INDEX

□□□

Analysis of Algorithm

Statistical Analysis

Chapter No.	May 2012	Dec. 2012	May 2013	Dec. 2013
Chapter 1	15 Marks	05 Marks	10 Marks	15 Marks
Chapter 2	20 Marks	30 Marks	25 Marks	30 Marks
Chapter 3	30 Marks	30 Marks	30 Marks	25 Marks
Chapter 4	25 Marks	25 Marks	15 Marks	20 Marks
Chapter 5	25 Marks	15 Marks	20 Marks	20 Marks
Chapter 6	10 Marks	20 Marks	20 Marks	10 Marks
Chapter 7	20 Marks	15 Marks	20 Marks	20 Marks
Repeated Questions	–	50 Marks	65 Marks	120 Marks

May 2012

Chapter 1 : Introduction to Analysis of Algorithm [Total Marks - 15]

Q. 1(a) Explain Big-oh, Omega and Theta Notations with help of example. How do we analyse and measure time and space complexity of algorithms ? **(10 Marks)**

Ans. : Two important ways to characterize the effectiveness of an algorithm are its space complexity and time complexity. Time complexity of an algorithm concerns determining an expression of the number of steps needed as a function of the problem size. Since the step count measure is somewhat coarse, one does not aim at obtaining an exact step count. Instead, one attempts only to get asymptotic bounds on the step count. Asymptotic analysis makes use of the O (Big Oh) notation. Two other notational constructs used by computer scientists in the analysis of algorithms are θ (Big Theta) notation and Ω (Big Omega) notation. The performance evaluation of an algorithm is obtained by totaling the number of occurrences of each operation when running the algorithm. The performance of an algorithm is evaluated as a function of the input size n and is to be considered modulo a multiplicative constant.

The following notations are commonly use notations in performance analysis and used to characterize the complexity of an algorithm.

θ-Notation (Same order)

This notation bounds a function to within constant factors. Write $f(n) = \theta(g(n))$ if there exist positive constants n_0, c_1 and c_2 such that to the right of n_0 the value of $f(n)$ always lies between $c_1g(n)$ and $c_2g(n)$ inclusive. $f(n) = \theta(g(n))$

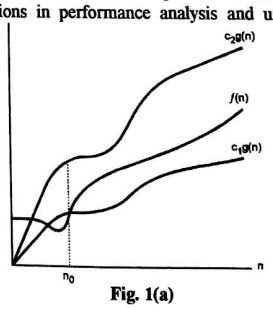


Fig. 1(a)

O-Notation (Upper Bound) :

This notation gives an upper bound for a function to within a constant factor. Write $f(n) = O(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or below $cg(n)$.

The functions used in this estimation often include the following :
1, log (n), n, n log (n), n^2, 2^n, n!
It has two main areas of application :

In computer science, it is useful in the analysis of the complexity of algorithms. In mathematics, it is usually used to characterize the residual term of a truncated infinite series, especially an asymptotic series. Big-O gives us a formal way of expressing asymptotic upper bounds, a way of bounding from above the growth of a function.

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

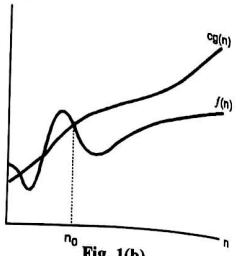


Fig. 1(b)

Ω-Notation (Lower Bound)

This notation gives a lower bound for a function to within a constant factor. Write $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or above $cg(n)$.

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

$g(n)$ is an asymptotic lower bound for $f(n)$.

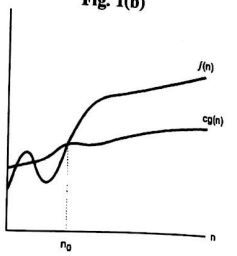


Fig. 1(c)

Time Complexity :

The time complexity of a problem is defined as the number of steps required to solve the entire problem using some efficient algorithm.

The time complexity of an algorithm is the amount of computer time required by an algorithm to run to completion. It is difficult to compute the time complexity in terms of physically clocked time. For instance in multiuser system, executing time depends on many factors such as :

1. System load
2. Number of other programs running
3. Instruction set used
4. Speed of underiying hardware

The time complexity is therefore given in terms of frequency count. Frequency count is a cont denoting number of times of execution of statement.

Space complexity :

The space complexity of a problem is defined as the amount of space and memory required by an algorithm to solve the problem. Space complexity is measured in terms of Big O notation. There are her measures of computational complexity. For instance, communication complexity is a measure of

complexity for distributed computations. A different measure of problem complexity, which is useful in some cases, is circuit complexity. This is a measure of the size of a Boolean circuit needed to compute the answer to a problem, in terms of the number of logic gates required to build the circuit. Such a measure is useful, for example, when designing hardware microchips to compute the function instead of software.

Q. 7(b) Write short note on : Randomized Algorithm **(5 Marks)**

Ans. : Randomized Algorithms :

A randomized algorithm are also called as probabilistic algorithm. The algorithm typically uses uniformly random bits as an input to guide its behavior and it tries to achieve good performance in the "average case". Basically, the algorithm's performance will be a random variable and thus either the running time, or the output are random variables. In common practice, randomized algorithms are approximated using a pseudorandom number generator in place of a true source of random bits; such an implementation may deviate from the expected theoretical behavior.

Consider the following example : The problem of finding 'a' in an array of n elements :

In a given array of n elements, let half the elements are 'a' and that of half are 'b' . One of the approach is to look at each element of the array and this is an expensive one and will take n/2 operations, if the array were ordered as 'b's first followed by 'a's. With this approach, we can not guarantee that the algorithm will complete quickly. On the other hand, if we look randomly then we can find 'a' quickly with high probability.

Randomized algorithms are particularly useful when faced with a malicious "adversary" or attacker who deliberately tries to feed a bad input to the algorithm. It is for this reason that randomness is ubiquitous in cryptography. In cryptographic applications, pseudo-random numbers cannot be used, since the adversary can predict them, making the algorithm effectively deterministic. Therefore either a source of truly random numbers or a cryptographically secure pseudo-random number generator is required. Another area in which randomness is inherent is quantum computing.

In the example above, the randomized algorithm always outputs the correct answer, but its running time is a random variable. Sometimes we want the algorithm to complete in a fixed amount of time , but allow a small probability of error. The former types are called Las Vegas algorithms, and the latter are Monte Carlo algorithms.

Chapter 2 : Divide and Conquer [Total Marks - 20]

Q. 3(a) Write Randomized Quick Sort Algorithm and explain with help of example. **(10 Marks)**

Ans. :

Randomized Version of Quick Sort Method and Analysis :

1. In the randomized version of Quick sort impose a distribution on input.
2. In this version choose a random key for the pivot.
3. Assume that procedure Random (a, b) returns a random integer in the range (a, b); there are $b - a + 1$ integers in the range and procedure is equally likely to return one of them.

