SYLLABUS

Academic Session (2015-16)

ALGORITHM DESIGN AND ANALYSIS [ETCS-301]

UNIT I

Asymptotic notations for time and space complexity, Big-Oh notation, Θ notation, Ω notation, the little-oh notation, the little-omega notation, Recurrence relations: iteration method, recursion tree method, substitution method, master method (with proof), subtract and conquer master method(with proof), Data Structures for Disjoint Sets, Medians and Order statistics. Complexity analysis, Insertion sort, Merge Sort, Quick sort. Strassen's algorithm for Matrix Multiplications.

[T1][R1][R2][No. of Hrs. 10]

UNIT II

Dynamic Programming: Ingredients of Dynamic Programming, emphasis on optimal substructure, overlapping substructures, memorization. Matrix Chain Multiplication, Longest common subsequence and optimal binary search trees problems, 0-1 knapsack problem, Binomial coefficient computation through dynamic programming. Floyd Warshall algorithm.

[T1][T2][R1] [R3][No. of Hrs. 10]

UNIT III

Greedy Algorithms: Elements of Greedy strategy, overview of local and global optima, matroid, Activity selection problem, Fractional Knapsack problem, Huffman Codes, A task scheduling problem. Minimum

Spanning Trees: Kruskal's and Prim's Algorithm, Single source shortest path: Dijkstra's and Bellman Ford Algorithm(with proof of correctness of algorithms).

[T1][T2][R4] [No. of Hrs. 10]

UNIT IV

String matching: The naïve String Matching algorithm, The Rabin-Karp Algorithm, String Matching with finite automata, The Knuth-Morris Pratt algorithm.

NP-Complete Problem: Polynomial-time verification, NP-Completeness and Reducibility, NP-Completeness Proof, NP –hard, Case study of NP-Complete problems (vertex cover problem, clique problem).

[T1][R1] [No. of Hrs. 10]

FIFTH SEMESTER (B.TECH)
FIRST TERM EXAMINATION–[2014]
ALGORITHM ANALYSIS & DESIGN–[ETCS-301]

Time : 1.30 hrs. **M.M. : 30**

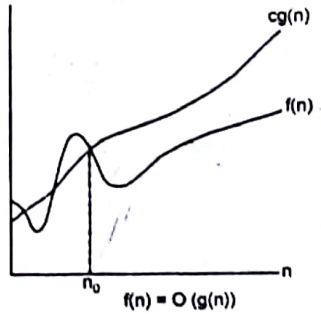
Note: ***Attempt Q.No. 1 and any two more Questions. All questions carry equal marks.***

Q.1. (a) Define Big-Oh nation. **(5×2=10)**

Ans. O notation is used to denote asymptotic upper bound. For a given function $g(n)$, the set of functions.

$O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0\}$

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



Q.1. (b) Define Worst Cause and Best Case complexity of Quick Sort algorithm.

Ans. Quicksort (sometimes called partition-exchange sort) is an efficient sorting algorithm, serving as a systematic method for placing the elements of an array in order. Quicksort is a divide and conquer algorithm. Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays.

Complexity of Quick Sort

Worst Case Analysis:

Now consider the case, when the pivot happened to the least element of the array, so that we had $k = 1$ and $n - k = n - 1$. In such a case, we have:

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

Now let us analyse the time complexity of quick sort in such a case in detail by solving the recurrence as follows:

$$T(n) = T(n-1) + T(1) + \alpha n$$
$$= [T(n-2) + T(1) + \alpha(n-1)] + T(1) + \alpha n$$
$$= T(n-2) + 2T(1) + \alpha(n-1+n) \text{ (by simplifying and grouping terms together)}$$
$$= [T(n-3) + T(1) + \alpha(n-2)] + 2T(1) + \alpha(n-1+n)$$
$$= T(n-3) + 3T(1) + \alpha(n-2+n-1+n)$$

$= [T(n-4) + T(1) + \alpha(n-3)] + 3T(1) + \alpha(n-2+n-1+n)$
$= T(n-4) + 4T(1) + \alpha(n-3+n-2+n-1+n)$
$= T(n-i) + iT(1) + \alpha(n-i+1+\ldots+n-2+n-1+n)$
(Continuiting likewise till the i^{th} step.)

$= T(n-i) + iT(1) + \alpha\left(\sum_{j=0}^{i-1}(n-j)\right)$.

Now clearly such a recurrence can only go on until $i = n-1$. So, substitute $i = n-1$ in the above equation, which gives us:

$$T(n) = T(1) + (n-1)T(1) + \alpha\sum_{j=0}^{n-2}(n-j)$$
$$= nT(1) + \alpha(n(n-2) - (n-2)(n-1)/2)$$

which is $O(n^2)$.

Best Case Analysis: The best case of quicksort occurs when the pivot we pick happens to divide the array into two exactly equal parts, in every step. Thus we have $k = n/2$ and $n - k = n/2$ for the original array of size n.

Consider, therefore, the recurrence:

$$T(n) = 2T(n/2) + \alpha n$$
$$= 2(2T(n/4) + \alpha n/2) + \alpha n$$
$$= 2^2T(n/4) + 2\alpha n$$

(By simplifying and grouping terms together).

$$= 2^2(2T(n/8) + \alpha n/4) + 2\alpha n$$
$$= 2^3T(n/8) + 3\alpha n$$
$$= 2^kT(n/2^k) + k\alpha n)$$

(Continuiting likewise till the k^{th} step)

Notice that this recurrence will continue only until $n = 2^k$ (otherwise we have $n/2^k < 1$), i.e., until $k = \log n$. Thus, by putting $k = \log n$, we have the following equation:

$$T(n) = nT(1) + \alpha n \log n, \text{ which is } O(n \log n).$$

This is the best case for quick sort.

Q.1. (c) What are the differences between Divide & Conquer and Dynamic Programming approach?

Ans.

Divide and Conquer Approach	Dynamic Programming
1. The sub problems in divide and conquer approach are more ore 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. (d) What are the various operations that can be performed on linked list data structure for Disjoint sets.

Ans. Disjoint sets structure is also called "union-find structure". So union, find and Make Set operations should be supported anyway. Other operations are not what this structure is all about, and whether they're supported depends on implementation and the aims we have. Sometimes we need to choose particular implementation specifically to fit our project's needs of additional opertations.

Other than that it would be nice if we supported the other basic set-related operations. Let's enumerate them:

- intersection of two sets. Since sets are disjoint, it's always empty unless these two sets coincide.
- union of two sets -- supported out of the box.
- get an element from the set -- supported, it's most likely the result of find.
- delete an element from the set - depends on implementation. When sets are implemented as forests, it's tricky and requires slower additional operations. When sets are implemented as linked lists, it's simple.
- enumerate the set, i.e. iterate each element in the given set. This one depends on implementation again: for linked lists it's simple, for forest-like implementation it requires additional structures to support.

Q.1. (e) Define Optimal Binary Search Tree problem.

Ans. Refer to End Term Examination May-June 2014 Q. 4(a)

Q.2. (a) Write an algorithm for Matrix Chain Multiplication problem. Also discuss lists complexity. **(5)**

Ans. Refer to End Term Examination May-June 2014 (Q.4b).

Q.2. (b) Determine the LCS of X = <A, B, C, B, D, A, B > and Y = < B, D, C, A, B, A>. **(5)**

Ans. Given $X = (x_1, x_2 \ldots x_m)$ and $Y = (y_1, y_2 \ldots y_n)$.

Let $C[0\ldots m, 0\ldots n]$ be an array of integer and

Let $B[0\ldots m, 0\ldots n]$ be an array of characters

Initialize $C[0, j] = 0$ and $C[i, 0] = 0$ for all i, j such that $0 \le i \le m$ and $0 \le j \le n$.

Initialize $B[i, j] =$ " for $0 \le i \le m$ and $0 \le j \le n$.

```
for i = 1 to m do
    for j = 1 to n do
        if (x_i = y_j)
            C[i, j] = 1 + C[i – 1, j – 1]
            B[i, j] = 'D'
        else if (C[i – 1, j] ≥ C[i, j – 1])
            C[i, j] = C[i – 1, j]
            B[i, j] = 'U'
        else
            C[i, j] = C[i, j – 1]
            B[i, j] = 'L'
return C[m, n] //This is the length of the LCS
```

The purpose of creating the array B is to be able to cacluate the actual longest common subsequence. The following pseudocode will use the array B and the sequence X to print the LCS.

```
Algorithm print LCS
    Let integer i = m //The length of sequence X
    Let integer j = n //The Length of Sequence Y
    while (i > 0 AND j > 0)
        if (B [i, j] = 'D')
            print x_i
            set i = i – 1
            set j = j – 1
        else if (B[i, j] = 'U')
            set i = i – 1
        else
            set j = j – 1
```

Given X = (A, B, C, B, D, A, B) and Y = (B, D, C, A, B, A), the algorithms above

C	0	1	2	3	4	5	6 (j)	B	B	D	C	A	B	A (Y)
0	0	0	0	0	0	0	0	A	U	U	U	D	L	D
1	0	0	0	0	1	1	1	B	D	L	L	U	D	L
2	0	1	1	1	1	2	2	C	U	U	D	L	U	U
3	0	1	1	2	2	2	2	B	D	U	U	U	D	L
4	0	1	1	2	2	3	3	D	U	D	U	U	U	U
5	0	1	2	2	2	3	3	A	U	U	U	D	U	D
6	0	1	2	2	3	3	4	B	D	U	U	U	D	U
7	0	1	2	2	3	4	4	(X)						

(i) The bold, italic values in the array B form the path the print LCS algorithm follows. The values of the sequence X in the rows where a 'D' occurs form the LCS. For this problem, the LCS is: BCBA

Q.3. Write the Quick Sort algorithm, Perform the Quick Sort to sort the following numbers: 4, 5, 1, 7, 8, 9, 2, 8, 8. **(10)**

Ans. End Term Examination May-June 2014 Q.1, (*a*)

Solution for the sequence 4, 5, 1, 7, 8, 9, 2, 8, 8

Procedure:

Pivot element key = 4

Step 1.

(*a*) Set $p = 1$ compare the indexed value to the key Do 1b. until it return the index values greater than key element.

(*b*) Set $r = 8$ compare the indexed value to the key

Do 1 (*b*) until it return the index values lesser than key element.

Step 2. Replace the p indexed value to the r indexed value and follow the step 1 again.

Step 3. If $p >= r$ then replace the r indexed value to the key element.

After Iteration 1 the sequence will be 1, 2, 4, 7, 8, 9, 5, 8, 8

Now we have two sub problems

1. 1, 2
2. 7, 8, 9, 5, 8, 8

Subproblem 1 is already sorted

Hence took subproblem 2

7, 8, 9, 5, 8, 8

After Iteration 2

Key = 7

5, 7, 9, 8, 8, 8

Now we have two sub problems.

1. 5
2. 9, 8, 8, 8

Subproblem 1 is already sorted

Hence took subproblem 2

9, 8, 8, 8

After Iteration 3

Key = 9

8, 8, 8, 9

Hence sorted.

Q.4. (*a*) Solve the recurrence $T_{(n)} = 9T(n/3) + n$ using Master method. **(5)**

Ans. $T(n) = 9T(n/3) + n$

According to Master Method.

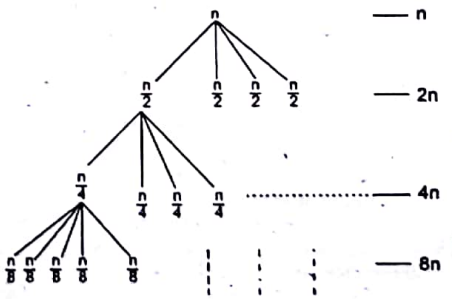
$$n^{\log_b a} = n^{\log_3 9} = n^2$$

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

So the Complexity of $T(n) = 9T(n/3) + n$ is $O(n^2)$.

Q.4. (*b*) Solve the recurrence $T(n) = 4T([n/2]) + n$ using Recurrence tree method. **(5)**

Ans.



$$\frac{n}{2^i} = 1$$

$$n = 2^i$$

$$\log_2 n = i$$

$8n^2$ Hence $T(n) = n^2$

FIFTH SEMESTER (B.TECH)
SECOND TERM EXAMINATION–[2014]
ALGORITHM ANALYSIS & DESIGN–[ETCS-301]

Time : 1.30 hrs. **M.M. : 30**

Note: *Attempt Q.No. 1 and any two more Questions. All questions carry equal marks.*

Q.1. (a) Write down the differences between DFS and BFS?

Ans.

DFS	BFS
1. DFS is an algorithm for traversing or searching a tree, tree structure or graph. One starts at the root (selecting some node as the root is the graph) and explores as for as possible along each branch before backtracking.	BFS is a graph search algorithm that begins at the root and explores all the neighbouring nodes. The for each of those nearest nodes, it explores their unexplored neighbor nodes and so on until it finds the goal.
2. DFS uses stack implementation i.e LIFO.	BFS uses queue implemntation i.e. FIFO.
3. DFS is faster than BFS and requires less memory than BFS.	BFS is slower than DFS and requires more memory than DFS.
4. It is used to find connected and strongly connected components, nopological sorting.	It is used in minimum spanning tree and shortest paths.
5. There is backtracking.	There is no backtracking.

Q.1. (b) Distinguish between Greedy approach and Dynamic Programming approach.

Ans. Dynamic programming is applicable to problems exhibiting the properties of

- overlapping subroplems, and
- optimal substructure.

Optimal substructure means that you can greedily solve subproblems and combine the solutions to solve the larger problem.

The difference between dynamic programming and greedy algorithms is that the subproblems overlap.

Greedy algorithms are indeed a special case of Dynamic Programming.

In dynamic programming we examine a set of solutions to smaller problems and pick the best among them. In a greedy algorithm, we believe one of the solutions to smaller problems to be the optimal one (depending on certain properties of the problem) and don't examine other solutions.

Sometimes it is easy to see a DP solution to a problem, but some observation is needed to find a greedy solution which reduces time and space complexity. Here's an example: Given an array of integers, find the longest zig-zag subarray (which may be non-contagious). A naive DP solution would be to store the longest zig-zag subarray ending at the i'th position in d[i]. Then use the recurrence relation, d[i+1] = 1 + max{d[j]: j <= i and zig-zag property is maintained}. This runs in O(n^2) time. A greedy approach will start from the first index and move forward to include a point where the 'derivative' changes sign. This will run in O(n) time. The choice to pick and believe one sub-solution ('derivative' change point) to be optimal saves a lot of time here, but is essentially just DP with sub-solution sample space pruned.

Q.1. (c) Justify that Prim's algorithm follows Greedy strategy?

