done at root is $f(n)$ and work done at all leaves is $\Theta(n^c)$ where c is $\text{Log}_b a$. And the height of recurrence tree is $\text{Log}_b n$



In recurrence tree method, we calculate total work done. If the work done at leaves is polynomially more, then leaves are the dominant part, and our result becomes the work done at leaves (Case 1). If work done at leaves and root is asymptotically same, then our result becomes height multiplied by work done at any level (Case 2). If work done at root is asymptotically more, then our result becomes work done at root (Case 3).

EXAMPLE:

Consider the recurrence

$T(n) = 4T(n/2) + n^3$.

For this recurrence, there are again $a=4$ subproblems, each dividing the input by $b=2$, but now the work done on each call is $f(n)=n^3$. Again $n^{\log_b a}$ is n^2, and $f(n)$ is thus $\Omega(n^{2+\epsilon})$ for $\epsilon=1$. Moreover, $4(n/2)^3 \le kn^3$ for $k=1/2$, so Case 3 applies. Thus $T(n)$ is $\Theta(n^3)$.

EXAMPLE:

Consider the recurrence

$$T(n) = 4T(n/2) + n^2.$$

For this recurrence, there are again $a=4$ subproblems, each dividing the input by $b=2$, but now the work done on each call is $f(n)=n^2$. Again $n^{\log_b a}$ is n^2, and $f(n)$ is thus $\Theta(n^2)$, so Case 2 applies. Thus $T(n)$ is $\Theta(n^2 \log n)$. Note that increasing the work on each recursive call from linear to quadratic has increased the overall asymptotic running time only by a logarithmic factor.

Q.3. (*a*) Explain Strassen's algorithm and explain. **(6.25)**

Ans. Refer Q.2. (*b*) of First term 2015

Q.3. (*b*) Compute following using strassen's algorithm. **(6.25)**

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 3 & 2 \\ 0 & 5 & 1 \end{bmatrix} \times \begin{bmatrix} 2 & 4 & 3 \\ 4 & 8 & 9 \\ 3 & 8 & 9 \end{bmatrix}$$

Ans.

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 3 & 2 \\ 0 & 5 & 4 \end{bmatrix} \times \begin{bmatrix} 2 & 4 & 3 \\ 4 & 8 & 9 \\ 3 & 8 & 9 \end{bmatrix}$$

A B

We can divide the matrix A and B in four square parts if n is a degree of 2. But A and B N is not a degree of 2 so,

$$\begin{bmatrix} 1 & 2 & 1 & 0 \\ 0 & 3 & 2 & 0 \\ 0 & 5 & 4 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 2 & 4 & 3 & 0 \\ 4 & 8 & 9 & 0 \\ 3 & 8 & 9 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

A B

We divide the matrices A& B into submatrices of size N/2 × N/2

$$\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right] \times \left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right]$$

A B

In Strassen method the flour submatrices of result are calculated using following formula

$$P1 = a\,(f\text{-}h);\ P2 = (a+b)\,h$$
$$P3 = (c+d)\,e;\ P4 = d\,(g\text{-}e)$$
$$P5 = (a+b)\,(e+h);\ P6 = (b\text{-}d)\,(g+h)$$
$$P7 = (a-c)\,(e+f)$$

The A × B can be calculated using above seven multiplications. Following are values of four sub-matrix of result C.

$$\underbrace{\begin{bmatrix} a & b \\ c & d \end{bmatrix}}_{A} \times \underbrace{\begin{bmatrix} e & f \\ g & h \end{bmatrix}}_{B} = \underbrace{\begin{bmatrix} P5+P4-P2+P6 & P1+P2 \\ P3+P4 & P1+P5-P3-P7 \end{bmatrix}}_{C}$$

A, B, and C are square matrices of N × N. a, b, c, d are submatrices of A, of size N/2 × N/2 e, f, g, h, are submatrices B, of size N/2 × N/2. P1, P2, P3, P4, P5, P6, P7 are submatrices of size N/2 × N/2

So,

$$A = \begin{array}{c} \begin{matrix} a & \quad & b \end{matrix} \\ \left[\begin{array}{cc|cc} 1 & 2 & 1 & 0 \\ 0 & 3 & 2 & 0 \\ \hline 0 & 5 & 4 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right] \\ \begin{matrix} c & \quad & d \end{matrix} \end{array}$$

$$B = \begin{array}{c} \begin{matrix} e & \quad & f \end{matrix} \\ \left[\begin{array}{cc|cc} 2 & 4 & 3 & 0 \\ 4 & 8 & 9 & 0 \\ \hline 3 & 8 & 9 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right] \\ \begin{matrix} g & \quad & h \end{matrix} \end{array}$$

So,

$$P1 = a\,(f-h)$$
$$= \begin{bmatrix} 1 & 2 \\ 0 & 3 \end{bmatrix}\left(\begin{bmatrix} 3 & 0 \\ 9 & 0 \end{bmatrix} - \begin{bmatrix} 9 & 0 \\ 0 & 0 \end{bmatrix}\right)$$
$$= \begin{bmatrix} 1 & 2 \\ 0 & 3 \end{bmatrix}\left(\begin{bmatrix} -6 & 0 \\ 9 & 0 \end{bmatrix}\right)$$
$$= \begin{bmatrix} 12 & 0 \\ 27 & 0 \end{bmatrix}$$
$$P2 = (a+b)h$$
$$= \left(\begin{bmatrix} 1 & 2 \\ 0 & 3 \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 2 & 0 \end{bmatrix}\right)\begin{bmatrix} 9 & 0 \\ 0 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 2 & 2 \\ 2 & 3 \end{bmatrix} \times \begin{bmatrix} 9 & 0 \\ 0 & 0 \end{bmatrix} \Rightarrow \begin{bmatrix} 18 & 0 \\ 18 & 0 \end{bmatrix}$$

$P3 = (c + d)\,e$

$$= \left(\begin{bmatrix} 0 & 5 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 4 & 0 \\ 0 & 0 \end{bmatrix} \right) \begin{bmatrix} 2 & 4 \\ 4 & 8 \end{bmatrix}$$

$$= \begin{bmatrix} 4 & 5 \\ 0 & 0 \end{bmatrix} \times \begin{bmatrix} 2 & 4 \\ 4 & 8 \end{bmatrix} \Rightarrow \begin{bmatrix} 28 & 56 \\ 0 & 0 \end{bmatrix}$$

$P4 = d\,(g–e)$

$$= \begin{bmatrix} 4 & 0 \\ 0 & 0 \end{bmatrix} \left(\begin{bmatrix} 3 & 8 \\ 0 & 0 \end{bmatrix} - \begin{bmatrix} 2 & 4 \\ 4 & 8 \end{bmatrix} \right)$$

$$= \begin{bmatrix} 4 & 0 \\ 0 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 4 \\ -4 & -8 \end{bmatrix} \Rightarrow \begin{bmatrix} 4 & 16 \\ 0 & 0 \end{bmatrix}$$

$P5 = (a + d)\,(e+h)$

$$= \left(\begin{bmatrix} 1 & 2 \\ 0 & 3 \end{bmatrix} + \begin{bmatrix} 4 & 0 \\ 0 & 0 \end{bmatrix} \right) \times \left(\begin{bmatrix} 2 & 4 \\ 4 & 8 \end{bmatrix} + \begin{bmatrix} 9 & 0 \\ 0 & 0 \end{bmatrix} \right)$$

$$= \begin{bmatrix} 5 & 2 \\ 0 & 3 \end{bmatrix} \times \begin{bmatrix} 11 & 4 \\ 4 & 8 \end{bmatrix} \Rightarrow \begin{bmatrix} 63 & 36 \\ 12 & 24 \end{bmatrix}$$

$P6 = (b-d)\,(g+h)$

$$= \left(\begin{bmatrix} 1 & 0 \\ 2 & 0 \end{bmatrix} - \begin{bmatrix} 4 & 0 \\ 0 & 0 \end{bmatrix} \right) \left(\begin{bmatrix} 3 & 8 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 9 & 0 \\ 0 & 0 \end{bmatrix} \right)$$

$$= \begin{bmatrix} -3 & 0 \\ 2 & 0 \end{bmatrix} \times \begin{bmatrix} 12 & 8 \\ 0 & 0 \end{bmatrix} \Rightarrow \begin{bmatrix} -36 & -24 \\ 24 & 16 \end{bmatrix}$$

$P7 = (a - c)\,(e + f)$

$$= \left(\begin{bmatrix} 1 & 2 \\ 0 & 3 \end{bmatrix} - \begin{bmatrix} 0 & 5 \\ 0 & 0 \end{bmatrix} \right) \left(\begin{bmatrix} 2 & 4 \\ 4 & 8 \end{bmatrix} + \begin{bmatrix} 3 & 0 \\ 9 & 0 \end{bmatrix} \right)$$

$$= \begin{bmatrix} 1 & -3 \\ 0 & 3 \end{bmatrix} \times \begin{bmatrix} 5 & 4 \\ 13 & 8 \end{bmatrix} \Rightarrow \begin{bmatrix} -34 & 20 \\ 39 & 24 \end{bmatrix}$$

So, now, calculating Matrix C as following:

$$P5 + P4 - P2 + P6 = \begin{bmatrix} 63 & 36 \\ 12 & 24 \end{bmatrix} + \begin{bmatrix} 4 & 16 \\ 0 & 0 \end{bmatrix} - \begin{bmatrix} 18 & 0 \\ 18 & 0 \end{bmatrix} + \begin{bmatrix} -36 & -24 \\ 24 & 16 \end{bmatrix}$$

$$= \begin{bmatrix} 13 & 28 \\ 18 & 40 \end{bmatrix}$$

$$P1 + P2 = \begin{bmatrix} 12 & 0 \\ 27 & 0 \end{bmatrix} + \begin{bmatrix} 18 & 0 \\ 18 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 30 & 0 \\ 43 & 0 \end{bmatrix}$$

$$P3 + P4 = \begin{bmatrix} 28 & 56 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 4 & 16 \\ 0 & 0 \end{bmatrix} \Rightarrow \begin{bmatrix} 42 & 72 \\ 0 & 0 \end{bmatrix}$$

$$P1 + P5 - P3 - P7 = \begin{bmatrix} 12 & 0 \\ 27 & 0 \end{bmatrix} + \begin{bmatrix} 63 & 36 \\ 12 & 24 \end{bmatrix} - \begin{bmatrix} 28 & 56 \\ 0 & 0 \end{bmatrix} - \begin{bmatrix} -34 & 20 \\ 39 & 24 \end{bmatrix}$$

$$= \begin{bmatrix} 81 & -40 \\ 0 & 0 \end{bmatrix}$$

So,

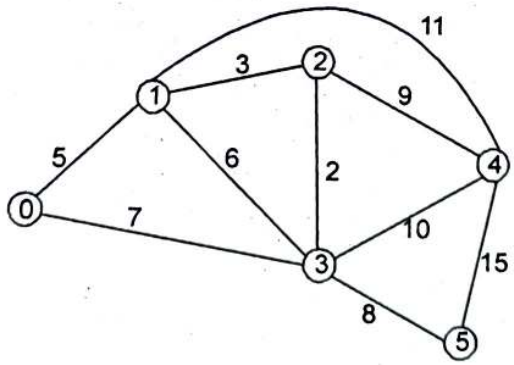
$$C \text{ is as } \left[\begin{array}{c|c} P5+P4-P2+P6 & P1+P2 \\ \hline P3+P4 & P1+P5-P3-P7 \end{array}\right]$$

$$C = \left[\begin{array}{cc|cc} 13 & 28 & 30 & 0 \\ 18 & 40 & 43 & 0 \\ \hline 42 & 72 & 81 & -40 \\ 0 & 0 & 0 & 0 \end{array}\right]$$

Q.4. (*a*) Explain any one algorithm for generating spanning tree with minimum cost. Write code/algorithm neatly. **(6.25)**

Ans. 4(a) Refer Q.No.3 of Important Questions (Pg. No. 6)

Q.4. (*b*) Using same algorithm, find minimum spanning tree for the following: **(6.5)**

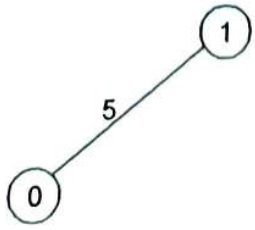


Ans. In prim's algorithm, first we initialize the priority queue Q to contain all the vertices and the keys of each vertex to ∞ except the root, whose key is set to 0.

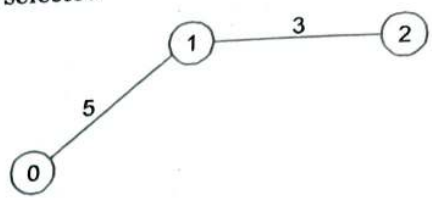
Let (o) is the root vertex.

So,

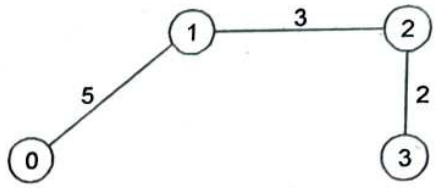
Step 1: edge {0,1} is selected



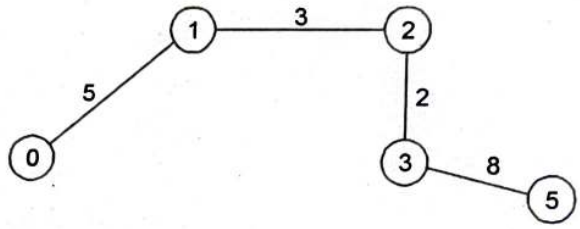
Step 2: edge {1,2} is selected



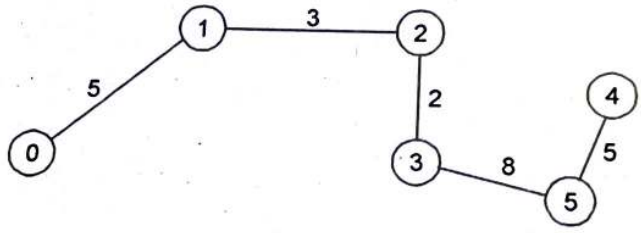
Step 3: edge {2,3} is selected



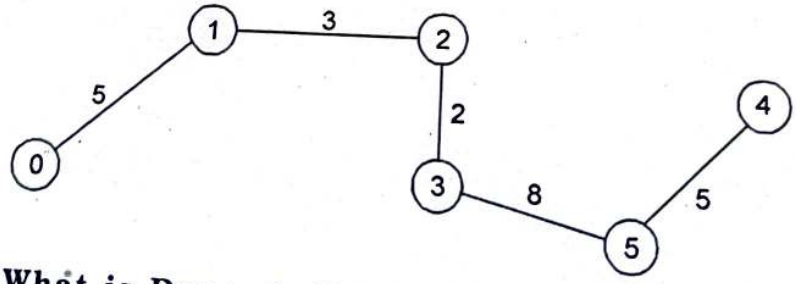
Step 4: edge {3,5} is selected



Step 5: edge {5,4} is selected



So, final spanning tree is



Q.5. (a) What is Dynamic Programming Paradigm? Explain its characterestics. **(6.25)]**

Ans. Dynamic programming is typically applied to **optimization problems**. In such problems there can be many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value. We call such a solution *an* optimal solution to the problem, as opposed to *the* optimal solution, since there may be several solutions that achieve the optimal value.

The development of a dynamic-programming algorithm can be broken into a sequence of four steps.

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.

Like divide and conquer, divide the problem into 2 or more optimal parts recursively this helps to define what the solution will look like.

3. Compute the value of an optimal solution in a bottom-up fashion(starting with the smallest subproblem).
4. Construct an optimal solution from computed values of smaller subproblems.

Dynamic programming is best applied to problems having these two characteristics:

- **Optimal substructure:** Consider a problem P, broken down into two subproblems, P_1 and P_2. We must be able to efficiently combine optimal solutions to P_1 and P_2 into an optimal solution to P. The structure of an optimal solution must contain optimal solutions to the recursive subproblems.
- **Overlapping subproblems:** Having defined in the second step the value of an optimal solution, a first attempt at solving the problem may be to simply implement a solution as a recursive algorithm. Dynamic programming requires that the recursive sub-solutions be computed many times, i.e. that the straight recursive solution does a lot of unnecessary computation. Dynamic programming works by eliminating this redundancy.

Characteristics of dynamic programming:

There are a no of characteristics that are to all dynamic programming problems:

Theses are:

1.The problem can be divided into no. of stages. With a decision required at each stage.

For example, in the shortest path problem, these were defined by the structure of the graph the decision was where to go next.

2. Each stage has a number of states associated with it. The states for the shortest path problem was the node reached.

3. The decision at one stage transforms one state into a state in the next stage. The decision of where to go next defined where we arrived in the next stage.

4.Given the current state, the optimal decision for each of the remaining states does not depend on the previous states or decisions. In the shortest path problems, it was not necessary to know how we got to a node, only that we did.

5.There exists a recursive relationship that identifies the optimal decision for stage j, given that stage j + 1 has already been solved.

6.The final stage must be solvable by itself.

Suppose you want to compute the nth Fibonacci number, F_n. The optimal solution to the problem is simply F_n (this is a somewhat contrived use of the word "optimal" to illustrate dynamic programming :-).

Recall that the Fibonacci numbers are defined: $F(n)$ =

1, if n = 1 or 2,

$F_{n-2} + F_{n-1}$ otherwise.

So the Fibonacci numbers are:

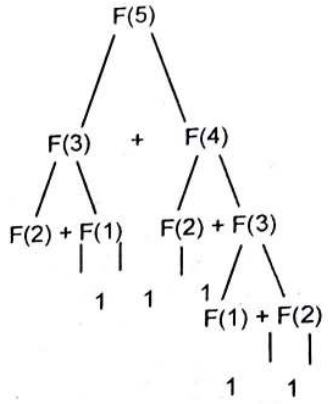
n : 1 2 3 4 5 6 7 8 9 10 ...

F_n: 1 1 2 3 5 8 13 21 34 55 ...

Thus we have completed steps 1 and 2 for designing a dynamic programming algorithm to compute F_n. We are prepared to write a non-dynamic algorithm as simple

```
F (n)
    if (n = 1) or (n = 2) then return 1
                  return F(n–2) + F(n–1)
```

This is not an efficient algorithm for computing F_n. Let's look at the recursive calls made for F_4:



Note that F_2 is computed three times, and F_3 is computed twice. Each recomputation incurs extra recusive work that has already been done elsewhere. Note also the shape of this diagram; it looks like a binary tree of calls. We can see the height of the tree by going from the root to the rightmost leaf. The height is $\Theta(n)$.

This exponential behavior is very inefficient.

We can do much better with dynamic programming. Our problem satisfies the optimal substructure property: each solution is the sum of two other solutions.

Using a dynamic programming technique called *memoization*, we can make the recursive algorithm much faster. We assume there is an array A of integers whose first and second elements have been initialized to 1, and there is an integer called *unknown* initially two, that keeps track of the index of the least Fibonacci number whose value is not known:

```
F (n)
    if n < unknown then return A[n]
                  A[n] = F(n-2) + F(n-1)
                  return A[n]
```

This algorithm is very similar to the previous one, but it uses the array A as a sort of "scratch" area to record previous results, rather than recompute compute them.

What is the running time of this recursive algorithm? If F_i is computed, it is stored in the array and never recomputed, so the algorithm basically traces a path from root to rightmost leaf of the tree, adding up all the results at each level in one addition. Thus the algorithm runs in time – (n). This is much better than the exponential-time algorithm.

Q.5. (b) Solve any problem using dynamic programming paradigm. In the example clearly show how it meet the dynamic paradigm criterion. (6.25)

Ans. Moving on a grid example the following is a very simple, although somewhat artificial, example of a problem easily solvable by a dynamic programming algorithm.

Imagine a climber trying to climb on top of a wall. A wall is constructed out of square blocks of equal size, each of which provides one handhold. Some handholds are

more dangerous/complicated than other. From each block the climber can reach three blocks of the row right above: one right on top, one to the right and one to the left (unless right or left are no available because that is the end of the wall). The goal is to find the least dangerous path from the bottom of the wall to the top, where danger rating (cost) of a path is the sum of danger ratings (costs) of blocks used on that path.

We represent this problem as follows. The input is an n × m grid, in which each cell has a positive cost C(i, j) associated with it. The bottom row is row 1, the top row is row n. From a cell (i, j) in one step you can reach cells (i + 1, j –1) (if j > 1), (i + 1, j) and (i + 1, j + 1) (if j < m).

Here is an example of an input grid. The easiest path is high-lighted. The total cost of the easiest path is 12. Note that a greedy approach - choosing the lowest cost cell at every step — would not yield an optimal solution: if we start from cell (1,2) with cost 2, and choose a cell with minimum cost at every step, we can at the very best get a path with total cast 13.

Grid example.

2	8	9	5	8
4	4	6	2	3
5	7	5	6	1
3	2	5	4	8

Step 1. The first step in designing a dynamic programming algorithm is defining an array to hold intermediate values. For $1 < i < n$ and $1 < j < m$, define $A(i,j)$ to be the cost of the cheapest (least dangerous) path from the bottom to the cell $(i. j)$. To find the value of the best path to the top, we need to find the minimal value in the last row of the array, that is,

$$\min_{1 \le j \le m} A(n,j).$$

Step 2. This is the core of the solution. We start with the initialization. The simplest way is to set $A(1,j) = C(1,j)$ for $1 < j < m$. A somewhat more elegant way is to make an additional zero row, and set $A(0. j) = 0$ for $1 < j < m$.

There are three cases to the recurrence: a cell might be in the middle (horizontally), on the leftmost or on the rightmost sides of the grid. Therefore, we compute $A(i. j)$ for $1 < i < n$, $1 < j < m$ as follows:

A (i,g) for the above grid.

∞	0	0	0	0	0	∞
∞	3	2	5	4	8	∞
∞	7	9	7	10	5	∞
∞	11	11	13	7	8	∞
∞	13	19	16	12	15	∞

$$A(i, j) = \begin{cases} C(i,j) + \min\{A(i-1,j-1), A(i-1,j)\} & \text{if } j = m \\ C(i,j) + \min\{A(i-1,j), A(i-1,j+1)\} & \text{if } j = 1 \\ C(i,j) + \min\{A(i-1,j-1)A(i-1,j), A(i-1,j+1)\} & \text{if } j \neq 1 \text{ and } j \neq m \end{cases}$$

We can eliminate the cases if we use some extra storage. Add two columns 0 and $m+1$ and initialize them to some very large number ∞; that is, for all $0 \le i \le n$ set .A(i,0) $= A(i.m + 1) = \infty$. Then the recurrence becomes, for $1 \le i \le n$. $1 \le j' \le ,m$.

$A(i,j) = C(i,j) + min\{A(i-1, j-1), A(i-1j), A(i-1j+1)\}$

Step 3. Now we need to write a program to compute the array; call the array B. Let *INF* denote some very large number, so that /INF > c for any c occurring in the program (for example, make INF the sum of all costs +1).

```
//initialization
for j = 1 to m do
    B (0, j) ← 0
for i = 0 to n do
    B (i, m+1) ← INF
//recurrence
for i = 1 to n do
    for j = 1 to m do
        B (i,j) ← C(i, j) + min {B(i-1, j – 1),B (i-1,j), B (i – 1, j + 1)}
//finding the cost of the least dangerous path
cost ← INF
for j = 1 to m do
    if (B (n,j) < cost) then
        cost ← B (n,j)
return cost
```

Step 4. The last step is to compute the actual path with the smallest cost. The idea is to retrace the decisions made when computing the array. To print the cells in the correct order, we make the program recursive. Skipping finding j such that $A(n,j)$ = cost the first call to the rpgram will printOpt (n,j).

```
Procedure PrintOpt (i j)
        if (i = 0) then return
else if (B(i, j) = C (i,j) + B(i-1 j–1) then PrintOpt(i-1,j-1)
else if (B (i,j) = C ( i, j) + B (i – 1,j)) then Print Opt (i – 1, j)
else if (B(i,j) = C (i,j) + B (i – 1, j + 1) then print (i – 1, j +1)
end if
put "Cell" (i, j)
end PrintOpt
```

Q.6. (a) Write the Huffman's Algorithm. Explain, Discuss its applications. **(6.25)**

Ans. Data can be encoded efficiently using Huffman codes. Each character is represented as a byte of 8 bits when characters are coded using standard code much as ASCII. It can be seen that characters in such codes have fixed-length codeward representation. These codes provide an effective way in which a string can break into sequency of characters, and later these characters can be accessed individually.

Consider an example, here we want to encode the string over 4-character alphabet,

$$S = (a, b, c, d)$$

for this we could use the following 3-bit fixed length code.

Character	*a*	*b*	*c*	*d*
Fixed-Length code	000	001	010	111

If given a string *"aabbabcabddadbcaa",it* can be easily encoded by considering individual character, and by replacing these with respective binary codewords.

a	b	b	a	b	c	a	b	d	d	a	d	b	c	a
000	001	001	000	001	010	000	001	111	111	000	111	001	010	000

The final 48 character binary string would be

"000 000 001 001 000 001 010 000 001 111 111 000 111 001 010 000"

Let us assume that we know the relative probabilities of the occurrence of these characters in advance. The character with higher probability of occur rence use fewer bits for representation while the characters having lower probability of occurrence use more bits for representation.

For example, we can design a variable length code for the string whose characters have varying probability of occurrence.

Character	a	b	c	d
Probability of occurrence	0.50	0.24	0.20	0.24
Variable-length codeward	0	110	10	11

It is not necessary that codeword follows some order. Having the variable length codeword the same string can be encoded as

a	a	b	b	a	b	c	a	b	d	d	a	d	b	c	a
0	0	110	110	0	110	10	0	110	111	111	0	111	110	10	0

The final 34 character binary string would be

"00 110 110 0 110 100 110 111 111 0 111 110 100"

(A): Prefin (free) Codes: By this we mean that no codeword is a prefix of codeword of another character. The examples for variable-length codeword is based on prefix (free) codes.

Let the two character share common prefix,

$$a = (01)$$
$$b = (0101).$$

and we have the following string:

$S = (0101)$, then we might think that the string will be decoded as "*aa*" or "*b*".

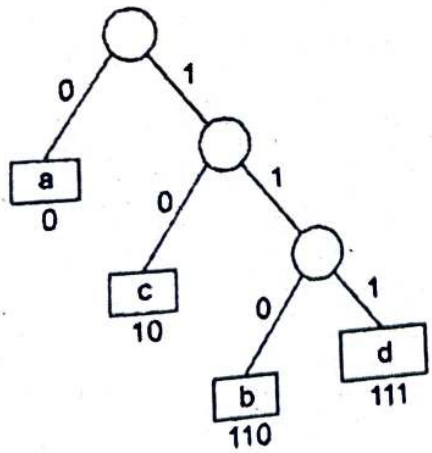


Fig. Corresponding binary tree for prefix codes.

By the property of the prefix (free) code we know that no codeword is prefix of codeword of another character, thus we decode the above string by choosing some longer codeword here "*b*"

It is noticeable that any binary prefix coding can be converted into a binary tree. For this tree leaves represent the codewords, and left branch indicates "0" and right branch indicates "1".

The length of the codeword is equal to the depth of the binary tree. The code for the above example in a prefix code arid the corresponding binary tree is shown in fig. A

For decoding a given prefix code, we have to traverse the corresponding binary tree from root to leaf depending upon the input character. Once a leaf is reached then we output the corresponding character, and the process continues from the from the root for rostof the input characters.

(B) Greedy Strategy: According to Huffman algorithm we are building a bottom up tree, starting from the leaf. Initially there are *n* singleton trees in the forest, as each tree is a leaf, The greedy strategy says that first find the two trees having the minimum probabilities. Then merge these two trees in a single tree where the probability of this tree is the total sum of two merged trees. The whole process is **repeated** until there is only one tree in the forest

Q.6. (b) Generate Huffman's tree for the following data and obtain Huffman code for each character. (6.25)

Character	A	B	C	D	E	F
Probability	0.5	0.35	0.5	0.1	0.4	0.2

Ans.