4. The new partition procedure, simply implemented the swap before actually partitioning.

Algorithm : Randomized partition

```
RANDOMIZED_PARTITION (A, p, r)
i ← RANDOM (p, r)
Exchange A[p] ↔ A[i]
return PARTITION (A, p, r)
```

Now randomized quick sort call the above procedure in place of PARTITION

Algorithm : Randomized version of QuickSort

```
RANDOMIZED_QUICKSORT (A, p, r)
If p < r then
  q ← RANDOMIZED_PARTITION (A, p, r)
  RANDOMIZED_QUICKSORT (A, p, q)
  RANDOMIZED_QUICKSORT (A, q+1, r)
```

Analysis of Quick Sort for Randomized Version :

(1) Worst-case :

Let T(n) be the worst-case time for QUICK SORT on input size n. We have a recurrence

$$T(n) = \max_{1 \le q \le n-1} (T(q) + T(n-q)) + \theta(n) \quad \ldots(1)$$

where q runs from 1 to n–1, since the partition produces two regions, each having size at least 1.

Now $T(n) \le cn^2$ for some constant c.

Substituting our guess in equation 1.We get

$$T(n) = \max_{1 \le q \le n-1} (cq^2) + c(n-q^2)) + \theta(n)$$

$$= c \max (q^2 + (n-q)^2) + \theta(n)$$

Since the second derivative of expression $q^2 + (n-q)^2$ with respect to q is positive. Therefore, expression achieves a maximum over the range $1 \le q \le n-1$ at one of the endpoints.

This gives the bound $\max (q^2 + (n-q)^2)) \; 1 + (n-1)^2 = n^2 + 2(n-1)$.

Continuing with our bounding of T(n) we get

$$T(n) \le c[n^2 - 2(n-1)] + \theta(n)$$

$$= cn^2 - 2c(n-1) + \theta(n)$$

Since we can pick the constant so that the $2c(n-1)$ term dominates the $\theta(n)$ term we have

$$T(n) \le cn^2$$

Thus the worst-case running time of quick sort is $\theta(n^2)$.

(2) Average-case analysis :

If the split induced by RANDOMIZED_PARTITION puts constant fraction of elements on one side of the partition, then the recurrence tree has depth $\theta(\lg n)$ and $\theta(n)$ work is performed at $\theta(\lg n)$ of these level. This is an intuitive argument why the average-case running time of RANDOMIZED_QUICKSORT is $\theta(n \lg n)$.

The procedure RANDOMIZED_QUICKSORT is called with a 1 element array :

$$T(1) = \theta(1).$$

Where T(n) is the average time required to sort an array of n elements

The procedure RANDOMIZED_QUICKSORT calls itself to sort two subarrays. The average time to sort an array A[1 . . q] is T[q] and the average time to sort an array A[q+1 . . n] is T[n – q].

We have :

$$T(n) = 1/n \, (T(1) + T(n-1) \sum_{q=1}^{n-1} T(q) + T(n-q)) + \theta(n) \quad \ldots(2)$$

We know from worst-case analysis

$$T(1) = \theta(1) \text{ and } T(n-1) = O(n^2)$$

$$T(n) = 1/n \, (\theta(1) + O(n^2)) + 1/n \sum_{q=1}^{n-1} (r(q) + T(n-q)) + \theta(n)$$

$$= 1/n \sum_{q=1}^{n-1} (T(q) + T(n-q)) + \theta(n) \quad \ldots(3)$$

$$= 1/n \left[2 \sum_{q=1}^{n-1} (T(k)) \right] + \theta(n)$$

$$= 2/n \sum_{q=1}^{n-1} (T(k) + \theta(n)) \quad \ldots(4)$$

Solve the above recurrence using substitution method. Assume inductively that $T(n) \le a \, n\lg n + b$ for some constants $a > 0$ and $b > 0$

If we can pick 'a' and 'b' large enough so that $n \lg n + b > T(1)$. Then for $n > 1$, we have

$$T(n) \ge \sum_{q=1}^{n-1} 2/n \, (aklgk + b) + \theta(n)$$

$$= 2a/n \sum_{q=1}^{n-1} klgk - 1/8(n^2) + 2b/n \, (n-1) + \theta(n) \quad \ldots(5)$$

At this point we are claiming that

$$\sum_{q=1}^{n-1} klgk \leq 1/2\, n^2 lgn - 1/8(n^2)$$

Stick this claim in the Equation 5 above and we get

$$T(n) \leq 2a/n\,[1/2\, n^2\, lgn - 1/8(n^2)] + 2/n\, b(n-1) + \theta(n)$$
$$\leq anlgn - an/4 + 2b + \theta(n)$$

From Equation 6, we have $\theta(n) + b$ and $an/4$ as two polynomials. ...(6)

QUICKSORT's average running time is $\theta(n \lg(n))$.

Q. 5(a) Explain Strassen's Matrix multiplication and derived its time complexity. **(10 Marks)**

Ans. :

Strassen's Matrix Multiplication :

Suppose, to multiply two matrices of size N × N: for example A × B = C.

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

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

2 × 2 matrix multiplication can be accomplished in 8 multiplication. ($2^{\log_2^8} = 2^3$)

Algorithm

```
void matrix_mult ()
{
   for (i = 1; i <= N; i++)
   {
      for (j = 1; j <= N; j++)
      {
         compute Ci,j;
      }
   }
}
```


Time analysis

$$C_{ij} = \sum_{k-1}^{N} a_{i,k}\, b_{k,j}$$

$$\text{Thus } T(N) = \sum_{i=1}^{N} \sum_{j=1}^{N} \sum_{k=1}^{N} c = cN^3 = O(N^3)$$

Strassen showed that 2 × 2 matrix multiplication can be accomplished in 7 multiplication and 18 additions or subtractions. ($2^{\log_2^7} = 2^{2.807}$). This reduce can be done by Divide and Conquer Approach.