Ans. PRIM's algorithm has the property that the edges in the set A (this set A contains the edges of the minimum spanning tree, when algorithm proceed step-by step) always form a single tree, i.e. at each step we have only one connected component.

- We begin with one starting vertex (say v) of a given graph G(V,E).
- Then, in each iteration, we choose a minimum weight edge (u, v) connecting a vertex v in the set A to the vertices in the set. That is, we always find an edge (u, v) of minimum weight such that v∈ A and u∈ V – A. Then we modify the set A by adding u i.e. $A \leftarrow A \cup \{u\}$
- This process is repeated until, i.e. until all the vertices are not in the set A PRIM's algorithm works as follows:

1. Initially the set A of nodes contains a single arbitrary node (i.e. starting vertex) and the set T of edges are empty.

2. At each step PRIM's algorithm looks for the shortest possible edge such that u∈ V – A and v∈ A

3. In this way the edges in T form at any instance a minimal spanning tree for the nodes in A. We repeat this process until A! = V.

Q.1. (d) What is Matroid, Prove that Undirected graph is Matroid.

Ans. A matroid is an ordered pair $M = (S, l)$ satisfying the 3 conditions below:

1. S is a finite non empty set.

2. l is a non empty family of subsets of S, called the independent subsets of S, such that if $B \in l$ and $A \leq B$, then $A \in l$. We say that l is pereditarvey if it satisfies this property. The empty set $\emptyset$ is necessarly a member of l.

3. If $A \in l$, $B \in l$ and $|A| < |B|$, then there is some element $x \in B\text{-}A$ such that $A \cup \{x\} \in l$. We say that M satisfies the exchange property.

The graphical matroid $M_G = (S_G, l_G)$ is defined in terms of a given undirected graph $G = (V, E)$ as

1. The set S_G is defined to be E, the set of edges of G.

2. A is a subset of E, then $A \in l_G$ if and only if A is a cyclic. That is, a set of edges A is independent if and only if the subgraph $G_A = (V, A)$ forms a forest and only if the subgraph $G_A = (V, A)$ forms a forest.

We need to prove

If $G = (V_1E)$ is an indirected graph, the $M_G = (S_G, l_G)$ is a matrioid.

Prove that undirected graph is matroid.

Clearly, $S_G = E$ is a finite set. Furthermore, I_G is hereditary, since a subset of a forest is a forest. Putting it another way, removing edges from an acyclic set of edges cannot create cycles. Thus, it remains to show that M_G satisfies the exchange property. Suppose that $G_A = (V, A)$ and $G_B = (V, B)$ are forests of G and that $|B| > |A|$. That is, A and B are acyclic sets of edges, and B contains more edges than A does. We know that the forest having k edges contains exactly $|V| - k$ trees. (To prove this another way, begin with $|V|$ trees, each consisting of a single vertex, and no edges. Then, each edge that is added to the forest reduces the number of trees by one.) Thus, forest G_A contains $|V| - |A|$ trees, and forest G_B contains $|V| - |B|$ trees. Since forest G_B has fewer trees than forest G_A does, forest G_B must contain some tree T whose vertices are in two different trees in forest G_A. Moreover, since T is connected, it must contain an edge (u, v) such that vertices u and v are in different trees in forest G_A. Since the edge $(u,$

v) connects vertices in two different trees in forest G_A, the edge (u, v) can be added to forest G_A without creating a cycle. Therefore, M_G satisfies the exchange property, completing the proof that M_G is a matroid.

Q.1. (e) What is the difference between Suffix function and Prefix function. (5 × 2)

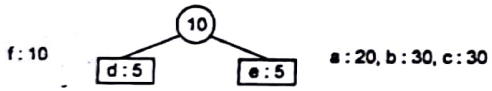
Ans. A prefix of a string $T = t_1 \ldots t_n$ is a string $\tilde{T} = t_1 \ldots t_m$, where $m \leq n$. A proper prefix of a string is not equal to the string itself ($0 \leq m < n$)[1] some sources[2] in addition restrict a proper prefix to be non-empty ($0 < m < n$). A prefix can be seen as a special case of a substring.

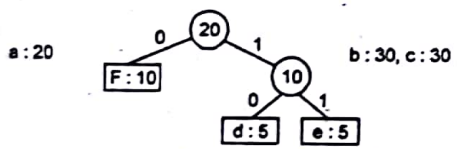
A suffix of a string is any substring of the string which includes its last letter, including itself. A proper suffix of a string is not equal to the string itself. A more restricted interpretation is that it is also not empty. A suffix can be seen as a special case of a substring.

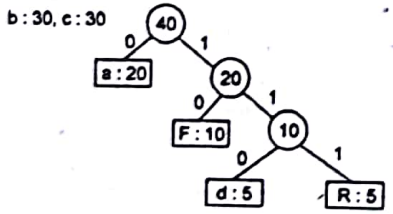
Q.2. (a) Find the Huffman code for the following set of frequencies. a:20, b: 30, c:30, d:05, e:05, f:10. (5)

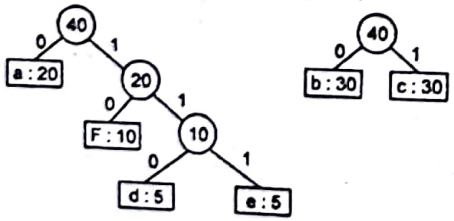
Ans.

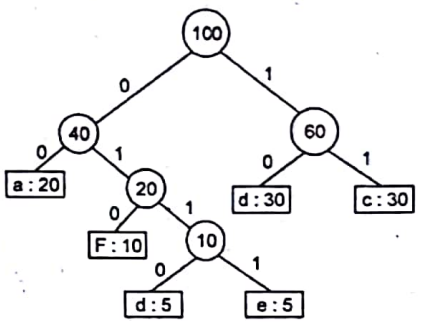
a : 20	b : 30	c : 30	d : 05	e : 05	f : 10
d : 5	e : 5	f : 10	a : 20	b : 30	c : 30











$$B(T) = f(e).d_T(e) + f(d)\cdot d_T(d) + f(F) + d_T(f) + f(a)\cdot d_T(a) + f(b)\cdot d_T(b) + f(c)\cdot d_T(c)$$
$$= 5 \times 4 + 5 \times 4 + 10 \times 3 + 20 \times 2 + 30 \times 2 + 30 \times 2$$
$$= 20 + 20 + 30 + 40 + 60 + 60 = 230$$

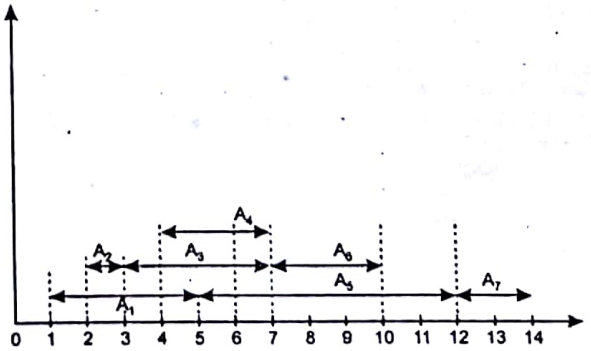
Q.2. (b) Write the Greedy algorithm for Optimal Activity Selection problem. Find out the Set which contain in maximum activity that is compatable to each other Given A = <A 1, A2, A3, A4 , A5, A 6, A7, Si = <1, 2, 3, 4, 5, 6, 12, Fi = < 5, 3, 6, 7, 12, 10 14. (50)

Ans.

A	A_1	A_2	A_3	A_4	A_5	A_6	A_1
Si	1	2	3	4	5	6	12
fi	5	3	6	7	12	10	14

Greedy-Activity-selection ($A_1 S_1 f$)

1. $n \leftarrow$ length [S]
2. $A \leftarrow [i]$
3. $j \leftarrow 1$
4. for $i \leftarrow 2$ to n
5. do if $Si \geq fi$
6. then $A \leftarrow AU[i]$
7. $j \leftarrow i$
8. Return A.



So, $\langle A_1 A_6 A_7 \rangle$ is the optimal activity schedule for the given set.

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

Compute-prefix-function (P)

1. $m \leftarrow$ length [P]
2. $\pi[i] \leftarrow 0$
3. $K \leftarrow 0$
4. For $q \leftarrow 2$ to m
5. do while $K > 0$ and $P[K+1] \neq P[q]$
6. do $k \leftarrow \pi[K]$
7. if $P[K+1] = P[q]$
8. then $K \leftarrow K + 1$
9. $\pi[q] \leftarrow K$
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 $f(O) = 0$.

Q.3. (b) Construct the Finite automata for the following pattern:
P = <abbacabb> & find its occurrences in the text T = abbabbcabcabbcbabb. **(5)**

Ans. (*i*)

A	B	C	B	D	B	A	E	A
1	2	5	6	8	11	12	15	20

(*ii*)

A	B	C	B	D	B	A	E	A
1	2	5	6	8	11	12	15	20

$A = 0 + 10 + 3$ $\qquad$ $13 + 5$

$B = 3 + 2$ $\qquad$ $\frac{18}{5}$

$C = 0$

$D = 0$

$E = 0$

Average waiting time = 3.6

$$\text{Turn around time} = \frac{20+10+3+2+3}{5} = \frac{38}{5}$$

$$= 7.6$$

Q.4. (a) What is Minimum Spanning tree? Explain the differences between Prim's and algorithm with the help of suitable example. **(5)**

Ans. Minimum spanning tree: An edge-weighted graph is a graph where we associate weights or costs with each edge. A minimum spanning tree (MST) of an edge-weighted graph is a spanning tree whose weight (the sum of the weights of its edges) is no larger than the weight of any other 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.

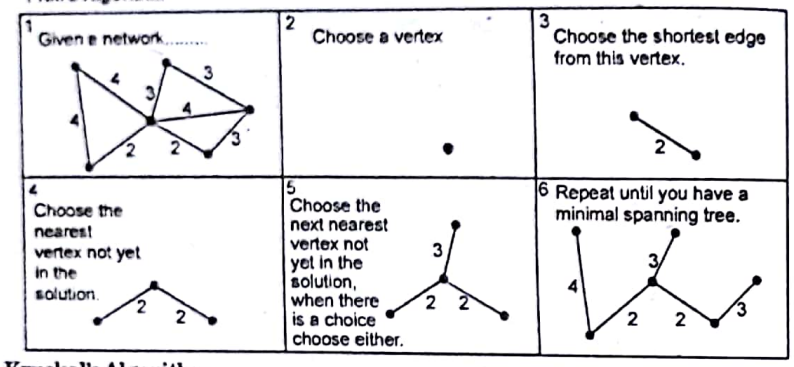
Differences between Prims and Kruskal Algorithm

Prim's algorithm intializes with a node, whereas Kruskal's algorithm initiates with an edge.

- Prim's algorithms span from one node to another while Kruskal's algorithm select the edges in a way that the position of the edge is not based on the last step.
- In prim's algorithm, graph must be a connected graph while the Kruskal's can function on disconnected graphs too.
- Prim's algorithm has a time complexity of $O(V^2)$ and Kruskal's time complexity is O(log V).

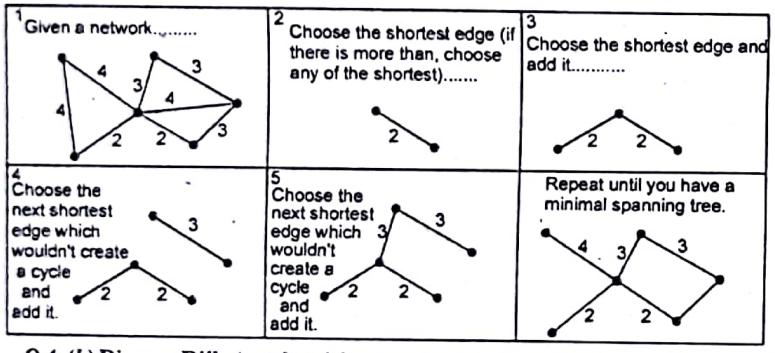
Prim's Algorithm

Prim's Algorithm



Kruskal's Algorithm

Kruskal's algorithm



Q.4. (*b*) Discuss Dijkstra algorithm for single source shortest path problem through an example: Discuss its complexity also. **(5)**

Ans. Dijkstra's algorithm solves the single-source shortest-path problem when all edges have non-negative weights. It is a greedy algorithm and similar to Prim's algorithm. Algorithm starts at the source vertex, s, it grows a tree, T, that ultimately spans all vertices reachable from S. Vertices are added to T in order of distance i.e., first S, then the vertex closest to S, then the next closest, and so on. Following implementation assumes that graph G is represented by adjacency lists.

Dijkstras (G, w, s)

1. Initialize Single-Source (G, s)
2. S ← { } //S will ultimately contains vertices of final shortest-path weights from s
3. Initialize priority queue Q i.e., Q ← V[G]
4. While priority queue Q is not empty do
5. u ← EXTRACT_MIN(Q) //Pull out new vertex
6. S ← S ∪ {u} //Perform relaxation for each vertex v adjacent to u.
7. for each vertex v in Adj[u] do
8. Relax (u, v, w)

Analysis

Like Prim's algorithm, Dijkstra's algorithm runs in O(|E|lg|V|) time.

FIFTH SEMESTER (B.TECH)
END TERM EXAMINATION–[2014]
ALGORITHM ANALYSIS & DESIGN–[ETCS-301]

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

Q.1. (*a*) What is Quick sort? Show its functioning on data. **(5 × 5 = 25)**

Ans. Quicksort (sometimes called partition-exchange sort) is an efficient sorting algorithm, serving as a systematic method for placing the elements of an array in order. Quicksort is a divide and conquer algorithm. Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays.

The steps are:

1. Pick an element, called a pivot, from the array.

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

3. Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

Algorithm

STEP 1. Choosing the pivot

Choosing the pivot is an essential step.

Depending on the pivot the algorithm may run very fast, or in quadric time.:

Some fixed element: e.g. the first, the last, the one in the middle

This is a bad choice - the pivot may turn to be the smallest or the largest element, then one of the partitions will be empty. Randomly chosen (by random generator) - still a bad choice.

The median of the array (if the array has N numbers, the median is the [N/2] largest number. This is difficult to compute - increases the complexity.

The median-of-three choice: take the first, the last and the middle element. Choose the median of these three elements.

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

The first element is 8, the middle - 10, the last - 5.

The median of [8, 10, 5] is 8

STEP 2. Partitioning

Partitioning is illustrated on the above example.