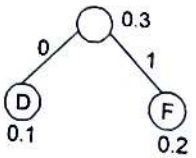
Character	A	B	C	D	E	F
Probability	0.5	0.35	0.5	0.1	0.4	0.2

arrange in dec. order of probability

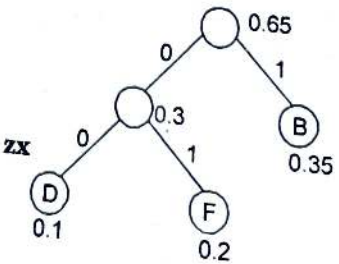
Character	Probability
A	0.5
C	0.5
E	0.4
B	0.35
F	0.2
D	0.1

Huffman tree is as following:

Step I:

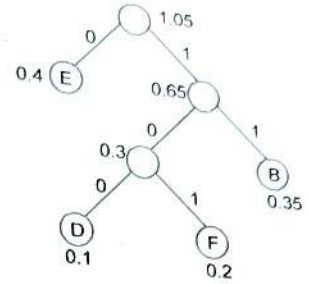


Step 2:

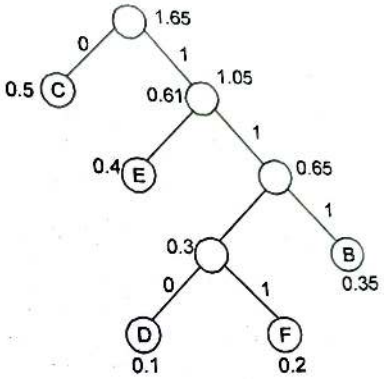


zx

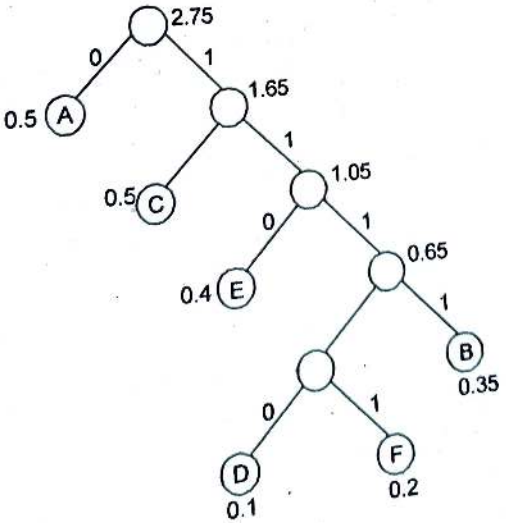
Step 3:



Step 4:



Step 5:



Codes for all characters are

A = 0
B : 1111
C : 10
D : 11100
E : 110
F : 11101

Q.7. (a) Define P, NP and NPC problems . (6.25)

Ans. Refer Q.No.3 (b) Second Term Exam 2015 (Page No.: 15-2015)

Q.7. (b) Give atleast 3 examples of NP complete problems ? (6.25)

Ans. Refer Q.No.2 Important Questions (Page No.: 4)

Q.8. Write short note on following 6.25*2=12.5)

Q.8. (a) Khuth - Mories Pratt Algorithm

Ans. Refer Q.No.9 (b) End Term Exam 2017 (Page No.: 24-2017)

Q.8. (b) OBST problems ?

Ans. Refer Q.No.1 Important Questions (Page No.: 3)

MID TERM EXAMINATION
FIFTH SEMESTER [B.TECH.] [SEPT. 2016]
ALGORITHM DESIGN AND ANALYSIS (ETCS–301)

Time : 1.5 hrs. M.M. : 30

Note: Attempt any three questions including Q.No.1 which is compulsory

Q.1. (a) Distinguish between O(big Oh) and o(little Oh) notations.

Ans: O(big Oh) notation: If $f(s)$ and $g(s)$ are functions of a real or complex variable s and S is an arbitrary set of (real or complex) numbers s (belonging to the domains of f and g), we write f(s) = O(g(s)) (s ∈ S), if there exists a constant c such that $|f(s)| \leq c|g(s)|$ ($s \in S$). To be consistent with our earlier definition of "big oh" we make the following convention: If a range is not explicitly given, then the estimate is assumed to hold for all sufficiently large values of the variable involved, i.e., in a range of the form $x \geq x_0$, for a suitable constant x_0.

Little-oh notation(o): Asymptotic upper bound provided by O-notation may not be asymptotically tight.$o(g(n))$ = { $f(n)$: for any +ve $c > 0$,if a constant $n_0>0$ such that $0<=f(n)<cg(n)$ for all $n>=n_0$}

The function $f(n)$ becomes insignificant relative to g(n) as n approaches infinity

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

Q.1. (b) Define subtract and conquer Master Theorem

Ans: Subtract and Conquer Master Theorem. Suppose we have a recurrence relation of the form

$$T(n) = aT(n-b) + f(n) \text{ when } n>1$$
$$= c \text{ when } n<=1$$

Here $f(n) = O(n^d)$

Therefore, $T(n) = aT(n-b) + O(n^d)$ when n>1

Now to find order of complexity for this kind of recurrence relation we can use subtract and conquer Master theorem. It states as follows:

$$T(n) = O(n^d) \quad \text{if } a<1$$
$$= O(n^{d+1}) \quad \text{if } a = 1$$
$$= O(n^d a^{n/b}) \quad \text{if } a>1$$

This is very helpful for finding order of complexity for recurrence relation of the form $T(n) = aT(n-b)+f(n)$ Generally we come across Master Theorem but that is not helpful for this kind of recurrence.

Q.1. (c) Prove that $(n + a)^b = O(n^b)$

Ans: It means, we have to show that

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

$$(n+a)^b = (n+a)(n+a)(n+a)\ldots(n+a) \text{ b times}$$
$$= n^b + k_1 n^{b-1} + k_2 n^{b-2} + \ldots + k_{b-1} n^1 + k_b$$

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

Q.2. c) Explain Overlapping Subproblems.

Ans: A problem is said to have overlapping subproblems if the problem can be broken down into subproblems which are reused several times or a recursive algorithm for the problem solves the same subproblem over and over rather than always generating new subproblems.

For example, the problem of computing the Fibonacci sequence exhibits overlapping subproblems. The problem of computing the nth Fibonacci number F_n can be broken down into the subproblems of computing F_{n-1} and F_{n-2}, and then adding the two. The subproblem of computing F_{n-1} can itself be broken down into a subproblem that requires computing F_{n-2}. Therefore the computation of F_{n-2} is reused, and the Fibonacci sequence thus exhibits overlapping subproblems.

A naive recursive approach to such a problem generally fails due to an exponential complexity. If the problem also shares an optimal substructure property, dynamic programming is a good way to work it out.

" simple recursive program for Fibonacci numbers "

```
int fib(int n)
{
  if (n <= 1)
    return n;
  return fib(n-1) + fib(n-2);
}
```

Q.2. d) What are the key features of Dynamic Programming?

Ans: 1. The problem can be divided into stages with a decision required at each stage in the optimal subproblem. The stages were the allocations to a single plant. The decision was how much to spend in the current year problem. They were defined by the structure of the graph. The decision was where to go next.

2. Each stage has a number of states associated with it. The states for the capital budgeting problem corresponded to the amount spent at that point in time. The states for the shortest path problem was the node reached.

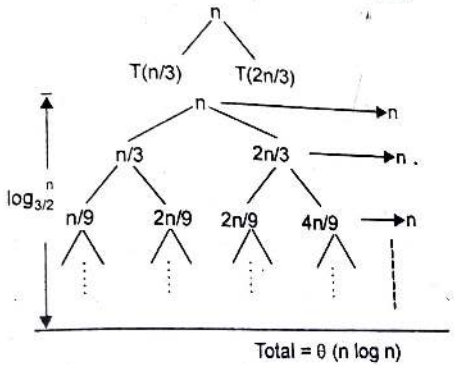
3. The decision at one stage transforms one state into a state in the next stage. The decision of how much to spend gave a total amount spent for the next stage. The decision of where to go next defines you arrive in the next stage.

4. Given the current state, the optimal decision for each of the remaining states does not depend on the previous states or decisions. In the budgeting problem, it is not necessary to know how the money was spent in previous stages, only how much was spent. In the path problem, it was not necessary to know how you got to a node, only that you did.
5. There exists a recursive relationship that identifies the optimal decision for stage j given that stage j+1 has already been solved.
6. The final stage must be solvable by itself.

Q.2. Solve the following reccurence relations: **(4×2.5)**

(a) T(n) = T(n/3) + T(2n/3) +n (using reccurence tree)

Ans: The given recurrence has the following recursion tree.



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

$$n \rightarrow 2/3n \rightarrow (2/3)^2 n$$

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

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

$T(n) = n+n+n+$ ---------$+\log_{3/2} n$ times

$= O(n\log n)$

Q.2. (b) $T(n) = 4T(n/2) + n^3$ (using Master Method)

Ans: Here $a = 4\ b = 2\ f(n) = n^3$

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

$F(n) = n^3$ using master method $T(n) = \theta(f(n)\ \log n) = \theta(n^3 \log n)$

Q.2. (c) T(n) =2T(⌊n/2⌋) + n (using substitution method)

Ans: $T(n) = 2T(\lfloor n/2 \rfloor)+n$

Let's guess that the solution is $T(n) = O(n\log_2 n)$

So, $T(n) \leq cn\log_2 n$

$T(n/2) \leq cn\log_2 n/2$

Substituting into recurrence, it yields

$T(n) = 2T(\lfloor n/2 \rfloor)+n$

$\leq 2T(n/2)+n$

$\leq 2cn\log_2 n/2+n$

$\leq cn\log_2 n/2+n$

$\leq cn\log_2 n - c\log_2 2+n$

$\leq cn\log_2 n - c+n$

$\leq cn\log_2 n$

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

Q.2. (d) T(n) = 1 if n =1
2T(n/2) + n if n>1 **(using iteration method)**

Ans:

$$T(n) = 2\,T(n/2)$$
$$= 2[2\,T(n/4)] = 4T(n/4)$$
$$= 2^3T(n/8)$$

In general $T(n) = 2^i\,T(n/i)$

Put $i = n/2$ we get $T(n) = 2^{n/2}T(1) = 2^{n/2}$

Q.3. (a) Write Insertion Sort algorithm.Explain best case and worst case time complexity of Insertion Sort Algorithm (6)

Ans: Insertion–Sort (A)

1. for $j \leftarrow 2$ to length [A]
2. do key $\leftarrow$ A [j]
3. Insert A[j] into the sorted sequence A[1..j – 1]
4. $i \leftarrow j - 1$
5. while $i > 0$ and A [i] > key
6. do A[i + 1] $\leftarrow$ A [i]
7. $i \leftarrow i - 1$.
8. A[i + 1] $\leftarrow$ key

Analysis of Insertion sort

The time taken by INSERTION - SORT procedure depends on the input. We start by presenting the INSERTION - SORT procedure with the "cost" of each statement and the number of times each statement is executed.

Insertion-Sort (A)	Cost	times
1. for $j \leftarrow 2$ to length [A]	c_1	n
2. do key $\leftarrow$ A[j]	c_2	$n-1$
3. insert A[j] into the sorted sequence A[$i..j-1$]	0	$n-1$
4. $i \leftarrow j-1$	c_4	$n-1$
5. while $i > 0$ and A[i] > key	c_5	$\sum_{j=2}^{n} t_j$
6. do A[$i+1$] $\leftarrow$ A [i]	c_6	$\sum_{j=2}^{n}(t_j-1)$
7. $i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8. A[$i+1$] $\leftarrow$ key	c_8	$n-1$

...running time of the algorithm is the sum of running times for each statements ...

...compute $T(n)$ the running time, we sum the product of the cost and times columns ...

$$= c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j - 1) + c_7\sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

...best case occurs if the array is already sorted. For each $j = 2,3,...n$, we then find that ...key in line 5 when $i = j - 1$.

...thus $t_j = 1$ for $j = 2,3,...n$

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \\ &= an + b, \text{ where } a \text{ and } b \text{ are constants.} \\ &= \text{linear function of } n = O(n). \end{aligned}$$

...worst case, the array is in reverse order. We must compare each element $A[j]$ with ...element in the entire sorted subarray $A[1..j-1]$ and $t_j = j$ for $j = 2,3,...n$.

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5$$

$$\sum_{j=2}^{n} j + c_6\sum_{j=2}^{n}(j-1) + c_7\sum_{j=2}^{n}(j-1) + c_8(n-1)$$

$$= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$

$$\left(\because \sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \text{ and } \sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}\right)$$

$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8)$$

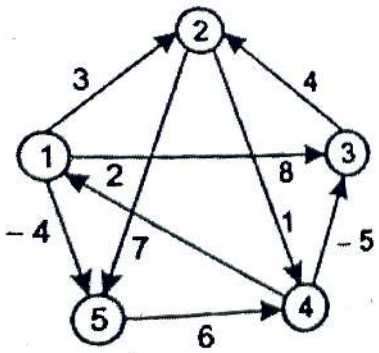
$= an^2 + bn + c$ for a, b and c are constant

= quadratic function of n

$= O(n^2)$.

Q.3. (b) Find all pairs shortest path for the tfollowing graph using Floyd ...rshall Algorithm.

Ans.



...rst case time (6)

...We start by ...ent and the

Ans.

$$d_{ij}^{(k)} = \min\left[d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right]$$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} \text{if } d_{ij}^{(k-1)} \le d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} \text{ if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \pi^{(0)} = \begin{bmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \pi^{(1)} = \begin{bmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \pi^{(2)} = \begin{bmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \pi^{(3)} = \begin{bmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \pi^{(4)} = \begin{bmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{bmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \pi^{(5)} = \begin{bmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{bmatrix}$$

Q.4. (a) Find the optimal parenthesization of a matrix chain product whose sequence of dimensions are,<4,10,3,12,20,7> 2015-16 (SI)

Ans: The matrices have sizes $4*10, 10*3, 3*12, 12*20, 20*7$. We need to compute $M[i,j]$, $0<=i,j<=5$. We know $M[i,i] = 0$ for all i.

	1	2	3	4	5	
1	0					1
2		0				2
3			0			3
4				0		4
5					0	5

We proceed, working away from the diagonal. We compute the optical solutions for products of 2 matrices.

	1	2	3	4	5	
1	0	120				1
2		0	360			2
3			0	720		3
4				0	1680	4
5					0	5

{4, 10, 3, 12, 20, 7}

Now products of 3 matrices

$$M[1,3] = \min \begin{cases} M[1,2] + M[3,3] + p_0 p_2 p_3 = 120 + 0 + 4.3.12 = 264 \\ M[1,1] + M[2,3] + p_0 p_1 p_3 = 0 + 360 + 4.10.12 = 840 \end{cases} = 264$$

$$M[2,4] = \min \begin{cases} M[2,3] + M[4,4] + p_1 p_3 p_4 = 360 + 0 + 10.12.20 = 2760 \\ M[2,2] + M[3,4] + p_1 p_2 p_4 = 0 + 720 + 10.3.20 = 1320 \end{cases} = 1320$$

$$M[3,5] = \min \begin{cases} M[3,4] + M[5,5] + p_2 p_4 p_5 = 720 + 0 + 3.20.7 = 1140 \\ M[3,3] + M[4,5] + p_2 p_3 p_5 = 0 + 1680 + 3.12.7 = 1932 \end{cases} = 1140$$

	1	2	3	4	5	
1	0	120				1
2		0	360			2
3			0	720		3
4				0	1680	4
5					0	5

⇒

	1	2	3	4	5	
1	0	120	264			1
2		0	360	1320		2
3			0	720	1140	3
4				0	1680	4
5					0	5

Now products of 4 matrices

$$M[1,4] = \min \begin{cases} M[1,3] + M[4,4] + p_0 p_3 p_4 = 264 + 0 + 4.12.20 = 1224 \\ M[1,2] + M[3,4] + p_0 p_2 p_4 = 120 + 720 + 4.3.20 = 1080 = 1080 \\ M[1,1] + M[2,4] + p_0 p_1 p_4 = 0 + 1320 + 4.10.20 = 2120 \end{cases}$$

$$M[2,5] = \min \begin{cases} M[2,4] + M[5,5] + p_1 p_4 p_5 = 1320 + 0 + 10.12.7 = 2720 \\ M[2,3] + M[4,5] + p_1 p_3 p_5 = 360 + 1680 + 10.12.7 = 2880 = 1350 \\ M[2,2] + M[3,5] + p_1 p_2 p_5 = 0 + 1140 + 10.3.7 = 1350 \end{cases}$$

	1	2	3	4	5	
1	0	120	264			1
2		0	360	1320		2
3			0	720	1140	3
4				0	1680	4
5					0	5

$\Rightarrow$

	1	2	3	4	5	
1	0	120	264	1080		1
2		0	360	1320	1350	2
3			0	720	1140	3
4				0	1680	4
5					0	5

Now products of 5 matrices

$$M[1,5] = \min \begin{cases} M[1,4] + M[5,5] + p_0 p_4 p_5 = 1080 + 0 + 4.20.7 = 1544 \\ M[1,3] + M[4,5] + p_0 p_3 p_5 = 264 + 1680 + 4.12.7 = 2016 \\ M[1,2] + M[3,5] + p_0 p_2 p_5 = 120 + 1140 + 4.3.7 = 1344 \\ M[1,1] + < [2,5] + p_0 p_1 p_5 = 0 + 1350 + 4.10.7 = 1630 \end{cases} = 1344$$

	1	2	3	4	5	
1	0	120	264	1080		1
2		0	360	1320	1350	2
3			0	720	1140	3
4				0	1680	4
5					0	5

$\Rightarrow$

	1	2	3	4	5	
1	0	120	264	1080	1344	1
2		0	360	1320	1350	2
3			0	720	1140	3
4				0	1680	4
5					0	5

To print the optimal parenthesization, we use the PRINT-OPTIMAL–PARENS procedure.

Print-Optimal-Parens (s, i, j)

1. if i = j
2. then print "A_i"
3. else print "("
4. Print-Optimal-Parens (s, i, s[i, j]
5. Print-Optimal-parens (s, s [i,j]+1,j)
6. print ")"

Now for optimal parenthesization, Each time we find the optimal value for M[i, j] we so store the value of k that we used, If we did this for the example, we would get

	1	2	3	4	5	
1	0	120/1	264/2	1080/2	1344/2	1
2		0	360/2	1320/2	1350/2	2
3			0	720/3	1140/4	3
4				0	1680/4	4
5					0	5

The k value for the solution is 2, so we have $((A_1 A_2)(A_3 A_4 A_5))$. The first half is done. The optimal solution for the second half comes from entry M[3,5]. The value of k here is 4, so now we have $((A_1 A_2)(A_3 A_4 A_5))$. Thus the optimal solution is to parenthesize $((A_1 A_2)((A_3 A_4) A_5)))$.

Q.4. (b) Determine LCS of <1,0,0,1,0,1,0,1> and<0,1,0,1,1,0,1,1,0> 2016–9

Ans. Lex X = <1,0,0,1,0,1,0,1> and Y = <0,1,0,1,1,0,1,1,0>. We know (5)

$$C[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ c[i-1,j-1]+1 & \text{if } i,j>0 \text{ and } x_i = y_j \\ \max(c[i,j-1], c[i-1,j] & \text{if } i,j>0 \text{ and } x_i \neq y_j \end{cases}$$

We are looking for c[8, 9]. The following tables is built.

x = (1,0,0,1,0,1,0,1) y = (0,1,0,1,1,0,1,1,0)

		0	1	2	3	4	5	6	7	8	9
		y_j	0	1	0	1	1	0	1	1	0
0	x_i	0	0	0	0	0	0	0	0	0	0
1	1	0	↑ 0	↖ (1)	← 1	↖ 1	↖ 1	← 1	↖ 0	↖ 1	← 1
2	0	0	↖ 1	↑ 1	↖ (2)	← 2	← 2	↖ 2	← 2	← 2	↖ 2
3	0	0	↖ 1	↑ 1	↖ 2	↑ 2	↑ 2	↖ (3)	← 3	← 3	↖ 3
4	1	0	↑ 1	↖ 2	↑ 2	↖ 3	↖ 3	↑ 3	↖ (4)	↖ 4	← 4
5	0	0	↖ 1	↑ 2	↖ 3	↑ 3	↑ 3	↖ 4	↑ 4	↑ 4	↖ 5
6	1	0	↑ 1	↖ 2	↑ 3	↖ 4	↖ 4	↑ 4	↖ 5	↖ (5)	↑ 5
7	0	0	↖ 1	↑ 2	↖ 3	↑ 4	↑ 4	↖ 5	↑ 5	↑ 5	↖ (6)
8	1	0	↑ 1	↖ 2	↑ 3	↖ 4	↖ 5	↑ 5	↖ 6	↖ 6	↑ 6

From the table, we can deduce that LCS = 6. There are several such sequences, for instance (1,0,0,1,1,0), (0,1,0,1,0,1) and (0,0,1,1,0,1).

FIFTH SEMESTER [B.TECH.] [DEC. 2016]
END TERM EXAMINATION
ALGORITHM DESIGN AND ANALYSIS (ETCS–301)

Time : 3 hrs. **M.M. : 75**

Note: *Attempt any five questions including Q.No. 1 which is compulsory. Select one question from each unit.*

Q.1. (a) Define little omega and little oh notation. (2.5)

Ans: ω – Notation: We use **ω-notation** to denote a lower bound that is not asymptotically tight. Formally, however, we define ***ω(g(n))*** (little-omega of g of n) as the set ***f(n)*** = ***ω(g(n))*** for any positive constant **C > 0** and there exists a value $n_0>0$, such that $0 \le c.g(n)<f(n)$.

For example, $n^2/2=\omega(n)$, but $n^2/2 \neq \omega(n^2)$. The relation $f(n)=\omega(g(n))$ implies that the following limit exists

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

That is, ***f(n)*** becomes arbitrarily large relative to ***g(n)*** as **n** approaches infinity.

Example

Let us consider same function, $f(n) = 4.n^3+10.n^2+5.n+1$

Considering $g(n)=n^2$,

$\lim n\to\infty((4.n^3+10.n^2+5.n+1)/n^2) = \infty$

Hence, the complexity of $f(n)$ can be represented as $o(g(n))$, i.e. $\omega(n^2)$

O – Notation

The asymptotic upper bound provided by **O–notation** may or may not be asymptotically tight. The bound $2.n^2 = O(n^2)$ is asymptotically tight, but the bound $2.n = O(n^2)$ is not.

We use **o-notation** to denote an upper bound that is not asymptotically tight.

We formally define ***o(g(n))*** (little-oh of g of n) as the set f(n) = o(g(n)) for any positive constant $c>0$ and there exists a value $n0>0$, such that $0 \le f(n) \le c.g(n)$.

Intuitively, in the **o–notation**, the function ***f(n)*** become insignificant relative to ***g(n)*** as **n** approaches infinity; that is,

$(\lim n\to\infty(f(n)g(n))=0)$

Example

Let us consider the same function, $f(n)=4.n^3+10.n^2+5.n+1$

Considering $g(n)=n^4$,

$\lim n\to\infty((4.n^3+10.n^2+5.n+1)/n^4)=0$

Hence, the complexity of $f(n)$ can be represented as $o(g(n))$, i.e. $o(n^4)$.

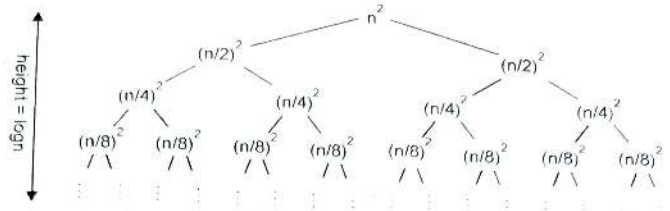
Q.1. (b) What is recursion tree method? **(2.5)**

Ans: A *recursion tree* is useful for visualizing what happens when a recurrence is iterated. It diagrams the tree of recursive calls and the amount of work done at each call.

For instance, consider the recurrence

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

The recursion tree for this recurrence has the following form:

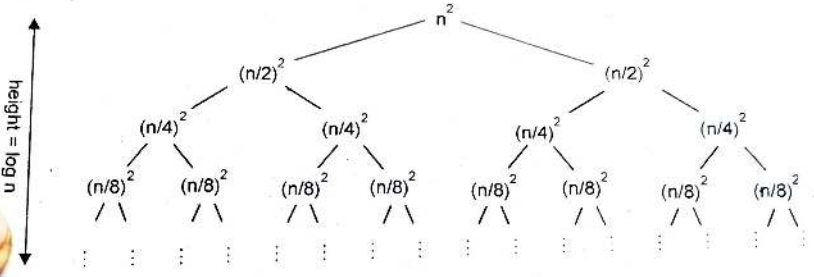


A recursion tree is useful for visualizing what happens when a recurrence is iterated. It diagrams the tree of recursive calls and the amount of work done at each call.

For instance, consider the recurrence

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

The recursion tree for this recurrence has the following form:



Q.1. (c) Write a short note on memoization. **(2.5)**

Ans: memoization or **memoisation** is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again. Memoization has also been used in other contexts (and for purposes other than speed gains), such as in simple mutually recursive descent parsing in a general top-down parsing algorithm that accommodates ambiguity and left recursion in polynomial time and space. Although related to caching, memoization refers to a specific case of this optimization, distinguishing it from forms of caching such as buffering or page replacement. In the context of some logic programming languages, memoization is also known as tabling; see also lookup table.

Q.1. (d) What is optimal substructure and overlapping substructure? **(2.5)**

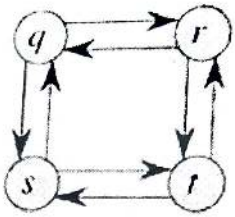
Ans: Optimal Substructure :

A given problems has Optimal Substructure Property if optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.

For example, the Shortest Path problem has following optimal substructure property:

If a node x lies in the shortest path from a source node u to destination node v then the shortest path from u to v is combination of shortest path from u to x and shortest path from x to v. The standard All Pair Shortest Path algorithms like Floyd–Warshall and Bellman–Ford are typical examples of Dynamic Programming.

On the other hand, the Longest Path problem doesn't have the Optimal Substructure property. Here by Longest Path we mean longest simple path (path without cycle) between two nodes. Consider the following unweighted graph given in the CLRS book. There are two longest paths from q to t: $q \to r \to k$ and $q \to s \to t$. Unlike shortest paths, these longest paths do not have the optimal substructure property. For example, the longest path $q \to r \to k$ is not a combination of longest path from q to r and longest path from r to t, because the longest path from q to r is $r \to q \to s \to t$ and the longest path from r to t is $r \to q \to s \to t$



Overlapping Substructure :

Like Divide and Conquer, Dynamic Programming combines solutions to sub-problems. Dynamic Programming is mainly used when solutions of same subproblems are needed again and again. In dynamic programming, computed solutions to subproblems are stored n a table so that these don't have to recomputed. So Dynamic Programming is not useful hen there are no common (overlapping) subproblems because there is no point storing he solutions if they are not needed again. For example, Binary Search doesn't have common subproblems. If we take example of following recursive program for Fibonacci Numbers, there are many subproblems which are solved again and again.

```
/* simple recursive program for Fibonacci numbers */
intfib(intn)
{
if(n <= 1 )
returnn;
returnfib(n-1) + fib(n-2);
}
```

Q.1. (e) What are the applications of minimum spanning tree? **(2.5)**

Ans. Minimum spanning trees have direct applications in the design of networks, including computer networks, telecommunications networks, transportation networks, water supply networks, and electrical grids (which they were first invented for, as mentioned above). They are invoked as subroutines in algorithms for other problems, including the Christofides algorithm for approximating the traveling salesman problem, approximating the multi-terminal minimum cut problem (which is equivalent in the single-terminal case to the maximum flow problem), and approximating the minimum-cost weighted perfect matching.

Other practical applications based on minimal spanning trees include:

• **Taxonomy.**

• Cluster analysis: clustering points in the plane, single-linkage clustering (a method of hierarchical clustering), graph-theoretic clustering, and clustering gene expression data.

• Constructing trees for broadcasting in computer networks. On Ethernet networks this is accomplished by means of the Spanning tree protocol.

• Image registration and segmentation – 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.

• Minimum spanning trees can also be used to describe financial markets. A correlation matrix can be created by calculating a coefficient of correlation between any two stocks.This matrix can be represented topologically as a complex network and a minimum spanning tree can be constructed to visualize relationships.

Q.1. (f) Explain fractional knapsack problem. **(2.5)**

Ans: Fractional Knapsack Problem

Given weights and values of n items, we need put these items in a knapsack of capacity W to get the maximum total value in the knapsack.

In the 0-1 Knapsack problem, we are not allowed to break items. We either take the whole item or don't take it.

Input:

Items as (value, weight) pairs

arr[] = {{60, 10}, {100, 20}, {120, 30}}

Same as above

Knapsack Capacity, W = 50;

Output:

Maximum possible value = 220

by taking items of weight 20 and 30 kg

In **Fractional Knapsack**, we can break items for maximizing the total value of knapsack. This problem in which we can break item also called fractional knapsack problem.

Input :

arr[] = {{60,10}, {100,20}, {120,30}}

Output :

Maximum possible value = 240

By taking full items of 10 kg, 20 kg and

2/3rd of last item of 30 kg

A brute-force solution would be to try all possible subset with all different fraction but that will be too much time taking.

An efficient solution is to use Greedy approach. The basic idea of greedy approach is to calculate the ratio value/weight for each item and sort the item on basis of this ratio. Then take the item with highest ratio and add them until we can't add the next item as whole and at the end add the next item as much as we can. Which will always be optimal solution of this problem.

Algo: Fractional Knapsack(Array v, Array w, int W)

1. for i = 1 to size(v)
2. do p[i] = v[i]/w[i]
3. Sort – Descending(p)

4.i <- 1

5 while (w>0)

6. do amount = min(W,w[i])

7.solution[i] = amount

8 W = W-amount

9 i<- i +1

10 return solution.

Q.1. (g) Differentiate between Dynamic programming and Greedy approach

Ans: Greedy method: In this approach, the decision is taken on the basis of current available information without worrying about the effect of the current decision in future.

Greedy algorithms build a solution part by part, choosing the next part in such a way, that it gives an immediate benefit. This approach never reconsiders the choices taken previously. This approach is mainly used to solve optimization problems. Greedy method is easy to implement and quite efficient in most of the cases. Hence, we can say that Greedy algorithm is an algorithmic paradigm based on heuristic that follows local optimal choice at each step with the hope of finding global optimal solution.