Divide and conquer matrix multiply

A × B = R

A0	A1	×	B0	B1	=	A0 × B0 + A1 × B2	A0 × B1 + A1 × B3
A2	A3		B2	B3		A2 × B0 + A3 × B2	A2 × B1 + A3 × B3

Algorithm : Strassen Algorithm

```
void matmul(int *A, int *B, int *R, int n)
{
 if (n == 1) then
 {
    (*R) += (*A) * (*B);
 }
else
{
    matmul(A, B, R, n/4);
    matmul(A, B+(n/4), R+(n/4), n/4);
    matmul(A+2*(n/4), B, R+2*(n/4), n/4);
    matmul(A+2*(n/4), B+(n/4), R+3*(n/4), n/4);
    matmul(A+(n/4), B+2*(n/4), R, n/4);
    matmul(A+(n/4), B+3*(n/4), R+(n/4), n/4);
    matmul(A+3*(n/4), B+2*(n/4), R+2*(n/4), n/4);
    matmul(A+3*(n/4), B+3*(n/4), R+3*(n/4), n/4);
 }
```

Time analysis :

$T(1) = 1$ (assume $N = 2^k$)

$T(N) = 7T(N/2)$

$T(N) = 7^k T(N/2^k) = 7^k$

$T(N) = 7^{\log N} = N^{\log 7} = N^{2.81}$

Chapter 3 : Greedy Method [Total Marks - 30]

Q. 2(b) Write down Prim's Algorithm and solve following problem : **(10 Marks)**

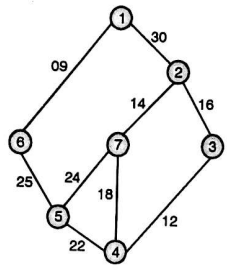


Fig. 2

Ans. :

Prim's Algorithm :

Prim's algorithm builds the tree edge by edge. The simplest way to choose next edge is an edge that results in a minimum increase in the sum of the costs of the edges added. If A is the set of edges selected so far, then A forms a tree. The next edge (u, v) to be added in A is a minimum-cost edge not in A with the property that $A \cup \{(u, v)\}$ is also a tree.

Algorithm of Prim's method :

(i) In this algorithm only a tree that includes only minimum-cost edge of G are included. And the edges are added one by one.

(ii) The next edge(i, j) that is to be added and in this i is a vertex already added and j is a vertex yet to be included, and the cost of (i, j) is minimum among all edges(k, *l*) such that vertex k is the tree and vertex *l* is not in the tree.

(iii) To determine the edge(i, j) efficiently, associate each vertex j not yet included in the tree a value near[j].

(iv) The value near[j] is a vertex in the tree such that cost[j, near[j]] is minimum among all choices for near[j].

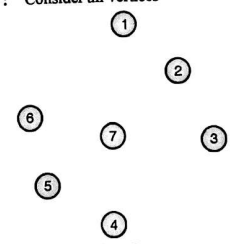
(v) Define near[j] = 0 for all vertices j that are already in the tree.

(vi) The next edge to include is defined to by the vertex j such that near[j] ≠ 0 (j not already in the tree) and cost[j, near [j]] is minimum.

Example :

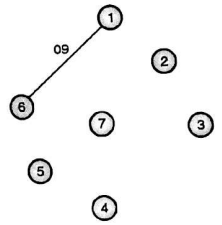
Consider all the vertices first and then select an edge with minimum weight. Select the adjacent edges with minimum weight. Avoid forming circuit.

Step 1 : Consider all vertices



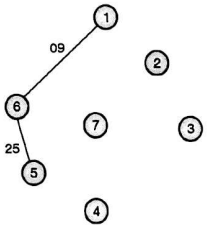
Total weight = 0

Step 2 : Select edge 1-6



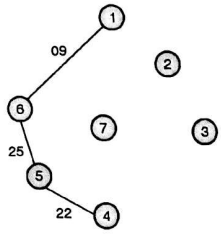
Total weight = 09

Step 3 : Edge 6-5 is selected.



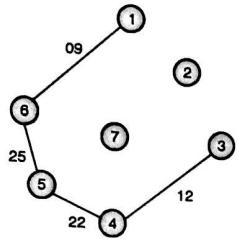
Total weight = 34

Step 4 : Edge 5-4 selected.



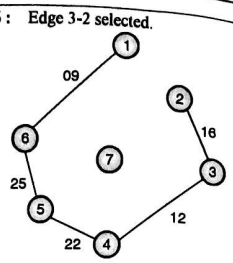
Total weight = 56

Step 5: Edge 4-3 is selected.



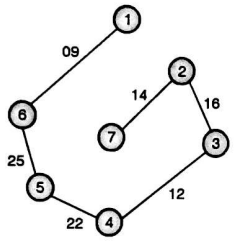
Total weight = 68

Step 6 : Edge 3-2 selected.



Total weight = 84

Step 7 : Edge 2-7 is selected.



Total weight = 98

Thus minimum cost of tree = 98, using Prim's Algorithm.

Q. 6(a) Write algorithm of Job Sequencing with Deadlines. Solve the following problem n = 5. $(P_1, P_2, P_3, P_4, P_5) = (20,15,10,5,1)$ and $(d_1, d_2, d_3, d_4, d_5) = (2,2,1,3,3)$. **(10 Marks)**

Ans. :

Algorithm :

```
1  Algorithm GreedyJob(d, J, n)
2  // J is a set of jobs that can be completed by their deadlines.
3  {
4  J := {1};
```

easy solution


```
5  for i := 2 to n do
6  {
7  if (all jobs in J ∪ {i} can be completed
8  by their deadlines) then J := J ∪ {i};
9  }
10 }
```

Let J be a set of k jobs and $\sigma = i_1, i_2, \ldots, i_k$ a permutation of jobs in J such that $d_{i1} \le d_{i2} \ldots \le d_{ik}$. Then J is feasible solution if and only if the jobs in J can be processed in the order σ without violating any deadline. This theorem determine whether all jobs in J U{i} can be completed by their deadlines or not. Use an array d[1 : n] to store the deadlines of the jobs in the order of their p-values. The set J can be represented as J[1 : k] such that J[r], $1 \le r \le k$ are jobs in J and $d[J[1]] \le d[J[2]] \le \ldots \le d[J][k]]$

To test whether J U{i} is feasible we have to just insert i into J preserving the deadline ordering and then verify that $d[[r]] \le r$, $1 \le r \le k + 1$. The insertion of i into J is simplified by the use of fictitious job 0 with d[0] = 0 and J[0] = 0. If job i is to be inserted at position 9, then only the positions of jobs J[9], J[9 + 1], ...J[k] are changed after the insertion. Hence it is necessary to verify only that these jobs do not violate their deadlines following the insertion.

Example :

Let n = 5

$p_i = (20, 15, 10, 5, 1)$

$d_i = (2, 2, 1, 3, 3)$

we have

n	p_i	d_i
1	20	2
2	15	2
3	10	1
4	5	3
5	1	3

We need one unit of time to process each job and we can do at most one job each time. We can earn the profit p_i, if job i is completed by its deadline d_i. If not so, no profit will be gained.