1. The first action is to get the pivot out of the way - swap it with the last element 5,3,25,6,10,17,1,2,18,8.

2. We want larger elements to go to the right and smaller elements to go to the left. Two "fingers" are used to scan the elements from left to right and from right to left:

$$\left[\underset{i}{5}, 3, 25, 6, 10, 17, \underset{j}{1}, 2, 18, 8\right]$$

While i is to the left of j, we move i right, skipping all the elements less than the pivot. If an element is found greater then the pivot, i stops

While j is to the right of i, we move j left, skipping all the elements greater than the pivot. If an element is found less then the pivot, j stops

When both i and j have stopped, the elements are swapped.

When i and j have crossed, no swap is performed, scanning stops, and the element pointed to by i is swapped with the pivot.

In the example the first swapping will be between 25 and 2, the second between 10 and 1.

3. Restore the pivot.

After restoring the pivot we obtain the following partitioning into three groups:

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

STEP 3. Recursively quicksort the left and the right parts

Q.1. (b) Explain the elements of Greedy strategy.

Ans. Elements of Greedy Algorithms

1. Greedy choice property
2. Optimal substructure (ideally)

Greedy choice property: Globally optimal solution can be arrived by making a locally optimal solution (greedy). The greedy choice property is preferred since then the greedy algorithm will lead to the optimal, but this is not always the case-the greedy algorithm may lead to a suboptimal solution. Similar to dynamic programming, but does not solve subproblems. Greedy strategy more top-down, making one greedy choice after another without regard to sub-solutions.

Optimal substructure: Optimal solution to the problem contains within it optimal solutions to subproblems. This implies we can solve subproblems and build up the solutions to solve larger problems.

Q.1. (c) Define Depth first search.

Ans. Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking.

Input: A graph G and a vertex v of G

Output: All vertices reachable from v labeled as discovered

A recursive implementation of DFS: [5]

```
1 procedure DFS(G,v):
2    label v as discovered
3    for all edges from v to w in G.adjacentEdges(v) do
4       if vertex w is not labeled as discovered then
5          recursively call DFS(G,w)
```

A non-recursive implementation of DFS:[6]

```
1 procedure DFS-iterative(G,v):
2  let S be a stack
3  S.push(v)
4  while S is not empty
5        v = S.pop()
6        if v is not labeled as discovered:
7          label v as discovered
8          for all edges from v to w in G.adjacentEdges(v) do
9             S.push(w)
```

Applications

Algorithms that use depth-first search as a building block include:

- Finding connected components.
- Topological sorting.
- Finding 2-(edge or vertex)-connected components.
- Finding 3-(edge or vertex)-connected components.
- Finding the bridges of a graph.
- Generating words in order to plot the Limit Set of a Group.
- Finding strongly connected components.
- Planarity testing
- Solving puzzles with only one solution, such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)
- Maze generation may use a randomized depth-first search.
- Finding biconnectivity in graphs.

Q.1. (d) Write Floyd-Warshall algorithm.

Ans. In computer science, the Floyd-Warshall algorithm is an algorithm for finding shortest paths in a weighted graph with positive or negative edge weights (but with no negative cycles). The Floyd-Warshall algorithm compares all possible paths through the graph between each pair of vertices. It is able to do this with $\Theta(|V|3)$ comparisons in a graph. This is remarkable considering that there may be up to $\Omega(|V|2)$ edges in the graph, and every combination of edges is tested. It does so by incrementally improving an estimate on the shortest path between two vertices, until the estimate is optimal.

Consider a graph G with vertices V numbered 1 through N. Further consider a function shortest Path (i, j, k) that returns the shortest possible path from i to j using vertices only from the set {l, 2, ..., k} as intermediate points along the way. Now, given this function, our goal is to find the shortest path from each i to each j using only vertices 1 to $k + 1$.

For each of these pairs of vertices, the true shortest path could be either (1) a path that only uses vertices in the set {1,..., k} or (2) a path that goes from i to $k + 1$ and then from $k + 1$ to j. We know that the best path from i to j that only uses vertices 1 through k is defined by shortest Path(i, j, k), and it is clear that if there were a better path from i to $k + 1$ to j, then the length of this path would be the concatenation of the shortest path from i to $k + 1$ (using vertices in {1,..., k}) and the shortest path from $k + 1$ to j (also using vertices in {1,..., k}).

If $w(i, j)$ is the weight of the edge between vertices i and j, we can define shortest Path $(i, j, k + 1)$ in terms of the following recursive formula: the base case is

$$\text{shortest Path}\,(i, j, 0) = w(i, j)$$

and the recursive case is

shortest Path$(i, j, k + 1)$ = min(shortest Path(i, j, k), shortest Path$(i, k + 1, k)$ + shortest Path $(k + 1, j, k)$)

This formula is the heart of the Floyd-Warshall algorithm. The algorithm works by first computing shortest Path(i, j, k) for all (i, j) pairs for $k = 1$, then $k = 2$, etc. This process continues until $k = N$, and we have found the shortest path for all (i, j) pairs using any intermediate vertices.

Q.1. (e) What is string matching? Name some algorithms for string matching.

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 strings (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 ($\Sigma = \{0, 1\}$, or DNA alphabet $\Sigma = \{A, C, G, T\}$ in bioinformatics.

Algorithm	Preprocessing time	Matching time[3]
Naive string search algorithm	0 (no preprocessing)	$\Theta((n-m)m)$
Rabin-Karp string search algorithm	$\Theta(m)$	average $\Theta(n+m)$, worst $\Theta((n-m)m)$
Finite-state automaton based search	$\Theta(m\lvert\Sigma\rvert)$	$\Theta(n)$
Knuth-Morris-Pratt algorithm	$\Theta(m)$	$\Theta(n)$
Boyer-Moore string search algorithm	$\Theta(m + \lvert\Sigma\rvert)$	$\Omega(n/m)$, $O(nm)$
Bitap algorithm (shift-or, shift and, Baeza-Yates-Gonnet	$\Theta(m + \lvert\Sigma\rvert)$	$O(mn)$

UNIT-I

Q.2 Write short note on the following: **(12.5)**

(a) Substitution method.

Ans. While solving some recurrences it is good to recognize some nice things about the recurrence you are actually solving. For instance in this recurrence, notice that at each step you are dividing n by 2. So strictly speaking through the recurrence you can only obtain $T(2^m)$, where $m \in N$. For instance, $T(3)$ cannot be obtained using this recurrence. However, such recurrences come up when you do certain things, when n is really large, recursively using say a binary tree. When n is really large you are interested mainly in how the solution behaves in an asymptotic sense and not the precise expression for the solution. In such cases, you assume that $n = 2^m$ for the simplicity of calculations and the cost for all n (even when n is not a power of 2) can be shown to obey the solution, obtained by the assuming $n = 2^m$, in an asymptotic sense.

Example:

$T(n) = n \log_2(n) + n$ is the guess you made in step (a). Using that replace $n = \frac{n}{2}$,

$$T\left(\frac{n}{2}\right) = \frac{n}{2}\log_2\left(\frac{n}{2}\right) + \frac{n}{2}.$$

$$T\left(\frac{n}{2}\right) = \frac{n}{2}(\log_2(n) - \log_2(2)) + \frac{n}{2} = \frac{n}{2}\log_2(n) - \frac{n}{2} + \frac{n}{2} = \frac{n}{2}\log_2(n)$$

$$\text{Hence, } T(n) = 2T\left(\frac{n}{2}\right) + n = 2\frac{n}{2}\log_2(n) + n = n\log_2(n) + n$$

If you were to rewrite the recurrence in terms of m, and call $T(2^m) = S(m)$, then we would get

$$S(m) = 2S(m-1) + 2^m \text{ since } \frac{n}{2} = 2^{m-1}$$

$$S(m-1) = 2S(m-2) + 2^{m-1}.$$

$$S(m) = 4S(m-2) + 2^m + 2^m$$

and so on (hidden induction here) to get

$$S(m) = 2^m S(0) + 2^m + 2^m + \ldots + 2^m + 2^m = m2^m + 2^m$$

Hence $T(n) = n\log_2(n) + n$ since $n = 2^m$ i.e., $m = \log_2(n)$

(b) Iteration method.

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 $T(n) = 8T(n/2) + n^2 (T(1) = 1)$

$$T(n) = n^2 + 8T(n/2)$$

$$= n^2 + 8\left(8T\left(\frac{n}{2^2}\right) + \left(\frac{n}{2}\right)^2\right)$$

$$= n^2 + 8^2 T\left(\frac{n}{2^2}\right) + 8\left(\frac{n^2}{4}\right)$$

$$= n^2 + 2n^2 + 8^2 T\left(\frac{n}{2^2}\right)$$

$$= n^2 + 2n^2 + 8^2\left(8T\left(\frac{n}{2^3}\right) + \left(\frac{n}{2^2}\right)^2\right)$$

$$= n^2 + 2n^2 + 8^3 T\left(\frac{n}{2^3}\right) + 8^2\left(\frac{n^2}{4^2}\right)$$

$$= n^2 + 2n^2 + 2^2 n^2 + 8^3 T\left(\frac{n}{2^3}\right)$$

$$= \ldots.$$

$$= n^2 + 2n^2 + 2^2 n^2 + 2^3 n^2 + 2^4 n^2 + \ldots$$

Recursion depth: How long (how many iterations) it takes until the subproblem has constant size? i times where $\frac{n}{2^i} = 1 \Rightarrow i = \log n$

What is the last term? $8^i T(1) = 8^{\log n}$

$$T(n) = n^2 + 2n^2 + 2^2 n^2 + 2^3 n^2 + 2^4 n^2 + \ldots + 2^{\log n - 1} n^2 + 8^{\log n}$$

$$= \sum_{k=0}^{\log n - 1} 2^k n^2 + 8^{\log n}$$

$$= n^2 \sum_{k=0}^{\log n - 1} 2^k + (2^3)^{\log n}$$

Now $\sum_{k=0}^{\log n - 1} 2^k$ is a geometric sum so we have $\sum_{k=0}^{\log n - 1} 2^k = \Theta(2^{\log n - 1}) = \Theta(n)$

$(2^3)^{\log n} = (2^{\log n})^3 = n^3$

$$T(n) = n^2 \cdot \Theta(n) + n^3$$

$$= \Theta(n^3)$$

Q.3. (a) What will be the complexity of insertion sort, quick sort and merge sort on following sequence.

Sequence 1–1,2,3,4,5 Sequence–5,4,3,2,1

Ans. For the sequence of 1, 2, 3, 4, 5 the complexity of

Insertion Sort is $O(n)$: It is the best case for the Insertion sort because all the elements in this sequence are sorted.

Quick Sort is $O(n^2)$: It would be worst case for the Quick sort because it will select the pivot element always from the beginning and generate a Left derivation Tree.

Merge Sort is $O(n \log n)$: Complexity of Merge sort in all cases remains same i.e., $O(n \log n)$.

For the Sequence of 5, 4, 3, 2, 1 the Complexity of

Insertion Sort is $O(n^2)$: It is the worst case for the Insertion sort because all the elements in this sequence are inversely sorted.

Quick Sort is $O(n^2)$: It would be worst case for the Quick sort because it will select the pivot element always from the beginning and generate a Right derivation Tree

Merge Sort is $O(n \log n)$: Complexity of Merge sort in all case remains same i.e., $O(n \log n)$.

Q.3. (b) Can we improve the worst case complexity of quick sort O (n^2) further? If yes then give its algorithm. (12.5)

Ans. In quick sort, we pick an element called the pivot in each step and re-arrange the array in such a way that all elements less than the pivot now appear to the left of the pivot, and all elements larger than the pivot appear on the right side of the pivot. In all subsequent iterations of the sorting algorithm, the position of this pivot will remain unchanged, because it has been put in its correct place. The total time taken to re-arrange the array as just described, is always $O(n)$, or an where is some constant. Let us suppose that the pivot we just chose has divided the array into two parts-one of size k and the other of size $n-k$. Notice that both these parts still need to be sorted. This gives us the following relation:

$$T(n) = T(k) + T(n-k) + \alpha n$$

where $T(n)$ refers to the time taken by the algorithm to sort n elements.

Worst Case Analysis:

Now consider the case, when the pivot happened to be the least element of the array, so that we had $k = 1$ and $n - k = n - 1$. In such a case, we have:

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

Now let us analyse the time complexity of quick sort in such a case in detail by solving the recurrence as follows:

$$T(n) = T(n-1) + T(1) + \alpha n$$
$$= [T(n-2) + T(1) + \alpha(n-1)] + T(1) + \alpha n.$$

(Note: I have written $T(n-1) = T(1) + T(n-2) + \alpha(n-1)$ by just substituting $n-1$ instead of n. Note the implicit assumption that the pivot that was chosen divided the original subarray of size $n-1$ into two parts: one of size $n-2$ and the other of size 1.)

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

(by simplifying and grouping terms together)

$$= [T(n-3) + T(1) + \alpha(n-2)] + 2T(1) + \alpha(n-1+n)$$
$$= T(n-3) + 3T(1) + \alpha(n-2+n-1+n)$$
$$= [T(n-4) + T(1) + \alpha(n-3)] + 3T(1) + \alpha(n-2+n-1+n)$$

$$= T(n-4) + 4T(1) + \alpha(n-3+n-2+n-1+n)$$
$$= T(n-i) + iT(1) + \alpha(n-i+1+\ldots$$
$$+ n-2+n-1+n) \text{ (Continuing likewise till the } i^{th} \text{ step).}$$
$$= T(n-i) + iT(1) + \alpha\left(\sum_{j=0}^{i-1}(n-j)\right)$$

(Look carefully at how the summation is being written.)

Now clearly such a recurrence can only go on until $i = n-1$ (Why? because otherwise $n-i$ would be less than 1). So, substitute $i = n-1$ in the above equation, which gives us:

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

$= nT(1) + \alpha(n(n-2) - (n-2)(n-1/2)$ (Notice that $\sum_{j=0}^{n-2} j = \sum_{j=1}^{n-2} j = (n-2)(n-1)/2$ by a formula we earlier derived in class)

which is $O(n^2)$.

This is the worst case of quick-sort, which happens when the pivot we picked turns out to be the least element of the array to be sorted, in every step (*i.e.* in every recursive call). A similar situation will also occur if the pivot happens to be the largest element of the array to be sorted.