In many problems, it does not produce an optimal solution though it gives an approximate (near optimal) solution in a reasonable time.

Components of Greedy Algorithm

Greedy algorithms have the following five components

- **A candidate set–** A solution is created from this set.
- **A selection function–** Used to choose the best candidate to be added to the solution.
- **A feasibility function–** Used to determine whether a candidate can be used to contribute to the solution.
- **An objective function–** Used to assign a value to a solution or a partial solution.
- **A solution function–** Used to indicate whether a complete solution has been reached.

Areas of Application

Greedy approach is used to solve many problems, such as

- Finding the shortest path between two vertices using Dijkstra's algorithm.

- Finding the minimal spanning tree in a graph using Prim's /Kruskal's algorithm, etc.

Dynamic Programming is also used in optimization problems. Like divide-and-conquer method, Dynamic Programming solves problems by combining the solutions of subproblems. Moreover, Dynamic Programming algorithm solves each sub-problem just once and then saves its answer in a table, thereby avoiding the work of re-computing the answer every time

Dynamic Programming algorithm is designed using the following four steps:

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

Applications of Dynamic Programming Approach

- Matrix Chain Multiplication
- Longest Common Subsequence
- Travelling Salesman Problem

Q.1. (*h*) What is Matroid?

Ans: A matroid M = (S, I) is a finite ground set S together with a collection of sets I $\subseteq$ 2 S, known as the independent sets, satisfying the following axioms:

1. If I $\in$ I and J $\subseteq$ I then J $\in$ I.

2. If I, J $\in$ I and |J| > |I|, then there exists an element z $\in$ J \ I such that

I $\cup$ {z} $\in$ I.

A second original source for the theory of matroids is **graph theory**.

Every finite graph (or **multigraph**) G gives rise to a matroid $M(G)$ as follows: take as E the set of all edges in G and consider a set of edges independent if and only if it is a **forest**; that is, if it does not contain a simple cycle. Then $M(G)$ is called a **cycle matroid**. Matroids derived in this way are **graphic matroids**. Not every matroid is graphic, but all matroids on three elements are graphic. Every graphic matroid is regular.

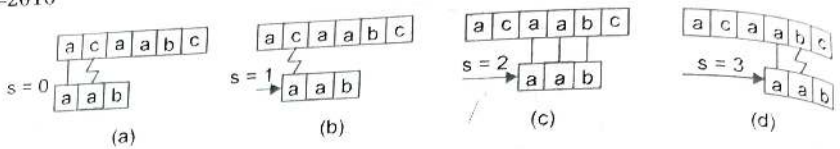
(2.5)

Q.1. (*i*) Write Naïve String Matching Algorithm.

Ans: The naive algorithm finds all valid shifts using a loop that checks the condition $P[1 \ldots m] = T[s + 1 \ldots s + m]$ for each of the $n - m + 1$ possible values of s.

```
NAIVE-STRING-MATCHER(T, P)
1 n ← length [T]
2 m ← length[P]
3 for s ← 0 to n - m
4 do if P[1 . . m] = T[s + 1 . . s + m]
5 then print "Pattern occurs with shift" s
```

The naive string-matching procedure can be interpreted graphically as sliding a "template" containing the pattern over the text, noting for which shifts all of the characters on the template equal the corresponding characters in the text, The **for** loop beginning on line 3 considers each possible shift explicitly. The test on line 4 determines whether the current shift is valid or not; this test involves an implicit loop to check corresponding character positions until all positions match successfully or a mismatch is found. Line 5 prints out each valid shift s.



Q.1. (*j*) Explain NP hard Problem briefly.

Ans: NP-Hard Problems

- We say that a decision problem Pi is NP-hard if every problem in NP is polynomialtime reducible to Pi .
- In symbols, Pi is NP-hard if, for every Pj $\in$ NP, Pj poly $\rightarrow$ Pi .
- Note that this doesn't require Pi to be in NP.
- Highly informally, it means that Pi is 'as hard as' all the problems in NP. – If Pi can be solved in polynomial-time, then so can all problems in NP. – Equivalently, if any problem in NP is ever proved intractable, then Pi must also be intractable.

Some problems can be translated into one another in such a way that a fast solution to one problem would automatically give us a fast solution to the other. There are some problems that every single problem in NP can be translated into, and a fast solution to such a problem would automatically give us a fast solution to every problem in NP. This group of problems are known as **NP-Hard**. Some problems in NP-Hard are actually not themselves in NP; the group of problems that are in *both* NP and NP-Hard is called **NP-Complete**.

UNIT-I

Q.2. (*a*) Explain Merge sort and compute the analysis of merge sort. (6)

Ans. Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n), it is one of the most respected algorithms.Merge sort first divides the array into equal halves and then combines them in a sorted manner.

```
Merge(A[], Temp[], left, mid, right)
{ Int i, j, k, l, target
i = left
j = mid + 1
target = left
while (i < mid && j < right) {
if (A[i] < A[j])
Temp[target] = A[i++]
else
Temp[target] = A[j++]
target++
}
if (i > mid) //left completed//
for (k = left to target-1)
A[k] = Temp[k];
```

```
if (j > right) //right completed//
k = mid
l = right
while (k > i)
A[l--] = A[k--]
for (k = left to target-1)
A[k] = Temp[k]
}
MainMergesort(A[1..n], n) {
Array Temp[1..n]
Mergesort[A, Temp, l, n]
}
Mergesort(A[], Temp[], left, right)
if (left < right) {
mid = (left + right)/2
Mergesort(A, Temp, left, mid)
Mergesort(A, Temp, mid+1, right)
Merge(A, Temp, left, mid, right)
}}
```

Complexity Analysis of Merge Sort

Worst Case Time Complexity: O(n log n)

Best Case Time Complexity : O(n log n)

Average Time Complexity : O(n log n)

Space Complexity : O(n)

• Time complexity of Merge Sort is O(n Log n) in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.

Q.2. (*b*) Perform the quick sort to sort the following numbers.

8, 3, 25, 6, 10, 17, 1, 2, 18, 5

Ans. 8, 3, 25, 6, 10, 17, 1, 2, 18, 5

first element: 8

middle element: 10

last element: 5

Therefore the median on [8,10,5] is 8.

Step 1. Choosing the Pivot Element

Choosing the pivot element can determine the complexity of the algorithm i.e. whether it will be n*logn or quadratic time:

a. Normally we choose the first, last or the middle element as pivot. This can harm us badly as the pivot might end up to be the smallest or the largest element, thus leaving one of the partitions empty.

b. We should choose the Median of the first, last and middle elements. If there are N elements, then the ceiling of N/2 is taken as the pivot element.

Step 2. Partitioning

a. First thing is to get the pivot out of the way and swapping it with the last number

5, 3, 25, 6, 10, 17, 1, 2, 18, **8**

b. Now we want the elements greater than pivot to be on the right side of it and similarly the elements less than pivot to be on the left side of it.

For this we define 2 pointers, namely i and j. i being at the first index and j being and the last index of the array.

- While **i** is less than **j** we keep in incrementing **i** until we find an element greater than pivot.
- Similarly, while **j** is greater then **i** keep decrementing **j** until we find an element less than pivot.
- After both **i** and **j** stop we swap the elements at the indexes of **i** and **j** respectively.

c. Restoring the pivot

When the above steps are done correctly we will get this as our output:

[5, 3, 2, 6, 1] [8] [10, 25, 18, 17]

Step 3. Recursively Sort the left and right part of the pivot.

Q.3. (*a*) Explain Strassen's algorithm for matrix multiplication. (6)

Ans: Following is simple Divide and Conquer method to multiply two square matrices.

(1) Divide matrices A and B in 4 sub-matrices of size N/2 x N/2 as shown in the below diagram.

(2) Calculate following values recursively. ae + bg, af + bh, ce + dg and cf + dh.

$$\underset{A}{\begin{bmatrix} a & b \\ c & d \end{bmatrix}} \times \underset{B}{\begin{bmatrix} e & f \\ g & h \end{bmatrix}} = \underset{C}{\begin{bmatrix} ae + & bg & af + & bh \\ ce + & dg & cf + & dh \end{bmatrix}}$$

A,B and C are square methices of size N × N

a, b, c and d are submathices of size N/2 × N/2

e, f, g and h are sub matrics of B, of size N/2 × N/2

In the above method, we do 8 multiplications for matrices of size N/2 x N/2 and 4 additions. Addition of two matrices takes $O(N^2)$ time. So the time complexity can be written as

$T(N) = 8T(N/2) + O(N^2)$

From Master's Theorem, time complexity of above method is $O(N^3)$

which is unfortunately same as the above naive method.

Simple Divide and Conquer also leads to $O(N^3)$, can there be a better way?

In the above divide and conquer method, the main component for high time complexity is 8 recursive calls. The idea of **Strassen's method** is to reduce the number of recursive calls to 7. Strassen's method is similar to above simple divide and conquer method in the sense that this method also divide matrices to sub-matrices of size N/2 x N/2 as shown in the above diagram, but in Strassen's method, the four sub-matrices of result are calculated using following formulae.

$$P1 = a(f-h) \quad P2 = (a+b)h$$
$$P3 = (c+d)e \quad P4 = d(g-e)$$
$$P5 = (a+d)(e+h) \quad P6 = (b-d)(g+h)$$
$$P7 = (a-c)(e+f)$$

The A × B can be calculated using above seven multiplications.

Following are values of four sub-matrices of result C

$$\underset{A}{\begin{bmatrix} a & b \\ c & d \end{bmatrix}} \times \underset{B}{\begin{bmatrix} e & f \\ g & h \end{bmatrix}} = \underset{C}{\begin{bmatrix} (p5+p4-p2+p6) & p1+p2 \\ p3+p4 & (p1+p5-p3+p7) \end{bmatrix}}$$

A,B and C are square metrices of size N × N

a, b, c and d are submatrices of A, of size N/2 × N/2

e, f, g and h are submatrices of B, of size N/2 × N/2

p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 × N/2

Time Complexity of Strassen's Method

Addition and Subtraction of two matrices takes $O(N^2)$ time. So time complexity can be written as

$T(N) = 7T(N/2) + O(N^2)$

From Master's Theorem, time complexity of above method is

$O(N^{Log7})$ which is approximately $O(N^{2.8074})$

Generally Strassen's Method is not preferred for practical applications for following reasons.

(1) The constants used in Strassen's method are high

(2) For Sparse matrices, there are better methods especially designed for them.

(3) The submatrices in recursion take extra space.

Q.3. (*b*)Apply Strassen's matrix multiplication algorithm to multiply the following matrices. **(6.5)**

$$\begin{bmatrix} 1 & 3 \\ 5 & 7 \end{bmatrix} \begin{bmatrix} 8 & 4 \\ 6 & 2 \end{bmatrix}$$

Ans: let A = $[A_{11} A_{12}]$ and B = $[B_{11} \quad B_{12}]$

$[A_{21} \quad A_{22}]$ $[B_{21} \quad B_{22}]$

So

$A_{11} = 1$ $\quad B_{11} = 8$

$A_{12} = 3$ $\quad B_{12} = 4$

$A_{21} = 5$ $\quad B_{21} = 6$

$A_{22} = 7$ $\quad B_{22} = 2$

Now compute P,Q,R,S,T,U,V as follows:

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

$= (1 + 7)(8 + 2) = 8 * 10 = 80$

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

$= (5 + 7) 8 = 12 * 8 = 96$

$R = A_{11}(B_{12} - B_{22})$
$= 1(4 - 2) = 1 * 2 = 2$

$S = A_{22}(B_{21} - B_{11})$
$= 7(6 - 8) = 7 * (-2) = -14$

$T = (A_{11} + A_{12}) B_{22}$
$= (1 + 3) 2 = 4 * 2 = 8$

$U = (A_{21} - A_{11})(B_{11} + B_{12})$
$= (5 - 1)(8 + 4) = 4 * 12 = 48$

$V = (A_{12} - A_{22})(B_{21} + B_{22})$
$= (3 - 7)(6 + 2) = (-4) * (8) = -32$

Now compute C_{ij}'s as follows :

$C_{11} = P+S-T+V$
$= 80-14-8-32 = 26$

$C_{12} = R + T$
$= 2 + 8 = 10$

$C_{21} = Q + S$
$= 96 - 14 = 82$

$C_{22} = P + R - Q + U$
$= 80+2-96+48 = 34$

Therefore,

$C = \begin{bmatrix} 26 & 10 \\ 82 & 34 \end{bmatrix}$

Thus the required matrix product is $\begin{bmatrix} 26 & 10 \\ 82 & 34 \end{bmatrix}$

UNIT-II

Q.4. (a) Explain Floyd Warshall algorithm. **(6)**

Ans: Floyd-Warshall algorithm (sometimes known as the Roy-Floyd algorithm, since Bernard Roy described this algorithm in 1959) is a graph analysis algorithm for finding shortest paths in a weighted, directed graph. A single execution of the algorithm will find the shortest path between all pairs of vertices. It does so in $\theta(V^3)$ time, where V is the number of vertices in the graph. Negative-weight edges may be present, but we shall assume that there are no negative-weight cycles.

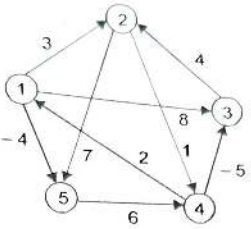
FLOYD-WARSHALL (W)

1. $n \leftarrow$ rows[W]
2. $D^{(0)} \leftarrow W$
3. for $k \leftarrow 1$ to n
4. do for $I \leftarrow 1$ to n
5. do for $j \leftarrow 1$ to n
6. do $d_{ij}^{(k)} \leftarrow \min (d_{ij}^{(k-1)}, d_{jk}^{(k-1)}, d_{kj}^{(k-1)})$
7. return $D^{(n)}$

The strategy adopted by the Floyd-Warshall algorithm is dynamic programming.

The running time of the Floyd-Warshall algorithm is determined by the triply nested for loops of lines 3-6. Each execution of line 6 takes O(1) time. The algorithm thus runs in time $\Theta(n^3)$.

Q.4. (b) Apply Floyd Warshall algorithm for constructing shortest path.



Ans. $d_{ij}^{(k)} = \min\left[d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right]$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} \text{if } d_{ij}^{(k-1)} \le d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{ik}^{(k-1)} \text{if } d_{ij}^{(k-1)} \le d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

$$d^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \pi^{(0)} = \begin{bmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

$$d^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \pi^{(1)} = \begin{bmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

$$d^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \pi^{(2)} = \begin{bmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

$$d^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \pi^{(3)} = \begin{bmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

$$d^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \pi^{(4)} = \begin{bmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{bmatrix}$$

$$d^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \pi^{(5)} = \begin{bmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{bmatrix}$$

Q.5. (a) What does dynamic programming have common with divide and conquer and what are differences? (6)

Ans: 1.The problem can be divided into *stages* with a *decision* required at each stage. In the capital budgeting problem the stages were the allocations to a single plant. The decision was how much to spend. In the shortest path problem, they were defined by the structure of the graph. The decision was were to go next.

2. Each stage has a number of *states* associated with it.The states for the capital budgeting problem corresponded to the amount spent at that point in time. The states for the shortest path problem was the node reached.

3.The decision at one stage transforms one state into a state in the next stage.The decision of how much to spend gave a total amount spent for the next stage. The decision of where to go next defined where you arrived in the next stage.

4. Given the current state, the optimal decision for each of the remaining states does not depend on the previous states or decisions.In the budgeting problem, it is not necessary to know how the money was spent in previous stages, only how much was spent. In the path problem, it was not necessary to know how you got to a node, only that you did.

5. There exists a recursive relationship that identifies the optimal decision for stage j, given that stage j+1 has already been solved.

6. The final stage must be solvable by itself.

S.No.	Divide-and-conquer algorithm	Dynamic Programming
1.	Divide-and-conquer algorithms splits a problem into separate subproblems, solve the subproblems, and combine the results for a solution to the original problem. Example : Quick sort, Merge sort, Binary search.	Dynamic Programming splits a problem into subproblems, some of which are common, solves the subproblems, and combines the results for a solution to the original problem. Example : Matrix Chain Multiplication, Longest Common Subsequence.
2.	Divide-and-conquer algorithms can be thought of as top-down algorithms.	Dynamic programming can be thought of as bottom-up.
3.	In divide and conquer, sub-problems are independent.	In Dynamic Programming, sub-problems are not independent.

4.	Divide & Conquer solutions are simple as compared to Dynamic programming.	Dynamic Programming solutions can often be quite complex and tricky.
5.	Divide & Conquer can be used for any kind of problems.	Dynamic Programming is generally used for Optimization problems.
6.	Only one decision sequence is ever generated.	Many decision sequences may be generated.

Q.5. (b) Determine the LCS of <A,B,C,B,D,A,B> AND <B,D,C,A,B,A> (6.5)

Ans: LCS-Length (X, Y)

```
1.  m ¬ length[X]
2.  n ¬ length[Y]
3.  for i ¬ 1 to m
4.     do c[i,0] ¬ 0
5.  for j ¬ 0 to n
6.     do c[0,j] ¬ 0
7.  for i¬ 1 to m
8. do for j ¬ 1 to n
9. do if x_i = y_j
10.   then c[i,j] ¬ c[i-1, j-1] + 1
11.  b[i,j] ¬ "↖"
12.  else if c[i - 1¸j]³ c[i, j-1]
13. then c[i,j] ¬ c[i-1, j]
14.   b[i,j] ¬ "Ý"
15.  else c[i,j] ¬ c[i, j-1]
16.  b[i,j] ¬ "¬"
17. return c and b
```

X = <A, B, C, B, D, A, B> X = <A, B, C, B, D, A, B>

Y = <B, D, C, A, B, A> Y = <B, D, C, A, B, A>

— **<B, C, B, A>** and **<B, D, A, B>** are

longest common subsequences of X and Y (*length* = 4)

— <B, C, A>, however, is not a LCS of X and Y

		j	0	1	2	3	4	5	6
i			y_j	***B***	***D***	***C***	***A***	***B***	***A***
0	x_i		0	0	0	0	0	0	0
1	***A***		0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	***B***		0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	***C***		0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	***B***		0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	***D***		0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	***A***		0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	***B***		0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Figure The c and b tables computed by LCS-LENGTH on the sequences X = [A, B, C, B, D, A, B] and Y = (B, D, C, A, B, A). The square in row i and column j contains the value of c [i, j] and the appropriate arrow for the value of b[i, j]. The entry 4 in c[7, 6] –the lower right-hand corner of the table–is the length of an LCS [B, C, B, A) of X and Y. For i,j> 0, entry c[i, j] depends only on whether $x_i = y_j$ and the values in entries c[i – 1,j], c [i, j – 1] and c [i–1, j–1] which are computed before c[i, j]. To reconstruct the elements of an LCS, follow the b[i, j] arrows from the lower right-hand corner; the path is shaded. Each "↖" on the path corresponds to an entry (highlighted) for which $x_i = y_i$ is a member of an LCS.

Q.6. (a) Explain the difference between Kruskal's and Prim's algorithm with the help of suitable example.

Ans:

Minimum connector algoritm

Kruskal's algorithm	Prim's algoritm
1. Slect the shortest edge in a network	1. Select any vertex
2. Select the next shortest edge which does not create a cycle	2. Select the shortest edge connected to that vertex
3. Repeat step 2 until all vertices have been connected.	3. Select the shortest edge connected to any vertex already connected
	4. Repeat step 3 until all vertices have been connected

MST-KRUSKAL (G,w)

1. A ← ϕ
2. for each vertex v ∈ V[G]
3. do MAKE-SET(v)
4. sort the edges of E into non decreasing order by weight w
5. for each edge (u,v) ∈ E, taken in non decreasing 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

MST-PRIM (G,w,r)

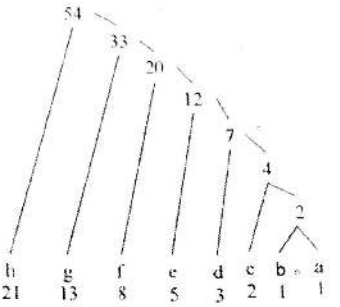
1. for each u ∈ V[G]
2. do key [u] ← ∞
3. π [u] ← NIL
4. key [r] ← 0
5. Q ← V[G]
6. while Q ≠ ϕ
7. do u ← EXTRACT-MIN(Q)
8. for each v ∈ Adj [u]
9. do if v ∈ Q and w (u,v) < key [v]
10. then π[v] ← u
11. key [v] ← w (u,v)

Q.6. (*b*) What is an optimal Huffman code for the following set of frequencies?

a : 1 b : 1 c : 2 d : 3 e :5 f : 8 g :13 h:21 **(6.5)**

Ans: A Since there are 8 letters in the alphabet, the initial queue size is n = 8, and 7 merge steps are required to build the tree. The final tree represents the optimal prefix code. The codeword for a letter is the sequence of the edge labels on the path from the root to the letter. Thus the optimal Huffman code is as follows.

h	0						
g	1	0					
f	1	1	0				
e	1	1	1	0			
d	1	1	1	1	0		
c	1	1	1	1	1	0	
b	1	1	1	1	1	1	0
a	1	1	1	1	1	1	1



Q.7. Illustrate Dijkstra's and Bellman Ford Algorithm for finding the shortest path. **(12.5)**

Ans: DIJKSTRA (G, w, s)

1. INITIALIZE-SINGLE-SOURCE (G,s)
2. S ← ϕ
3. Q ← V[G]
4. while Q ≠ ϕ
5. do u ← EXTRACT-MIN (Q)
6. S ← S ∪ {u}
7. for each vertex v ∈ Adj [u]
8. do RELAX (u,v,w)

BELLMAN-FORD (G, w, s)

1. INITIALIZE-SINGLE-SOURCE (G, s)
2. for i ← 1 to | V[G] | - 1
3. do for each edge (u,v) ∈ E[G]
4. do RELAX (u,v,w)
5. for each edge (u,v) ∈ E[G]
6. do if d [v] > d [u] + w (u,v)
7. then return FALSE
8. return TRUE

UNIT-IV

Q.8. (*a*) Give Knuth Morris Pratt Algorithm for pattern matching? **(6.5)**

Ans. Knuth–Morris– Pratt string searching algorithm (or KMP algorithm) searches for occurrences of a "word" W within a main "text string" S by employing the observation that when a mismatch occurs, the word itself embodies sufficient information to determine where the next match could begin, thus bypassing re-examination of previously matched

character consider a (relatively artificial) run of the algorithm, where W = "ABCDABD" and S = "ABC ABCDAB ABCDABCDABDE". At any given time, the algorithm is in a state determined by two integers:

- m, denoting the position within S where the prospective match for W begins,
- i, denoting the index of the currently considered character in W.

```
COMPUTE-PREFIX-FUNCTION(p)
1 m←  length[p]
2 π [1] ←0
3 k←0
4 for q ←2 to m
5  do while k >0 and p[k+1] ≠ p[q]
6 do k ←  π [k]
7 if p[k+1] = p[q]
8  then k←k+1
9 π [q]←k
10 return π
```

The KMP matching algorithm is given in KMP-MATCHER.

```
KMP-MATCHER(t,p)
1 n←length[t]
2 m←length[p]
3 π ← COMPUTE-PREFIX-FUNCTION(p)
4 q ← 0
5 for i ←1 to n
6 do while q>0 and p[q+1] ≠ t[i]
7 do q ← π[q]
8 if p[q+1] =  t[i]
9 q ←q+1
10 if q= m
11 then print "Pattern oocurs with shift"i-m
12  q←  π [ q]
```

***Q.8. (b)* Explain NP-Completeness reduction with an example.**

Ans: Hardest problems in NP. All problems in NP can be "reduced to" an NP-Complete problem„ "Reduced to" means NP problem can be converted into a NPC problem in polynomial-time „ Solution to NPC problem can be converted back into a solution to the NP problem.

The theory of NP-completeness is a solution to the practical problem of applying complexity theory to individual problems. NP-complete problems are defined in a precise sense as the hardest problems in P. Even though we don't know whether there is any problem in NP that is not in P, we can point to an NP-complete problem and say that if there are any hard problems in NP, that problems is one of the hard ones.

(Conversely if everything in NP is easy, those problems are easy. So NP-completeness can be thought of as a way of making the big P=NP question equivalent to smaller questions about the hardness of individual problems.)

So if we believe that P and NP are unequal, and we prove that some problem is NP-complete, we should believe that it doesn't have a fast algorithm.

For unknown reasons, most problems we've looked at in NP turn out either to be in P or NP-complete. So the theory of NP-completeness turns out to be a good way of showing that a problem is likely to be hard, because it applies to a lot of problems. But there are problems that are in NP, not known to be in P, and not likely to be NP-complete; for instance the code-breaking example I gave earlier.

Reduction

Formally, NP-completeness is defined in terms of "reduction" which is just a complicated way of saying one problem is easier than another.

We say that A is easier than B, and write A < B, if we can write down an algorithm for solving A that uses a small number of calls to a subroutine for B (with everything outside the subroutine calls being fast, polynomial time). There are several minor variations of this definition depending on the detailed meaning of "small" — it may be a polynomial number of calls, a fixed constant number, or just one call.

Then if A < B, and B is in P, so is A: we can write down a polynomial algorithm for A by expanding the subroutine calls to use the fast algorithm for B.

So "easier" in this context means that if one problem can be solved in polynomial time, so can the other. It is possible for the algorithms for A to be slower than those for B, even though A < B.

As an example, consider the Hamiltonian cycle problem. Does a given graph have a cycle visiting each vertex exactly once? Here's a solution, using longest path as a subroutine:

```
for each edge (u,v) of G
    if there is a simple path of length n-1 from u to v
        return yes    // path + edge form a cycle
return no
```

This algorithm makes m calls to a longest path subroutine, and does O(m) work outside those subroutine calls, so it shows that Hamiltonian cycle < longest path. (It doesn't show that Hamiltonian cycle is in P, because we don't know how to solve the longest path subproblems quickly.)

As a second example, consider a polynomial time problem such as the minimum spanning tree. Then for every other problem B, B < minimum spanning tree, since there is a fast algorithm for minimum spanning trees using a subroutine for B. (We don't actually have to call the subroutine, or we can call it and ignore its results.)

Q.9. (a) Explain Rabin-Karp String matching Algorithm. **(6)**

Ans: RABIN-KARP-MATCHER (T,P,d,q)

1. n ← length[T]
2. m ← length[P]
3. h ← $d^{m-1} \bmod q$
4. p ← 0
5. t_0 ← 0
6. for i ← 1 to m
7. do p ← (dp + P[i]) mod q
8. t_0 ← (dt_0 + T[i]) mod q
9. for s ← 0 to n-m

10. do if $p = t_s$
11. then if $P[1..m] = T[s+1..s+m]$
12. then ("Pattern occurs with shift" s)
13. if $s < n-m$
14. then $t_{s+1} \leftarrow (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q$

Q.9. (b) What is finite automata and its significance to match a string with algorithm and complexity. (6.5)

Ans: A FA can be represented by a 5-tuple $(Q, \Sigma, \delta, q_0, A)$ where

- **Q** is a finite set of states.
- Σ is a finite set of symbols called the alphabet.
- δ is the transition function where $\delta: Q \times \Sigma \rightarrow Q$
- $\mathbf{q_0}$ is the initial state from where any input is processed ($q_0 \in Q$).
- **A** is a set of final state/states of Q ($A \subseteq Q$).

We define the string-matching automation corresponding to a given pattern $P[1..m]$ as follows.

The state set Q is {0,1,..,m}. The start state q_0 is state 0, and state m is the only accepting state.

The transition function δ is defined by the following equation for any state q and character a:

$$\delta(q,a) = \sigma(P_q a)$$

As for any string-matching automation for a pattern of length m, the state set Q is {0,1,...,m}, the start state is 0, and the only accepting state is state m.

FINITE-AUTOMATION-MATCHER (T, ,δ,m)

1. $n \leftarrow$ length [T]
2. $q \leftarrow 0$
3. for $i \leftarrow 1$ to n
4. do $q \leftarrow \delta(q, T[i])$
5. if $q = m$
6. then $s \leftarrow i - m$
7. print ("Pattern occurs with shift" s)

COMPUTE-TRANSITION-FUNCTION (P, Σ)

1. $m \leftarrow$ length [P]
2. for $q \leftarrow 0$ to m
3. do for each character $a \in \Sigma^*$
4. do $k \leftarrow \min(m+1, q+2)$
5. repeat $k \leftarrow k - 1$
6. until
7. $\delta(q, a) \leftarrow k$
8. return δ

FIRST TERM EXAMINATION [SEP-2017]
FIFTH SEMESTER [B.TECH]
ALGORITHM DESIGN AND ANALYSIS
[ETCS–301]

Time : 1.30 hrs. **M.M. : 30**

Note: *Attempt any three questions including Q.1 is compulsory.*

Q. 1. (a) Define problem statement, problem instance and problem space with reference to algorithm with an example. (2)

Ans. A problem statement is a concise description of an issue to be addressed or a condition to be improved upon. A simple and well-defined problem statement will be used by the project team to understand the problem and work toward developing a solution.

In computational complexity theory, a problem refers to the abstract question to be solved. In contrast, an instance of this problem is a rather concrete utterance, which can serve as the input for a decision problem.