The optimal solution = {1, 2, 4}

The total profit = 20 + 15 + 5 = 40

Time complexity = $O(n^2)$

easy solution

Q. 7(a) Difference between Prim's Algorithm and Kruskal's Algorithm. **(5 Marks)**

Ans. :

Sr. No.	Prim's algorithm	Kruskal's algorithm
1.	Prim's algorithm builds the tree edge by edge.	While Kruskal's algorithm is started with the empty graph.
2.	The simplest way to choose next edge is an edge that results in a minimum increase in the sum of the costs of the edges added.	It then selects an edge that have a minimum weight and that doesn't produce a cycle.
3.	If A is the set of edges selected so far, then A forms a tree. The next edge (u, v) to be added in A is a minimum-cost edge not in A with the property that $A \cup \{(u, v)\}$ is also a tree.	Hence the tree is constructed edge by edge and cycles are avoided and always the cheapest edge is selected.

Q. 7(e) Write short note on : Optimal Storage on Tapes. **(5 Marks)**

Ans. :

Optimal Storage on Tapes :

These are n programs that are to be stored on a computer tape of length l. For each program i, l_i is length, $1 \le i \le n$. All programs can be stored on the tape if and only if the sum of the lengths of programs is at most l. Whenever a program is to be retrieved from this tape, the tape is initially positioned at the front. If the programs are stored in the order $I = i_1, i_2, \ldots, i_n$ the time t_j is needed to retrieve program i_j is proportional to $\sum_{1 \le k \le j} l_{ik}$.

If all programs retrieved equally often, then expected or mean retrieval time (MRT) is $(1/n) \sum_{1 \le j \le n} t_j$

Here we are required to find a permutation for the n programs so that when they are stored on the tape in this order the MRT is minimized. Minimizing the MRT is equivalent to minimizing.

$$d(I) = \sum_{1 \le j \le n} \sum_{1 \le k \le j} l_{ik}$$

For example, let $n = 3$ and $(l_1, l_2, l_3) = (5, 10, 3)$. There are $n ! = 6$ possible orderings. These ordering and their respective d values are

Ordering I	d (I)
1, 2, 3	5 + 5 +10 + 5 + 10 + 3 = 38
1, 3, 2	5 + 5 + 3 + 5 + 3 + 10 = 31
2, 1, 3	10 + 10 + 5 + 10 + 5 + 3 = 43
2, 3, 1	10 + 10 + 3 + 10 + 3 + 5 = 41
3, 1, 2	3 + 3 + 5 + 3 + 5 + 10 = 29
3, 2, 1	3 + 3 + 10 + 3 + 10 + 5 = 34

The optimal ordering is 3, 1, 2.

A greedy approach would choose next program on the basis of some optimisation measure. That measure would be the d value of the permutation. The next program to be stored on the tape would be one that minimizes the increase in d.

If we have already constructed the permutation $i_1, i_2, \ldots, i_r$, the appending program j gives the permutation $i_1, i_2, \ldots, i_r, i_{r+1} = j$. This increase the d value by $\sum_{1 \le k \le r} l_{ik} + l_j$

The greedy method simply require us to store the programs in non-decreasing order of their lengths .This ordering can be carried out in O(n logn) time using an efficient sorting algorithm (For example, – heap sort)

Algorithm :

```
1  Algorithm Store (n,m)
2  // n is number of programs and m is number of tapes
3  {
4  j = 0 ;
5  for i = 1 to n do
6          {
7             write ( " append program", i;
8                 " to permutation for tape ", j );
9             j = ( j + 1 ) mod m;
           }
10    }
```

Chapter 4 : Dynamic Programming [Total Marks - 25]

Q. 1(b) Construct the Optimal Binary Search Tree for the identifier set (a_1, a_2, a_3, a_4) = (cout, float, if, while).

With $P(1:4) = \left(\frac{1}{20}, \frac{1}{5}, \frac{1}{10}, \frac{1}{20}\right)$ and $q(0:4) = \left(\frac{1}{5}, \frac{1}{10}, \frac{1}{5}, \frac{1}{20}, \frac{1}{20}\right)$ **(10 Marks)**

Ans. :

$w(i, i) = q(i), c(i, i) = 0, r(i, i) = 0$

Let multiply p and q by 20 for convenience p (1 : 4) = {1, 4, 2, 1}

$q(0:4) = \{4, 2, 4, 1, 1\}$

$w(i, i) = p(j) + q(j) + w(i, j-1)$

$w(0, 1) = p(1) + q(1) + w(0, 0) = 1 + 2 + 4 = 7$

$r(0, 1) = w(0, 1) + \min\{e(0, 0) + c(1, 1)\} = 7$

$r(0, 1) = 1$

$p(1:4) = \{1, 4, 2, 1\}$

$q(1:4) = \{4, 2, 4, 1, 1\}$

$w(1, 2) = p(2) + q(2) + w(1, 1) = 4 + 4 + 2 = 10$

$c(1, 2) = w(1, 2) + \min\{c(1, 1) + c(2, 2)\} = 10$

$r(1, 2) = 2$

$w(2, 3) = p(3) + q(3) + w(2, 2) = 2 + 1 + 4 = 7$

$c(2, 3) = w(2, 3) + \min\{c(2, 2) + c(3, 3)\} = 7$

$r(2, 3) = 3$

$r(3, 4) = p(4) + q(4) + w(3, 3) = 1 + 1 + 1 = 3$

$c(3, 4) = w(3, 4) + \min\{c(3,3) + c(4,4)\} = 3$

$r(3,4) = 4$

$w_{02} = p(2) + q(2) + w(0,1) = 4 + 4 + 7 = 15$

$c_{02} = w_{02} + \min\{c(0,0) + c(1,2), c(0,1) + c(2,2)\}$

$= 15 + \min\{10, 7 + 0\} = 22$

$r_{02} = 2$

$w_{13} = p(3) + q(3) + w(1,2) = 2 + 1 + 10 = 13$

$c_{13} = w_{13} + \min\{c(1,1) + c(2,3) + c(1,2) + c(3,3)\}$

$= 13 + \min\{7, 10\} = 20$

$r_{13} = 2$

$w_{24} = p(4) + q(4) + w(2,3) = 1 + 1 + 7 = 9$

$c_{24} = 9 + \min\{c(2,2) + c(3,4) + c(2,3) + c(4,4)\} = 9 + \{3, 7\} = 12$

$r_{24} = 3$

$w_{03} = p(3) + q(3) + w(0,2) = 2 + 1 + 15 = 18$

$c_{03} = w_{03} + \min\{c(0,0) + c(1,3), c(0,1) + c(2,3) + c(0,2) + c(3,3)\}$

$= 18 + \min\{20, 7 + 7, 22 + 0\}$

$c_{03} = 18 + 14 = 32$

$r_{03} = 2$

$w_{14} = p(4) + q(4) + w(1,3) = 1 + 1 + 13 = 15$

$c_{14} = w_{14} + \min\{c(1,1) + c(2,4), c(1,2) + c(3,4), c(1,3) + c(4,4)\}$

$= 15 + \min\{12, 10 + 3, 20 + 0\} = 27$

$r_{14} = 2$

$w_{04} = p(4) + q(4) + w(0,3) = 1 + 1 + 18 = 20$

$c_{04} = 20 + \min\{c(0,0) + c(1,4), c(0,1) + c(2,4), c(0,2) + c(3,4), c(0,3) + c(4,4)\}$