Algorithm

The steps are:

1. Pick an element, called a pivot, from the array.

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

3. Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

```
quicksort (A, l0, hi):
  if l0 < hi :
    p := partition (A, l0, hi)
    qicksort (A, l0, p – 1)
    quicksort (A, p + 1, hi)
//l0 is the index of the leftmost element of the subarray
// hi is the index of the rightmost element of the subarray (inclusive) partition (A, l0, hi)
    pivotIndex := choosePivot (A, l0, hi)
    pivotValue := A[pivotIndex]
    //put the chosen pivot at A[hi]
    swap A[pivotIndex] and A[hi]
    storeIndex := l0
    // Compare remaining array elements against pivotValue = A[hi] for i from l0 to hi-1, inclusive
    if A[i] <=pivotvalue
        swap A[i] and A[storeIndex]
        storeIndex := storeIndex + 1
    swap A[storeIndex] and A[hi] //Move pivot to its final place return storeIndex.
```

UNIT–II

Q.4. (a) Write note on the optimal binary search tree problems. (12.5)

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.4. (b) What is matrix chain multiplication? Explain.

Ans. Dynamic programming is an algorithm that solves the problem of matrixchain multiplication. We are given a sequence (chain) $\_A_1, A_2, ..., A_n\_$ of n matrices to be multiplied, and we wish to compute the product $A_1 A_2 .... A_n$.

We can evaluate the expression above, using the standard algorithm for multiplying pairs of matrices as a subroutine once we have parenthesized it to resolve all ambiguities in how the matrices are multiplied together. A product of matrices is fully parenthesized if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses. Matrix multiplication is associative, and so all parenthesizations yield the same product. For example, if the chain of matrices is $\_A_1, A_2, A_3, A_{4-1}$ can the product $A_1 A_2 A_3 A_4$ can be fully parenthesized in five distinct ways:

$(A_1 (A_2 (A_3 A_4)))$,
$(A_1 ((A_2 A_3) A_4))$,
$((A_1 A_2) (A_3 A_4))$,
$((A_1 (A_2 A_3)) A_4)$,
$((A_1 A_2) A_3) A_4)$,

The way we parenthesize a chain of matrices can have a dramatic impact on the cost of evaluating the product. Consider first the cost of multiplying two matrices. The standard algorithm is given by the following pseudocode. The attributes rows and columns are the numbers of rows and columns in a matrix.

Step 1: The structure of an optimal parenthesization

Our first step in the dynamic-programming paradigm is to find the optimal substructure and then use it to construct an optimal solution to the problem from optimal solutions to subproblems. let us adopt the notation A_{i_j}, where $i \le j$, for the matrix that results from evaluating the product $A_i A_{i+1} + A_j$. Observe that if the problem is nontrivial, *i.e.*, $i < j$. then any parenthesization of the product $A_i A_{i+1} A_j$ must split the product between A_k and A_{k+1} for some integer k in the range $i \le k < j$. Suppose that an optimal parehensization of $A_i A_{j+1} A_j$ splits the product between A_k and A_{k+1}. Then the parenthesization of the "prefix" subchain $A_i A_{i+1} A_k$ within this optimal parenthesization of $A_i A_{i+1} A_j$ must be an optimal parenthesization of $A_i A_{i+1} A_k$.

Step 2: A recursive solution

Next, we define the cost of an optimal solution recursively in terms of the optimal solutions to subproblems. For the matrix-chain multiplication problem, we pick as our subproblems the problems of determining the minimum cost of a parenthesization of $A_i A_{i+1} A_j$ for $1 \le i \le j \le n$.

Let $m[i, j]$ be the minimum number of scalar multiplications needed to compute the matrix A_{i_j}; for the full problem, the cost of a cheapest way to compute A_{1_n} would thus be $m[1, n]$.

To compute $m[i, j]$ when $i < j$, we take advantage of the structure of an optimal solution from step 1. Let us assume that the optimal parenthesization splits the product $A_i A_{i+1} A_j$ between A_k and A_{k+1}, where $i \le k < j$. Then, $m[i, j]$ is equal to the minimum cost for computing the subproducts $A_{i\ldots k}$ and $A_{k+1\ldots j}$, plus the cost of multiplying these two matrices together. Recalling that each matrix A_i is $p_{i-1} \times p_i$, wee see that computing the matrix product $A_{i\ldots k} A_{k+1\ldots j}$ takes $p_{i-1} pk\, pj$ scalar multiplications. Thus, we obtain

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

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

Step 3: Computing the optimal costs

It is a simple matter to write a recursive algorithm based on recurrence to computer the minimum cost $m[1, n]$ for multiplying $A_1, A_2 A_n$. This algorithm takes exponential time, which is no better than the brute-force method of checking each way of parenthesizing the product. The important observations that we can make at this point is that we have relatively few subproblems: one problem for each choice of i and j satisfying $1 \le i \le j \le n$, or ${}^nC_2 + n = \Theta(n^2)$ in all. A recursive algorithm may encounter each subproblem many time in different branches of its recursion tree. This property of overlapping subproblems is the second hallmark of the applicability of dynamic programming.

Matrix-Chain-Order (p)

1. $n \leftarrow$ length $[p] - 1$
2. for $i \leftarrow 1$ to n
3. do $m[i, i] \leftarrow 0$
4. for $1 \leftarrow 2$ to $n \rightarrow 1$ is the chain length.
5. do for $i \leftarrow 1$ to $n - l + 1$
6. do $j \leftarrow i + l - 1$
7. $m[i, j] \leftarrow \infty$
8. for $k \leftarrow i$ to $j - 1$
9. do $q \leftarrow m[i, k] + m[k + 1, j] + k$
10. if $q < m[i, j]$
11. then $m[i, j] \leftarrow q$
12. $s[i, j] \leftarrow k$
13. **return** m and s

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 ")"

Q.5. What are Huffman codes? Why we use them? Discuss in detail. (12.5)

Ans. A Huffman code is a particular type of optimal prefix code that is commonly used for lossless data compression. The process of finding and/or using such a code proceeds by means of Huffman coding. The output from Huffman's algorithm can be viewed as a variable-length code table for encoding a source symbol (such as a character in a file). The algorithm derives this table from the estimated probability or frequency of occurrence (weight) for each possible value of the source symbol. As in other entropy

encoding methods, more common symbols are generally represented using fewer bits than less common symbols. Huffman's method can be efficiently implemented, finding a code in linear time to the number of input weights if these weights are sorted.

Given: A set of symbols and their weights (usually **proportional** to probabilities).

Find

A prefix-free binary code (a set of codewords) with minimum expected codewords length (equivalently, a tree with minimum **weighted path length from the root).**

Input:

Alphabet $A = \{a_1, a_2, \ldots, a_n\}$, which is the symbol alphabet of size n.

Set $W = \{w_1, w_2, \ldots w_n\}$, which is the set of the (positive) symbol weights (usually proportional to probabilities), i.e., w_i = weight (a_i), $1 \le i \le n$.

Output:

Code $C(A, W) = \{c_1, c_2, \ldots, c_n\}$, which is the set of (binary) codewords, where C_i is the codeword for a_i, $1 \le i \le n$.

Goal:

$$L(C) = \sum_{i=1}^{n} w_i \times \text{length}\,(c_i)$$

Let be the weighted path length of code C.

Condition: $L(C) \le L(T)$ for any code $T(A, W)$.

We give an example of the result of Huffman coding for a code with five words and given weights. We will not verify that it minimizes L over all codes (it does of course), but we will compute L and compare it to the Shannon entropy H of the given set of weights; the result is nearly optimal.

Input (A, W)	Symbol (a_i)	a	b	c	d	e	Sum
	Weights (W_i)	0.10	0.15	0.30	0.16	0.29	= 1
	Codewords (c_i)	010	011	11	00	10	
Output C	Codeword length (in bits) (I_i)	3	3	2	2	2	
	Contribution to weight Path length ($I_i W_i$)	0.30	0.45	0.60	0.32	0.58	$L(C) = 2.25$
Optimality	Probability budget (2^{-I})	1/8	1/8	1/4	1/4	1/4	= 1.00
	Information contents in bits ($-\log_2 w_i$)	3.32	2.74	1.74	2.64	1.79	
	Contribution to entropy ($-w_i \log_2 w_i$)	0.332	0.411	0.521	0.423	0.518	$H(A)$ = 2.205

For any code that is biunique, meaning that the code is uniquely decodeable, the sum of the probability budgets across all symbols is always less than or equal to one. In this example, the sum is strictly equal to one; as a result, the code is termed a complete code. If this is not the case, you can always derive an equivalent code by adding extra symbols (with associated null probabilities), to make the code complete while keeping it biunique.

Basic technique

Compression

The technique works by creating a binary tree of nodes. These can be stored in a regular array, the size of which depends on the number of symbols, n. A node can be

either a leaf node or an internal node. Initially, all nodes are leaf nodes, which contain the symbol itself, the weight (frequency of appearance) of the symbol and optionally, a link to a parent node which makes it easy to read the code (in reverse) starting from a leaf node. Internal nodes contain symbol weight, links to two child nodes and the optional link to a parent node. As a common convention, bit '0' represents following the left child and bit '1' represents following the right child. A finished tree has up to n leaf nodes and $n - 1$ internal nodes. A Huffman tree that omits unused symbols produces the most optimal code lengths.

The simplest construction algorithm uses a priority queue where the node with lowest probability is given highest priority:

1. Create a leaf node for each symbol and add it to the priority queue.
2. While there is more than one node in the queue:
   (*i*) Remove the two nodes of highest priority (lowest probability) from the queue
   (*ii*) Create a new internal node with these two nodes as children and with probability equal to the sum of the two nodes' probabilities.
   (*iii*) Add the new node to the queue.
3. The remaining node is the root node and the tree is complete.

Decompression

The process of decompression is simply a matter of translating the stream of prefix codes to individual byte values, usually by traversing the Huffman tree node by node as each bit is read from the input stream (reaching a leaf node necessarily terminates the search for that particular byte value). Before this can take place, however, the Huffman tree must be somehow reconstructed. In the simplest case, where character frequencies are fairly predictable, the tree can be preconstructed (and even statistically adjusted on each V compression cycle) and thus reused every time, at the expense of at least some measure of compression efficiency. Otherwise, the information to reconstruct the tree must be sent a priori. A naive approach might be to prepend the frequency count of each character to the compression stream. Unfortunately, the overhead in such a case could amount to several kilobytes, so this method has little practical use. If the data is compressed using canonical encoding, the compression model can be precisely reconstructed with just $B2^B$ bits of information (where B is the number of bits per symbol).

Optimality

Although Huffman's original algorithm is optimal for a symbol-by-symbol coding (i.e., a stream of unrelated symbols) with a known input probability distribution, it is not optimal when the symbol-by-symbol restriction is dropped, or when the probability mass functions are unknown. Also, if symbols are not independent and identically distributed, a single code may be insufficient for optimality.

UNIT-III

Q.6. Discuss in detail any one algorithm for finding the cost spanning tree. (12.5)

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 C[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.7. Explain Dijkstra's and Bellman Ford algorithm for finding single source shortest path in detail. **(12.5)**

Ans. Dijkstra's algorithm

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. The algorithm exists in many variants; Dijkstra's original variant found the shortest path between two nodes, but a more common variant fixes a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a shortest path tree. For a given source node in the graph, the algorithm finds the shortest path between that node and every other. It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined. For example, if the nodes of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest route between one city and all other cities. As a result, the shortest path algorithm is widely used in network routing protocols, most notably IS-IS and Open Shortest Path First (OSPF).

Dijkstra's original algorithm does not use a min-priority queue and runs in time $O(|V|^2)$ (where $|V|$ is the number of nodes). The implementation based on a min-priority queue implemented by a Fibonacci heap and running in $O(|E| + |V| \log |V|$ (where $|E|$ is the number of edges). This is asymptotically the fastest known single-sources shortest-path algorithm for arbitrary directed graphs with unbounded non-negative weights.

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

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

2. Set the initial node as current. Mark all other nodes unvisited. Create a set of all the unvisited nodes called the unvisited set.

3. For the current node, consider all of its unvisited neighbors and calculate their tentative distances. Compare the newly calculated tentative distance to the current assigned value and assign the smaller one. For example, if the current node A is marked with a distance of 6, and the edge connecting it with a neighbor B has length 2, then the distance to B (through A) will be 6 + 2 = 8. If B was previously marked with a distance greater than 8 then change it to 8. Otherwise, keep the current value.

4. When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the univisited set. A visited node will never be checked again.

5. If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the unvisited set is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.

6. Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and go back to step 3.

Bellman-Ford algorithm

The **Bellman-Ford algorithm** is an algorithm that computes shortest paths from a single source vertex to all of the other vertices in aweighted digraph. It is slower than Dijkstra's algorithm for the same problem, but more versatile, as it is capable of handling graphs in which some of the edge weights are negative numbers. Negative edge weights are found in various applications of graphs, hence the usefulness of this algorithm. If a graph contains a "negative cycle" that is reachable from the source, then there is no cheapest path: any path can be made cheaper by one more walk around the negative cycle. In such a case, the Bellman-Ford algorithm can detect negative cycles and report their existence.

Like Dijkstra's Algorithm. Bellman-Ford is based on the principle of relaxation, in which an approximation to the correct distance is gradually replaced by more accurate values until eventually reaching the optimum solution. In both algorithms, the approximate distance to each vertex is always an overestimate of the true distance, and is replaced by the minimum of its old value with the length of a newly found path. However, Dijkstra's algorithm greedily selects the minimum-weight node that has not yet been processed, and performs this relaxation process on all of its outgoing edges; by contrast, the Bellman-Ford algorithm simply relaxes all the edges, and does this $|V| - 1$ times, where $|V|$ is the number of vertices in the graph. In each of these repetitions, the number of vertices with correctly calculated distances grows, from which it follows that eventually all vertices will have their correct distances. This method allows the Bellman-Ford algorithm to be applied to a wider class of inputs than Dijkstra.

Bellman-Ford runs in $O(|V| \cdot |E|)$ time, where $|V|$ and $|E|$ are the number of vertices and edges respectively.

```
function BellmanFord (list vertices, list edges, vertex source)
 ::distance [], predecessor []
 //This implementation takes in a graph, represented as
 // Lists of vertices and edges and fills two arrays.
 // (distance and predecessor) with shortest-path
 //(less cost/distance/metric) information
 //step 1: initialize graph
 for each vertex v in vertices:
 if v is source then distance [v] : = 0
 else distance [v] := inf
 predecessor [v] := null
 //step 2: relax edges repeatedly
 for i from 1 to size (vertices) – 1:
 for each edge (u, v) in Graph with weight w in edges:
 if distance [u] + w < distance [v] :
 distance [v] : = distance [u] + w
 predecessor [v] := u
 //Step 3: check for negative-weight cycles
```

for each edge (u, v) in Graph with weight w in edges:

if distance $[u] + w <$ distance $[v]$:

error "Graph contains a negative-weight cycle"

return distance [], predecessor []

UNIT–IV

Q.8. What is finite automation? How to match strings with finite automata? Discuss in detail. **(12.5)**

Ans. A finite automaton M is a 5-tuple $(Q, q0, A, \Sigma, \delta)$, where

- Q is a finite set of states,
- $q_0 \in Q$ is the start state,
- $A \subseteq Q$ is a distinguished set of accepting states,
- Σ is a finite input alphabet,
- δ is a function from $Q \times \Sigma$ into Q, called the **transition function** of M.