Problem Space refers to the entire range of components that exist in the process of finding a solution to a problem.

Q .1. (b) Define Algorithm and Asymptotic notations. (2)

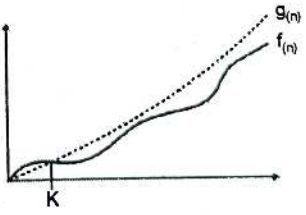
Ans. Algorithm: An algorithm (pronounced AL-go-rith-um) is a procedure or formula for solving a problem, based on conductiong a sequence of specified actions. A computer program can be viewed as an elaborate algorithm. In mathematics and computer science, an algorithm usually means a small procedure that solves a recurrent problem.

The commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- O Notation
- Ω Notation
- θ Notation

Big Oh Notation,O

The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

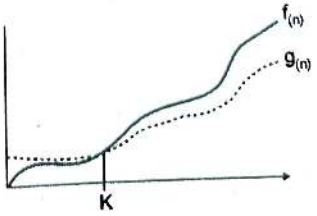


For example, for a function $f(n)$

$O(f(n)) = \{ g(n) :$ there exists $c > 0$ and n_θ such that $f(n) \leq c.g(n)$ for all $n > n_\theta. \}$

Omega Notation, Ω

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.



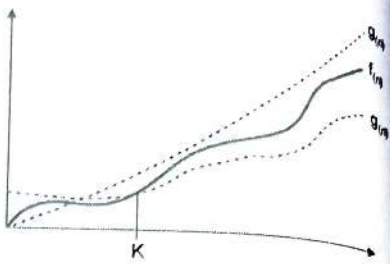
For example, for a function $f(n)$

$\Omega(f(n)) \geq \{g(n)$: there exists $c > 0$ and n_0 such that $g(n) \leq c.f(n)$ for all $n > n_0$. $\}$

Theta Notation, θ

The notation $\theta(n)$ is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows -

$\theta(f(n)) = \{g(n)$ if and only if $g(n) = O(f(n))$ and $g(n) = \Omega(f(n))$ for all



Q. 1. (c) List out Approaches to design an algorithms known to you. (2)

Ans.

- Divide and Conquer Method. In the divide and conquer approach, the problem is divided into several small sub-problems.
- Greedy Method. In greedy algorithm of optimizing solution, the best solution is chosen at any moment.
- Dynamic Programming.
- Backtracking Algorithm.
- Branch and Bound.
- Linear Programming.

Q. 1. (d) How correctness of algorithm is checked? (2)

Ans. The main steps in the formal analysis of the correctness of an algorithm are:

- Identification of the properties of input data (the so-called problem's preconditions).

Identification of the properties which must be satisfied by the output data (the so-called problem's postconditions).

- Proving that starting from the preconditions and executing the actions specified in the algorithms one obtains the postconditions.

When we analyze the correctness of an algorithm a useful concept is that of state. The algorithm's state is the set of the values corresponding to all variables used in the algorithm

Q. 1. (e) State master method to solve a recurrence relation with all the cases. (2)

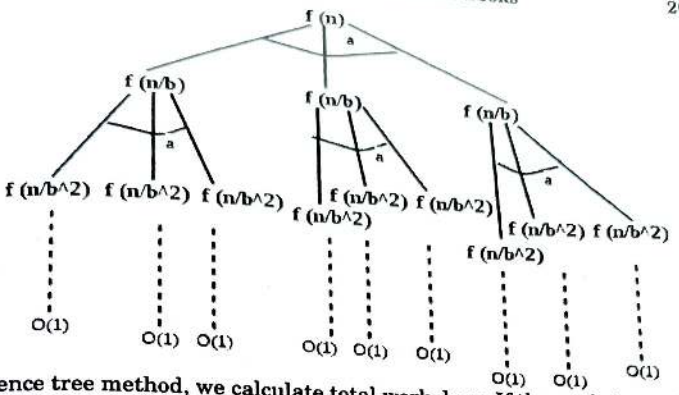
Ans. Master Method is a direct way to get the solution. The master method works only for following type of recurrences or for recurrences that can be transformed to following type.

$T(n) = aT(n/b) + f(n)$ where $a >= 1$ and $b > 1$

There are following three cases:

1. If $f(n) = \Theta(n^c)$ where $c < \text{Log}_b a$ then $T(n) = \Theta(n^{\text{Log}_b a})$
2. If $f(n) = \Theta(n^c)$ where $c = \text{Log}_b a$ then $T(n) = \Theta(n^c \text{Log } n)$
3. If $f(n) = \Theta(n^c)$ where $c > \text{Log}_b a$ then $T(n) = \Theta(f(n))$

How does this work?: Master method is mainly derived from recurrence tree method. If we draw recurrence tree of $T(n) = aT(n/b) + f(n)$, we can see that the work done at root is $f(n)$ and work done at all leaves is $\Theta(n^c)$ where c is $\text{Log}_b a$. And the height of recurrence tree is $\text{Log}_b n$



recurrence tree method, we calculate total work done. If the work done at leaves is ynomially more, then leaves are the dominant part, and our result becomes the work e at leaves (Case 1). If work done at leaves and root is asymptotically same, then our ult becomes height multiplied by work done at any level (Case 2). If work done at root symptotically more, then our result becomes work done at root (Case3).

Examples of some standard algorithms whose time complexity can be evaluated ing Master Method

Merge Sort: $T(n) = 2T(n/2) + \Theta(n)$. It falls in case 2 as c is 1 and $\text{Log}_b a$] is also 1. So solution is $\Theta(n \text{ Log}n)$

Binary Search: $T(n) = T(n/2) + \Theta(1)$. It also falls in case 2 as c is 0 and $\text{Log}_b a$ is also So the solution is $\Theta(\text{Log}n)$

Notes:

1. It is not necessary that a recurrence of the form $T(n) = aT(n/b) + f(n)$ can be solved ing Master Theorem. The given three cases have some gaps between them. For ample, the recurrence $T(n) = 2T(n/2) + n/\text{Log}n$ cannot be solved using master method.

2. Case 2 can be extended for $f(n) = \Theta(n^c \text{Log}^k n)$

If $f(n) = \Theta(n^c \text{Log}^k n)$ for some constant $k >= 0$ and $c = \text{Log}_b a$, then $T(n) = \Theta(n^c \text{Log}^{k+1} n)$

Q. 2. (a) Can master method solve the recurrence relation t(n) = 3T(n/4) + nl n? If "no" explain,if "yes" solve it. **(5)**

Ans.

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

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

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

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

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

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

Therefore, the solution is $T(n) = \theta(n \lg n)$.

Q .2. (b) Can master method solve the recurrence relation $T(n) = 2T(n/2) + n$ log n? If "no" explain,if "yes" solve it. (5)

Ans. Yes it can be solved by master method a = 2 b = 2 f(n) = nlogn

$$n^{\log_b a} = n^{\log_2 2} = n \text{ but } f(n) = n \log n$$

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

Q. 3. (a) Discuss the essence of Dynamic Programming. (5)

Ans. Greedy algorithms makes the best local decision at each step, but may not guarantee the global optimal. Exhaustive search algorithms explore all possibilities and always select the optimal, but the cost is too high.

Thus leads to dynamic programming: search all possibilities (correctness) while restoring results to avoid recomputing (efficiency).

Dynamic programming can efficiently implement recursive algorithm by storing partial results.

If recursive algorithm computes the same subproblems over and over and over again, storing the answer for each subproblem in a table to look up instead of recompute.

Generally for optimization problem for left-to right-order objects such as characters in string, elements of a permutation, points around a polygon, leaves in a search tree, integer sequences. Because once the order is fixed, there are relatively few possible stopping places or states.

Use when the problem follow the *principle of optimality*: Future decisions are made based on the overall consequences of previous decisions, not the actual decisions themselves.

Independent subproblems: solution to one subproblem doesn't affect solution to another subproblem of the same problem. Using resources to solve one subproblem renders them unavailable to solve the other subproblem. (longest path)

Q. 3. (b) Give the optimal parenthesis for matrix multiplication problem with input of 6 matrix of size: << 4, 10, 3, 12, 20, 7 >>.

Ans. Refer Q. 4. (b) of End Term Examination 2017.

Q. 4. (a) Give the problem statement of 0/1 knapsack. Consider the following input instance of 0/1 knapsack problem;

3 items weight 20,30,40 units and profit associated with them 10,20,50 units respectively with knapsack of capacity 60 units.Solve it using dynamic programming approach.

Ans. Problem can be solved in this way

0–1 Knapsack Problem

value [] = {60, 100, 120};

weight [] = {10, 20, 30};

W = 50;

Solution: 220

Weight = 10; Value = 60;
Weight = 20; Value = 100;
Weight = 30; Value = 120;
Weight = (20 + 10); Value = (100 + 60)
Weight = (30 + 10); Value = (120 + 60);
Weight = (30 + 20); Value = (120 + 100);
Weight = (30 + 20 + 10) > 50.

Q. 4. (b) Write down the psuedocode of insertion sort and analyze its complexity in all cases of input instance. (5)

Ans. Insertion-Sort (A)

```
1. for j ← 2 to length [A]
2.     do key ← A[j]
3.        ▷ Insert A[j] into the sorted sequence A[1..j – 1]
4.        i ← j – 1
5.        while i > 0 and A[i] > key
6.              do A [i + 1] ← A[i]
7.                 i ← i – 1.
8.        A[i + 1] ← key
```

Analysis of Insertion Sort

The time taken by INSERTION - SORT procedure depends on the input. We start by presenting the INSERTION - SORT procedure with the time "cost" of each statement and the number of time each statement is executed.

Insection-Sort (A)	Cost	times
1. for $j \leftarrow 2$ to length [A]	c_1	n
2. do key $\leftarrow A[j]$	c_2	$n-1$
3. ▷insert $A[j]$ into the sorted sequence $A[i..j-1]$	0	$n-1$
4. $i \leftarrow j-1$	c4	$n-1$
5. While $i > 0$ and $A[i] >$ key	c_5	$\sum_{j=2}^{n} t_j$
6. do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7. $i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8. $A[i+1] \rightarrow$ key	c_8	$n-1$

The running time of the algorithm is the sum of running times for each statement excuted.

To compute T(n) the running time, we sum the product of the cost and time columns.

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Best case occurs if the array is already sorted. For each $j = 2, 3, \ldots n$, we find that $A[i] \leq$ key in line 5 when $i = j - 1$.

$$\text{Thus } t_j = 1 \text{ for } j = 2, 3, \ldots\ldots n$$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$

$$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

$= an + b$, where a and b are constants.

$=$ linear function of $n = O(n)$.

In worst case, the array is in reverse order. We must compare each elements $A[j]$ with each element in the entire sorted, subarray $A[1...j-1]$ and $t_j = j$ for $j = 2, 3, ...n$.

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} j + c_6 \sum_{j=2}^{n} (j-1) + c_7 \sum_{j=2}^{n} (j-1) + c_8(n-1)$$

$$= c_1 n + c_2(n-1) + c_4(n-1) + c_5$$

$$\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$

$$\left(\because \sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \text{ and } \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}\right)$$

$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n - (c_2 + c_4 + c_5 + c_8)$$

$= an^2 + bn + c$ for a, b and c are constant.

$=$ quadratic function of n.

$= O(n^2)$

END TERM EXAMINATION DEC-2017
FIFTH SEMESTER [B.TECH]
ALGORITHM DESIGN AND ANALYSIS
[ETCS–301]

Time : 3 hrs. M.M. : 75

Note: *Attempt any five questions including Q.1 is compulsory.*

Q. 1. (a) Define Aymptotic notation?

Ans. Refer Q. 1. (b) of First Term Examination 2017.

Q. 1. (b) What is Substitution method ?

Ans. The substitution method for solving recurrences is famously described using two steps:

- **1:** Guess the form of the solution
- **2:** Use induction to show that the guess is valid

Determine a tight asymptotic lower bound for the following recurrence

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

Let us guess that $T(n) = n^2 lg(n)$. Therefore, our induction hypothesis is there exists c and an n_0 such that

$$T(n) \geq cn^2 lg(n)$$

$\delta\, n_0 > n$ and $c > 0$.

For the **base case**($n = 1$). we have $T(1) = 1 > c1^2 lg1$. This is true for all $c > 0$.

Now for the inductive step, assume the hypothesis is true for m < n, thus:

$$T(m) \leq cm^2 lg(m)$$

So: $T(n) = 4T(n/2) + n^2 \geq 4c \frac{n^2}{4} \lg \frac{n}{2} + n^2 = cn^2 lg(n) - cn^2 lg(2) + n^2 = cn^2 lg(n)$

$+(1-c)n^2$ If we now pick as $c < 1$, then.

Q. 1. (c) Explain Hashing and elaborate its disadvantages over linear and binary search

Ans. Hashing is generating a value or values from a string of text using a mathematical function.

Hashing is one way to enable security during the process of message transmission when the message is intended for a particular recipient only. A formula generates the hash, which helps to protect the security of the transmission against tampering.

Hashing is also a method of sorting key values in a database table in an efficient manner.

Advantages: The main advantage of hash tables over other table data structures is speed. This advantage is more apparent when the number of entries is large (thousands or more).

Hash tables are particularly efficient when the maximum number of entries can be predicted in advance, so that the bucket array can be allocated once with the optimum size and never resized.

Disadvantages: Hash tables can be more difficult to implement than self-balancing binary search trees.

Although operations on a hash table take constant time on average, the cost of a good hash function can be significantly higher than the inner loop of the lookup algorithm for a sequential list or search tree.

For certain string processing applications, such as spell-checking, hash tables may be less efficient than tries, finite automata, or Judy arrays.

There are no collisions in this case.

The entries stored in a hash table can be enumerated efficiently (at constant cost per entry), but only in some pseudo-random order. In comparison, ordered search trees have lookup and insertion cost proportional to log(n), but allow finding the nearest key at about the same cost, and ordered enumeration of all entries at constant cost per entry.

Hash tables in general exhibit poor locality of reference—that is, the data to be accessed is distributed seemingly at random in memory. Hash tables become quite inefficient when there are many collisions.

Q. 1. (d) Differentiate between dynamic programming and divide and conquer approach.

Ans.

S.No.	Divide-and-conquer algorithm	Dynamic Programming
1.	Divide-and-conquer algorithms splits a problem into separate subproblems, solve the subproblems, and combine the results for a solution to the original problem. Example : Quick sort, Merge sort, Binary search.	Dynamic Programming splits a problem into subproblems, some of which are common, solves the subproblems, and combines the results for a solution to the original problem. Example : Matrix Chain Multiplication, Longest Common Subsequence.
2.	Divide-and-conquer algorithms can be thought of as top-down algorithms.	Dynamic programming can be thought of as bottom-up
3.	In divide and conquer, sub-problems are independent.	In Dynamic Programming, sub-problems are not independent.
4.	Divide & Conquer solutions are simple as compared to Dynamic programming	Dynamic Programming solutions can often be quite complex and tricky.
5.	Divide & Conquer can be used for any kind of problems.	Dynamic Programming is generally used for Optimization problems.
6.	Only one decision sequence is ever generated.	Many decision sequences may be generated.

Q. 1. (e) Explain the concept of overlapping subproblems.

Ans. A problem is said to have overlapping subproblems if the problem can be broken down into subproblems which are reused several times or a recursive algorithm for the problem solves the same subproblem over and over rather than always generating new subproblems.

For example, the problem of computing the Fibonacci sequence exhibits overlapping subproblems. The problem of computing the nth Fibonacci number F(n), can be broken down into the subproblems of computing F(n – 1) and F(n – 2), and then adding the two. The subproblem of computing F(n – 1) can itself be broken down into a subproblem that involves computing F(n – 2). Therefore the computation of F(n – 2) is reused, and the Fibonacci sequence thus exhibits overlapping subproblems.

A naive recursive approach to such a problem generally fails due to an exponential complexity. If the problem also shares an optimal substructure property, dynamic programming is a good way to work it out.

```
/* simple recursive program for Fibonacci numbers */
intfib(intn)
{
 if( n <= 1 )
  returnn;
 returnfib(n-1) + fib(n-2);
}
```

Q. 1. (f) What are the advantages of optimal binary search tree over binary search tree?

Ans. • Binary search trees are used to organize a set of keys for fast access: the tree maintains the keys in-order so that comparison with the query at any node either results in a match, or directs us to continue the search in left or right subtree.

• A balanced search tree achieves a worst-case time O(log n) for each key search, but fails to take advantage of the structure in data.

• For instance, in a search tree for English words, a frequently appearing word such as "the" may be placed deep in the tree while a rare word such as "machiocolation" may appear at the root because it is a median word.

• In practice, key searches occur with different frequencies, and an Optimal Binary Search Tree tries to exploit this non-uniformity of access patterns, and has the following formalization.

• The input is a list of keys (words) w1, w2, . . . , wn, along with their access probabilities p1, p2, . . . , pn. The prob. are known at the start and do not change.

• The interpretation is that word wi will be accessed with relative frequency (fraction of all searches) pi . The problem is to arrange the keys in a binary search tree that minimizes the (expected) total access cost.

Q. 1. (g) Explain 0-1 knapsack problem.

Ans. In 0-1 Knapsack, items cannot be broken which means the thief should take the item as a whole or should leave it. This is reason behind calling it as 0-1 Knapsack.

Hence, in case of 0-1 Knapsack, the value of x_i can be either *0* or *1*, where other constraints remain the same.

0 – 1 Knapsack cannot be solved by Greedy approach. Greedy approach does not ensure an optimal solution. In many instances, Greedy approach may give an optimal solution.

The following examples will establish our statement.

Example-1

Let us consider that the capacity of the knapsack is W = 25 and the items are as shown in the following table.

Item	A	B	C	D
Profit	24	18	18	10
Weight	24	10	10	7

Without considering the profit per unit weight (p_i/w_i), if we apply Greedy approach to solve this problem, first item *A* will be selected as it will contribute max$_i$mum profit among all the elements.

After selecting item *A*, no more item will be selected. Hence, for this given set of items total profit is *24*. Whereas, the optimal solution can be achieved by selecting items, *B* and C, where the total profit is 18 + 18 = 36.

Q. 1. (h) What are the elements of greedy strategy?

Ans. Elements of greedy strategy are:

1. Optimal substructure
2. 0/1 knapsack
3. Activity selection problem
4. Huffman coding

Q. 1. (i) Define Matroid with an example.

Ans. A matroid M = (S, I) is a finite ground set S together with a collection of sets I ⊆ 2 S, known as the independent sets, satisfying the following axioms: 1. If I ∈ I and J ⊆ I then J ∈ I. 2. If I, J ∈ I and |J| > |I|, then there exists an element z ∈ J \ I such that I U {z} ∈ I.

A second original source for the theory of matroids is graph theory.

Every finite graph (or multigraph) *G* gives rise to a matroid *M*(*G*) as follows: take as *E* the set of all edges in *G* and consider a set of edges independent if and only if it is a forest; that is, if it does not contain a simple cycle. Then *M*(*G*) is called a **cycle matroid**. Matroids derived in this way are **graphic matroids**. Not every matroid is graphic, but all matroids on three elements are graphic. Every graphic matroid is regular.

Q. 1. (j) Explain P and NP briefly.

Ans. P- Polynomial time **solving** . Problems which can be solved in polynomial time, which take time like O(n), O(n2), O(n3). Eg: finding maximum element in an array or to check whether a string is palindrome or not. so there are many problems which can be solved in polynomial time.

NP- Non deterministic Polynomial time solving. Problem which can't be solved in polynomial time like TSP(travelling salesman problem) or An easy example of this is subset sum: given a set of numbers, does there exist a subset whose sum is zero?.

but NP problems are checkable in polynomial time means that given a solution of a problem , we can check that whether the solution is correct or not in polynomial time.

Q. 2. (a) Explain quick sort and compute the analysis of quick sort perform quick sort on following data 14,15,25,28,30,32,35,40

What is the problem with quick sort ,if the data is already sorted? Discuss.

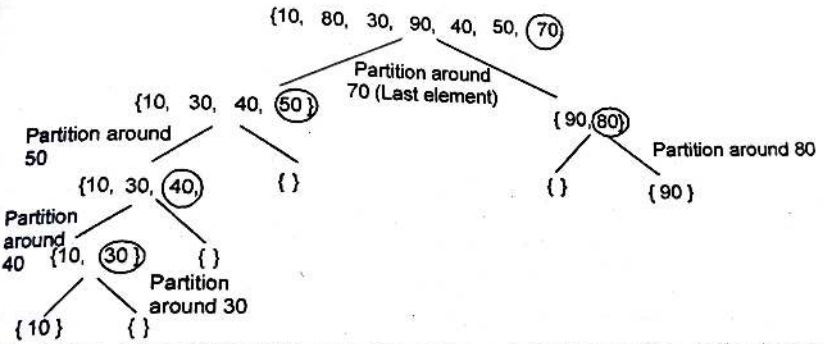
Ans. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Pseudo Code for recursive QuickSort function :

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
  if (low < high)
  {
    /* pi is partitioning index, arr[pi] is now
       at right place */
    pi = partition(arr, low, high);
    quickSort(arr, low, pi - 1);  // Before pi
    quickSort(arr, pi + 1, high); // After pi
  }
}
```



Partition Algorithm: There can be many ways to do partition, following pseudo code adopts the method given in CLRS book. The logic is simple, we start from the leftmost element and keep track of index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap current element with arr[i]. Otherwise we ignore current element.

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
  if (low < high)
  {
    /* pi is partitioning index, arr[p] is now
       at right place */
    pi = partition(arr, low, high);
    quickSort(arr, low, pi - 1);  // Before pi
    quickSort(arr, pi + 1, high); // After pi
  }
}
```

Array 14,15,25,28,30,32,35,40 can be sorted acc to the above method but we see that its already sorted than it shows the worst case of quicksort.

Worst-case running time: When quicksort always has the most unbalanced partitions possible, then the original call takes cncnc, n time for some constant ccc, the recursive call on n−1 $n-1$n, minus, 1elements takes c(n−1) $c(n-1)$c, left parenthesis, n, minus, 1, right parenthesis time, the recursive call on n-2$n-2$n, minus, 2 elements takes c(n−2)c$(n-2)$c, left parenthesis, n, minus, 2, right parenthesis time, and so on. Here's a tree of the subproblem sizes with their partitioning times:

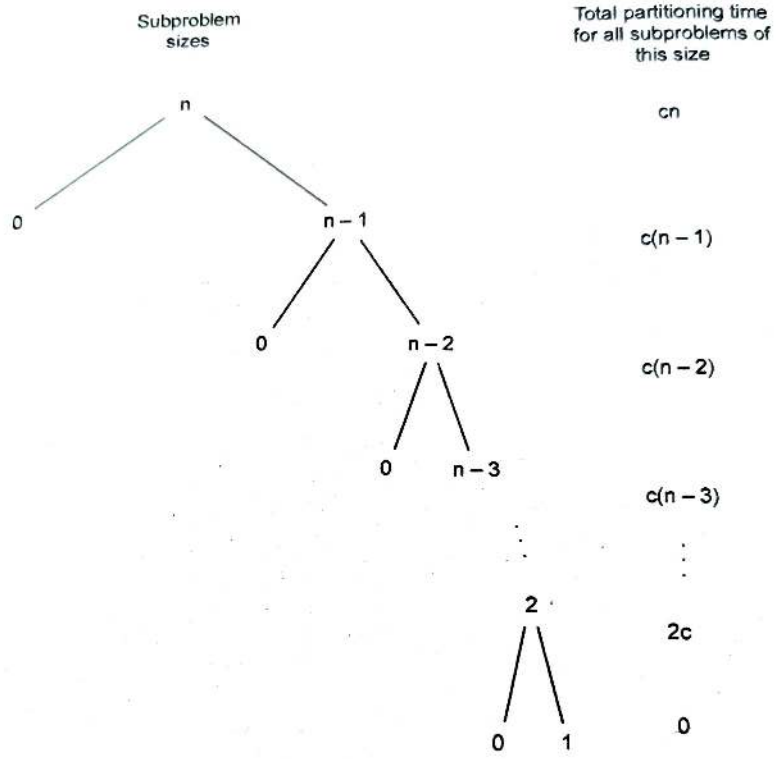


Diagram of worst case performance for Quick Sort

When we total up the partitioning times for each level, we get

\begin{aligned} cn + c(n-1) + c(n-2) + \cdots + 2c &= c(n + (n-1) + (n-2) + \cdots + 2) \\ &= c((n+1)(n/2) - 1) \ .

\end{aligned}$cn+c(n-1)+c(n-2)+...+2c$

$=c(n+(n-1)+(n-2)+...+2)=c((n+1)(n/2)-1)$.

The last line is because 1 + 2 + 3 + \cdots + n1+2+3+"+n is the arithmetic series, as we saw when we analyzed selection sort. (We subtract 1 because for quicksort, the summation starts at 2, not 1.) We have some low-order terms and constant coefficients, but when we use big-Θ notation, we ignore them. In big-Θ notation, quicksort's worst-case running time is \Theta(n^2) Θ (n2).

Best-case running time: Quicksort's best case occurs when the partitions are as evenly balanced as possible: their sizes either are equal or are within 1 of each other. The former case occurs if the subarray has an odd number of elements and the pivot is right in the middle after partitioning, and each partition has $(n-1)/2 (n-1)/2$left parenthesis, n, minus, 1, right parenthesis, slash, 2elements. The latter case occurs if the subarray has an even number nnn of elements and one partition has n/2n/2n, slash, 2 elements with the other having n/2-1n/2−1n, slash, 2, minus, 1. In either of these cases, each partition has at most n/2n/2n, slash, 2 elements, and the tree of subproblem sizes looks a lot like the tree of subproblem sizes for merge sort, with the partitioning times looking like the merging times:

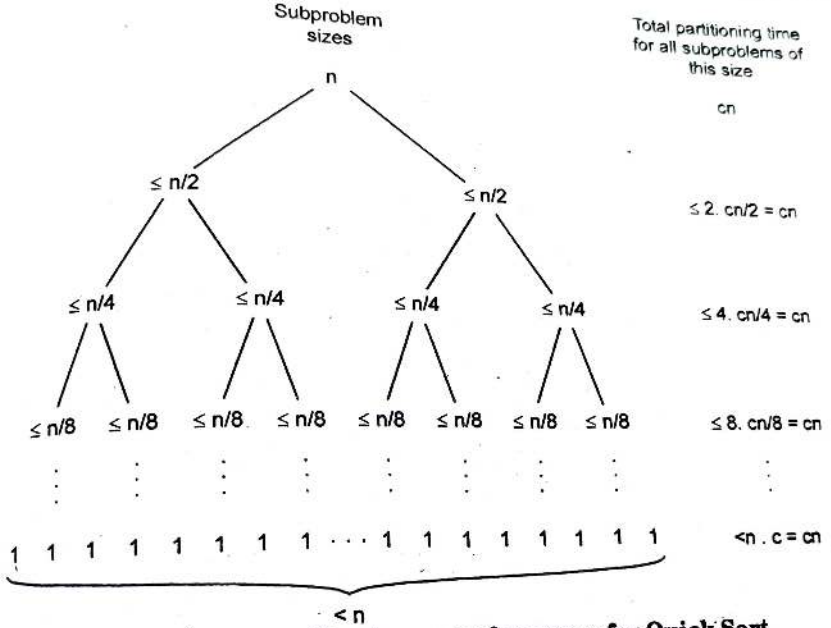
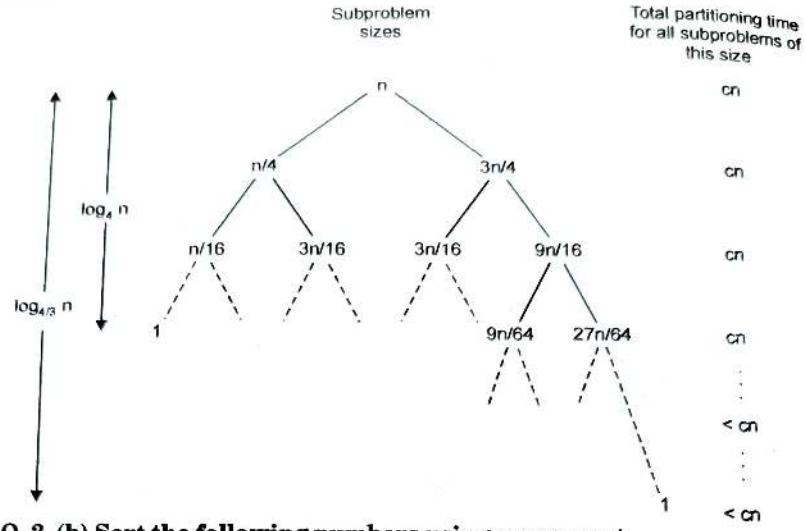


Diagram of best case performance for Quick Sort

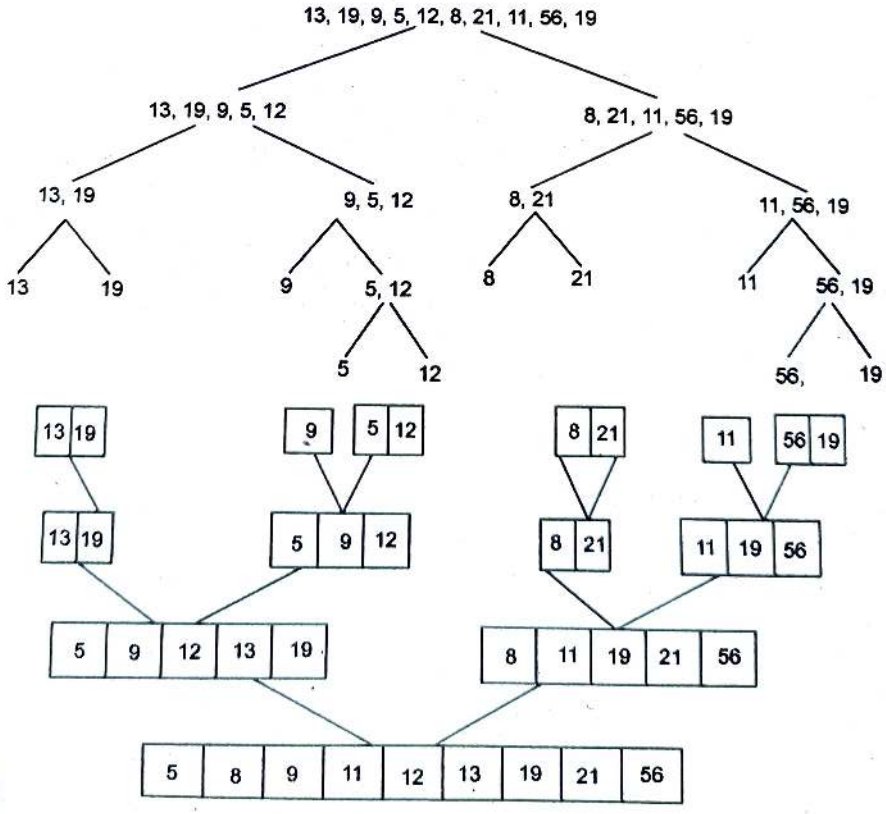
Using big-Θ notation, we get the same result as for merge sort: \Theta(n \log_2 n)Θ (*n*log2*n*).