$c_{04} = 20 + \min\{27, 7 + 12, 22 + 3, 32\} = 20 + 19 = 39$

$r_{04} = 2$

$c_{04} = 39$ is minimum cost of binary search tree for (a_1, a_2, a_3, a_4)

	0	1	2	3	4
0	$w_{00} = 4$ $c_{00} = 0$ $r_{00} = 0$	$w_{11} = 2$ $c_{11} = 0$ $r_{11} = 0$	$w_{22} = 4$ $c_{22} = 0$ $r_{44} = 0$	$w_{33} = 1$ $c_{33} = 0$ $r_{33} = 0$	$w_{44} = 1$ $c_{44} = 0$ $r_{44} = 0$
1	$w_{01} = 7$ $c_{01} = 7$ $r_{01} = 1$	$w_{12} = 10$ $c_{12} = 10$ $r_{12} = 2$	$w_{23} = 7$ $c_{23} = 7$ $r_{23} = 3$	$w_{34} = 3$ $c_{34} = 3$ $r_{34} = 4$	
2	$w_{02} = 15$ $c_{02} = 22$ $r_{02} = 2$	$w_{13} = 13$ $c_{13} = 20$ $r_{13} = 2$	$w_{24} = 9$ $c_{24} = 12$ $r_{24} = 3$		
3	$w_{03} = 18$ $c_{03} = 32$ $r_{03} = 2$	$w_{14} = 15$ $c_{14} = 27$ $r_{14} = 2$			
4	$w_{04} = 20$ $c_{04} = 39$ $r_{04} = 2$				

Root of tree is t_{04} is a_2 i.e. float. Hence the left sub tree is t_{01} and right sub tree t_{24}. t_{01} has root a_1 and sub trees t_{00} and t_{11}. Tree t_{24} has root 3. Tree t_{24} has left sub tree t_{22} and right subtree t_{34}. t_{34} has left sub tree t_{33} and right sub tree t_{44} Final tree is

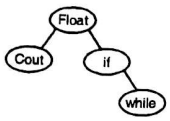


Fig. 3

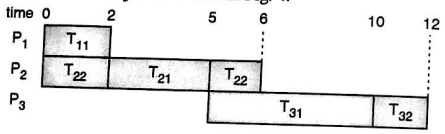
Q. 2(a) Explain Flow Shop Scheduling with help of suitable examples. **(10 Marks)**

Ans. : Flow Shop Scheduling :

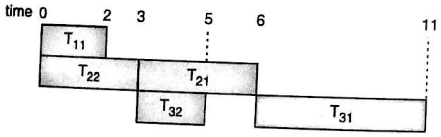
There are use n jobs each requiring m tasks $T_{1i}, T_{2i}, \ldots, T_{mi}$, $1 \le i \le n$ to be performed. Task T_{ji} is to be performed on processes P_j, $1 \le j \le m$. The time required to complete task T_{ji} is t_{ji}. No processor may have more than one task assigned to it in any time interval. Processing of T_{ji}, $j > 1$ cannot be started until task $T_j - 1$, i has been completed. For example, two jobs have to be scheduled on three processors. The task times are given by the matrix J.

$$J = \begin{bmatrix} 2 & 0 \\ 3 & 3 \\ 5 & 2 \end{bmatrix}$$

Two possible schedules for the job are shown in Fig. 4.



(a) Preemptive schedule



(b) Non-preemptive schedule

Fig. 4

Preemptive Schedule :

In preemptive schedule the processor is terminated before completing the task.

Non-preemptive Schedule :

Processor is not terminated until the task is complete. The finish time (f_i (s)) of job i is the line at which all tasks of job i have been completed.

In Fig. 4(a) f_1 (S) = 10, f_2 (S) = 12

In Fig. 4(b) f_1 (S) = 11, f_2 (S) = 5

Finish time F(S) of a schedule S is given by

$$F(S) = \max_{1 \le i \le n} \{ f_i(S) \}$$

Mean flow line MFT (S) is defined to be

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

An optimal finish time (OFT) schedule for a given set of jobs is a non-preemptive schedule for which F (S) is minimum over all non-preemptive schedule S. Use a_i to represent t_{1i}, and b_i to represent t_{2i}. It is easy to see that an optimal permutation, (schedule) has the properly that given the first job in the permutation, the remaining permutation is optimal with respect to the state the two processors are in the following the completion of the first job.

Let $\sigma_1, \sigma_2, \ldots \sigma_k$ be a permutation prefix defining a schedule for jobs $T_1, T_2, \ldots T_k$. For this schedule let f_1 and f_2 be the times at which the processing of jobs $T_1, T_2 \ldots, T_k$ is completed on processor P_1 and P_2 respectively. Let g (S, t) be the length of an optimal schedule for the subset of jobs S. Let us assume that processor 2 is not available until line t.

The length of an optimal schedule for the job set {1, 2...., n} is g ({1, 2, ..., n), 0} since the principle of optimality holds, we obtain

$$g(\{1, 2, \ldots, n), 0\} = \min_{1 \le i \le n} \{a_i + g(\{1, 2, \ldots, n)\} - \{i\}, b_i)\}$$

$$g(\phi, t) = \max\{t, 0\} \text{ and that } a_i \ne 0, 1 \le i \le n$$

$$g(S, t) = \min_{i \in S} \{a_i + g(S - \{i\}, b_i + \max\{t - a_i, 0\})\}$$

Consider any schedule R for a subset of jobs S. Assume that P_2 is not available until line t. Let i and j be the first two jobs in this schedule.