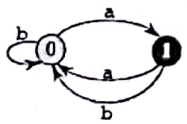
The finite automaton begins in state q_0 and reads the characters of its input string one at a time. If the automaton is in state q and reads input character a, it moves ("makes a transition") from state q to state $\delta(q, a)$. Whenever its current state q is a member of A, the machine M is said to have accepted the string read so far. An input that is not accepted is said to be rejected. Figure below illustrates these definitions with a simple two-state automaton. A finite automaton M induces a function φ, called the **final-state function,** from Σ^* to Q such that $\varphi(w)$ is the state M ends up in after scanning the string w.

Thus, M accepts a string w if and only if $\varphi(w) \in A$. The function φ is defined by the recursive relation

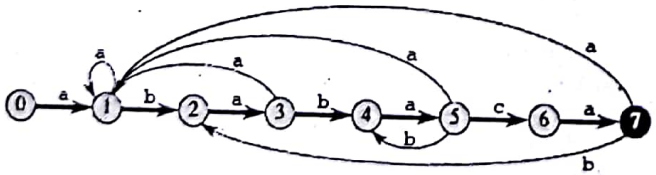
$$\varphi(\varepsilon) = q_0$$

$$\varphi(wa) = \delta(\varphi(w), a) \text{ for } w \in \Sigma^*, a \in \Sigma$$

state	input a	b
0	1	0
1	0	0



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 $x : \sigma(x) = \max \{k : P_k] x\}$.

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 $P] x$. It follows from the definition of the suffix function that if $x] y$, then $\sigma(x) \le \sigma(y)$.

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.9. What is NP-Completeness? Discuss any five NP-complete problems in detail. **(12.5)**

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 set

of 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.

Satisfiability Problem

SATISFIABILITY, or SAT, is a problem of great practical importance, with applications ranging from chip testing and computer design to image analysis and software engineering. It is also a canonical hard problem. Here's what an instance of SAT looks like:

$$(x \vee y \vee z)\,(x \vee \overline{y})\,(y \vee \overline{z})\,(z \vee \overline{x})\,(\overline{x} \vee \overline{y} \vee \overline{z}).$$

This is a Boolean formula in conjunctive normal form (CNF). It is a collection of clauses (the parentheses), each consisting of the disjunction (logical or, denoted V) of several literals, where a literal is either a Boolean variable (such as x) or the negation of one (such as x). A satisfying truth assignment is an assignment of false or true to each variable so that every clause contains a literal whose value is true. The SAT problem is the following: given a Boolean formula in conjunctive normal form, either find a satisfying truth assignment or else report that none exists.

In the instance shown previously, setting all variables to true, for example, satisfies every clause except the last. Is there a truth assignment that satisfies all clauses?

With a little thought, it is not hard to argue that in this particular case no such truth assignment exists.

(Hint: The three middle clauses constrain all three variables to have the same value.) But how do we decide this in general? Of course, we can always search through all truth assignments, one by one, but for formulas with n variables, the number of possible assignments is exponential, 2n.

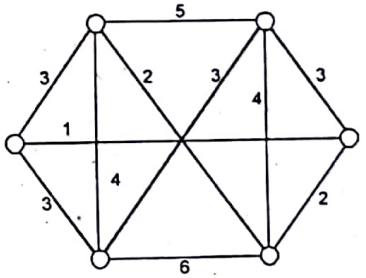
SAT is a typical search problem. We are given an instance I (that is, some input data specifying the problem at hand, in this case a Boolean formula in conjunctive normal form), and we are asked to find a solution S (an object that meets a particular specification, in this case an assignment that satisfies each clause). If no such solution exists, we must say so.

More specifically, a search problem must have the property that any proposed solution S to an instance I can be quickly checked for correctness. What does this entail? For one thing, S must at least be concise (quick to read), with length polynomially

bounded by that of I. This is clearly true in the case of SAT, for which S is an assignment to the variables. To formalize the notion of quick checking, we will say that there is a polynomial-time algorithm that takes as input I and S and decides whether or not S is a solution of I. For SAT, this is easy as it just involves checking whether the assignment specified by S indeed satisfies every clause in I. Later in this chapter it will be useful to shift our vantage point and to think of this efficient algorithm for checking proposed solutions as defining the search problem. Thus: A search problem is specified by an algorithm C that takes two inputs, an instance I and a proposed solution S, and runs in time polynomial in jlj. We say S is a solution to I if and only if C(l; S) = true.

Given the importance of the SAT search problem, researchers over the past 50 years have tried hard to find efficient ways to solve it, but without success. The fastest algorithms we have are still exponential on their worst-case inputs. Yet, interestingly, there are two natural variants of SAT for which we do have good algorithms. If all clauses contain at most one positive literal, then the Boolean formula is called a Horn formula, and a satisfying truth assignment.

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:

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

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.

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 we

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

$$F = (x_1 \vee x_1 \vee \bar{x}_4) \wedge (\bar{x}_1 \vee \bar{x}_2) \wedge (\bar{x}_2 \vee \bar{x}_2) \wedge \ldots$$

Then in this case we would create nodes likes this:

$(x_1 = 0, x_2 = 0, x_4 = 0)$ $(x_2 = 0, x_4 = 0)$ $(x_2 = 0, x_3 = 0)$...
$(x_1 = 0, x_{3-1} 1, x_4 = 0)$ $(x_2 = 0, x_4 = 1)$ $(x_2 = 0, x_3 = 1)$
$(x_1 = 0, x_2 = 1, x_4 = 1)$ $(x_2 = 1, x_4 = 1)$ $(x_2 = 1, x_3 = 0)$
$(x_1 = 1, x_2 = 0, x_4 = 0)$
$(x_1 = 1, x_2 = 0, x_4 = 1)$
$(x_1 = 1, x_2 = 1, x_4 = 0)$
$(x_1 = 1, x_2 = 1, x_4 = 1)$

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.

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. **(5x2)**

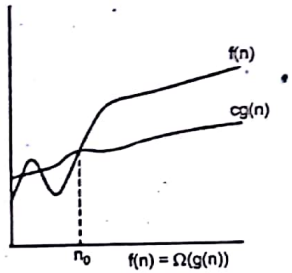
Ans. Ω Notation:

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

$0 \leq cg(n) \leq f(n)$ for all $n \geq n_0\}$

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

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

• 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 $n \ge n_0$,

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

$$f(n) = n!$$
$$= n(n-1)(n-2)\ldots 1$$
$$f(n) \le n^m$$
$$\boxed{f(n) = O(n^m)}$$

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

(ii)

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

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

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

$$T(\sqrt{n}) \le C(\sqrt{n})\lg(\sqrt{n})$$
$$T(n) \le 2(C\sqrt{n}\lg(\sqrt{n})) + 1$$
$$\le 2C\sqrt{n}\lg n^{1/2} + 1$$
$$\le 2C\left(\sqrt{n}\frac{1}{2}\lg(n) + 1\right)$$
$$\le C\sqrt{n}\lg(n) + 1$$
$$\le C\, n \log n$$

where the last step holds as long as $c \ge 1$.

(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:

$$T(n) = 4T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n$$
$$= 4\left[4T\left(\left\lfloor \frac{n}{2^2} \right\rfloor\right) + \frac{n}{2}\right] + n$$
$$= 4\left[4.\left(4T\left(\left\lfloor \frac{n}{2^3} \right\rfloor\right) + \frac{n}{2^2}\right) + \frac{n}{2}\right] + n$$
$$= 4^3 T\left(\left\lfloor \frac{n}{2^3} \right\rfloor\right) + 4^2\frac{n}{2^2} + 4.\frac{n}{2} + n$$
$$\le 4^i T\left(\frac{n}{2i}\right) + \ldots. + 4^2\frac{n}{2^2} + 4\frac{n}{2^1} + 4^0\frac{n}{2^0}$$

The series Terminaters when

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

So,

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

$$\leq \left(\frac{4^0}{2^0}\right)n + \left(\frac{4^1}{2^1}\right)n + \left(\frac{4^2}{2^2}\right)n + \ldots + 4^{\log_2 n}$$

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

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

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

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

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.

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

$$a = 5, b = 3$$

$$n^d = n^2$$

$$d = 2$$

$$d > 1$$

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

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

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

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 $[D^0]$ and the initial precedence matrix $[P^0]$ 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 $[D^0]$ and the inital distance Precedence $[P^0]$ 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. **(5×2)**

Ans. Quick sort Algorithm

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 $A[j] \le x$
5. then $i \leftarrow i + 1$
6. exchange $A[i] \leftarrow A[j]$
7. exchange $A[i+1] \leftrightarrow A[x]$
8. return $i + 1$

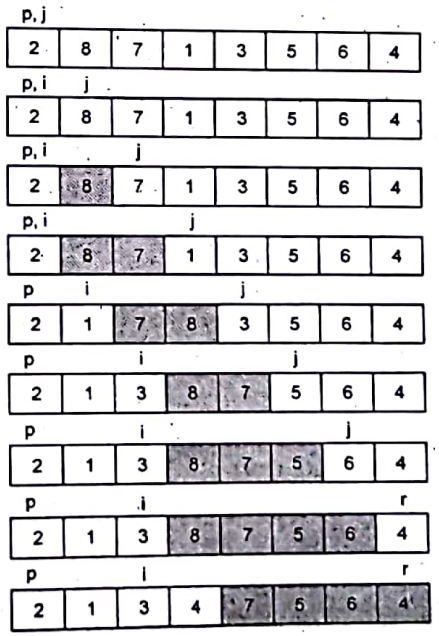
Figure shows example of partitioning on an array where $x = A[r]$ is a pivot element, we consider the following loop invariant for the quick sort algorithm to show its correctness.

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

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

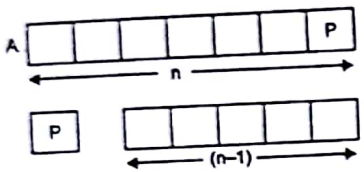
3. If $k = r$, then $A[k] = x$

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

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

Ans.

I i Pj 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

$A[i+1] \leftrightarrow A[r]$

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 |

P j r

(h) | 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 |

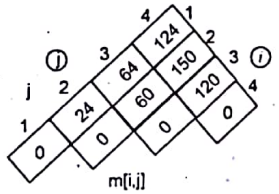
and finally the array becomes

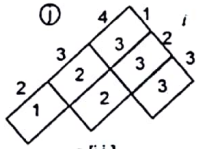
| 8 | 10 | 12 | 19 | 25 | 30 | 34 | 40 |

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





Way of paranthesization

s[1,4] = 3

(A1 A2 A3) (A4)

s[1,3] = 2

Optimal paranthesization (((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>

Ans. Refer Q.2. (b) of First Term 2014

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.1. (d) of Second Term 2014

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

Ans. Refer Q.3. (a) of Second Term 2014

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 $\pi : \{1, 2, ..., m\} \to \{0, 1, ..., m-1\}$ 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.

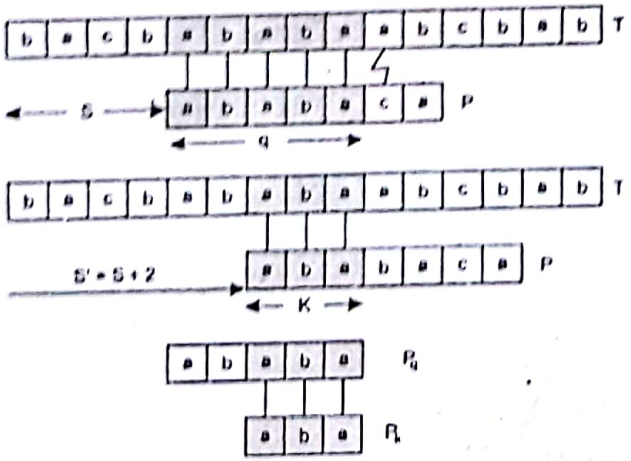


Figure shows the pattern P matches with text T so that $q = 5$. The valid shift occured at S = 4. The figure shows the pattern will not match text T when the shift $S' = S + 1$, but its valid when $S' = S + 2$. When the pattern Pq is compared with P_k, we find the array π can be represented as $\pi[5]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

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 = $O(mn)$ (quadratic in worst case)

KMP:

Running time for prefix function

calculation = $\theta(m)$

and for KMP matcher = $O(n)$

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

Finite Automata

Running time = $O(n)$

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: **(5)**

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

Ans. Optimal Huffman code.

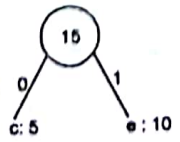
A:45 b: 15 C: 5 d:25 e: 10

(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

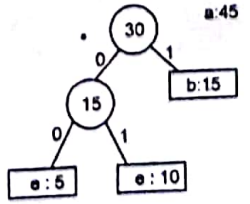
(a) c: 5 e: 10 b : 15 d: 25 a: 45

(b)

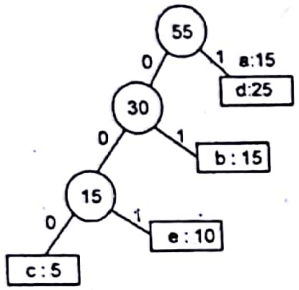


b : 15 d: 25 a: 45

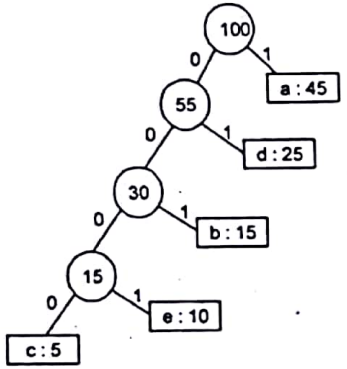
(c) d:25



(d)



(e)



Hence the final coding for each of the given characters is

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

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

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 equal 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:

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

$\mathbf{P} = \{L \mid L = L(M)$ 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 $L_R \in \mathbf{P}$. 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 $w \in \Sigma^*$, $w \in L \Leftrightarrow \exists y(|y| \le |w|_k$ 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.