Average-case running time: Showing that the average-case running time is also \Theta(n\log_2 *n*) Θ (*n*log2 *n*) takes some pretty involved mathematics, and so we won't go there. But we can gain some intuition by looking at a couple of other cases to understand why it might be *O*(*n* \log_2 n)*O*(*n*log2*n*)*O*, left parenthesis, n, log, start subscript, 2, end subscript, n, right parenthesis. (Once we have *O*(*n*\log_2 *n*) *O*(*n*log2*n*)*O*, left parenthesis, n, log, start subscript, 2, end subscript, n, right parenthesis, the \Theta(*n*\log_2 *n*) Θ (*n*log2*n*) bound follows because the average-case running time cannot be better than the best-case running time.) First, let's imagine that we don't always get evenly balanced partitions, but that we always get at worst a 3-to-1 split. That is, imagine that each time we partition, one side gets 3*n*/43*n*/43, n, slash, 4 elements and the other side gets *n*/4*n*/4*n*, slash, 4. (To keep the math clean, let's not worry about the pivot.) Then the tree of subproblem sizes and partitioning times would look like this:



Q. 2. (b) Sort the following numbers using merge sort
13,19,9,5,12,8,21,11,56,19

Ans.



Q.3. (a) Explain the data structure for disjoint set,its operations and its applications.

Ans. A disjoint set is a data structure which keeps track of all elements that are separated by a number of disjoint (not connected) subsets. With the help of disjoints sets, you can keep a track of the existence of elements in a particular group.

Let's say there are 6 elements A, B, C, D, E, and F. B, C, and D are connected and E and F are paired together. This gives us 3 subsets that have elements (A), (B, C, D), and (E, F).

Disjoint sets help us quickly determine which elements are connected and close and to unite two components into a single entity.

MAKE-SET(x): make a new set Si = {x} and add Si to S

- UNION(x, y): if x ∈ Si and y ∈ Sj , then S ← S – {Si , Sj} ∪ {Si ∪ Sj} – Representative of Si ∪ Sj is typically the representative of Si or Sj
- FIND(x): returns the representative of the set containing x
- Analyse complexity of sequence of m MAKE-SET, FIND and UNION operations, n of which are MAKE-SET operations
- Complexity is analysed in terms of n and m

Application:

MST-KRUSKAL(G, ẃ)

1 A ← ∅

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(u)

6 = FIND(v)

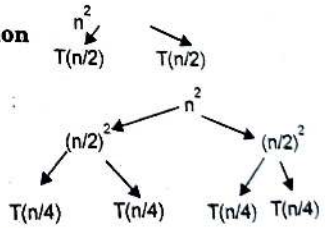
7 then A ← A ∪ {(u, v)}

8 UNION(u, v)

9 return A

Q. 3. (b) Solve the following recurrence relation

(i) T(n) = 4T(n/2) + n^2(using recurrence tree)



Ans.

T(n) = n^2+n^2/4+n^2/4^2+.....lg n times

<= n^2(1/1–1/4)

T(n) = Theta(n^2)

(ii) T(n) = 5T(n/4) + (n^3) (using master theorem)

Ans. By Master theorem

a = 5 b = 4f(n) = n^3

$n^{\log_b a} = n^{\log 45} = n^2$

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

therefore ∈ = 1

hence by case 3 sol is T(n) = theta(n^3)

Q. 4. What are the basic steps of dynamic programming?

Ans. Dynamic programming is used where we have problems, which can be divided into similar sub-problems, so that their results can be re-used. Mostly, these algorithms are used for optimization. Before solving the in-hand sub-problem, dynamic algorithm will try to examine the results of the previously solved sub-problems. The solutions of

sub-problems are combined in order to achieve the best solution.

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Q. 4. (b) Find optimal parenthesization of a matrix chain product whose sequence of dimension is< <4,10,3,12,20,7>

Ans. Sequence of dimensions are,<4,10,3,12,20,7>. The matrices have sizes 4*10,10*3,3*12,12*20,20*7 .we need to compute M[i, j],0 < =i, j<=5.We know M[i, i] = 0 for all i

1	2	3	4	5	
0					1
	0				2
		0			3
			0		4
				0	5

We proceed, working away from the diagonal. We compute the optimal solutions for products of 2 matrices.

1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

{4, 10, 3, 12, 20, 7}

Now products of 3 matrices

$$M[1,3] = \min \begin{cases} M[1,2] + M[3,3] + p_0 p_2 p_3 = 120 + 0 + 4 \cdot 3 \cdot 12 = 264 \\ M[1,1] + M[2,3] + p_0 p_1 p_3 = 0 + 360 + 4 \cdot 10 \cdot 12 = 840 \end{cases} = 264$$

$$M[2,4] = \min \begin{cases} M[2,3] + M[4,4] + p_1 p_3 p_4 = 360 + 0 + 10 \cdot 12 \cdot 20 = 2760 \\ M[2,2] + M[3,4] + p_1 p_2 p_4 = 0 + 720 + 10 \cdot 3 \cdot 20 = 1320 \end{cases} = 1320$$

$$M[3,5] = \min \begin{cases} M[3,4] + M[5,5] + p_2 p_4 p_5 = 720 + 0 + 3 \cdot 20 \cdot 7 = 1140 \\ M[3,3] + M[4,5] + p_2 p_3 p_5 = 0 + 1680 + 3 \cdot 12 \cdot 7 = 1932 \end{cases} = 1140$$

1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

⇒

1	2	3	4	5	
0	120	264			1
	0	360	1320		2
		0	720	1140	3
			0	1680	4
				0	5

Now, products of 4 matrices

$$M[1,4] = \min \begin{cases} M[1,3] + M[4,4] + p_0 p_3 p_4 = 264 + 0 + 4.12.20 = 1224 \\ M[1,2] + M[3,4] + p_0 p_2 p_4 = 120 + 720 + 4.3.20 = 1080 \\ M[1,1] + M[2,4] + p_0 p_1 p_4 = 0 + 1320 + 4.10.20 = 2120 \end{cases} = 1080$$

$$M[2,5] = \min \begin{cases} M[2,4] + M[5,5] + p_1 p_4 p_5 = 1320 + 0 + 10 \cdot 20 \cdot 7 = 2720 \\ M[2,3] + M[4,5] + p_1 p_3 p_5 = 360 + 1680 + 10 \cdot 12 \cdot 7 = 2880 \\ M[2,2] + M[3,5] + p_1 p_2 p_5 = 0 + 1140 + 10 \cdot 3 \cdot 7 = 1350 \end{cases}$$

1	2	3	4	5	
0	120	264			1
	0	360	1320		2
		0	720	1140	3
			0	1680	4
				0	5

⇒

1	2	3	4	5	
0	120	264	1080		1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

Now products of 5 matrices

$$M[1,5] = \min \begin{cases} M[1,4] + M[5,5] + p_0 p_4 p_5 = 1080 + 0 + 4 \cdot 20 \cdot 7 = 1544 \\ M[1,3] + M[4,5] + p_0 p_3 p_5 = 264 + 1680 + 4 \cdot 12 \cdot 7 = 2016 \\ M[1,2] + M[3,5] + p_0 p_2 p_5 = 120 + 1140 + 4 \cdot 3 \cdot 7 = 1344 \\ M[1,1] + M[2,5] + p_0 p_1 p_5 = 0 + 1350 + 4 \cdot 10 \cdot 7 = 1630 \end{cases} = 1344$$

1	2	3	4	5	
0	120	264	1080		1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

⇒

1	2	3	4	5	
0	120	264	1080	1344	1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

To print the optimal parenthesization, we use the PRINT-OPTIMAL-PARENS procedure.

PRINT-OPTIMAL-PARENS (*s, i, f*)

1. if $i = j$
2. then print "A"
3. else print "("
4. PRINT-OPTIMAL-PARENS ($s, i, s[i, j]$)
5. PRINT-OPTIMAL-PARENS ($s, \ldots + 1, j$)
6. print ")"

Now for optimal paranthesization, Each ... we find the optimal value for $M[i,j]$ we also store the value of k that we used. ... his for the example, we would get.

1	2	3	4	5	
0	120/1	264/2	1080/2	1344/2	1
	0	360/2	1320/2	1350/2	2
		0	720/3	1140/4	3
			0	1680/4	4
				0	5

The k value for the solution is 2, so we have $((A_1 A_2)(A_3 A_4 A_5))$. The first half is done. The optimal solution for the second half comes from entry $M[3, 5]$. The value of k here is 4, so now we have $((A_1 A_2)((A_3 A_4) A_5))$. Thus the optimal solution is to parenthesize $((A_1 A_2)((A_3 A_4) A_5))$.

Q. 5. (a) Compute binomial coefiicient using dynamic programming.

Ans: 1. A binomial coefficient C(n, k) can be defined as the coefficient of X^k in the expansion of (1 + X)^n.

2. A binomial coefficient C(n, k) also gives the number of ways, disregarding order, that k objects can be chosen from among n objects; more formally, the number of k-element subsets (or k-combinations) of an n-element set.

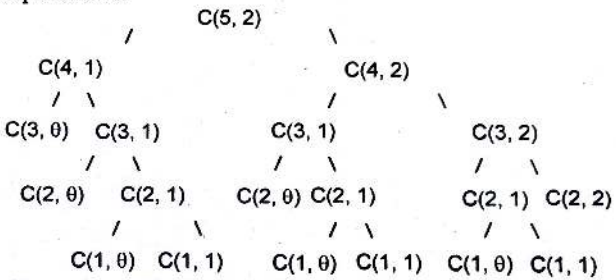
The Problem: *Write a function that takes two parameters n and k and returns the value of Binomial Coefficient C(n, k).* For example, your function should return 6 for n = 4 and k = 2, and it should return 10 for n = 5 and k = 2.

(1) Optimal Substructure: The value of C(n, k) can be recursively calculated using following standard formula for Binomial Coefficients.

C(n, k) = C(n–1, k–1) + C(n–1, k)

C(n, 0) = C(n, n) = 1

(2) Overlapping Subproblems: It should be noted that the above function computes the same subproblems again and again. See the following recursion tree for $n = 5$ an $k = 2$. The function $C(3, 1)$ is called two times. For large values of n, there will be many common subproblems.



Since same suproblems are called again, this problem has Overlapping Subproblem property. So the Binomial Coefficient problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, re-computations of same subproblems can be avoided by constructing a temporary array C[][] in bottom up manner. Following is Dynamic Programming based implementation.

Q. 5. (b) Explain the Floyd Warshall algorithm and discuss its complexity.

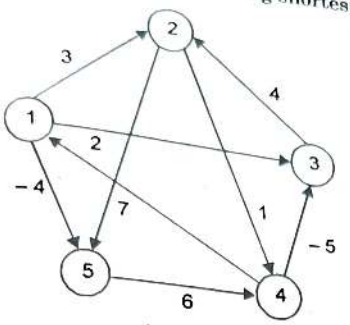
Ans. Floyd-Warshall algorithm (sometimes known as the Roy-Floyd algorithm, since Bernard Roy described this algorithm in 1959) is a graph analysis algorithm for finding shortest paths in a weighted, directed graph. A single execution of the algorithm will find the shortest path between all pairs of vertices. It does so in $\theta(V^3)$ time, where V is the number of vertices in the graph. Negative-weight edges may be present, but we shall assume that there are no negative-weight cycles.

FLOYD-WARSHALL (W)

1. n ← rows[W]
2. $D^{(0)} \leftarrow W$
3. for k ← 1 to n
4. do for I ← 1 to n
5. do for j ← 1 to n
6. do $d_{ij}^{(k)} \leftarrow \min (d_{ij}^{(k-1)}, d_{jk}^{(k-1)}, d_{kj}^{(k-1)})$
7. return $D^{(n)}$

The strategy adopted by the Floyd-Warshall algorithm is dynamic programming.
The running time of the Floyd-Warshall algorithm is determined by the triply nested for loops of lines 3-6. Each execution of line 6 takes O(1) time. The algorithm thus runs in time $O(n^3)$.

Apply Floyd Warshall algorithm for constructing shortest path



$$d_{ij}^{(k)} = \min\left[d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right]$$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} \text{ if } d_{ij}^{(k-1)} \le d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} \text{ if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \pi^{(0)} = \begin{bmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \pi^{(1)} = \begin{bmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \pi^{(2)} = \begin{bmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \pi^{(4)} = \begin{bmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{bmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \pi^{(5)} = \begin{bmatrix} NIL & 3 & 4 & 5 & 1 \\ 4 & NIL & 4 & 2 & 1 \\ 4 & 3 & NIL & 2 & 1 \\ 4 & 3 & 4 & NIL & 1 \\ 4 & 3 & 4 & 5 & NIL \end{bmatrix}$$

Q. 6. (a) Explain the difference between Dijkstra,s and Bellmanford algorithm with the help of example.

Ans. DIJKSTRA (G, w, s)

1. INITIALIZE-SINGLE-SOURCE (G,s)
2. S ← 00
3. Q ← V[G]
4. while Q ≠ 0
5. do u ← EXTRACT-MIN (Q)
6. S ← S ∪ {u}
7. for each vertex v ∈ Adj [u]
8. do RELAX (u,v,w)

BELLMAN-FORD (G, w, s)

1. INITIALIZE-SINGLE-SOURCE (G, s)
2. for i ← 1 to | V[G] | – 1
3. do for each edge (u,v)∈ E[G]
4. do RELAX (u,v,w)
5. for each edge (u,v) ∈ E[G]
6. do if d [v] > d [u] + w (u, v)
7. then return FALSE
8. return TRUE

Q. 6. (b) Find the optimal Schedule for the following jobs with profit (p1,p2,p3,p4,p5,p6) = (3,5,17,20,6,10) and deadlines(d1,d2,d3,d4,d5,d6) = (1,3,3,4,1,2)

Ans. above question can be solved in this way:

Step: 1 We will arrange the profits Pi in descending order, along with corresponding deadlines.

Profit	30	20	18	6	5	3	1
Job	P7	P3	P4	P6	P2	P1	P5
Deadlines	2	4	3	1	3	1	2

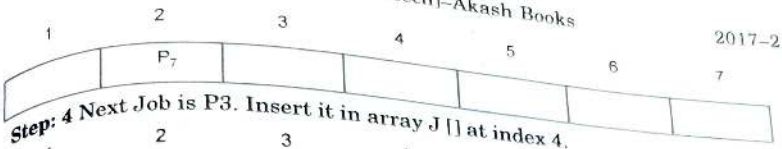
Step: 2 Create an array J [] which stores the jobs. Initially j [] will be

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

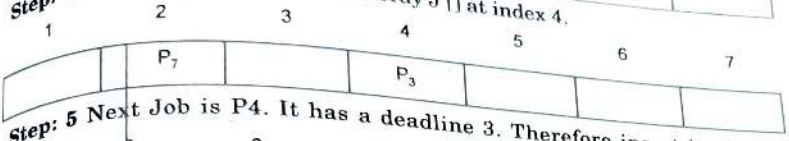
Step: 3 Add ith Job in array J [ ] at index denoted by its deadlines Di

First Job is P7, its deadline is 2.

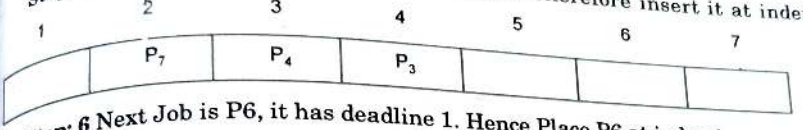
Hence insert P7 in the array J [] at 2nd index.



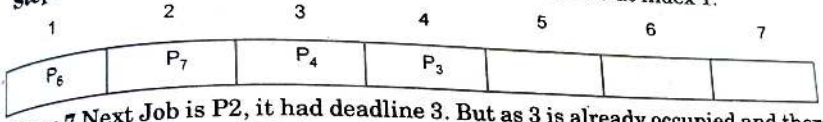
Step: 4 Next Job is P3. Insert it in array J [] at index 4.



Step: 5 Next Job is P4. It has a deadline 3. Therefore insert it at index 3.



Step: 6 Next Job is P6, it has deadline 1. Hence Place P6 at index 1.



Step: 7 Next Job is P2, it had deadline 3. But as 3 is already occupied and there is no empty slot at index < J [3]. Just discard job P2. Similarly Job P1 and P5 will get discarded.

Step: 8 Thus the optimal sequence which we will obtain will be 6-7-4-3. The maximum profit will be 74.

Q. 7. (a) Explain Prim's algorithm for finding the minimum spanning tree and analyze its complexity.

Ans. MST-PRIM (G,w,r)

1. for each u ∈ V[G]
2. do key [u] ← ∞
3. π[u] ← NIL
4. key [r] ← 0
5. Q ← V[G]
6. while Q ≠ ∅
7. do u ← EXTRACT-MIN(Q)
8. for each v ∈ Adj [u]
9. do if v ∈ Q and w (u,v) < key [v]
10. then π[v] ← u
11. key [v] ← w (u,v)

Q. 7. (b) Consider 5 items along their respective weights and values

$$I = \langle I_1, I_2, I_3, I_4, I_5 \rangle$$

$$w = \langle 5, 10, 20, 30, 40 \rangle$$

$$v = \langle 30, 20, 100, 90, 160 \rangle$$

The capacity of knapsack W = 60. Find the solution to the fractional

Ans. Initially,

Item	w_i	v_i
I_1	5	30
I_2	10	20
I_3	20	100
I_4	30	90
I_5	40	160

Taking value per weight ratio *i.e.*, $P_i = v_i / w_i$

Item	w_i	v_i	$P_i = v_i/w_i$
I_1	5	30	6.0
I_2	10	20	2.0
I_3	20	100	5.0
I_4	30	90	3.0
I_5	40	160	4.0

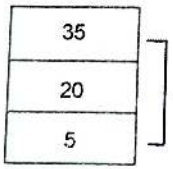
Now, arrange the value of P_i in decreasing order.

Item	w_i	v_i	$P_i = v/w_i$
I_1	5	30	6.0
I_3	20	100	5.0
I_5	40	160	4.0
I_4	30	90	3.0
I_2	10	20	2.0

Now, fill the knapsack according to the decreasing value of P_i.

First we choose item I_1 whose weight is 5, then choose item I_3 whose weight is 20. Now the total weight in knapsack is 5 + 20 = 25.

Now, the next item is I_5 and its weight is 40, bit we want only 35. So we choose fractional part of if i.e.,



The value of fractional part of I_5 is $\frac{160}{40} \times 35 = 140$

Thus the maximum value = 30 + 100 + 140 = 270

Q. 8. (a) Differntiate between P and NP Problems.Explain polynomial tim verification with an example.How it is different from polynolial time solution

Ans. P problems are the problems which can be solved in a Polynomial tim complexity.For example: finding the greatest number in a series of multiple number will take atmost 'N' number of steps , where N is the count of numbers in the sequenc

Where as NP problems may include those which cannot be solved(or are yet to b solved) in Polynomial time

Although some NP problems are yet to be solved in Polynomial time, once we ha a solution, it can be checked in Polynomial time.

For example: '*n* queens problem' where you have to arrange '*n*' number of queens in a chess board in such a way that no 2 queens are in the same row, column or diagonal.

Here, finding a suitable arrangement in Polynomial time may be impossible , but verifying that a certain arrangement works can be done in Polynomial time.

Polynomial time verification: For some problems, the answer can be verified to be correct in Polynomial Time, even if there is no known way of solving the original problem in Polynomial Time.

For example, consider solving NxN Sudoku. There is no known way to solve this where the time is bound by a polynomial function of N for sufficiently large values.

However, it is easy to check if an NxN Sudoku is solved correctly. Simply 1) verify that every row, column, and box contains each of the numbers 1..N exactly once; 2) verify that the original clues are respected in the solution. Both of these operations are polynomial-time.

Another example is integer factorization. While there is no known way to factor N-digit numbers in Polynomial time, it is easy to check whether a factorization is correct by simply multiplying the purported factors and comparing to the original number.

Q. 8. (b) Illustrate string matching with finite automata.

Ans. We define the string-matching automation corresponding to a given pattern P[1..m] as follows.

The state set Q is {0,1,..,m}. The start state q_0 is state 0, and state m is the only accepting state.

The transition function δ is defined by the following equation for any state q and character a:

$$\delta(q, a) = \sigma(P_q a)$$

As for any string-matching automation for a pattern of length m, the state set Q is {0,1,..,m}, the start state is 0, and the only accepting state is state m.

FINITE-AUTOMATION-MATCHER (T, δ, m)

1. n ← length [T]
2. q ← 0
3. for i ← 1 to n
4. do q ←δ (q,T[i])
5. if q = m
6. then s ← i – m
7. print "Pattern occurs with shift" s

COMPUTE-TRANSITION-FUNCTION (P,Σ)

1. m ← length [P]
2. for q ← 0 to m
3. do for each character a ∈ Σ^*
4. do k ← min (m+1, q+2)
5. repeat k ← k – 1
6. until
7. δ (q,a)← k
8. return δ

Q. 9. (a) Explain NP hard and NP Complete problems with the help of suitable example.

Ans. NP-complete problems are special kinds of NP problems. You can take any kind of NP problem and twist and contort it until it looks like an NP-complete problem. For example, the knapsack problem is NP. It can ask what's the best way to stuff a pieces of different precious metals lying on

the ground, and that you can't carry all of them in the bag.

Surprisingly, there are some tricks you can do to convert this problem into a travellin salesman problem. In fact, any NP problem can be made into a travelling salesma problem, which makes travelling salesman NP-complete.

(Knapsack is also NP-complete, so you can do the reverse as well!)

NP-Hard problems are worst than NP problems. Even if someone suggested you solution to a NP-Hard problem, it'd still take forever to verify if they were right.

For example, in travelling salesman, trying to figure out the absolute shortest pat through 500 cities in your state would take forever to solve. Even if someone walked u to you and gave you an itinerary and claimed it was the absolute shortest path, it'd sti take you forever to figure out whether he was a liar or not.

Q. 9. (b) Explain KNUTH_MORRIS_PRATT string matching algorithm

Ans. Knuth–Morris–Pratt string searching algorithm (or KMP algorithm) searche for occurrences of a "word" W within a main "text string" S by employing the observatio that when a mismatch occurs, the word itself embodies sufficient information t determine where the next match could begin, thus bypassing re-examination o previously matched character consider a (relatively artificial) run of the algorithm, wher W = "ABCDABD" and S = "ABC ABCDAB ABCDABCDABDE". At any given time, th algorithm is in a state determined by two integers:

- m, denoting the position within S where the prospective match for W begins,
- i, denoting the index of the currently considered character in W.

```
COMPUTE-PREFIX-FUNCTION(p)
 1  m <– length[p]
 2  π[1] <–0
 3  k <– 0
 4  for q <– 2 to m
 5  do while k>0 and p[k+1] ≠ p[q]
 6  do k <– π[k]
 7  if p[k+1] = p[q]
 8  then k<–k+1
 9  π[q]<–k
10  return π
```

The KMP matching algorithm is given in KMP-MATCHER.

```
KMP-MATCHER(t,p)
 1  n<-length[t]
 2  m<-length[p]
 3  π<-COMPUTE-PREFIX-FUNCTION(p)
 4  q<-0
 5  for i<-1 to n
 6  do while q>0 and p[q+1] ≠t[i]
 7   do q<-π[q]
 8  if p[q+1] = t[i]
 9  q <-q+1
10 if q= m
11 then print "Pattern occurs with shift"i-m
12 q<-π[q]
```

FIRST TERM EXAMINATION [SEP. 2018]
FIFTH SEMESTER [B.TECH]
ALGORITHM DESIGN & ANALYSIS [ETCS–301]

Time : 1½ hrs. **M.M. : 30**

Note: *Attempt Q.no. 1 which is compulsory and any two other questions from remaining.*

Q.1. (a) Solve the recurrence relation of Stassen's algorithm for matrix multiplication.

Ans. • Strassen observed the following. **(2)**

$$Z = \begin{Bmatrix} A & B \\ C & D \end{Bmatrix} \cdot \begin{Bmatrix} E & F \\ G & H \end{Bmatrix} = \begin{Bmatrix} (S_1 + S_2 - S_4 + S_6) & (S_4 + S_5) \\ (S_6 + S_7) & (S_2 + S_3 + S_5 - S_7) \end{Bmatrix}$$

where

$$\begin{aligned} S_1 &= (B - D) \cdot (G + H) \\ S_2 &= (A + D) \cdot (E + H) \\ S_3 &= (A - C) \cdot (E + F) \\ S_4 &= (A + B) \cdot H \\ S_5 &= A \cdot (F - H) \\ S_6 &= D \cdot (G - E) \\ S_7 &= (C + D) \cdot E \end{aligned}$$

• This leads to a divide-and-conquer algorithm with running time

$$T(n) = 7T(n/2) + \Theta(n^2)$$

• We only need to perform 7 multiplications recursively.

• Division/Combination can still be performed in $\Theta(n^2)$ time.

• Lets solve the recurrence using the iteration method

$$T(n) = 7T(n/2) + n^2$$

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

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

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

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

$$= n^2 + \left(\frac{7}{2^2}\right)n^2 + \left(\frac{7}{2^2}\right)^2 n^2 + \left(\frac{7}{2^2}\right)^3 n^2 \ldots + \left(\frac{7}{2^2}\right)^{\log n - 1} n^2 + 7^{\log n}$$

$$\sum_{i=0}^{\log n - 1} \left(\frac{7}{2^2}\right)^i n^2 + 7^{\log n}$$

$$= n^2 \cdot \Theta\left(\left(\frac{7}{2^2}\right)^{\log n - 1}\right) + 7^{\log n} \Rightarrow = n^2 \cdot \Theta \frac{7^{\log n}}{(2^2)^{\log n}} + 7^{\log n}$$

$$= n^2 \cdot \Theta\left(\frac{7^{\log n}}{n^2}\right) + 7^{\log n} \Rightarrow = \Theta(7^{\log n}).$$

— Now we have the following:

$$7^{\log n} = 7^{\frac{\log_7 n}{\log_7 2}} = (7^{\log_7 n})^{(1/\log_7 2)} = n^{(1/\log_7 2)}$$

$$= n^{\frac{\log_2 7}{\log_2 2}} = n^{\log 7}.$$

— Or in general: $a^{\log_k n} = n^{\log_k a}$

So the solution is $T(n) = \Theta(n^{\log 7}) = \Theta(n^{2.81, \cdots})$.

Q.1. (*b*) Differentiate between Big-Oh and Small-Oh notations. **(2)**

Ans. O(big Oh) notation: If $f(s)$ and $g(s)$ are functions of a real or complex variable s and S is an arbitrary set of (real or complex) numbers s (belonging to the domains of f and g), we write f(s) = O(g(s)) (s ∈ S), if there exists a constant c such that $|f(s)| \leq c\,|g(s)|$ (s ∈ S). To be consistent with our earlier definition of "big oh" we make the following convention: If a range is not explicitly given, then the estimate is assumed to hold for all sufficiently large values of the variable involved, i.e., in a range of the form $x \geq x_0$, for a suitable constant x_0.