Then from above equation we obtain :

$$g(S, t) = a_i + g(S - \{i\}, b_i + \max\{t - a_i, 0\})$$
$$g(S, t) = a_i + a_j + g(S - \{i, j\}, b_j + \max\{b_i + \max\{t - a_i, 0\} - a_j, 0\})$$

Above equation can be simplified using following result

$$t_{ij} = b_j + \max\{b_i + \max\{t - a_i, 0\} - a_j, 0\}$$
$$= b_j + b_i - a_j + \max\{\max\{t - a_i, 0\}, a_j - b_i\}$$
$$= b_j + b_i - a_j + \max\{t - a_i, a_j - b_i, 0\}$$
$$t_{ij} = b_j + b_i - a_j - a_i + \max\{t, a_i + a_j - b_i, a_i\}$$

If jobs i and j are interchanged in R, then the finish time g' (S, t) is

$$g'(S, t) = a_i + a_j + g(S - \{i, j\}, t_{ji})$$

where $t_{ji} = b_j + b_i - a_j - a_j + \max\{t, a_i + a_j - b_j, a_i\}$

Comparing g (S, t) and g' (S, t) then $g(S, t) \le g'(S, t)$

$\max\{t, a_i + a_j - b_i, a_i\} \le \max\{t, a_i + a_j - b_j, a_j\}$

In order to hold for all values of t_i we need.

$\max\{a_i + a_j - b_i, a_i\} \le \max\{a_i + a_j - b_j, a_j\}$

or $a_i + a_j + \max\{-b_i, -a_j\} \le a_i + a_j + \max\{-b_j, -a_i\}$

or $\min\{b_i, a_j\} \ge \min\{b_j, a_i\}$ ...(I)

From above equation, there exists an optimal schedule in which for every pair (i, j) of adjacent jobs, $\min\{b_i, a_j\} \ge \min\{b_j, a_i\}$

The scheduling rule resulting from (I) is therefore :

Sort all the a_i's and b_j's into non-decreasing order. Consider this sequence in this order. If the next numbers in the sequence is a_j and job j has not yet been scheduled, schedule job j at the leftmost available spot. If the next number is b_j and job j has not yet been scheduled, schedule job j at the rightmost available spot. If job j has already been scheduled, go to next number in the sequence. e.g. Let n = 4 $(a_1, a_2, a_3, a_4) = (3, 4, 8, 10)$ and $(b_1, b_2, b_3, b_4) = (6, 2, 9, 15)$. The sorted sequence of a's and b's is $(b_2, a_1, a_2, b_1, a_3, b_3, a_4, b_4) = (2, 3, 4, 6, 8, 9, 10, 15)$. Let σ_1, σ_2, σ_3 and σ_4 be the optimal schedule. Since the smallest number is b_2, we set $\sigma_4 = 2$.

The next number is a_1 and we set $\sigma_1 = a_1$. The next smallest number is a_2. Job 2 has already been scheduled. The next number of b_1. Job 1 has already been scheduled. The next is a_3 and we set σ_3. This leaves σ_3 free and job 4 unscheduled. Thus $\sigma_3 = 4$.

Q. 7(d) Write short note on : Advantages of dynamic programming. **(5 Marks)**

Ans. :

Advantages of Dynamic Programming :

1. Dynamic programming is an algorithm design method that can be used when the solution to a problem can be viewed as the result of a sequence of decisions.
2. One way to solve problems for which it is not possible to make a sequence of stepwise decisions leading to an optimal decision sequence is to try all possible decision sequences. We could enumerate all decision sequences and then pick out the best. But the time and space requirements may be prohibitive.
3. Dynamic programming often drastically reduces the amount of enumeration by avoiding the enumeration of some decision sequences that cannot possibly be optimal. In dynamic programming an optimal sequence of decisions is obtained by making explicit appeal to the principle of optimality.
4. Dynamic programming is applicable to the problem, when the sub-problems are not independent. Thus, a dynamic programming algorithm solves every sub-problem just once and then saves its answer in a table, thus avoiding the work of re-computing the answer every time the sub-problem is encountered.

Comparison between Dynamic Programming and Divide and Conquer :

1. Divide and Conquer method and dynamic programming for problem, solve the problem by combining the solutions to sub-problems. Divide and Conquer method partitions the problem into independent sub-problems, solves these problems recursively and then combine these solutions to obtain the solution for the original problem.
2. However the dynamic programming is applicable when the sub-problems are not independent. This shows that, divide and conquer method does more work than necessary.

Chapter 5 : Backtracking [Total Marks - 25]

Q. 4(b) Write algorithm of Sum of Subsets. Solve following problem and draw portion of state space tree. w = {5, 7, 10, 12, 15, 18, 20} and m = 35.
Find all possible subsets of w that sum to m. **(10 Marks)**

Ans. :

Algorithm of Sum of Subsets :

```
1. Algorithm samofsub (s, k, r)
2. {
3.     //generate left child,
4. if (s + w [k] = m) then write (x(1 : k)); //subset found
```

```
5.  else if (s + w [k] + w [k + 1] ≤ m)
6.  then sumofsub (s + w [k], k + 1, r – w[k]);
7.  //generate right child and evaluate Bk
8.  If ((s + r – w[k] ≥ m) and (s + w [k + 1)] ≤ m)) then
9.  {
10.     x[k] = 0;
11.     sumofsub (s, k + 1, r – w [k]);
12.}
13.}
```

Example :

The values of s, k and r are listed by the rectangular nodes on each of the calls to sumofsub. Circular nodes represent points at which subsets with sums m are printed out.

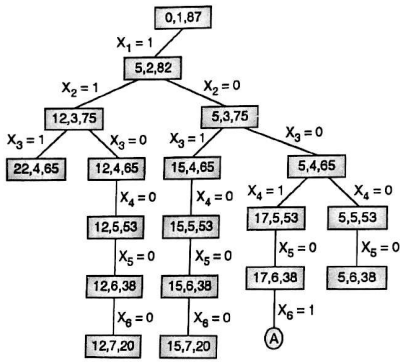


Fig. 5 : Left subtree of tree

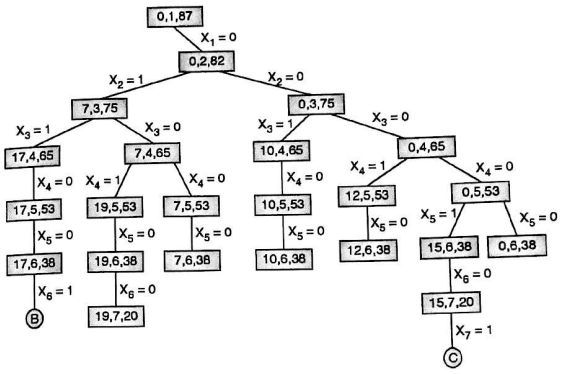


Fig. 5(a)

At nodes A, B and C the output is respectively.
A (1, 0, 0, 1, 0, 1) , B (0, 1, 1, 0, 0, 1), C (0, 0, 0, 0, 1, 0, 1)

Q. 6(b) State the applications of the graph theory. **(10 Marks)**

Ans. :

Applications of Graph Theory :

Graphs are among the most ubiquitous models of both natural and human-made structures. They can be used to model many types of relations and process dynamics in physical, biological and social systems. Many problems of practical interest can be represented by graphs. In computer science, graphs are used to represent networks of communication, data organization, computational devices, the flow of computation, etc. One practical example: The link structure of a website could be represented by a directed graph. The vertices are the web pages available at the website and a directed edge from page *A* to page *B* exists if and only if *A* contains a link to *B*. A similar approach can be taken to problems in travel, biology, computer chip design, and many other fields.