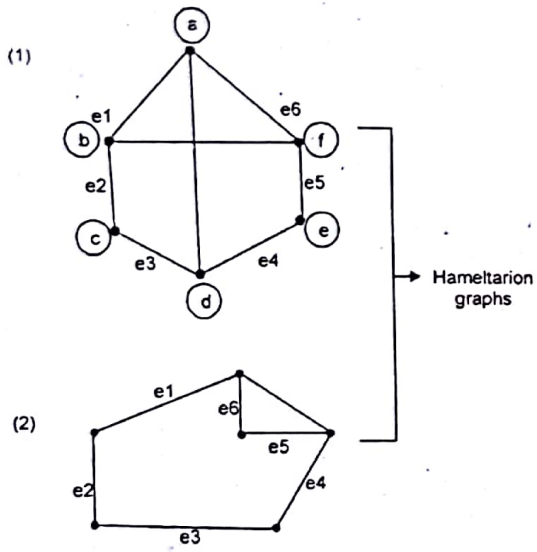
Q.4. Attempt (any two) **(5×2)**

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.

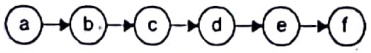
A Hamiltonian decomposition is an edge decomposition of a graph into Hamiltonian circuits. Examples of Hamiltonian graphs.



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 $1 \le d_i \le n$ 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) -

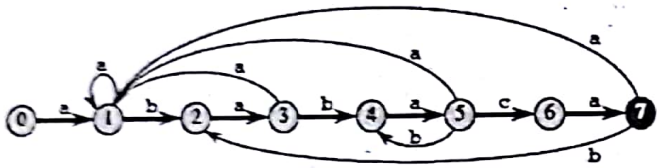
Ans. String matching algorithm using finite automata: There is a string-matching automaton for every pattern P; this automaton must be constructed from the pattern in a pre-processing step before it can be used to search the text string. Figure below illustrates this construction for the pattern $P =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

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 $\sigma(x) \le \sigma(y)$.

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 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. $n \leftarrow$ 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 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.4. (d) Proof of correctness of Bellman-Ford algorithm.

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

The correctness of algorithm can be shown by induction.

Lemma: After i repetition of for loop:

- If 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 the 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

0 ≤ 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)**

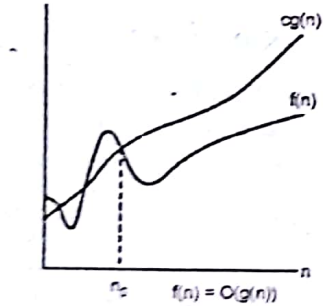
Ans. O Notation:

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

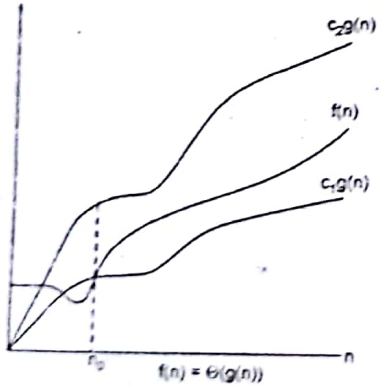
$0 \le f(n) \le cg(n)$ for all $n \ge n_0\}$

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 $0 \le c_1g(n) \le f(n) \le c2g(n)$ 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 $c_1g(n)$ and $c_2g(n)$, 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 $0(g(n))$. 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 $3n^2 + 2n^2 = 0\ (n^2)$; $3^n \ne 0\ (2^n)$

Ans.

Proof

when $n \ge 1$

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

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

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

$g(n) = 2^n$

$f(n) \not\le g(n)$ for all $n \ge 1$

Hence $f(n) \ne 0\ (g(n))$

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

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)
```

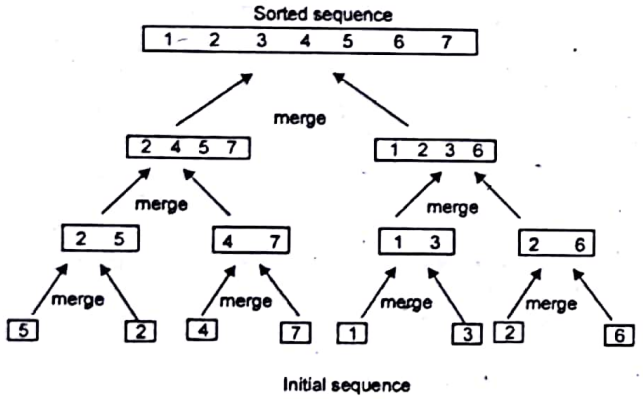


Fig.: The operation of merge sort on the array $A = 5, 2, 4, 7, 1, 3, 2, 6$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top

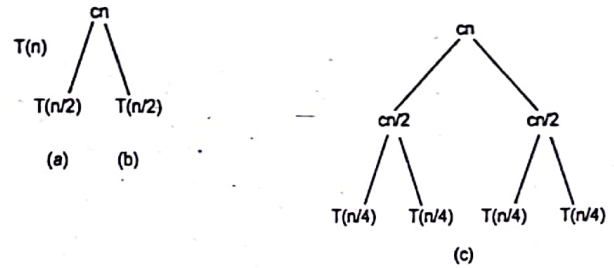
Analysis of merge sort: Merge sort on just one element takes constant time. When we have $n > 1$ elements, we break down the running time as follows.

- **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 $C(n) = \Theta\ (n)$.

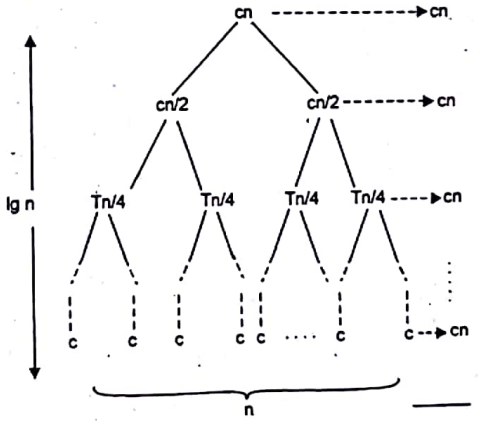
When we add the functions $D(n)$ and $C(n)$ for the merge sort analysis, we are adding a function that is $\Theta\ (n)$ and a function that is $\Theta\ (1)$. This sum is a linear function of n, that is, $\Theta\ (n)$. Adding it to the $2T(n/2)$ term from the "conquer" step gives the recurrence for the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

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

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_i-w_i] additional value. On other hand if theif decides 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-1, w-wi]
THEN c[i,w]=v_i + c [i -1, w-w_i]
ELSE c(i,w) = c[i-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-1, w] item i is not part of the solution, and we are continue tracing with c[i-1, w]. Otherwise item i is part of the solution, and we continue tracing with c[i-1, w-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_x is a function of m immediately preceding terms of the sequence. If m is a small positive integer, a_x depends on a finite amount of history [GK90]. If $n = m$, the value of a_x 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.

$$a_x = c_1 a_{x-1} + c_2 a_{x-2} + \dots + c_m a_{x-m}$$
$$a_x = c a_{x-1} + f(n) \quad f(n) \text{ is a function of } n$$
$$a_{x,n} = a_{x-1,n} + a_{x-1,n-1}$$

$$a_x = c a_{x-1}m + a_{x-1,n-1}$$
$$a_x = a_1 a_{x-1} + a_2 a_{x-2} + \dots + a_{x-1} a_1$$

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, $n \ge 1$. 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. **(6.25)**

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

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

where $a \ge 1$ 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 $a \ge 1$ 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:

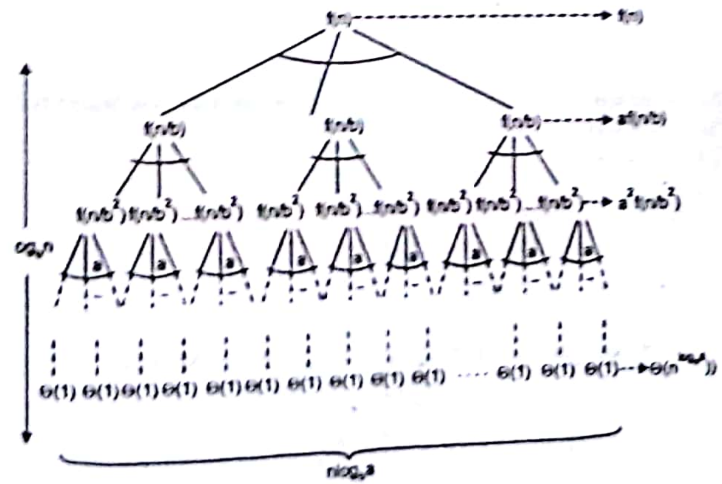
1. If $f(n) = O(n^{\log_b a-\epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

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 $\Omega(n^{3+\epsilon})$ 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.2. (a) Explain Strassen's algorithm and explain. **(6.25)**

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

Q.2. (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 x 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

$P1 = a(f-h)$; $P2 = (a+b)h$

$P3 = (c+d)e$; $P4 = d(g-e)$

$P5 = (a+b)(e+h)$; $P6 = (b-d)(g+h)$

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

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

$$\left[\begin{array}{c|c}a&b\\\hline c&d\end{array}\right] \times \left[\begin{array}{c|c}e&f\\\hline g&h\end{array}\right] = \left[\begin{array}{c|c}P5+P4-P2+P6 & P1+P2\\\hline P3+P4 & P1+P5-P3-P7\end{array}\right]$$

A B C

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

So,

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

c d

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

g h

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,

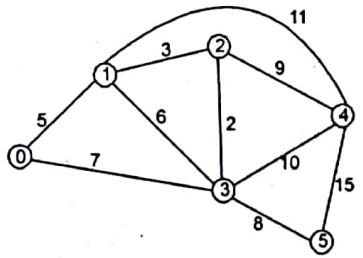
$$C \text{ is as } \left[\begin{array}{c|c} P5+P4-P2+P6 & P1+P2 \\ \hline P3+P4 & P1+P5-P3-P7 \end{array}\right]$$

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

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

Ans. Refer Q.6 of End Term 2014.

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

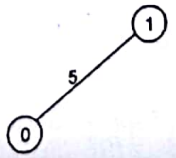


Ans. In prim's algorithm, first we initialize the priority queue Q to contain all the vertices and the keys of each vertex to ∞ except the root, whose key is set to 0.

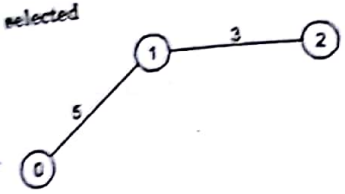
Let (0) is the root vertex.

So,

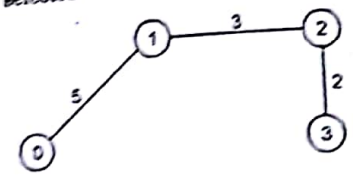
Step 1: edge {0,1} is selected



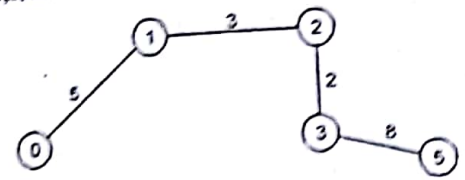
Step 2: edge (1,2) is selected



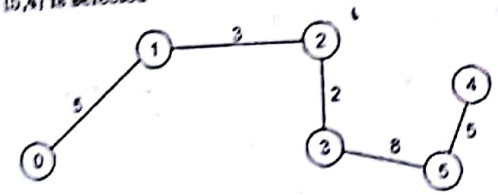
Step 3: edge (2,3) is selected



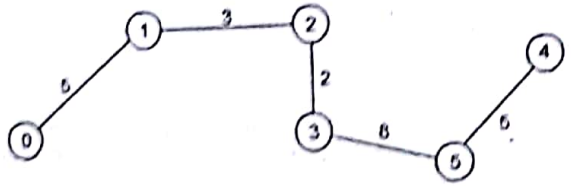
Step 4: edge (3,5) is selected



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:

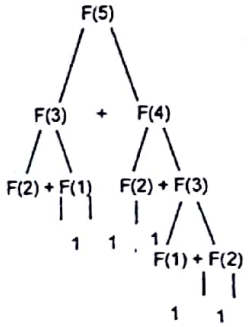
n : 1 2 3 4 5 6 7 8 9 10 ...

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 simply:

```
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	∞

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

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 $1 \le i \le n$. $1 \le j \le m$.

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

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	*a*	*b*	*b*	*a*	*b*	*c*	*a*	*b*	*d*	*d*	*a*	*d*	*b*	*c*	*a*
000	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*".

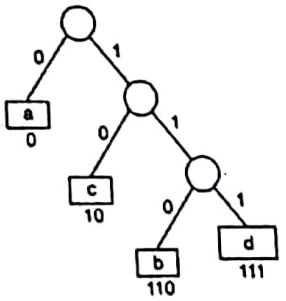


Fig. Corresponding binary tree for prefix codes.

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 thie 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.

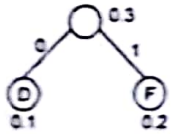
Character	A	B	C	D	E	F
Probability	0.5	0.35	0.5	0.1	0.4	0.2

arrange in dec. order of probability

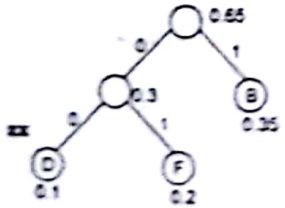
Character	Probability
A	0.5
C	0.5
E	0.4
B	0.35
F	0.2
D	0.1

Huffman tree is as following:

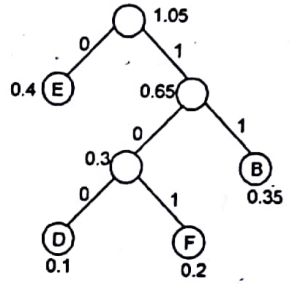
Step 1:



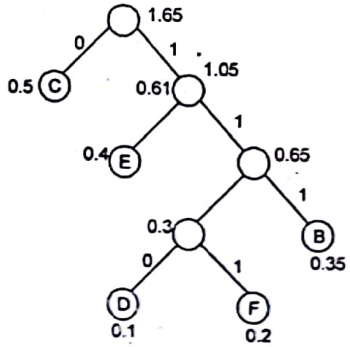
Step 2:



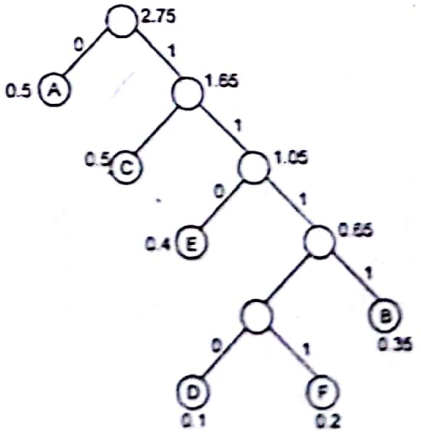
Step 3:



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.3. (*b*) Second Term Examination 2015

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

Ans. Refer Q.9. of End Term Examination 2014

Q.8. (a) Write short note on following. **(6.25x2=12.5)**

Q.8. (a) Khuth-Morics Pratt algorithm.