Little-oh notation(o): Asymptotic upper bound provided by O-notation may not be asymptotically tight.$O(g(n)) = \{f(n)$: for any +ve $c > 0$, if a constant $n_0 > 0$ such that $0 <= f(n) < cg(n)$ for all $n >= n_0\}$

The function $f(n)$ becomes insignificant relative to g(n) as n approaches infinity

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

Q.1. (*c*) List the applications for disjoint sets. **(2)**

Ans. Applications of Disjoint-set Data Structures

- Connected components (CCs)
- Minimum Spanning Trees (MSTs)

Determine the connected components of an undirected graph.

Connected-Components(G)

- for each vertex $v \in V[G]$
- do MAKE-SET (v)
- for each edge $(u, v) \in E[G]$
- do if FIND-SET (u) ≠ FIND-SET (v)
- then UNION (u, v)

Same-Component (u, v)

- if FIND-SET (u) = FIND-SET (v)
- then return TRUE
- else reture FALSE

Minimum Spanning Tree:

A *minimum spanning tree* (MST) or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

Algorithm Steps:

- Sort the graph edges with respect to their weights.
- Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
- Only add edges which doesn't form a cycle , edges which connect only disconnected components.

Q.1. (d) Compare Dynamic Programming with divide and conquer. (2)

Ans. Refer to Q.1 (d) End term Examination Dec-2017 [page no - 8-2017].

Q.1. (e) Discuss the ingredients of dynamic programming. (2)

Ans. Two key ingredients that an optimization problem must have in order for dynamic programming to apply: Optimal substructure and overlapping sub problems

Optimal substructure:

(a) A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to sub problems.

(b) Whenever a problem exhibits optimal substructure we have a good clue that dynamic programming might apply.

Common pattern in discovering optimal substructure

(a) You show that a solution to the problem consists of making a choice, such as choosing an initial cut in rod or choosing an index at which to split the matrix chain. Making this choice leaves one or more sub problems to be solved

(b) You supposed that for a given problem, you are given the choice that leads to an optimal solution. You do not concern yourself yet with how to determine this choice. You just assume that is has been given to you.

(c) Given this choice, you determine which sub problems ensure and how to best characterize the resulting space of sub problems.

(d) You show that the solutions to the sub problems used within an optimal solution to the problem must themselves be optimal by using a "cut-and-paste" technique. You do so by supposing that each of the sub problems solutions is not optimal and then deriving a contradiction.

Overlapping Sub problems:

- Dynamic Programming is not useful when there are no common (overlapping) sub problems because there is no point storing the solutions if they are not needed again.
- Take advantage of overlapping sub problems by solving each sub problems once and then storing the solution in a table, where it can be looked up when needed, using a constant time per lookup.

Q.2. Solve the following recurrence relations (Provide proper explanation) **(2 × 5 = 10)**

(a) $T(n) = 3T(n-2) + n^2$ **(b) $T(n) = 3T(n/2) + O(n)$**

(c) $T(n) = 4T(n/2) + O(n^2)$ **(d) Worst case of Quick sort**

(e) Tower of Hanoi

Ans. (a) $T(n) = 3T(n-2) + n^2$

We iteratively substitute Terms to arrive a general form

$$T(n) = 3*3T(n-4) + (n-2)^2 + n^2$$
$$= 3^2T(n-2*2) + (n-2)^2 + n^2$$
$$T(n) = 3^kT(n-2k) + (n-2(k-1))^2 + (n-2(k-2))^2 \ldots (n-2)^2 + n^2$$

Lets $n - 2k = 1$ for k, where if stops $(T(i))$

and insert the value $\frac{n-1}{2} = k$

$$T(n) = 3^{\frac{n}{2}-\frac{1}{2}} T(1) + (n-2k+2)^2 + (n-2k+4)^2 + \ldots (n-2)^2 + n^2$$

$$= 3^{\frac{n}{2}-\frac{1}{2}} + (1+2)^2 + (1+4)^2 + (1+6)^2 + \ldots (n-2)^2 + n^2$$

$$= 3^{\frac{n}{2}-\frac{1}{2}} + (3)^2 + (5)^2 + (7)^2 + \ldots (n-2)^2 + n^2$$

$$= 3^n + \frac{n(n+1)(2n-1)}{6} \approx O(n^3)$$

$\therefore$ $T(n) = O(n^3)$.

(b) $T(n) = 3T(n/2) + O(n)$

$a = 3, \quad b = 2, \quad c = 1$

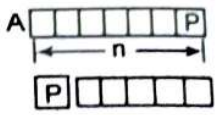
$n^{\log_b a} = n^{\log_2 3} > n$

It falls to case 1 of master's theorem

$\therefore$ $T(n) = 0(n^{\log 3})$.

(c) Refer Q.3(b) End term Examination Dec-2017 [page no - 15-2017].

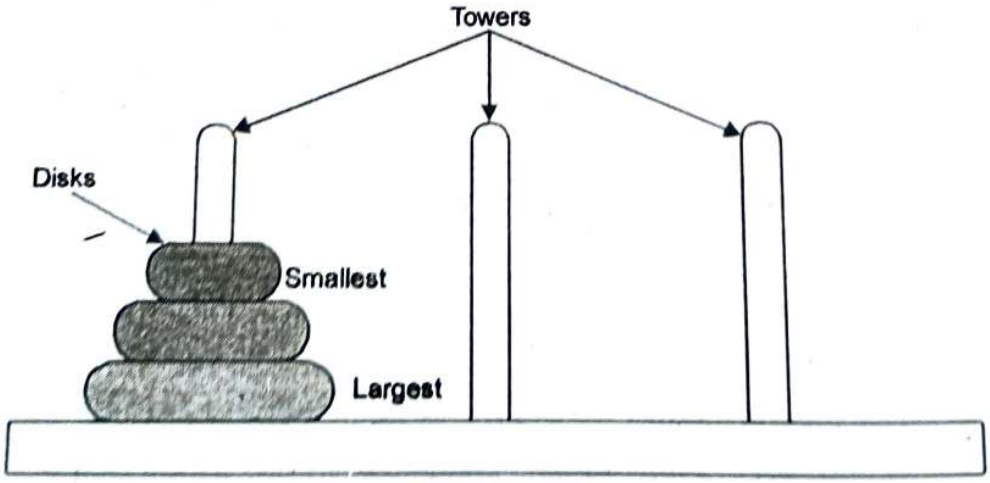
(d) Worst case: Worst case occures when the array is divided in two unbalanced sub arrays where one subarray is empty.



Thus, $T(n) = T(n-1) + T(0) + q(n)$

$= T(n-1) + q(n) = \theta(n^2)$.

(e) Tower of Hanoi, is a mathematical puzzle which consists of three towers (peges) and more than one rings is as depicted–



Rules: The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are–

- Only one disk can be moved among the towers at any given time.
- Only the "top" disk can be removed.
- No large disk can sit over a small disk.

Algorithm: If we have 2 disk–

- first, we move the smaller (top) disk to aux peg.
- Then, we move the larger (bottom) disk to destination peg.
- And finally, we move the smaller disk from aux to destination peg.

The steps to follow are –

- **Step 1** – Move $n-1$ disks from **source** to **aux**
- **Step 2** – Move n^{th} disk from **source** to **dest**
- **Step 3** – Move $n-1$ disks from **aux** to **dest**

Q.3. (a) Design an efficient algorithm to find maximum and minimum number, out of given n numbers simultaneously. Also calculate its time complexity. **(5)**

Ans. Maximum and Minimum:

MaxMin (i, j, max, min)

// a [1 : n] is a global array, parameters i and j are integers, $1 <= i <= j <= n$. The effect is to 4.

// Set max and min to the largest and smallest value 5 in a [i : j], respectively.

```
{
If (i = j) then Max = Min = a[i];
Else if (i = j – 1) then
{
if (a[i] < a[j]) then
        {
                Max = a[j];
                Min = a[ i];
    }
    Else
    {
                Max = a[i];
                Min = a[j];
    }
}Else
{
Mid = (i + j)/2;
MaxMin (I, Mid, Max, Min);
MaxMin (Mid + 1, j, Max 1, Min 1);
If (Max < Max 1) then Max = Max 1;
If (Min > Min 1) then Min = Min 1;
}}
```

The procedure is initially invoked by the statement, MaxMin (1, n, x, y)

Complexity:

If $T(n)$ represents this no., then the resulting recurrence relations is

$$T(n) = T([n/2] + T[n/2] + 2 \qquad n > 2$$

$$1\,n = 2$$

$$1\,n = 1$$

When 'n' is a power of 2, $n = 2k$ for some positive integer 'k', then

$$\begin{aligned} T(n) &= 2T(n/2) + 2 \\ &= 2(2T(n/4) + 2) + 2 \\ &= 4T(n/4) + 4 + 2 \\ &= 2k - 1\ T(2) + \Sigma\ 1 \le I \le k - 1 \le 2i \\ &= 2k - 1 + 2k - 2 \\ T(n) &= (3n/2) - 2 \end{aligned}$$

Q.3. (*b*) Find the optimal way to multiply following matrices $A1 = 2*5$, $A2 = 5*3$, $A3 = 3*4$ and $A4 = 4*6$ using dynamic programming. (5)

Ans. Matrix chain multiplication using Dynamic Programing

So we will make grid of 4 × 4 as we have 4 matrices and then will calculate cost of each sub matrix along with order and then will calculate final minimum cost of multiplying all the 4 matrices is a particular order

$$\begin{aligned} m(1,1) &= (2,2) = m(3,3) = m(4,4) \\ &= 0 \text{ (no cost in multiplying one matrix)} \\ m(1,2) &= 2 \times 5 \times 3 = 30 \\ m(2,3) &= 5 \times 3 \times 4 = 60 \\ m(3,4) &= 3 \times 4 \times 6 = 72 \\ m(1,3) &\Rightarrow A_1(A_2 \cdot A_3) \\ &= m(1,1) + m(2,3) + 2 \times 5 \times 4 \\ &= 0 + 60 + 40 = 100 \\ m(1,3) &\Rightarrow (A_1 . A_2)A_3 \\ &= m(1,2) + m(3,3) + 2 \times 3 \times 4 \\ &= 30 + 0 + 24 = 54 \text{ (minimum)} \end{aligned}$$

So we will take $m(1,3) = 54$

$A_2(A_3.A_4)$

$$\begin{aligned} m(2,4) &= m(2,2) + m(3,4) + 5 \times 3 \times 6 \\ &= 0 + 72 + 90 = 162 \text{ minimum} \end{aligned}$$

or

$(A_2 \cdot A_3), A_4$

$$\begin{aligned} &= m(2,3) + m(4,4) + 5 \times 4 \times 6 \\ &= 60 + 0 + 120 = 180 \end{aligned}$$

So, $m(2,4) = 162$

$$\begin{aligned} m(1,4) &= A_1(A_2 \cdot A_3 \cdot A_4) \\ &= m(1,1) + m(2,4) + 2 \times 5 \times 6 = 222 \end{aligned}$$

$(A_1 \cdot A_2)(A_3 \cdot A_4)$

$$= m(1,2) + m(3,4) + 2 \times 3 \times 6 = 138$$

$(A_1 \cdot A_2 \cdot A_3)A_4$

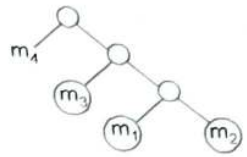
$$= m(1,3) + m(4,4) + 2 \times 4 \times 6 = 102 \text{ minimum}$$

So $m(1,4) = 102$ so final order as

$(A_1 \cdot A_2 \cdot A_3) A_4$

$m(1, 3) = (1, 2) \cdot 3$

$((A_1 \cdot A_2) \cdot A_3) A_4$



Final order of matrix chain multiplication.

Q.4. Demonstrate the optimal substructure of Longest Common Subsequence problem. Find the longest common subsequence of the following two substrings using dynamic programming. $A = abcbbac, B = bcbaab$. **(10)**

Ans. Optimal Substructure: Let the input sequences be $X[0 \ldots m-1]$ and $Y[0 \ldots n-1]$ of lengths m and n respectively. And let $L(X[0 \ldots m-1], Y[0 \ldots n-1])$ be the length of LCS of the two sequences X and Y.

Following is the recursive definition of $L(X[0 \ldots m-1], Y[0 \ldots n-1])$.

If last characters of both sequences match (or $X[m-1] = Y[n-1]$) then,

$L(X[0 \ldots m-1], Y[0 \ldots n-1]) = 1 + L(X[0 \ldots m-2], Y[0 \ldots n-2])$

If last characters of both sequences do not match (or $X[m-1] != Y[n-1]$) then,

$L(X[0 \ldots m-1], Y[0 \ldots n-1]) = \text{MAX}(L(X[0 \ldots m-2]), Y[0 \ldots n-1]), L(X[0 \ldots m-1], Y[0 \ldots n-2])$

Examples: Consider the input strings "AGGTAB" and "GXTXAYB". Last characters match for the strings. So length of LCS can be written as: L("AGGTAB", "GXTXAYB") = 1 + L("AGGTA", "GXTXAY")

	A	G	G	T	A	B
G	–	–	4	–	–	–
X	–	–	–	–	–	–
T	–	–	–	3	–	–
X	–	–	–	–	–	–
A	–	–	–	–	2	–
Y	–	–	–	–	–	–
B	–	–	–	–	–	1

Common Subsequence

$A = abcbbac$

$B = bcbaab$

Formula

$$m(i,j) = \begin{cases} 1 + m(i-1, j-1), \\ \rightarrow \text{if exact match} \\ \max(m(i-1, j), m(i, j-1) \\ \rightarrow \text{if no match} \end{cases}$$

$\Rightarrow$

using this formula we will fill the grid below:

			a	*b*	*c*	*b*	*b*	*a*	*c*
		0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0	0
b	1	0	0	1	1	1	1	1	1
c	2	0	0	1	2	2	2	2	2
b	3	0	0	1	2	3	3	3	3
a	4	0	1	1	2	3	3	4	4
a	5	0	1	1	2	3	3	4	4
b	6	0	1	2	2	3	4	4	4

if match → one more than diagonal element
if no match → max of previous 2 element
After filling the grid we will trace back to get the sequence

			a	b	c	b	b	a	c
		0	1	2	3	4	5	6	7
	0	0	(0)	0	0	0	0	0	0
b	1	0	0	(1)	1	1	1	1	1
c	2	0	0	1	(2)	2	2	2	2
b	3	0	0	1	2	(3)	3	3	3
a	4	0	1	1	2	(3)	3	4	4
a	5	0	1	1	2	(3)	3	4	4
b	6	0	1	2	2	3	(4)	(4)	(4)

b c b b

So the longest common subsequence is of 4 letters and is {*b c b b*}

END TERM EXAMINATION [DEC. 2018]
FIFTH SEMESTER [B.TECH]
ALGORITHM DESIGN AND ANALYSIS
[ETCS–301]

Time : 3 hrs. **M.M. : 75**

Note: *Attempt any five questions including Q.no.1 which is compulsory. Internal choice is indicated.*

Q.1. (*a*) Define Big *O*, Small *O* and Small *w* notation with examples? **(2.5)**

Ans. Big-*O*: Big-*O*, commonly written as *O*, is an Asymptotic Notation for the worst case, or ceiling of growth for a given function. It provides us with an asymptotic upper bound for the growth rate of runtime of an algorithm. Say $f(n)$ is your algorithm runtime, and $g(n)$ is an arbitrary time complexity you are trying to relate to your algorithm. $f(n)$ is $O(g(n))$, if for some real constants c $(c > 0)$ and n_0, $f(n) <= C\,g(n)$ for every input size $n(n > n_0)$

Small-o: Small-o, commonly written as o, is an Asymptotic Notation to denote the upper bound (that is not asymptotically tight) on the growth rate of runtime of an algorithm $f(n)$ is $o(g(n))$, if for some real constants c $(c > 0)$ and $n_0(n_0 > 0)$, $f(n)$ is $< C\,g(n)$ for every input size $n(n > n_0)$. The definitions of *O*-notation and *o*-notation are similar. The main difference is that in $f(n) = O(g(n))$, the bound $f(n) <= g(n)$ holds for some constant $c > 0$, but in $f(n) = o(g(n))$, the bound $f(n) < c\,g(n)$ holds for all constants $c > 0$.

Small-omega: Small-omega, commonly written as ω, is an Asymptotic Notation to denote the lower bound (that is not asymptotically tight) on the growth rate of runtime of an algorithm. $f(n)$ is $\omega(g(n))$, if for some real constants c $(c > 0)$ and $n_0(n_0 > 0)$, $f(n)$ is $> C\,g(n)$ for every input size $n(n > n_0)$. The definitions of Ω-notation and ω-notation are similar. The main difference is that in $f(n) = \Omega(g(n))$, the bound $f(n) >= g(n)$ hold for some constant $c > 0$, but in $f(n) = \omega(g(n))$, the bound $f(n) > c\,g(n)$ holds for all constants c > 0.

Q.1. (*b*) Define iteration method with example? **(2.5)**

Ans. In the iteration method we iteratively "unfold" the recurrence until we "see the pattern". The iteration method does not require making a good guess like the substitution method (but it is often more involved than using induction).

Example: Solve $T(n) = 8T(n/2) + n^2\,(T(1) = 1)$

$$T(n) = n^2 + 8T(n/2)$$

$$= n^2 + 8\left(8T\left(\frac{n}{2^2}\right) + \left(\frac{n}{2}\right)^2\right)$$

$$= n^2 + 8^2T\left(\frac{n}{2^2}\right) + 8\left(\frac{n^2}{4}\right)$$

$$= n^2 + 2n^2 + 8^2T\left(\frac{n}{2^2}\right)$$

$$= n^2 + 2n^2 + 8^2\left(8T\left(\frac{n}{2^3}\right) + \left(\frac{n}{2^2}\right)^2\right)$$

$$= n^2 + 2n^2 + 8^3T\left(\frac{n}{2^3}\right) + 8^2\left(\frac{n^2}{4^2}\right)$$

$$= n^2 + 2n^2 + 2^2n^2 + 8^3T\left(\frac{n}{2^3}\right)$$

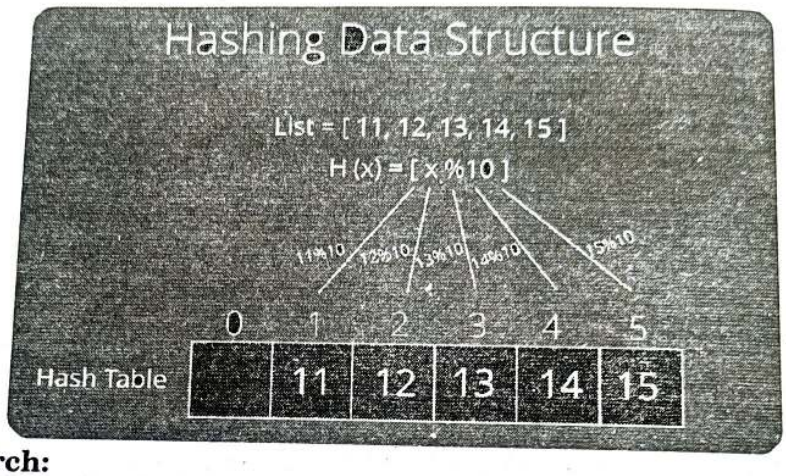
$$= \ldots.$$

$$= n^2 + 2n^2 + 2^2n^2 + 2^3n^2 + 2^4n^2 + \ldots$$

Q.1. (c) Explain hashing and elaborate its advantages over linear and binary search? **(2.5)**

Ans. Hashing is an important Data Structure which is designed to use a special function called the Hash function which is used to map a given value with a particular key for faster access of elements. The efficiency of mapping depends of the efficiency of the hash function used.

Let a hash function $H(x)$ maps the value at the index x % 10 in an Array. For example if the list of values is {11,12,13,14,15} it will be stored at positions {1,2,3,4,5} in the array or Hash table respectively.



Linear search:

- Checks through each element, from beginning to end, until value is found
- Slow and inefficient
- Stops at first value found
- Cannot take advantage of an ordered list
- Incredibly easy to implement (as in 1 line of code easy)
- Average complexity: $O(n/2)$

Hash search:

- Maps each label to a value
- Very fast to find what you're looking for
- More efficient than most indexing systems
- May require remapping when something new is added
- Average complexity: $O(1)$

Binary search:

- Starts at centre, goes halfway forward or halfway backward depending if value is smaller/larger, until value is reached
- Takes full advantage of an ordered list
- Requires a pre-ordered list
- Quite fast, easy to implement (less than a dozen lines of code)
- Average complexity: $O(\log n)$

Q.1. (d) What do you mean by optimality and correctness of algorithm? **(2.5)**

Ans. Optimality: The principle of optimality is the basic principle of dynamic programming, which was developed by Richard Bellman: that an optimal path has the property that whatever the initial conditions and control variables (choices) over some initial period, the control (or decision variables) chosen over the remaining period must be optimal for the remaining problem, with the state resulting from the early decisions taken to be the initial condition.

An algorithm can be said to be optimal if the function that describes its time complexity in the worst case is a lower bound of the function that describes the time complexity in the worst case of a problem that the algorithm in question solves.

Correctness:

The main steps in the formal analysis of the correctness of an algorithm are:

- Identification of the properties of input data (the so-called problem's preconditions).
- Identification of the properties which must be satisfied by the output data (the so-called problem's postconditions).
- Proving that starting from the preconditions and executing the actions specified in the algorithms one obtains the postconditions.

When we analyze the correctness of an algorithm a useful concept is that of state. The algorithm's state is the set of the values corresponding to all variables used in the algorithm.

Q.1. (e) Compare knapsack problem in Greedy, Dynamic and any other approach (backtracking) and given the different perspectives? **(2.5)**

Ans. The Knapsack Problem (KP): The Knapsack Problem is an example of a combinatorial optimization problem, which seeks for a best solution form among many other solutions. It is concerned with a knapsack that has positive integer volume (or capacity) V. There are n disinct items that may potentially be placed in the knapsack. Item i has a positive integer volume V_i and positive integer benefit B_i. In addition, there are Q_i copies of item i available, where quantity Q_i is a positive integer satisfying $1 \le Q_i \le \infty$.

Dynamic Programming Approach:

Consider an instance of the problem defined by the first i items, $1 \le i \le N$, with:

weights $w_1, \ldots, w_i$,

values $v_1, \ldots, v_i$,

and knapsack capacity j, $1 \le j \le$ Capacity.

ALGORITHM Dynamic Programming (Weights [1 ... N]. Values [1 ... N]. Table (0 ... N, 0 ... Capacity])

// Input: Array Weights contains the weights of all items

Array Values contains the values of all items

Array Table is initialized with O_s; it is used to store the results from the dynamic programming algorithm.

//Output: The last value of array Table (Table (N, Capacity]) contains the optimal solution of the problem for the given Capacity

for $i = 0$ to N do

for $j = 0$ to Capacity

if $j <$ Weights[i] then

Table $[i, j] \leftarrow$ Table $[i-1, j]$

else

Table $[i, j] \leftarrow$ *maximum* {Table $[i-1, j]$
AND
Values $[i]$ + Tablel $[i-1, j]$ – Weights$[i]$]

return Table $[N$, Capacity$]$

Greedy Algorithm Approach:

ALGORITHM Greedy Algorithm (Weights (1, ..., N]. Values [1, ..., N])

// Input: Array Weights contains the weights of all items
Array Values contains the values of all items

//Output: Array Solution which indicates the items are included in the knapsack ('1') or not ('0')

Integer CumWeight

Compute the value-to-weight ratios $r_i = v_i / w_i$, $i = 1, ..., N$, for the items given

Sort the items in non-increasing order of the value-to-weight ratios

for all items do

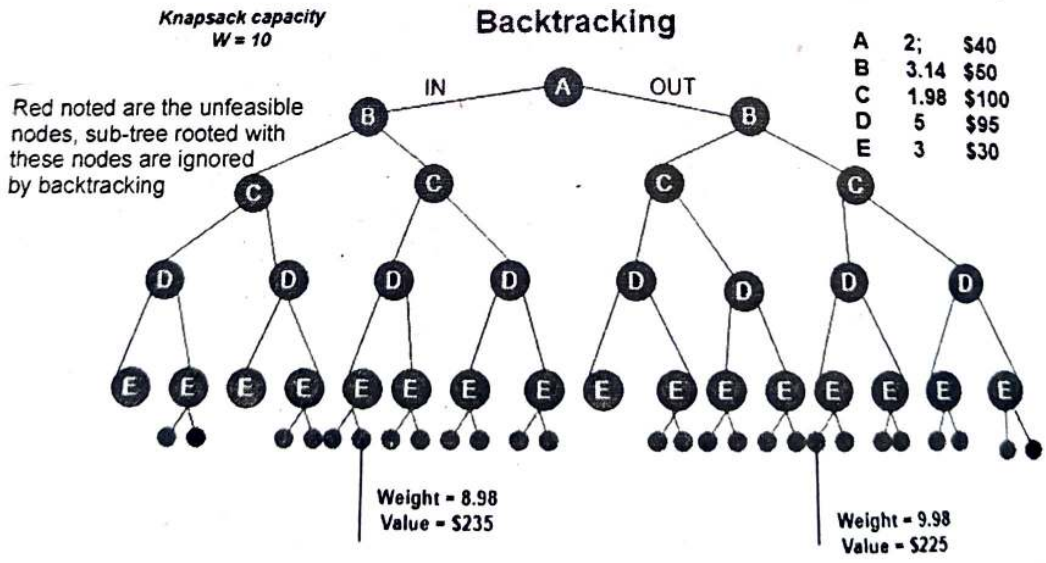
if the current item on the list fits into the knapsack then

place it in the knapsack

else

proceed to the next one

Backtracking Approach: We can use **Backtracking** to optimize the Brute Force solution. In the tree representation, we can do DFS of tree. If we reach a point where a solution no longer is feasible, there is no need to continue exploring. In example, backtracking would be much more effective if we had even more items or a smaller knapsack capacity.

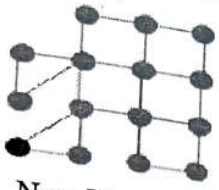


Q.1. (f) Explain polynomial time verfication problems with an example? (2.5)

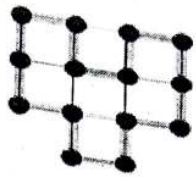
Ans. Polynomial-time "verify" membership in languages. For example, suppose that for a given instance $< G, u, v, k >$ of the decision problem PATH, we are also given a path p from u to v. We can easily check whether the length of p is at most k, and if so, we can view p as a "certificate" that the instance indeed belongs to PATH. For the decision problem PATH, this certificate doesn't seem to buy us much. After all, PATH belongs to

p. in fact, PATH can be solved in linear time-and so verifying membership from a given certificate takes as long as solving the problem from scratch. We shall now examine a problem for which we know of no polynomial-time decision algorithm yet.

Hamiltonian cycle problem: Consider the Hamiltonian cycle problem. Given an undirected graph G, does G have a cycle that visits each vertex exactly once? There is no known polynomial time algorithm for this dispute.



Non-Hamiltonian



Hamiltonian

Let us understand that a graph did have a Hamiltonian cycle. It would be easy for someone to convince of this. They would similarly say: "the period is $hv_3, v_7, v_1, ..., v_{13i}$. We could then inspect the graph and check that this is indeed a legal cycle and that it visits all of the vertices of the graph exactly once. Thus, even though we know of no efficient way to solve the Hamiltonian cycle problem, there is a beneficial way to verify that a given cycle is indeed a Hamiltonian cycle.

Polynomial Time Reduction:: We say that Decision Problem L_1 is Polynomial time Reducible to decision Problem L_2 ($L_{1 \leq_p} L_2$) if there is a polynomial time computation function f such that of all x, x E L_1 if and only if $x \in L_2$.

Q.1. (g) Compare MCM problem with memorization and recursive MCM? (2.5)

Ans. The Matrix Chain Multiplication Problem is the classic example for **Dynamic Programming.** If there are three matrices: A, B and C. The total number of multiplication for $(A * B) * C$ and $A * (B * C)$ is likely to be different. For example, if the dimensions for three matrices are: 2×3, 3×5, 5×9 (please note that the two matrices can be multiplied if and only if the columns of first matrix is equal to the rows of the second matrix),

Recursion: The $c(k)$ is the number of multiplication if you multiple matrix k and $k+1$. Therefore, the psudo code for the above equations can be written as:

```
1   intmatrix-chain (int* matrix, int m, int n){
2     if (m == n) return 0;
3     intans = MAXINT;
4     for (inti = m; i < n; i ++){
5       int cost = matrix-chain (matrix, m, i) + matrix-chain (matrix, i + 1, n)
                                  + matrix [i]. row* matrix [i]. columns*
6     matrix [i + 1]. columns;
7       if (cost < ans){
8         ans < cost;
9       }
10    }
11    returnans;
```

Memorization: The above recursion is straightforward implementation but as any intermediate results are computed over and over again, this approach is still slow practice. One easy method to improve this is to store the results in a look-up. This prevent the repetitive calculations.

```
// global lookup
intmatrix_chain (int*matrix, int m, int n){
 if (m == n){
   lookup [m, n] = 0;
   return 0;
  }
  if (lookup [m, n] ! = – 1)return lookup [m, n];
  intans = MAXINT;
  for (int i = m; i < n; i ++){
   int cost = matrix_chain (matrix, m, i) + matrix_chain (matrix, i + 1, n)
                                  + matrix [i]. rows* matrix[i]. columns*
matrix [i + 1]. columns;
    if (cost < ans){
     ans < cost;
    }
   }
   lookup [m, n] = ans; // storing the answer
   returnans;
  }
```

Q.1. (*h*) Define binomial Coefficient? **(2.5)**

Ans. Binomial coefficients are a family of positive integers that occur as coefficients in the binomial theorem. Binomial coefficients have been known for centuries, but they're best known from Blaise Pascal's work circa 1640. Below is a construction of the first 11 rows of Pascal's triangle.

```
                    1
                  1   1
                1   2   1
              1   3   3   1
            1   4   6   4   1
          1   5   10  10  5   1
        1   6   15  20  15  6   1
      1   7   21  35  35  21  7   1
    1   8   28  56  70  56  28  8   1
```

Following are common definition of Binomial Coefficients.