The development of algorithms to handle graphs is therefore of major interest in computer science. There, the transformation of graphs is often formalized and represented by graph rewrite systems. They are either directly used or properties of the rewrite systems. Complementary to graph transformation systems focusing on rule-based in-memory manipulation of graphs are graph databases geared towards transaction-safe, persistent storing and querying of graph-structured data.

Graph-theoretic methods, in various forms, have proven particularly useful in linguistics, since natural language often lends itself well to discrete structure. Traditionally, syntax and compositional semantics follow tree-based structures, whose expressive power lies in the Principle of Compositionality, modeled in a hierarchical graph. Within lexical semantics, especially as applied to computers, modeling word meaning is easier when a given word is understood in terms of related words; semantic networks are therefore important in computational linguistics.

Still other methods in phonology (e.g. Optimality Theory, which uses lattice graphs) and morphology (e.g. finite-state morphology, using finite-state transducers) are common in the analysis of language as a graph. Indeed, the usefulness of this area of mathematics to linguistics has borne organizations such as Text Graphs, as well as various 'Net' projects, such as WordNet, VerbNet, and others.

Graph theory is also used to study molecules in chemistry and physics. In condensed matter physics, the three dimensional structure of complicated simulated atomic structures can be studied quantitatively by gathering statistics on graph-theoretic properties related to the topology of the atoms. For example, Franzblau's shortest-path (SP) rings. In chemistry a graph makes a natural model for a molecule, where vertices represent atoms and edges bonds.

This approach is especially used in computer processing of molecular structures, ranging from chemical editors to database searching. In statistical physics, graphs can represent local connections between interacting parts of a system, as well as the dynamics of a physical process on such systems. Graph theory is also widely used in sociology as a way, for example, to measure actors' prestige or to explore diffusion mechanisms, notably through the use of social network analysis software.

Likewise, graph theory is useful in biology and conservation efforts where a vertex can represent regions where certain species exist (or habitats) and the edges represent migration paths, or movement between the regions. This information is important when looking at breeding patterns or tracking the spread of disease, parasites or how changes to the movement can affect other species.

In mathematics, graphs are useful in geometry and certain parts of topology, e.g. Knot Theory. Algebraic graph theory has close links with group theory. A graph structure can be extended by assigning a weight to each edge of the graph. Graphs with weights, or weighted graphs, are used to represent structures in which pair wise connections have some numerical values. For example if a graph represents a road network, the weights could represent the length of each road.

A digraph with weighted edges in the context of graph theory is called a network. Network analysis has many practical applications, for example, to model and analyze traffic networks. Applications of network analysis split broadly into three categories :

(1) First, analysis to determine structural properties of a network, such as the distribution of vertex degrees and the diameter of the graph. A vast number of graph measures exist, and the production of useful ones for various domains remains an active area of research.

(2) Second, analysis to find a measurable quantity within the network, for example, for a transportation network, the level of vehicular flow within any portion of it.

(3) Third, analysis of dynamical properties of networks.

Q. 7(c) Write short note on : N-Queens Problem. **(5 Marks)**

Ans. :

N-Queens Problem :

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, following is a solution for 4 Queen problem.

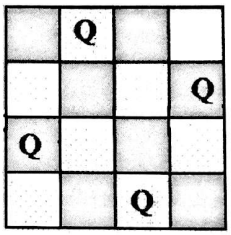


Fig. 6

The expected output is a binary matrix which has 1s for the blocks where queens are placed. For example following is the output matrix for above 4 queen solution.

{ 0, 1, 0, 0} { 0, 0, 0, 1}
{ 1, 0, 0, 0} { 0, 0, 1, 0}

Algorithm : All solutions to the n-queens problem

```
1.  Algorithm NQueens(k, n)
2.  // Using backtracking, this procedure prints all possible placements of n queens on
3.  // an n x n chessboard so that they are nonattacking.
4.  {
5.      for i := 1 to n do
6.     {
7.
8.       if Place(k,i) then
9.        {
10.       x[k] := i;
11.       if (k = n) then write (x[l : n]);
12.       else NQueens(k + l, n);
```

```
13.  }
14.  }
15.  }
```

Chapter 6 : String Matching Algorithms [Total Marks - 10]

Q. 5(b) Write down Knuth-Morris-Pratt Algorithm. **(10 Marks)**

Ans. :

The Knuth-Morris-Pratt Algorithm :

Knuth, Morris and Pratt discovered first linear time string-matching algorithm by following a tight analysis of the naive algorithm. Knuth-Morris-Pratt algorithm keeps the information that naive approach wasted gathered during the scan of the text.

By avoiding this waste of information, it achieves a running time of O(n + m), which is optimal in the worst case sense. That is, in the worst case Knuth-Morris-Pratt algorithm have to examine all the characters in the text and pattern at least once.

The failure function :

The KMP algorithm preprocesses the pattern P by computing a failure function f. The failure function indicates the largest possible shift *s* using earlier comparisons. Specifically, the failure function f(j) is defined as the length of the longest prefix of P that is a suffix of P[i . . j].

Knuth-Morris-Pratt Failure (P) :

```
Input  :  Pattern with m characters
Output :  Failure function f for P[i . . j]
          i ← 1
          j ← 0
          f(0) ← 0
          while i < m do
          if P[j] = P[i]
                f(i) ← j + 1
             i ← i + 1
             j← j + 1
          else if
             j ← f(j - 1)
          else
             f(i) ← 0
             i ← i + 1
```


Note that the failure function f for P, which maps j to the length of the longest prefix of P that is a suffix of P[1 . . j], encodes repeated substrings inside the pattern itself.

As an example, consider the pattern P = a b a c a b. The failure function, f(j), using above algorithm is :

j	a	1	2
P[i]	a	b	a
	c	a	b
f(j)	0	0	1
	0	1	2

By observing the above mapping we can see that the longest prefix of pattern, P, is "a b" which is also a suffix of pattern P.

Consider an attempt to match at position i, that is when the pattern P[0 . . m – 1] is aligned with text P[i . . i + m – 1].