Ans. Refer Q.3. (*a*) of End Term Examination 2014

Q.8. (b) OBST problem.

Ans. Refer Q.4. (*b*) of End Term Examination 2014

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 $f(s) = O(g(s))$ $(s \in S)$, 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 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

$$c_1 n^b \leq (n+a)^b \leq c_2 n^b.$$
$$(n+a)^b = (n+a)(n+a)(n+a)...(n+a) \text{ b times}$$
$$= n^b + k_1 n^{b-1} + k_2 n^{b-2} + ... + k_{b-1} n^1 + k_b$$
$$c_1 n^b \leq n^b + k_1 n^{b-1} + k_2 n^{b-2} + ... + k_{b-1} n^1 + k_b \leq c_2 n^b.$$

Now, $$c_1 n^b \le n^b + k_1 n^{b-1} + k_2 n^{b-2} + \ldots + k_{b-1} n^1 + k_b.$$

$$c_1 \le 1 + \frac{k_1}{n} + \frac{k_2}{n^2} + \ldots + \frac{k_b}{n^b}.$$

It is possible to find c_1 small enough such that for particular values of $k_1, k_2, \ldots k_{b-1}, k_b$ even if values are negative for $n \ge n_0$.

i.e., $$(n+a)^b = \Omega(n^b) \quad \ldots(1)$$

Now, $n^b + k_1 n^{b-1} + k_2 n^{b-2} + \ldots + k_{b-1} n^1 + k_b \le n^b + k_1 n^b + k_2 n^b + \ldots k_{b-1} n^b + k_b n^b = kn^b$

i.e. $n^b + k_1 n^{b-1} + k_2 n^{b-2} + \ldots + k_b \le c_2 n^b \; \forall \; n \ge n_0$.

$$(n+a)^b = O(n^b) \quad \ldots(2)$$

From (1) and (2), we get $(n+a)^b = \theta(n^b)$

Q.1. (d) 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 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 */
int fib(intn)
{
  if( n <= 1 )
    returnn;
  returnfib(n-1) + fib(n-2);
}
```

Q.1. (e) 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 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.

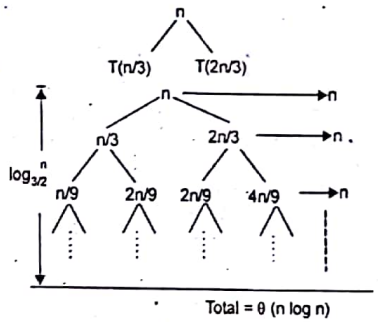
5. There exists a recursive relationship that identifies the optimal decision for stage j, given that stage j+1 has already been solved.

6.The final stage must be solvable by itself.

Q.2. Solve the following reccurence relations: **(4x2.5)**

(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

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

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, $T(n) \le cn\log_2 n$

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

Substituting into recurrence, it yields

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

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

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

$\Rightarrow n \log_2 n/2^k = n$

$\Rightarrow n \log_2 n - n \log_2 2^k = n$

$\Rightarrow n \log_2 n - kn = n$

$\Rightarrow n \log_2 n$

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

Q.2. (a) $T(n) = 1$ if $n = 1$ (using iteration method)

$2T(n/2) + n$ if $n > 1$

Ans: $T(n) = 2\,T(n/2)$

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

$= 2^3 T(n/8)$

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.2. (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	$n-1$

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

To compute T(n) the running time, we sum the product of the cost and times 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 $j = 2,3,...n$, we then find that A[i] ≤ key in line 5 when $i = j - 1$.

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

$T(n) = c_1 n + c_2 (n-1) + c_4 (n-1) + c_5 (n-1) + c_8 (n-1)$

$= (c_1 + c_2 + c_4 + c_5 + c_8) n - (c_2 + c_4 + c_5 + c_8)$

$= an + b$, 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 element A[j] with each element in the entire sorted subarray A[1...j – 1] and $t_j = j$ for $j = 2,3,...n$.

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

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

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

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

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

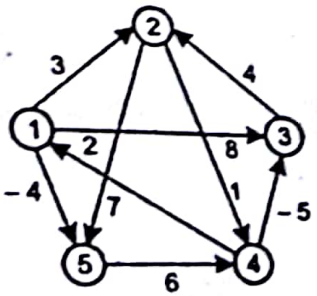
$= an^2 + bn + c$ for a, b and c are constant

= quadratic function of n

$= O(n^2)$.

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

Ans.



Ans.

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

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

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

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

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

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

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

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

Q.4. (a) Find the optimal parenthesization of a matrix chain product whose sequence of dimensions are <4,10,3,12,20,7> **(5)**

Ans: The matrices have sizes $4*10, 10*3, 3*12, 12*20, 20*7$. We need to compute $M[1,5]$ for $n=5$. 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

$$M[1,3] = \min\begin{cases} M[1,2] + M[3,3] + p_0p_2p_3 = 120 + 0 + 4.3.12 = 264 \\ M[1,1] + M[2,3] + p_0p_1p_3 = 0 + 360 + 4.10.12 = 840 \end{cases} = 264$$

$$M[2,4] = \min\begin{cases} M[2,3] + M[4,4] + p_1p_2p_4 = 360 + 0 + 10.12.20 = 2760 \\ M[2,2] + M[3,4] + p_1p_2p_4 = 0 + 720 + 10.3.20 = 1320 \end{cases} = 1320$$

$$M[3,5] = \min\begin{cases} M[3,4] + M[5,5] + p_2p_4p_5 = 720 + 0 + 3.20.7 = 1140 \\ M[3,3] + M[4,5] + p_2p_3p_5 = 0 + 1680 + 3.12.7 = 1932 \end{cases} = 1140$$

	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

$$M[1,4] = \min\begin{cases} M[1,3] + M[4,4] + p_0p_3p_4 = 264 + 0 + 4.12.20 = 1224 \\ M[1,2] + M[3,4] + p_0p_2p_4 = 120 + 720 + 4.3.20 = 1080 = 1080 \\ M[1,1] + M[2,4] + p_0p_1p_4 = 0 + 1320 + 4.10.20 = 2120 \end{cases}$$

$$M[2,5] = \min\begin{cases} M[2,4] + M[5,5] + p_1p_4p_5 = 1320 + 0 + 10.12.7 = 2720 \\ M[2,3] + M[4,5] + p_1p_3p_5 = 360 + 1680 + 10.12.7 = 2880 = 1350 \\ M[2,2] + M[3,5] + p_1p_2p_5 = 0 + 1140 + 10.3.7 = 1350 \end{cases}$$

	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

⇒

	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

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

	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

⇒

	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 also 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> **(5)**

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

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

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 $f(n) = \omega(g(n))$ 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. $o(n^4)$.

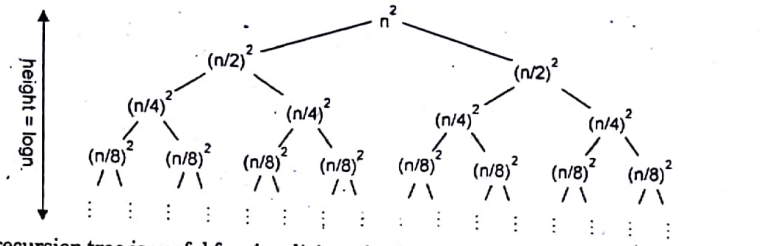
Q.1. (b) What is recursion tree method? **(2.5)**

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:

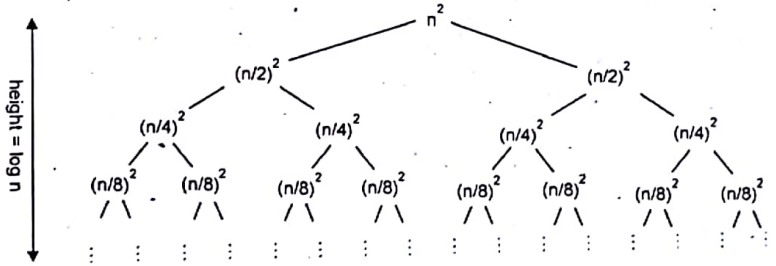


A recursion tree is useful for visualizing what happens when a recurrence is iterated. It diagrams the tree of recursive calls and the amount of work done at each call.

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)**

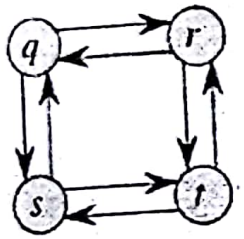
Ans: Optimal Substructure :

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 in a table so that these don't have to recomputed. So Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point storing the 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 ⊆ 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 ∪ {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.

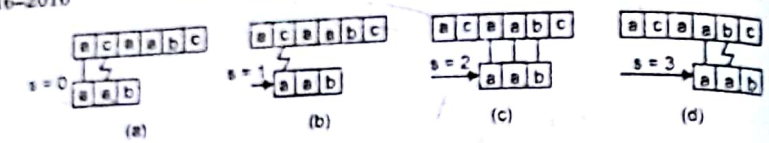
Q.1. (i) Write Naïve String Matching Algorithm. **(2.5)**

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 $n \leftarrow$ length [T]
2 $m \leftarrow$ length[P]
3 **for** $s \leftarrow 0$ **to** $n - m$
4 **do** *if* $P[1 \ldots m] = T[s + 1 \ldots s + m]$
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

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

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

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

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

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.

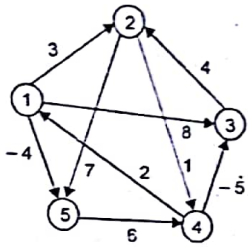
FLOYD-WARSHALL (W)

1. n ← 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 $d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)}, d_{kj}^{(k-1)})$
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] ¬ "↑"
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 ritght-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. Selέct 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

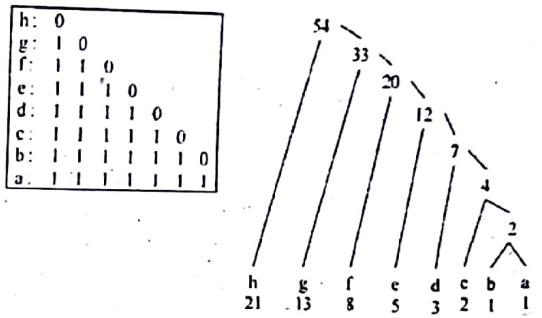
MST-PRIM (G,w,r)

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.



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 occurs 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) mod q
4. p ← 0
5. t0 ← 0
6. for i ← 1 to m
7.  do p ←(dp + P[i]) mod q
8.  t0 ←(dt0 + T[i]) mod q
9. for s ← 0 to n-m
```

10 do if p = t,
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. **(0.5)**

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 $\delta: Q \times \Sigma \rightarrow Q$
- 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 ← 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

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)**

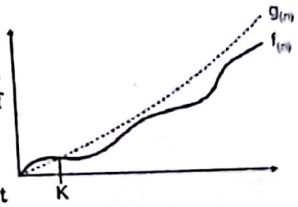
Ans. Algorithm: An algorithm (pronounced AL-go-rith-um) is a procedure or formula for solving a problem, based on conductiong a sequence of specified actions. A computer program can be viewed as an elaborate algorithm. In mathematics and computer science, an algorithm usually means a small procedure 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.

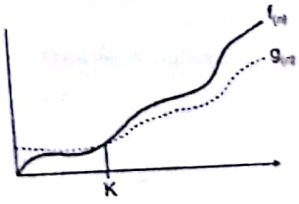


For example, for a function *f*(n)

$O(f(n)) = \{ g(n) :$ there exists $c > 0$ and n_0 such that $f(n) \leq c.g(n)$ for all $n > n_0. \}$

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.



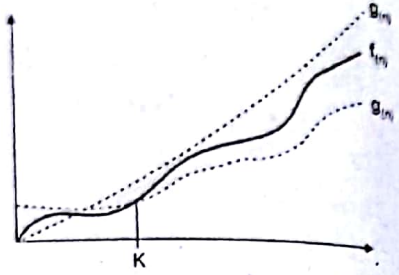
For example, for a function *f*(n)

$\Omega(f(n)) \geq \{g(n)$: 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 θ(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)

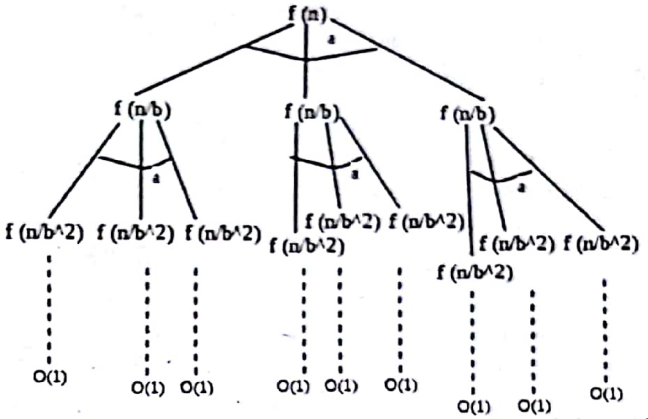
Ans. Master Method is a direct way to get the solution. The master method works only for following type of recurrences or for recurrences that can be transformed to following type.

$T(n) = aT(n/b) + f(n)$ 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 $\text{Log}_b n$



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 (Case3).

Examples of some standard algorithms whose time complexity can be evaluated using 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 the 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 0. 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 using Master Theorem. The given three cases have some gaps between them. For example, 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 $t(n) = 3T(n/4) + n \lg n$? 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.,

$$af\left(\frac{n}{b}\right) \leq c f(n)$$

$$3\left(\frac{n}{4}\right)\log\left(\frac{n}{4}\right) \leq \left(\frac{3}{4}\right) n \lg n = cf(n) \text{ for } c = \frac{3}{4}$$

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) + nlogn? 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) = nlogn

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. **(5)**

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 a c and an n_0 such that

$$T(n) \geq cn^2 lg(n)$$

ð $n_0 > n$ 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	15	15	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 maximum 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 15 + 15 = 30.

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 $I \subseteq 2^S$, known as the independent sets, satisfying the following axioms: 1. If $I \in I$ and $J \subseteq I$ then $J \in I$. 2. If $|J| < |I|$ and $|J| > |II|$, then there exists an element $x \in J \setminus I$ such that $I \cup \{x\} \in I$.

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.

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.