1. A binomial coefficient $C(n, k)$ can be defined as the coefficient of X^k in the expansion of $(1 + X)^n$.
2. A binomial coefficient $C(n, k)$ also gives the number of ways, disregarding order, that k objects can be chosen from among n objects; more formally, the number of k-element subsets (or k-combinations) of an n-element set.

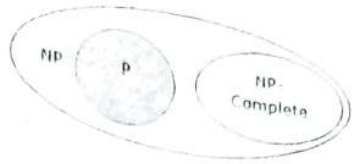
The Problem; Write a function that takes two parameters n and k and returns the value of Binomial Coefficient $C(n, k)$. For example, your function should return 6 for $n = 4$ and $k = 2$, and it should return 10 for $n = 5$ and $k = 2$.

Q.1. (*i*) Define NP hard and NP complete with example? **(2.5)**

Ans. A problem is in the class NPC if it is in NP and is as hard as any problem in

A problem is **NP-hard** if all problems in NP ar
polynomial time reducible to it, even though it may no
s in NP itself.

A language B is NP-complete if it satisfies tw
nditions



- B is in NP
- Every A in NP is polynomial time reducible to B.

If a language satisfies the second property, but not necessarily the first one, the language B is known as NP-Hard. Informally, a search problem B is NP-Hard if there exists some NP-Complete problem A that Turing reduces to B.

The problem in NP-Hard cannot be solved in polynomial time, until $P = NP$. If a problem is proved to be NPC, there is no need to waste time on trying to find an efficient algorithm for it. Instead, we can focus on design approximation algorithm.

***NP*-Complete Problems:** Following are some NP-Complete problems, for which no polynomial time algorithm is known.

- Determining whether a graph has a Hamiltonian cycle
- Determining whether a Boolean formula is satisfiable, etc.

***NP*-Hard Problems:** The following problems are NP-Hard

- The circuit-satisfiability problem
- Set Cover
- Vertex Cover
- Travelling Salesman Problem

Q.1. (*j*) Which string matching algorithm is better and when? **(2.5)**

Ans. Let m be the length of the pattern, n be the length of the searchable text and $= + |\Sigma|$ be the size of the alphabet.

Algorithm	Preprocessing time	Matching time [1]	Space
Naive string-search algorithm	none	Θ(nm)	none
Rabin-Karp algorithm	Θ(m)	average Θ(n + m), worst Θ((n – m)m),	O(1)
Knuth-Morris-Pratt algorithm	Θ(m)	Θ(n)	Θ(m)
Boyer-Moore string-search algorithm	Θ(m + k)	best Ω(n/m) worst O(m)	Θ(k)
Bitap algorithm (shift-or, shift-and, Baeza-Yates-Gonnet)	Θ(m + k)	O(mn)	
Two-way string-matching algorithm	O(m)	O(n + m)	O(1)
BNDM (Backward Non-Determinstic DAWG Matching)	O(m)	O(n)	
BOM (Backwared Oracle Matching)	O(m)	O(mn)	

The Boyer-Moore string-search algorithm has been the standard bench mark for the practical string-search literature.

Q.2. (*a*) Let *N* is number of guests attending party. If each of guests shakes hand with everyone else only once. How many handshakes will take place? Write recursive definition and algorithm? (4)

Ans. Let there be N no. of guests

The first guest will shake hands with the rest of them, that is n-1 guests.

Second guest will shake hands with everyone except the first one(cause they already shook hands once). This second guest shakes hand n-2 times.

This series continues

3: n-3

4: n-4

.

n-1 guest: 1 (or n-(n-1))

n guest: 0

The sum of these handshakes is the total no. of handshakes .Thus

This chain would go as (n–1) + (n–1) + (n–1)+...+ 2 + 1 = n!

The recursive formula is T(n) = T(n–1) + (n–1) and its closed formula solution is n*(n–1)/2

Algorithm (n)

Input: n

Output : number of handshakes

Begin number = n*(n–1)/2

return (number)

End

Q.2. (*b*) Quick sort is not a stable sorting algorithm. If key in *a*[*i*] is changed to *a*[*i*]* *n* + *i* – 1, then new keys are all distinct. After sorting which transformation will restore the keys to their original values? (4.5)

Ans. Quick sort array = 7, 4, 6, 9, 4, 5

a[1]* n + i – 1

Updated array = 41 24 37 56 27 $\boxed{34}$ → pivot

(a) i = –1, j = 0, a(j) = 41 > 34 no change

(b) i = –1, j = 1 a(j) = 24 < 34

i++ i = 0, j =1 swap a(i) and a(j)

array = 24 41 37 56 27 $\boxed{34}$ → pivot

(c) i = 0, j = 2 a(j) = 37 > 34 no change

(d) i = 0, j = 3 a(j) = 56 > 34 no change

(e) i = 0, j = 4 a(j) = 27 < 34

i++ i = 1 j = 4 swap a(i) and a(j)

array = 24 27 37 56 41 $\boxed{34}$ → pivot

(f) i = 1, j = 5 a(j) = pivot → stop

swap a(i + 1) and pivot.

Array = 24 27 34 56 41 37

Next pivot

Final array 24 27 34 37 41 56

Now this made quick sort stable as a(1) and a(4) both having values 4 were changed into 24 and 27 respectively and after sorting a(1) i.e. 24 apprears before a(4) i.e. 27 which is required for any sort to be stable.

Updated array = 24 27 34 37 41 56

Now transformation that will make array as original will be

$a(i) = a[i]/n \quad n = 6$

$24/6$	$27/6$	$34/6$	$37/6$	$41/6$	$56/6$
4	4.5	5.67	6.16	6.83	9.3

Rounding off to nearest integers

4 4 5 6 7 9

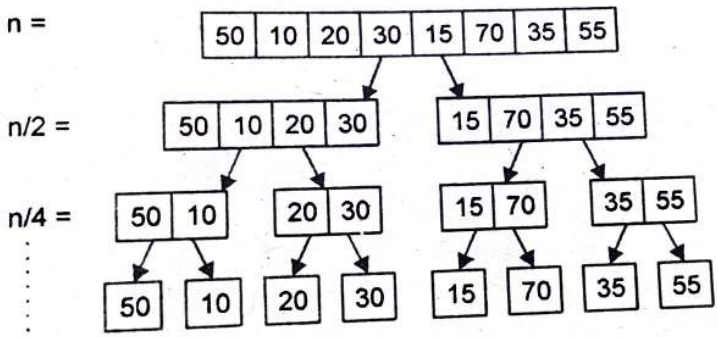
Q.2. (c) Sort the following numbers using Merge sort? The set is (50, 10, 20, 30, 15, 70, 35, 55) **(4)**

Ans. Merge sort → divide and conquor recursively

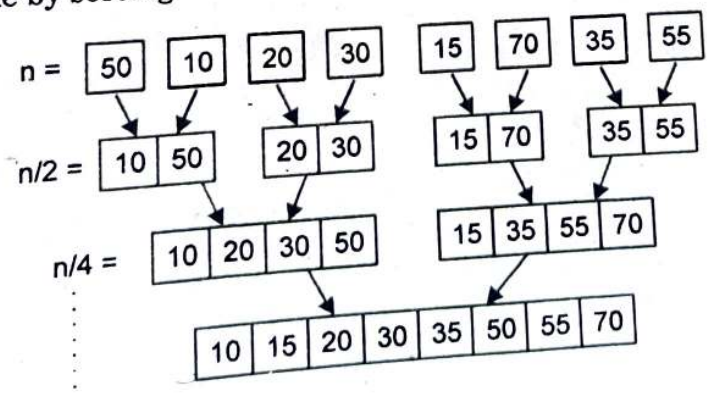
one disadvantage→ Requires extra space.

Array = {50, 10, 20, 30, 15, 70, 35, 55}

Merge sort will divide the array in n/2 size and sort the two parts individually. This is done recursively till the time the complete array gets sorted.



The array size is 1 now → no furthur division. Now combine the array by solving problem i.e. combine by sorting



Final merged array which is sorted

Main Advantage → guaranteed to num is θ(nlog n)

OR

Q.3. (*a*) Let *S* be sample of *s* elements from *X*. If *X* is partitioned into *s*+ 1 parts as in algorithm Rsort given below. Then show that size of each part is Big *O*(*n*/*s* log *n*) **(6.5)**

Algorithm Rsort (*a*, *n*)
{randomly sample *s* elements from a [ ];
Sort this sample;
Partition input using sorted samples as partition key;
Sort each part separately;}

Ans. Algorithm Rsort(a, n)
{randomly sample s elements from a[];
sort this sample;
partition input using sorted samples as partition key;
Sort each part separately;}
= Partition the input into S + 1 parts using the sample keys as splitters.
This can be done as follows;

- Assign on processor to each key so that the processors can perform a binary search in the sorted sample and identify the parts to which their keys belong. Now, sort the keys with respect to their part numbers using the algorithm.
- Sort each of the S + 1 parts
- Output the sorted parts in order

→ We realize that the size of each of the S + 1 parts is O ($\log^3 n$) in
∴ each X can be sorted in O(log |X|) time.

- Integer sorting in this step also needs O(logn) time using only n/log n processors
- Using the slow-down lemma, we can also sort X, in O((log $|X_i|^2$)time using $|X_i|$ processor.
- The size of each part is O (n/s logn)

Q.3. (*b*) Solve recurrence **(3)**

$$T(n) = 1 \qquad n <= 4$$
$$T(n) = T(n^{1/2}) + C \qquad n > 4$$

Ans. Introduce a change of variable by letting $n = 2^m$

$$\Rightarrow \quad T(2^m) = T(2^{m/2}) + C$$

Let us introduce another change by letting $S(m) = T(2^m)$.

$$S(m) = S\left(\frac{m}{2}\right) + C$$

$$S(m) = \left(S\left(\frac{m}{4}\right) + C\right) + C$$

$$S(m) = S\left(\frac{m}{2^2}\right) + 2C$$

By substituting further, we get

$$\Rightarrow \quad S(m) = S\left(\frac{m}{2^k}\right) + kC$$

To simplify the expression, assume $m = 2^k$

$$\Rightarrow \quad S\left(\frac{m}{2^k}\right) = S(1) = T(2)$$

$\therefore$ $T(2) = T(\sqrt{2}) + C$ which is appoximalely C

$S(m) = C + kC$

$= (k + 1) C$

$S(m) = (\log m + 1) C$

$\therefore$ $T(n) = (\log (\log n) + 1) C.$

$T(n) \simeq O(\log(\log n) + 1)$

Q.3. (c) Prove by induction

$\Sigma i = 0 \ldots n\ xi = (x^{n+1} - 1)/x - 1$ $x <> 1$ for $x >= 0$ **(3)**

Ans. To prove that $\sum_{j=0}^{n} x^j = \frac{1-x^{n+1}}{1-x}$, for $x \neq 1$

First check it for n = 0; i.e. $\sum_{j=0}^{0} x^j = x^0 = 1 = \frac{1-x^{0+1}}{1-x}$

In the inductive step assume it is true for n = k, so for n = k + 1

$$\sum_{j=0}^{k+1} x^j = \sum_{j=0}^{k} x^j + x^{k+1}$$

$$= \frac{1-x^{k+1}}{1-x} + x^{k+1}$$

$$= \frac{1-x^{k+1}+(1-x)(x^{k+1})}{1-x}$$

$$= \frac{1-x^{k+2}}{1-x}$$

Which, by the principle of induction, proves the result.

Q.4. (a) Find optimal binary Merge pattern for 10 files whose lengths are given in the set S, (28, 32, 12, 5, 8, 4, 53, 91, 35, 3, 11). Show that if all internal nodes in a tree have degree k, then number n of external nodes is such that n mod$(k - 1) = 1$. Draw optimal 3 way merge tree obtained using rule area when $(q1 \ldots q11) = (3, 7, 8, 9, 15, 16, 18, 20, 23, 25, 28)$. **(7.5)**

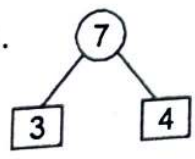
Ans.

S = (28, 32, 12, 5, 8, 4, 53, 91, 35, 3, 11)

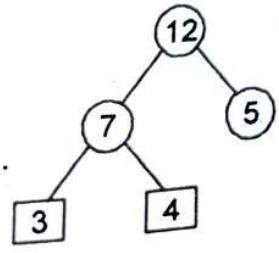
Ascending order (3, 4, 5, 8, 11, 12, 28, 32, 35, 53, 91)

Now take 2 minimum and merge

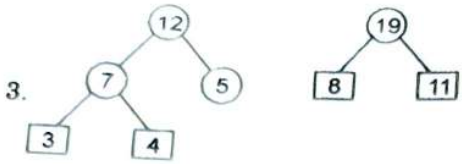
1. 5, 8, 11, 12, 28, 32, 35, 53, 91



2. 8, 11, 12, 28, 32, 35, 53, 91

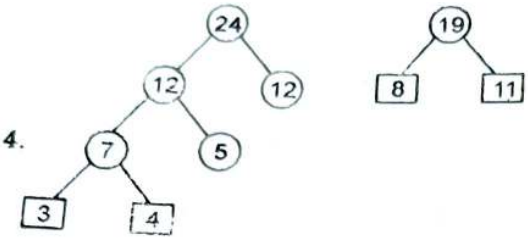


3.



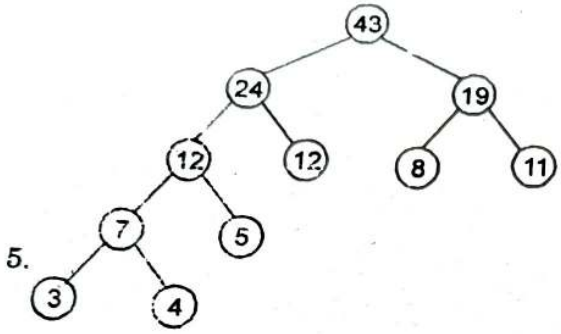
12,28, 32, 35, 53, 91

4.

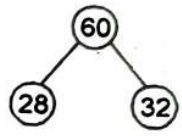


28, 32, 35, 53, 91

5.



28, 32, 35, 53, 91

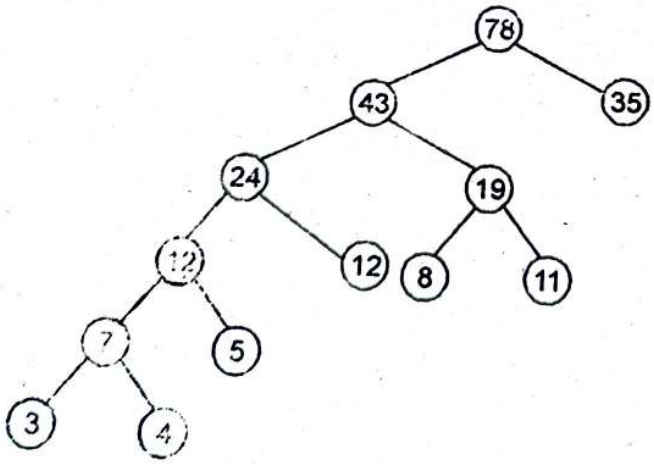


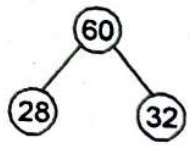
35, 53, 91

Same

6. ....

7.

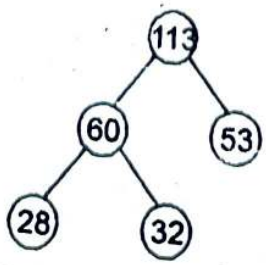




53,91

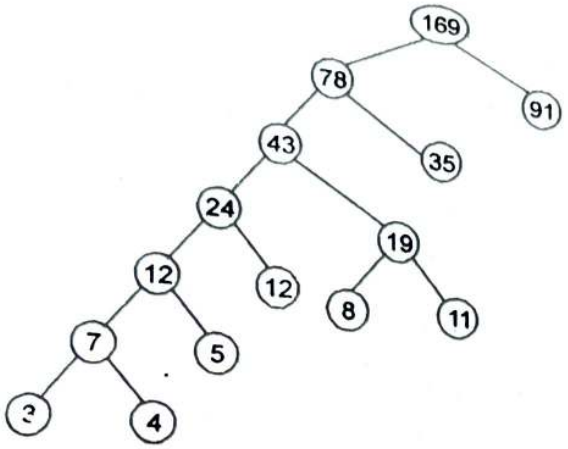
8.

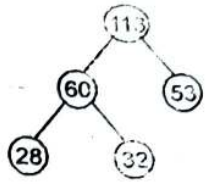
Same



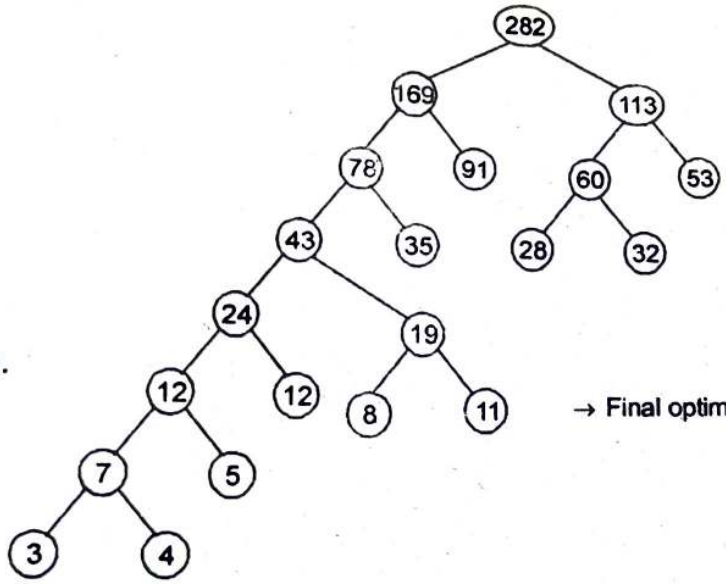
91.

9.





10.

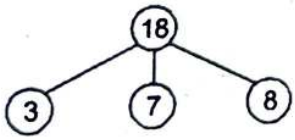


→ Final optimal merge pattern

(iii) $(q_1 \cdot q_{11}) = (2,7,8,9,15,16,18,20,23,25,28)$

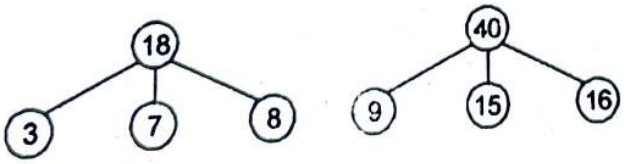
In this we will take 3 minimum values and merge to form 3-way merge tree

1.



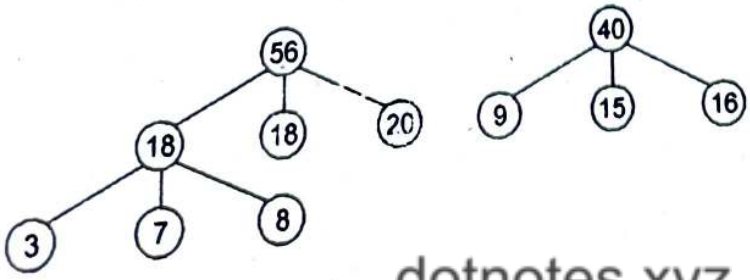
9, 15, 16, 18, 20, 23, 25, 28

2.



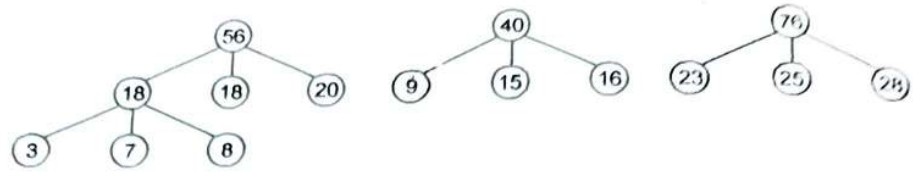
18, 20, 23,25, 28

3.

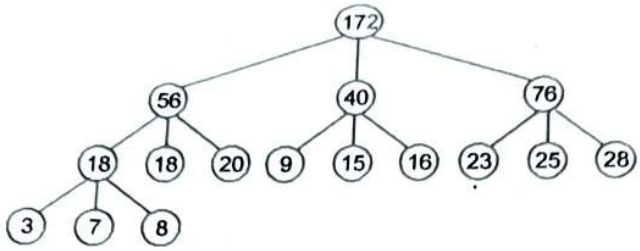


23, 25, 28

4.



5.



Final 3 way merge tree

Q.4. (*b*) The jobs are scheduled on 3 processors. The task times are given by matrix *J* as follows *J* = [20; 33; 52]. Draw two possible schedules. **(5)**

Ans.

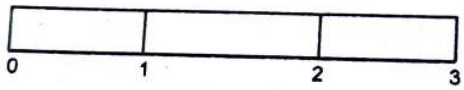
Job	1	2	3
Profit	2	3	5
Deadlines	0	3	2

Convert the matrix into profit table

(1) Arrange in descending order of profit

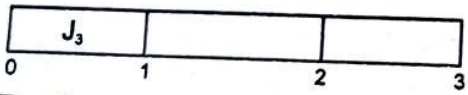
J	3	2	1
P	5	3	2
D	2	3	0

(2)



→ Dead line matrix

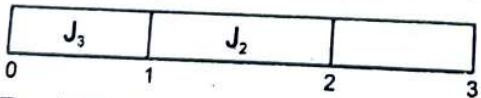
(3) Select J_3 P = 5, D = 2 → Can be placed in first 2 columns



P = 5

D = 2

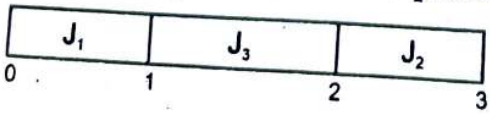
(4) Select J_2 P = 3, D = 3 → Can be placed in any column



P = 5 P = 3

D = 2 D = 3

(5) Select J_3, P = 2, D = 0 → placed in first column only



P = 2, P = 5, P = 3
D = 0, D = 2, D = 3
Maximum profit = 2 + 5 + 3 = 10
Schedule = $J_1 \rightarrow J_3 \rightarrow J_2$

Q.5. (a) Show that knapsack optimization problem reduces to knapsack decision problem when all the P's, W's and m are integers and complexity is measured as a function of input length. If input length is q, then $\Sigma pi <= n^2$, where n is no. of objects.

OR

Ans. 0/1 – KNAPSACK (4)

We are given n objects and a knapsack. Each object i has a positive weight wi and a positive value Vi. The knapsack can carry a weight not exceeding W. Fill the knapsack so that the value of objects in the knapsack is optimized.

A solution to the knapsack problem can be obtained by making a sequence of decisions on the variables x1, x2, . . ., xn. A decision on variable xi involves determining which of the values 0 or 1 is to be assigned to it. Let us assume that

decisions on the xi are made in the order xn, xn-1, . . .x1. Following a decision on xn, we may be in one of two possible states: the capacity remaining in m – wn and a profit of pn has accrued. It is clear that the remaining decisions xn-1, . . . , x1 must be optimal

With respect to the problem state resulting from the decision on xn. Otherwise, xn,. . . . , xl will not be optimal. Hence, the principal of optimality holds.

Fn (m) = max {fn-1 (m), fn-1 (m - wn) + pn} — ...(1)

For arbitrary fi (y), i > 0, this equation generalizes to:

Fi (y) = max {fi-1 (y), fi-1 (y - wi) + pi} - ...(2)

Equation-2 can be solved for fn (m) by beginning with the knowledge fo (y) = 0 for all y and fi (y) = - ~, y < 0. Then fl, f2, . . . fn can be successively computed using equation-2.

When the wi's are integer, we need to compute fi (y) for integer y, 0 <_ y <_ m. Since fi (y) = ~ ~ for y < 0, these function values need not be computed explicitly. Since each fi can be computed from fi - 1 in Θ (m) time, it takes Θ (m n) time to compute fn. When the wi's are real numbers, fi (y) is needed for real numbers y such that 0 < y <_ m. So, fi cannot be explicitly computed for all y in this range. Even when the wi's are integer, the explicit Θ (m n) computation of fn may not be the most efficient computation. So, we explore an alternative method for both cases.

The fi (y) is an ascending step function; i.e., there are a finite number of y's, 0 = yl < y2 <....< yk, such that fi (yl) < fi (y2) <.....< fi (yk); fi (y) = – ~ , y < yl; fi (y) = f (yk), y 3 yk; and fi (y) = fi (yj), yj ≤ y ≤ yj + 1. So, we need to compute only fi (yj), 1 <j < k. We use the ordered set S^i = {(f (yj), yj) | 1 ≤ j ≤ k} to represent fi (y). Each number of S^i is a pair (P, W), where P = fi (yj) and W = yj. Notice that S° = {(0, 0)}. We can compute S^{i+1} from Si by first computing:

Si 1 = {(P, W) | (P - pi, W - wi) e S^i

Now, S^{i+1} can be computed by merging the pairs in S^i and Si 1 together. Note that if Si+1 contains two pairs (Pj, Wj) and (Pk, Wk) with the property that Pj ≤ Pk and Wj > Wk, then the pair (Pj, Wj) can be discarded because of equation-2. Discarding or purging rules such as this one are also known as dominance rules. Dominated tuples get purged. In the above, (Pk, Wk) dominates (Pj, Wj).

Q.5. (*b*) Show job sequencing with deadlines problem is NP-hard? **(4)**

Ans. NP-hard in the ordinary sense

(pseudo polynomial time complexity):

The problem cannot be optimally solved by an algorithm with polynomial time complexity but with an algorithm of time complexity $O((n. \max p_j)^k)$.

Given positive integers $a_1 \ldots a_t$ and $b = \frac{1}{2}\sum_{j=1}^{t} a_j$ do there exist two disjoint subsets S_1 and S_2 such that $\sum_{j \in S_i} a_j = b$ for i = 1, 2?

Partition is NP-Hard in the ordinary sense.

given positive integers $a_1, \ldots a_{3t}$, b with

$$\frac{b}{4} < a_j < \frac{b}{2} \qquad j = 1, \ldots, 3t$$

and

$$\sum_{j=1}^{3t} a_j = tb$$

do there exist t pairwise disjoint three element subsets $S_i \subset \{1, \ldots, 3t\}$ such that $\sum_{j \in S_i} a_j = b$ for i = 1,...,t?

3-partition is strongly NP-hard

- 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.
- A scheduling problem is strongly NP-hard if
- 3-partition (or a similar problem) can be reduced to this problem with a polynomial time algorithm.

Q.5. (*c*) How we count the number of parenthesis in MCM Problem. Use substitution method to show solution to recurrence is (2^n). Assume symbol as omega. **(4.5)**

Ans. Number of Parenthesizations

Example:

- Given the matrices A_1, A_2, A_3, A_4

→ Assume the dimensions of $A_1 = d_0 \times d_1$, etc

- Below are the five possible parenthesizations of these arrays, along with the number of multiplications:

1. $(A_1A_2)(A_3A_4) : d_0d_1d_2 + d_2d_3d_4 + d_0d_2d_4$
2. $(A_1A_2)A_3)A_4) : d_0d_1d_2 + d_0d_2d_3 + d_0d_3d_4$
3. $(A_1(A_2A_3))A_4 : d_1d_2d_3 + d_0d_1d_3 + d_0d_3d_4$

4. $A_1(A_2A_3))A_4) : d_1d_2d_3 + d_1d_3d_4 + d_0d_1d_4$

5. $A_1(A_2(A_3A_4)) : d_2d_3d_4 + d_1d_2d_4 + d_0d_1d_4$

The number of parenthesizations is at least $T(n) > T(n-1) + T(n-1)$

- Since the number with the first element removed is $T(n-1)$, which is also the number with the last removed
- Thus the number of parenthesizations is $\Omega(2^n)$
- The number is actually $T(n) = \sum_{k=1}^{n-1} T(k)T(n-k)$ which is related to the *Catalan numbers.*
- This is because the original product can be split into 2 subproducts in k places. Each split is to be parenthesized optimally
- This recurrence is related to the ***Catalan numbers.***

Consider the following reccurence relation, which shows up fairly frequently for some types of algorithms:

$$T(1) = 1, \quad T(n) = 2T(n-1) + c_1$$

By expanding this out a bit (using the "iteration method"), we can guess that this will be $O(2^n)$. To use the substitution method to prove this bound, we now need to guess a closed-form upper bound based on this asymptotic bound. We will guess an upper bound of $k2^n - b$, where b is some constant. We include the b in anticipation of having to deal with the constant c_1 that appears in the recurrence relation, and because it does no harm. In the process of proving this bound by induction, we will generate a set of constraints on k and b, and if b turns out to be unnecessary, we will be able to set it to whatever we want at the end.

Our property, then, is $T(n) \le k2^n - b$, for some two constants k and b. Note that this property logically implies that $T(n)$ is $O(2^n)$, which can be verified with reference to the definition of O.

Base case: $n = 1$, $T(1) = 1 \le k2^1 - b = 2k - b$. This is true as long as $k \ge (b + 1)/2$.

Inductive case: We assume our property is true for $n-1$. We now want to show that it is true for n.

$$\begin{aligned} T(n) &= 2T(n-1) + c_1 \le 2(k2^{n-1} - b) + c_1 \quad \text{(by IH)} \\ &= k2^n - 2b + c_1 \\ &\le k2^n - b \end{aligned}$$

This is true as long as $b \ge c_1$.

So we end up with two constraints that need to be satisfied for this proof to work, and we can satisfy them simply by letting $b = c_1$ and $k = (b + 1)/2$, which is always possible, as the definition of O allows us to choose any constant. Therefore, we have proved that our property is true, and so $T(n)$ is $O(2^n)$.

Q.6. (a) Give $O(n^2)$ time algorithm to find longest monotonically increasing subsequence for n numbers? **(4)**

Ans. There is a simple DP solution in O(n^2) but it can also be implemented in O(n log n). O(n^2) solution here:

Let A[0..n-1] be the input array and LIS[0..n-1] be used to store lengths of longest increasing subsequences ending at LIS[i] for each i.

LIS[i] = 1, 0 <= i< n
for i = 1 to n-1:
for j = 0 to i-1:
if A[i] > A[j] and LIS[j] + 1 > LIS[i]:
LIS[i] = LIS[j] + 1 Required solution = max(LIS[i], 0<=i<n)

This gives the length of the longest increasing subsequence. The subsequence itself can be found by using the same logic in reverse i.e., including the number with maximum LIS[i], then the number A[j] from where we came here and so on.

Q.6. (*b*) How task scheduling problem is solved using matroids. If S is set of unit time tasks with deadlines and *I* is set of all independent tasks then prove that corresponding system (*S*, *I*) is matroid. **(5)**

Ans. A ***unit-time task*** is a job, such as a program to be run on a computer, that requires exactly one unit of time to complete. Given a finite set S of unit-time tasks, a ***schedule*** for S is a permutation of S specifying the order in which these tasks are to be performed. The first task in the schedule begins at time 0 and finishes at time 1, the second task begins at time 1 and finishes at time 2, and so on.

The problem of ***scheduling unit-time tasks with deadlines and penalties for a single processor*** has the following inputs:

- a set $S = \{a_1, a_2, ..., a_n\}$ of n unit-time tasks;
- a set of n integer ***deadlines*** $d_1, d_2, ..., d_n$, such that each d_i satisfies $1 \le d_i \le n$ and task a_i is supposed to finish by time d_i; and
- a set of n nonnegative weights or ***penalties*** $w_1, w_2, ..., w_n$, such that we incur a penalty of w_i if task a_i is not finished by time d_i and we incur no penalty if a task finishes by its deadline.

We are asked to find a schedule for S that minimizes the total penalty incurred for missed deadlines.

Consider a given schedule. We say that a task is ***late*** in this schedule if it finishes after its deadline. Otherwise, the task is ***early*** in the schedule. An arbitrary schedule can always be put into ***early-first form***, in which the early tasks precede the late tasks. To see this, note that if some early task a_i follows some late task a_j, then we can switch the positions of a_i and a_j, and a_i will still be early and a_j will still be late.

We similarly claim that an arbitrary schedule can always be put into ***canonical form***, in which the early tasks precede the late tasks and the early tasks are scheduled in order of monotonically increasing deadlines. To do so, we put the schedule into early-first form. Then, as long as there are two early tasks a_i and a_j finishing at respective times k and $k + 1$ in the schedule such that $d_j < d_i$, we swap the positions of a_i and a_j. Since a_j is early before the swap, $k + 1 \le d_j$. Therefore, $k + 1 < d_i$, and so a_i is still early after the swap. Task a_j is moved earlier in the schedule, so it also still early after the swap.

The search for an optimal schedule thus reduces to finding a set A of tasks that are to be early in the optimal schedule. Once A is determined, we can create the actual schedule by listing the elements of A in order of monotonically increasing deadline, then listing the late tasks (i.e., $S - A$) in any order, producing a canonical ordering of the optimal schedule.

We say that a set A of tasks is ***independent*** if there exists a schedule for these tasks such that no tasks are late. Clearly, the set of early tasks for a schedule forms an independent set of tasks.

If S is a set of unit-time tasks with deadlines, and i is the set of all independent sets of tasks, then the corresponding system (S,i) is a matroid.

Proof : Every subset of an independent set of tasks is certainly independent. To prove the exchange property, suppose that B and A are independent sets of tasks and that $|B| > |A|$. Let k be the largest t such that $N_t(B) \le N_t(A)$. (Such a value of t exists, since $N_0(A) = N_0(B) = 0$.) Since $N_n(B) = |B|$ and $N_n(A) = |A|$, but $|B| > |A|$, we must have that $k < n$ and that $N_j(B) > N_j(A)$ for all j in the range $k + 1 \le j \le n$. Therefore, B contains more tasks with deadline $k + 1$ than A does. Let a_i be a task in $B - A$ with deadline $k + 1$. Let $A' = A \cup \{a_i\}$.

We now show that A' must be independent by using property of Lemma.

For $0 < t < k$, we have $N_t(A') = N_t(A) \le t$, since A is independent. For $k < t = n$, we have $N_t(A') \le N_t(B) \le t$, since B is independent. Therefore, A' is independent, completing our proof that (S,i) is a matroid.

Q.6. (c) Explain Floyd Warshall algorithm and explain its time complexity. (3.5)

Ans. Refer Q.4. (a) End Term Examination 2016.

OR

Q.7. (a) Professor George proposes new divide and conquer algorithm for computing Minimum Spqnning tree (MST). If graph is partitioned into 2 sets V1 and V2. Let Let E1 and E2 are set of edges that are incident only on vertices in V1 and V2. Recursively solve MST problem on each of two sub-graphs G1 (V1, E1) and G2 (V2, E2). Finally select minimum weight edge in E that crosses cut (V1, V2) and use this edge to unite the resulting two minimum spanning trees into a single spanning tree. Sets V1 and V2 differ by maximum 1. (7)

Ans. We claim that the algorithm will fail. A simple counter example is shown in fig. Graph $G = (V.E)$ has four vertices: $\{v_1, v_2, v_3, v_4\}$, and is partitioned into subsets

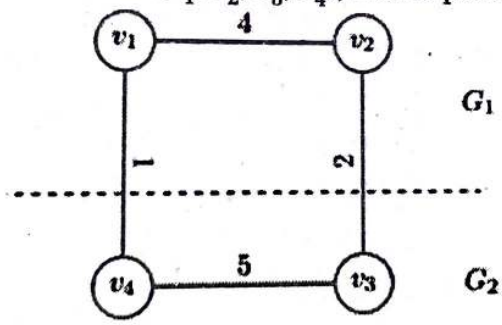


Figure : An counter example.

Gl with $V_1 = \{v_1, v_2\}$ and G_2 with $V_2 = \{v_3, v_4\}$ The minimum-spanning-tree(MST) of G_1 has weight 4, and the MST of G_2 has weight 5, and the minimum-weight edge crossing the cut $\{v_1, v_2\}$ has weight 1. in sum the spanning tree forming by the proposed algorithm $v_2 - v_1 - v_4 - v_3$ which has weight 10. On the contrary, it is obvious that the MST of G is $v_4 - v_1 - v_2 - v_3$ with weight 7. Hence the proposed algorithm fails to obtain an MST.

Q.7. (b) How would you extend Robin karp Method to problem of searching text string for an occurrence of anyone of given set of k patterns. Assume all patterns have some length Generalize your solution for different pattern strings. (5.5)

Ans. Rabin-Karp algorithm also slides the pattern one by one. But unlike the Naive algorithm, Rabin Karp algorithm matches the hash value of the pattern with the hash value of current substring of text, and if the hash values match then only it starts matching individual characters. So Rabin Karp algorithm needs to calculate hash values for following strings.

1) Pattern itself.

2) All the substrings of text of length m.

Since we need to efficiently calculate hash values for all the substrings of size m of text, we must have a hash function which has following property. Hash at the next shift must be efficiently computable from the current hash value and next character in text or we can say *hash(txt[s+1 .. s+m])* must be efficiently computable from *hash(txt[s .. s+m-1])* and *txt[s+m]* i.e., *hash(txt[s+1 .. s+m])= rehash(txt[s+m], hash(txt[s .. s+m-1])* and rehash must be O(1) operation

The hash function suggested by Rabin and Karp calculates an integer value. The integer value for a string is numeric value of a string. For example, if all possible characters are from 1 to 10, the numeric value of "122" will be 122. The number of possible characters is higher than 10 (256 in general) and pattern length can be large. So the numeric values cannot be practically stored as an integer. Therefore, the numeric value is calculated using modular arithmetic to make sure that the hash values can be stored in an integer variable (can fit in memory words).

To do rehashing, we need to take off the most significant digit and add the new least significant digit for in hash value. Rehashing is done using the following formula.

*hash(txt [s+1.. s+m]) = (d(hash(txt[s.. s+m-1]) – txt[s]*h) + txt[s + m])modq*

hash(txt [s.. s+m-1]): Hash value at shift *s*.

hash(txt[s+1.. s+m]): Hash value at next shift (or shift s+1)

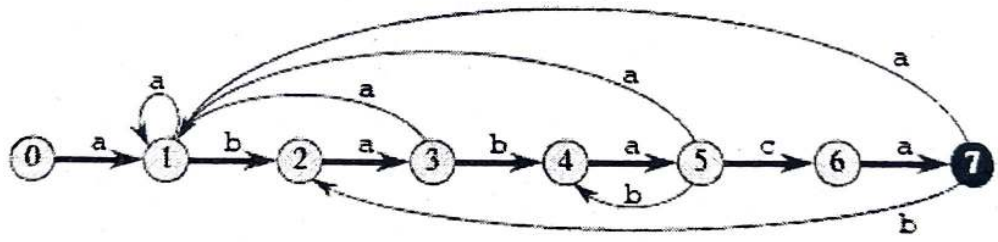
d : Number of characters in the alphabet

q : A prime number

h : d^(m-1)

Q.8. (*a*) Illustrate String matching with Finite automata? **(4.5)**

Ans. String matching algorithm using finite automata: There is a string matching automaton for every pattern *P*; this automaton must be constructed from the pattern in a pre-processing step before it can be used to search the text string. Figure below illustrates this construction for the pattern *P* = *ababaca*.



(a)

State	Input a	b	c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

(b)

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

(c)

We shall assume that P is a given fixed pattern string; for brevity, we shall not indicate the dependence upon P in our notation. In order to specify the string-matching automaton corresponding to a given pattern $P[1...m]$, we first define an auxiliary function σ, called the suffix function corresponding to P. The function σ is a mapping from Σ to $\{0, 1,...,m\}$ such that $\sigma(x)$ is the length of the longest prefix of P that is a suffix of $x : \sigma(x) = \max \{k : P_k \rceil x\}$.

The suffix function σ is well defined since the empty string $P_0 = \varepsilon$ is a suffix of every string. As examples, for the pattern $P = ab$, we have $\sigma(\varepsilon) = 0, \sigma(ccaca) = 1$, and $\sigma(ccab) = 2$. For a pattern P of length m, we have $\sigma(x) = m$ if and only if $P \rceil x$. It follows from the definition of the suffix function that if $x \rceil y$, then $\sigma(x) \leq \sigma(y)$.

We define the string-matching automaton that corresponds to a given pattern $P[1...m]$ as follows.

- The state set Q is $\{0, 1,...,m\}$. The start state q_0 is state 0, and state m is the only accepting state.
- The transition function σ is defined by the following equation, for any state q and character a : $\delta(q, a) = \sigma(P_q a)$.

To clarify the operation of a string-matching automaton, we now give a simple, efficient program for simulating the behaviour of such an automaton (represented by its transition function δ) in finding occurrences of a pattern P of length m in an input text $T[1...n]$.

```
FINITE-AUTOMATION-MATCHER (T, δ, m)
1. n ← length [T]
2. q ← 0
3. for i ← 1 to n
4.      do q ← d(q, T[i])
5.           if q = m
6.                then print "Pattern occurs shift" i – m
```

As for any string-matching automaton for a pattern of length m, the state set Q is $\{0, 1,...,m\}$, the start state is 0, and the only accepting state is state m. The simple loop structure of FINITE-AUTOMATON-MATCHER implies that its matching time on a text string of length n is $\theta(n)$.

Q.8. (*b*) Give pseudo code to reconstruct LCS from completed *c* taable and original sequences $X = (x_1, x_2, x_n)$ and $Y = (y_1, y_2, ..., y_n)$ without using *b* table in $O(m + n)$ time. Give its memorized version. (8)

Ans.

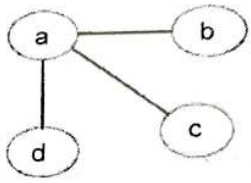
```
PRINT-LCS(c, X, Y, i, j)
if c[i, j] == 0
return
if X[i] == Y[j]
     PRINT-LCS(c, X, Y, i - 1, j - 1)
print X[i]
```

```
elseif c[i - 1, j] > c[i, j - 1]
    PRINT-LCS(c, X, Y, i - 1, j)
else
PRINT-LCS(c, X, Y, i, j - 1)
```

OR

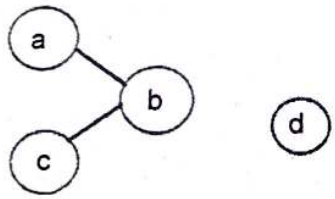
Q.9. (a) Let G be a connected undirected graph with n vertices. Show G must have at least $n - 1$ Edges and that all connected and undirected graphs with $n - 1$ edges are trees. What is minimum no. of edges in a strongly connected digraph with n vertices? What from do such digraphs have? **(6)**

Ans. We know that the minimum number of edges required to make a graph of n vertices connected are (n-1) edges. We can observe that removal of one edge from the graph G will make it disconnected. Thus a connected graph of n vertices and (n-1) edges cannot have a circuit. Hence a graph G is a tree.

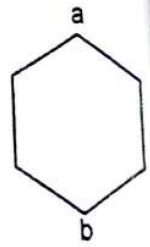


Let the graph G is disconnected then there exist at least two components G1 and G2 say. Each of the component is circuit-less as G is circuit-less. Now to make a graph G connected we need to add one edge e between the vertices Vi and Vj, where Vi is the vertex of G1 and Vj is the vertex of component G2.

Now the number of edges in **G = (n – 1)+1 =n.**



Now, G is connected graph and circuit-less with n vertices and n edges, which is impossible because connected circuit-less graph is a tree and tree with n vertices has **(n-1)** edges. So the graph G with n vertices, **(n-1)** edges and without circuit is connected. Hence the given statement is proved.



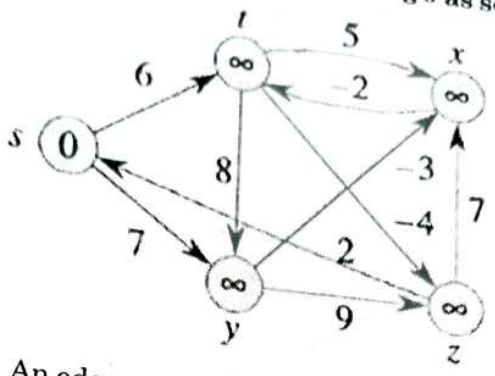
• A cycle is a sequence v1,v2,...vkv1,v2,...vk of vertices such that: v1=vkv1=vk, and there is a directed edge between vivi and vi+1vi+1 for i∈[1,k–2]i∈[1,k–2].

• A graph is said to be strongly connected or diconnected if every vertex is reachable from every other vertex. In particular, v2v2 is reachable from v1v1 and v1v1 is reachable from v2v2, which means that there is a cycle in the graph.

A (directed) cycle graph with nn vertices and nn edges is strongly connected. Conversely, any graph with n–1n–1 edges is not strongly connected. Furthermore, the complete graph is strongly connected. So a strongly connected graph with nn nodes will have between n and (n–1) edges.

Q.9. (b) Give properties of shortest path and relaxation. Run bellman ford algorithm on directed Graph. In each pass, relax edges in the same order as in

figure. Using vertex z as source, show d and pi values after each pass. now change the weight of edge (z, x) to 4 and run algorithm using s as source. **(6.5)**



Ans. Shortest paths: An edge-weighted digraph is a digraph where we associate weights or costs with each edge. A *shortest path* from vertex s to vertex t is a directed path from s to t with the property that no other such path has a lower weight.

Properties.

We summarize several important properties and assumptions.

- *Paths are directed.* A shortest path must respect the direction of its edges.
- *The weights are not necessarily distances.* Geometric intuition can be helpful, but the edge weights weights might represent time or cost.
- *Not all vertices need be reachable.* If t is not reachable from s, there is no path at all, and therefore there is no shortest path from s to t.
- *Negative weights introduce complications.* For the moment, we assume that edge weights are positive (or zero).
- *Shortest paths are normally simple.* Our algorithms ignore zero-weight edges that form cycles, so that the shortest paths they find have no cycles.
- *Shortest paths are not necessarily unique.* There may be multiple paths of the lowest weight from one vertex to another; we are content to find any one of them.
- *Parallel edges and self-loops may be present.* In the text, we assume that parallel edges are not present and use the notation v->w to refer to the edge from v to w, but our code handles them without difficulty.

Relaxation: Our shortest-paths implementations are based on an operation known as *relaxation*. We initialize distTo[s] to 0 and distTo[v] to infinity for all other vertices v.

- *Edge relaxation.* To relax an edge v->w means to test whether the best known way from s to w is to go from s to v, then take the edge from v to w, and, if so, update our data structures.

```
private void relax(DirectedEdge e) {
int v = e.from(), w = e.to();
  if (distTo[w] >distTo[v] + e.weight()) {
distTo[w] = distTo[v] + e.weight();
edgeTo[w] = e;
  }
}
```

Given graph above :

- Using vertex z as the source:
- d values:

s	t	x	y	z
∞	∞	∞	∞	∞
2	∞	7	∞	0
2	5	7	9	0
2	5	6	9	0
2	4	6	9	0

- π values:

s	t	x	y	z
NIL	NIL	NIL	NIL	NIL
z	NIL	z	NIL	NIL
z	x	z	s	NIL
z	x	y	s	NIL
z	x	y	s	NIL

- Changing the weight of edge (z, x) to 4:
- d values:

s	t	x	y	z
0	∞	∞	∞	∞
0	6	∞	7	∞
0	6	4	7	2
0	2	4	7	2
0	2	4	7	–2

- π values:

s	t	x	y	z
NIL	NIL	NIL	NIL	NIL
NIL	s	NIL	s	NIL
NIL	s	y	s	t
NIL	x	y	s	t
NIL	x	y	s	t

Consider edge (z, x), it'll return FALSE since $x.d = 4 > z.d - w(z, x) = -2 + 4$.

FIRST TERM EXAMINATION [SEPT. 2019]
FIFTH SEMESTER [B.TECH]
ALGORITHMS DESIGN & ANALYSIS [ETCS - 301]

Time : 1.5 Hrs. M.M. : 30

Note: 1. Attempt Q.No.1 which is compulsory and any two other questions from remaining questions. Each question carries 10 marks.

2. Necessary Data may be assumed wherever necessary and same may be clearly indicated.

Q.1. (a) Prove Master Theorem? (2)

Ans. The master method provides a "cookbook" method for solving recurrences of the form

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

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

The master method requires memorization of three cases.

The recurrence describes the running time of an algorithm that divides a problem of size n into a subproblems, each of size n/b, where a and b are positive constants. The a subproblems are solved recursively, each in time T(n/b). The cost of dividing the problem and combining the results of the subproblems is described by the functions f(n). For example, the recurrence arising from the MERGE-SORT procedure has $a = 2$, $b = 2$, and $f(n) = \Theta(n)$.

The master method depends on the following theorem.

The master theorem

Let $a \geq 1$ and $b > 1$ be constants, let f(n) be a function, and let T(n) be defined on the nonnegative integers by the recurrence

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

where we interpret n/b to mean either n/b or n/b. Then T(n) can be bounded asymptotically as follows:

There are 3 cases:

1. If $f(n) = 0\ (n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = (n^{\log_b a})$.
2. If $f(n) = \Theta\ (n^{\log_b a})$, then $T(n) = \Theta\ (n^{\log_b a}\ 1g\ n)$
3. If $f(n) = \Omega\ (n^{\log_b a + \epsilon})$ with ϵ, and f(n) satisfies the regularity condition, then $T(n) = Q\ (f(n))$. Regularity condition: $af\ (n/b) < cf\ (n)$ for some constant $c < 1$ and all sufficiently large n.

Q.1. (b) Differentiate between Big-Oh and Small-Oh notations. (2)

Ans. O(big Oh) notation: If f(s) and g(s) are functions of a real or complex variable s and S is an arbitrary set of (real or complex) numbers s (belonging to the domains of f and g), we write $f(s) = O(g(s))$ $(s \in S)$, if there exists a constant c such that $|f(s)| \leq c|g(s)|$ $(s \in S)$. To be consistent with our earlier definition of "big oh" we make the following convention: If a range is not explicitly given, then the estimate is assumed to hold for all sufficiently large values of the variable involved, i.e., in a range of the form $x \geq x_0$, for a suitable constants x_0.

Little-oh notation(o): Asymptotic upper bound provided by O-notation may not be asymptotically tight. O(g(n)) = (f(n): for any +ve c > 0, if a constant $n_0 > 0$ such that $0 <= f(n) < cg(n)$ for all $n >= n_0$}

The function f(n) becomes insignificant relative to g(n) as n approaches infinity

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

Q.1. (c) Describe various representation of disjoint Sets. (2)

Q.1. (d) Compare Dynamic Programming and Memoization. (2)

Ans. Dynamic Programming is also used in optimization problems. Like divide-and-conquer method, Dynamic Programming solves problems by combining the solutions of subproblems. Moreover, Dynamic Programming algorithm solves each sub-problem just once and then saves its answer in a table, thereby avoiding the work of re-computing the answer every time

Dynamic Programming algorithm is designed using the following four steps:

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

Applications of Dynamic Programming Approach

- Matrix Chain Multiplication
- Longest Common Subsequence
- Travelling Salesman Problem

Memoization or Memoisation is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again. Memoization has also been used in other contexts (and for purposes other than speed gains), such as in simple mutually recursive descent parsing in a general top-down parsing algorithm that accommodates ambiguity and left recursion in polynomial time ans space. Although related to caching, memoization refers to a specific case of this optimization, distinguishing it from forms of caching such as buffering or page replacement. In the context of some logic programming languages, memoization is also known as tabling; see also lookup table.

Q.1. (e) Discuss the ingredients of dynamic programming. (2)

Ans. Refer to Q.1. (e) Mid Term Examination 2016 (Pg. No. 2-2016)

Q.2. Solve the following recurrence relations (provide proper explanation)

(a) $T(n) = 2T(n-2) + n^3$

(b) $T(n) = 7T(n/2) + O(n^2)$

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

(d) $T(n) = T(\sqrt{n}) + n$ (10)

Q.3. (a) Define i^{th} order statistic. Describe the algorithm to find the i^{th} order statistic along with its time complexity. (5)

Q.3. (b) Find Optimal solution for following 0/1 knapsack problem using dynamic programming: Weights (w1, w2, w3) = (2, 3, 3), values (v1, v2, v3) = (1,2,4), capacity of knapsack = 6 (5)

Q.4. Demonstrate the optimal substructure of Longest Common Sub-Sequence problem. Find the longest common subsequence of the following two substrings using dynamic programming: A = ABCBDAB, B = BDCABA (10)

END TERM EXAMINATION [DEC. 2019]

Time : 3 Hrs. **M.M. : 75**

Note: Attempt any five questions including Q. No. 1 is compulsory.

Q.1. Answer any five questions in brief:

Q.1. (a) Define algorithm. Explain any three asymptotic notations with examples. (5)

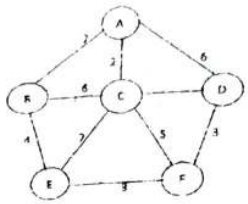
Ans. Algorithm: An algorithm (pronounced AL-go-rith-um) is a procedure or formula for solving a problem, based on conducting a sequence of specified actions. A computer program can be viewed as an elaborate algorithm. In mathematics and computer science, an algorithm usually means a small procedure that solves a recurrent

Q.2. (a) Write a recursive quick short algorithm and derive its time complexity for best, average and worst case. (6.5)

Q.2. (b) Show the result of running quick sort technique on the given sequence: 65,150,180,90,11,25,13,45,99,155 (6)

Q.3. (a) Explain the single source shortest path problem with an example. (6.5)

Q.3. (b) Find the minimum cost spanning tree for the following graph using Prim's algorithm. (6)



Q.4. Solve the following recurrence relation

(i) $T(n) = n + T\left(\frac{n}{5}\right) + T\left(\frac{4n}{5}\right)$ (4.5)

(ii) $T(n) = T(n-1) + n$ if $n > 1$ and $T(n) = 1$ if $n = 1$ (4)

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

Q.5. (a) Explain the characteristics of greedy algorithm. Does greedy technique always provide optimal solution? Justify your answer. (6.5)

Q.5. (b) Construct a Huffman tree and find the resulting code word of each character for the given set of frequencies. (6)

a : 5 b : 30 c : 45 d : 2 e : 4 f : 14

Q.6. (a) Write the Kruskal's algorithm for finding minimum cost spanning tree. (6.5)

Q.6. (b) Sort the following input sequence using insertion sort:

50, 80, 100, 90, 55, 15, 150, 65, 78 (6)

Q.7. (a) Write a recursive algorithm to solve Tower of Hanoi problem with an example. (6.5)

Q.7. (b) Explain the methodology of Dynamic programming. List the applications of Dynamic programming. (6)

Q.8. (a) Find the optimal solution using dynamic programming for a 0/1 knapsack problem where the number of item (n) is 5 and weight carrying capacity of knapsack (m) is 7. The profit and weight of each item is as given below (6.5)

Items	P	Q	R	S	T
Profit	95	75	85	35	60
Weight	4	3	2	1	2

Q.8. (b) Determine the longest common subsequence of the two given sequences ababab and baabaa using dynamic programming (6)

Q.9. (a) Define and explain P, NP, NP hard and NP complete problems. (6.5)

Ans. Refer to Q.1 (i) End Term Examination 2018 (Pg. No. 14-2018)

Q.9. (b) Explain reducibility with an example. (6)

problem.

The commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- O Notation
- Ω Notation
- θ Notation

Big Oh Notation, O : The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity of the longest amount of time an algorithm can possibly take the complete.

For example, for a function **f(n)**

$O(f(n)) = \{g(n)$: there exists $c > 0$ and n_0 such that $f(n) \leq c.g(n)$ for all $n > n_0.\}$

Omega Notation, Ω : The notation $\Omega(n)$ is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

For example, for a function **f(n)**

$\Omega(f(n)) \geq \{g(n)$: there exists $c > 0$ and n_0 such that $g(n) < c.\ f(n)$ for all $n > n_0.\}$

Theta Notation, θ : The notation $\theta(n)$ is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows-

$\theta(f(n)) = \{g(n)$ *if and only if* $g(n) = O(f(n))$ and $g(n) = \Omega(f(n))$ for all$\}$

Q.1. (b) Write a recursive algorithm for binary search and discus its time complexity for every case. (5)

Q.1. (c) Explain Master theorem. Apply master theorem to solve the following recurrence relation: (5)

$$T(n) = 64T\left(\frac{n}{8}\right) + n^5$$

Q.1. (d) Find the optimal solution to a job sequencing with deadline problem which has n = 7 jobs and following profits and deadlines: (5)

Jobs	A	B	C	D	E	F	G
Profit	55	23	15	70	90	45	35
Deadline	5	2	4	1	3	2	1

Q.1. (e) Distinguish between dynamic programming and greedy method. (5)

Ans. Refer to Q.1 (g) End Term Examination 2016 (Pg. No. 14-2016)

Q.1. (f) Explain matrix chain multiplication with an example. (5)

Q.1. (g) Write a short note on string matching algorithm. (5)

Ans. In computer science, string searching algorithms, sometimes called string matching algorithms, are an important class of string algorithms that try to find a place where one or several string (also called patterns) are found within a larger string or text.

Let Σ be an alphabet (finite set). Formally, both the pattern and searched text are vectors of elements of Σ. The Σ may be a usual human a alphabet (for example, the letters A through Z in the Latin alphabet). Other applications may use binary alphabet ($\Sigma = \{0, 1\}$) or DNA alphabet ($\Sigma = \{A, C, G, T\}$) in bioinformatics.

In practice, how the string is encoded can affect the feasible string search algorithms. In particular if a variable width encoding is in use then it is slow (time proportional to N) to find the Nth character. This will significantly slow down many of the more advanced search algorithms. A possible solution is to search for the sequence of code units instead, but doing so may produce false matches unless the encoding is specifically designed avoid it.