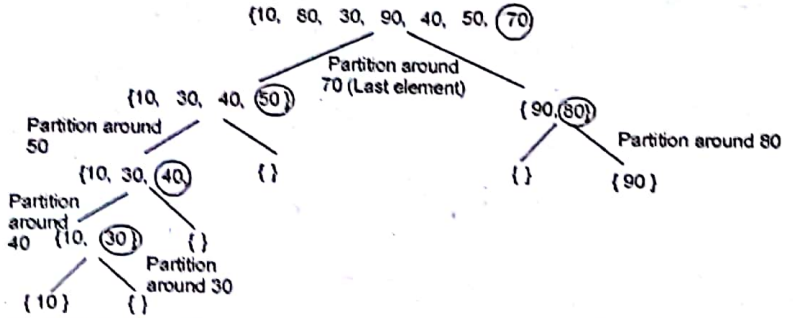
Ans. There are many different versions of quickSort that pick pivot in different ways.

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 cn*cn*c, n time for some constant ccc, the recursive call on n–1 *n*–1n, 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*–2n, 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:

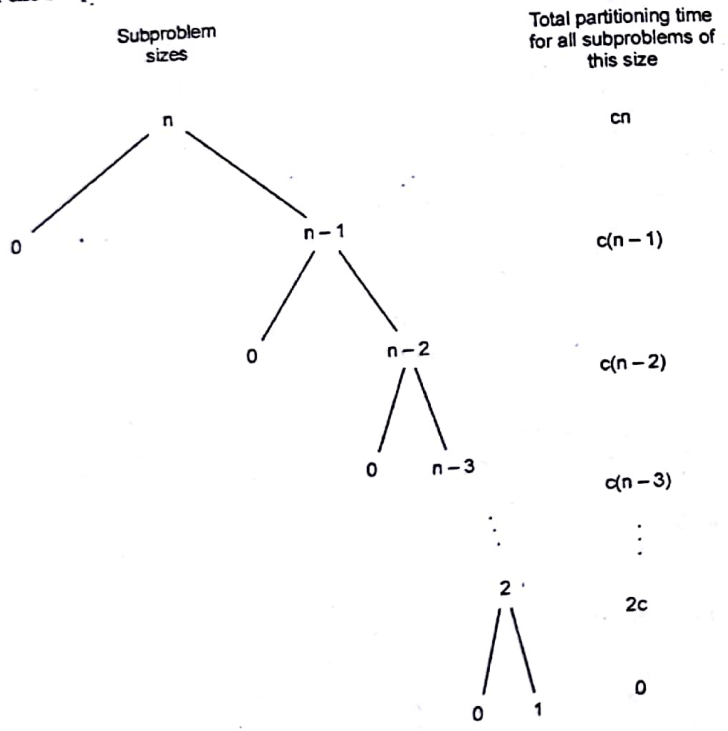


Diagram of worst case performance for Quick Sort

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)+...+2*c*

=*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 (*n*2).

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)/2left parenthesis, n, minus, 1, right parenthesis, slash, 2elements. The latter case occurs if the subarray has an even number n*n*n of elements and one partition has n/2*n*/2n, slash, 2 elements with the other having n/2-1*n*/2–1n, slash, 2, minus, 1. In either of these cases, each partition has at most n/2*n*/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:

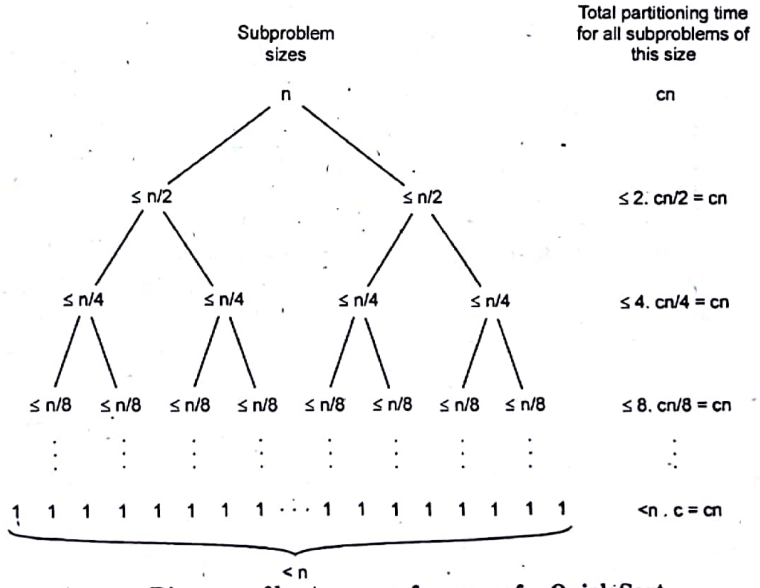
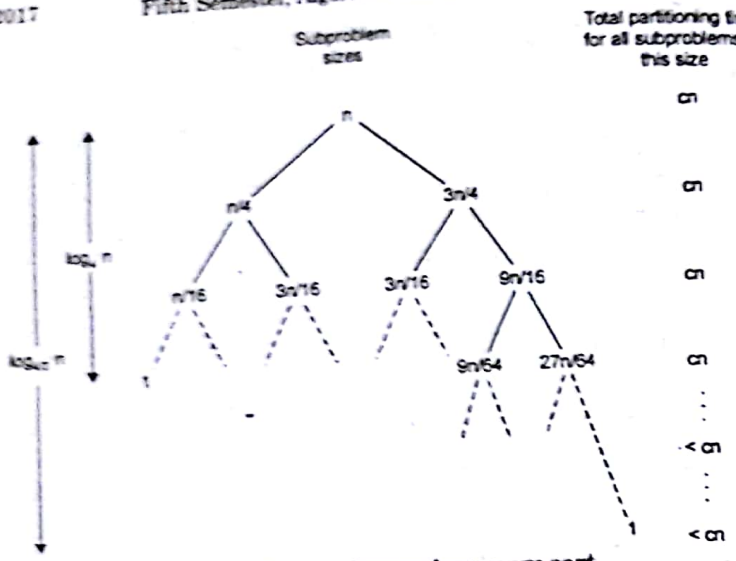


Diagram of best case performance for Quick Sort

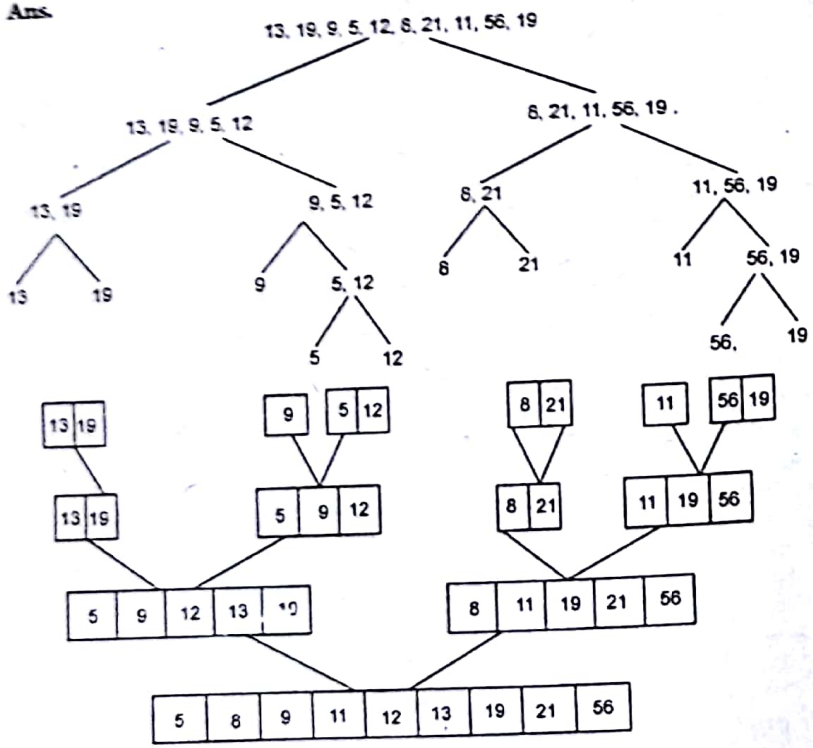
Using big-Θ notation, we get the same result as for merge sort: \Theta(n \log_2 n)Θ (*n*log2*n*).

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 3n/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.

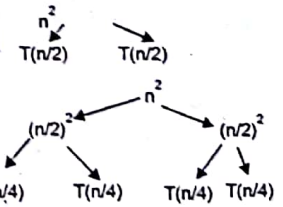
MAKE-SET(x): make a new set Si = {x} and add Si to S

- 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, w)
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) + n² (using recurrence tree)

Ans.

$T(n) = n^2+n^2/4+n^2/4^2+.....\lg n \text{ times}$
$<= n^2(1/1-1/4)$
$T(n) = \text{Theta}(n^2)$

(ii) T(n) = 5T(n/4) + (n³) (using master theorem)

Ans. By Master theorem

$a = 5 \quad b = 4 f(n) = n^3$

$n^{\log_b a} = n^{\log 45} = n^2$

$f(n) = n^3 = n \log^{a+\epsilon} = n^{2+1}$

therefore $\epsilon = 1$

hence by case 3 sol is $T(n) = \text{theta}(n^3)$

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	
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 optimal 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

$$M[1,3] = \min\begin{cases} M[1,2] + M[3,3] + p_0p_2p_3 = 120 + 0 + 4 \cdot 3 \cdot 12 = 264 \\ M[1,1] + M[2,3] + p_0p_1p_3 = 0 + 360 + 4 \cdot 10 \cdot 12 = 840 \end{cases} = 264$$

$$M[2,4] = \min\begin{cases} M[2,3] + M[4,4] + p_1p_3p_4 = 360 + 0 + 10 \cdot 12 \cdot 20 = 2760 \\ M[2,2] + M[3,4] + p_1p_2p_4 = 0 + 720 + 10 \cdot 3 \cdot 20 = 1320 \end{cases} = 1320$$

$$M[3,5] = \min\begin{cases} M[3,4] + M[5,5] + p_2p_4p_5 = 720 + 0 + 3 \cdot 20 \cdot 7 = 1140 \\ M[3,3] + M[4,5] + p_2p_3p_5 = 0 + 1680 + 3 \cdot 12 \cdot 7 = 1932 \end{cases} = 1140$$

	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

$$M[1,4] = \min\begin{cases} M[1,3] + M[4,4] + p_0p_3p_4 = 264 + 0 + 4.12.20 = 1224 \\ M[1,2] + M[3,4] + p_0p_2p_4 = 120 + 720 + 4.3.20 = 1080 \\ M[1,1] + M[2,4] + p_0p_1p_4 = 0 + 1320 + 4.10.20 = 2120 \end{cases} = 1080$$

$$M[2,5] = \min\begin{cases} M[2,4] + M[5,5] + p_1p_4p_5 = 1320 + 0 + 10 \cdot 20 \cdot 7 = 2720 \\ M[2,3] + M[4,5] + p_1p_3p_5 = 360 + 1680 + 10 \cdot 12 \cdot 7 = 2880 \\ M[2,2] + M[3,5] + p_1p_2p_5 = 0 + 1140 + 10 \cdot 3 \cdot 7 = 1350 \end{cases} = 1350$$

	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

⇒

	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

$$M[1,5] = \min\begin{cases} M[1,4] + M[5,5] + p_0p_4p_5 = 1080 + 0 + 4 \cdot 20 \cdot 7 = 1544 \\ M[1,3] + M[4,5] + p_0p_3p_5 = 264 + 1680 + 4 \cdot 12 \cdot 7 = 2016 \\ M[1,2] + M[3,5] + p_0p_2p_5 = 120 + 1140 + 4 \cdot 3 \cdot 7 = 1344 \\ M[1,1] + M[2,5] + p_0p_1p_5 = 0 + 1350 + 4 \cdot 10 \cdot 7 = 1630 \end{cases} = 1344$$

	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

⇒

	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, 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 ($s, s[i, j] + 1, j$)
6. print ")"

Now for optimal paranthesization, Each time we find the optimal value for $M[i, j]$ we also 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 $((A_1A_2)(A_3A_4A_5))$. 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 $((A_1A_2)((A_3A_4)A_5))$. Thus the optimal solution is to parenthesize $((A_1A_2)((A_3A_4)A_5))$.

Q. 5. (a) Compute binomial coefficient 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.

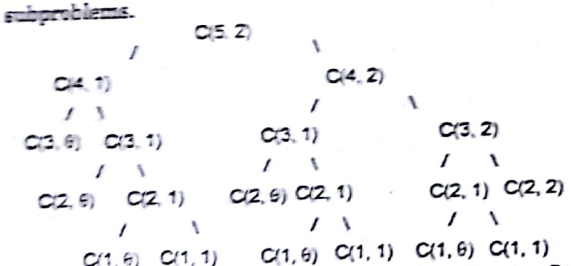
The Problem: *Write a function that takes two parameters n and k and returns the value of Binomial Coefficient C(n, k).* For example, your function should return 6 for n = 4 and k = 2, and it should return 10 for n = 5 and k = 2.

(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 Subproblems 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.

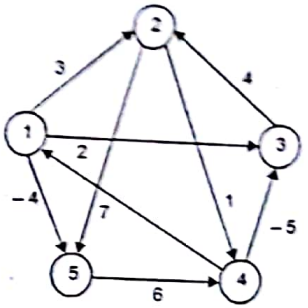
FLOYD-WARSHALL (W)

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 $\theta(n^3)$.

Apply Floyd Warshall algorithm for constructing shortest path



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

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

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

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

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

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

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

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 ← ∅
3. Q ← V[G]
4. while Q ≠ ∅
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.

1	2	3	4	5	6	7
	P_7					

Step: 4 Next Job is P3. Insert it in array J [] at index 4.

1	2	3	4	5	6	7
	P_7		P_3			

Step: 5 Next Job is P4. It has a deadline 3. Therefore insert it at index 3.

1	2	3	4	5	6	7
	P_7	P_4	P_3			

Step: 6 Next Job is P6, it has deadline 1. Hence Place P6 at index 1.

1	2	3	4	5	6	7
P_6	P_7	P_4	P_3			

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 knapsack problem.

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.,

35
20
5

– 60 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 time 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 time complexity.For example: finding the greatest number in a series of multiple numbers, will take atmost 'N' number of steps , where N is the count of numbers in the sequence.

Where as NP problems may include those which cannot be solved(or are yet to be solved) in Polynomial time

Although some NP problems are yet to be solved in Polynomial time, once we have 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 knapsack if you had lots of different sized 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 travelling salesman problem. In fact, any NP problem can be made into a travelling salesman 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 a 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 path through 500 cities in your state would take forever to solve. Even if someone walked up to you and gave you an itinerary and claimed it was the absolute shortest path, it'd still 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) 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 occurs with shift"i-m
12 q<-π[q]
```