T: a b a c a a b a c c
P: a b a c a b

Assume that the first mismatch occurs between characters T[i + j] and P[j] for 0 < j < m. In the above example, the first mismatch is T[5] = a and P[5] = b.

Then, T[i . . i + j – 1] = P[0 . . j – 1] = u

That is, T[0 . . 4] = P[0 . . 4] = u, in the example [u = a b a c a] and

T[i + j] ≠ P[j] i.e., T[5] ≠ P[5], In the example [T[5] = a ≠ b = P[5]].

When shifting, it is reasonable to expect that a prefix v of the pattern matches some suffix of the portion u of the text. In our example, u = a b a c a and v = a b a c a, therefore, 'a' a prefix of v matches with 'a' a suffix of u.

Let l(j) be the length of the longest string P[0 . . j -1] of pattern that matches with text followed by a character c different from P[j]. Then after a shift, the comparisons can resume between characters T[i + j] and P[l(j)], i.e., T(5) and P(1)

T: a b a c a a b a c c
P: a b a c a b

Note that no comparison between T[4] and P[1] needed here.

Knuth-Morris-Pratt (T, P)

```
Input  :  Strings T[0 . . n] and P[0 . . m]
Output :  Starting index of substring of T matching P
f ← compute failure function of Pattern P
i ← 0
j ← 0
```

```
while i < length[T] do
   if j ← m - 1 then
      return i - m + 1    // we have a match
i ← i +1
j ← j +1
else if j > 0
      j ← f(j - 1)
   else
      i ← i +1
```

Analysis :

The running time of Knuth-Morris-Pratt algorithm is proportional to the time needed to read the characters in text and pattern. In other words, the worst-case running time of the algorithm is O(m + n) and it requires O(m) extra space.

Chapter 7 : Branch and Bound [Total Marks - 20]

Q. 3(b) Explain 0/1 Knapsack problem using Branch and Bound Method. **(10 Marks)**

Ans. :

0/1 Knapsack Problem using Branch and Bound method :

To use Branch and Bound technique for 0/1 knapsack problem we replace the objective function $\Sigma p_i x_i$ by the function $-\Sigma p_i x_i$. Clearly $\Sigma p_i x_i$ is maximized if $-\Sigma p_i x_i$ is minimized. This modified knapsack problem is stated as

$$\text{Minimize} - \sum_{i=1}^{n} p_i x_i$$

$$\text{subject to} \sum_{i=1}^{n} w_i x_i \le m$$

$$x_i = 0 \text{ or } 1,\ 1 \le i \le n$$

Assume a fixed tuple size formulation for the solution space. This can be extended to a variable tuple size formulation. Every leaf node in the state space tree representing an assignment for which $\sum_{1 \le i \le n} w_i x_i \le m$ is an answer node.

easy solution

All other leaf nodes are infeasible. For a minimum-cost answer node to correspond to any optimal solution. We need to define $c(x) = -\sum_{1 \le i \le n} p_i x_i \le m$ for every answer node x. The cost $c(x) = \infty$ for infeasible leaf nodes. For nonleaf nodes c(x) is recursively defined to be min {c(lchild(x)), c (rchild))}. We now need two functions $\hat{c}(x)$ and u(x) such that $\hat{c}(x) \le c(x) \le u(x)$ for every node x. The cost $\hat{c}(\bullet)$ and $u(\bullet)$ satisfying this requirement may be obtained as follows.

Let x be a node at level j, $1 \le j \le n+1$. At node x assignments have already been made to xi., $1 \le i < j$. The cost of these assignments is $-\sum_{1 \le i < j} p_i x_i$.

So, $c(x) \le -\sum_{1 \le i \le j} p_i x_i$ and we may use $u(x) = -\sum_{1 \le i < j} p_i x_i$. If $q = -\sum_{1 \le i < j} p_i x_i$ then an improved upper bound function u(x) is $u(x) = \text{Ubound}(q, \sum_{1 \le i < j} w_i x_i, j\text{-}1, m)$

Algorithm : Function u(·) for Knapsack Problem :

```
Algorithm UBound(cp, cw, k, m)
//w[i] and p[i] are respectively
// the weight and profit of the ith object.
{
   b := cp; c := cw,
   for i := k + 1 to n do
   {
   if (c + w[i] < m) then
   {
         c := c + w[i];
         b := b — p[i];
      }
}
return b;
}
```

Q. 4(a) Describe Traveling Salesperson Problem. Explain how to solve using Branch and Bound Method. **(10 Marks)**

easy solution

Ans. :

Travelling Salesperson Problem :

Let G be the complete graph on five points with the following distance matrix :

0	14	4	10	20
14	0	7	8	7
4	5	0	7	16
11	7	9	0	2
18	7	17	4	0

Determining the shortest tour starting from node 1 that traverses exactly once through each other node before returning to node 1. The nodes in the implicit graph correspond to partially specified paths. For example, node (1,4,3) corresponds to two complete tours as follow :

(1,4,3,2,5,1) and (1,4,3,5,2,1)

The successors of a given node correspond to paths in which one additional node has been specified. At each node calculate a lower bound on the length of the corresponding complete tours. To calculate lower bound, half the distance between two points i and j is counted at the time when we leave i, and the remaining half when we arrive at j. To obtain a bound on the length of a path, it is sufficient to add elements of this kind. For instance, a complete tour must include a departure from node 1, a visit to each of the nodes 2,3,4, and 5 (not necessarily in this order) and a return to 1. Its length is therefore at least

$$2 + 6 + 4 + 3 + 3 + 2 = 20$$

This calculation does not imply the existence of a solution that costs only 20.

The starting point for our tour as node 1 and this arbitrary choice will not affect the length of the shortest tour. The search begins by generating the four possible successors of the root, namely, nodes (1,2), (1,3), (1,4) and (1,5). The bound for node (1,2), for example, is calculated as follows.

A tour that begins with (1,2) must include :

(i) The trip 1– 2 : 14 (formally , leaving 1 for 2 and arriving at 2 from 1: 7 + 7)

(ii) A departure from 2 toward 3,4, or 5: minimum 7/2

(iii) A visit to 3 that neither comes from 1 nor leaves for 2: minimum 11/2

(iv) A similar visit to 4: minimum 3

(v) A similar visit to 5: minimum 3

(vi) A return to 1 from 3,4, or 5: minimum 2.

(vi) Therefore the length of this tour is at least 31. The other bounds are calculated similarly.

Next, the most promising node seems to be (1,3), whose bound is 24. The three children (1,2), (1,3,4), and (1,3,5) of this node are therefore generated. Calculate the bound for node (1,3,2) as follows

(i) The trip 1-3-2: 9

(ii) A departure from 2 toward 4 or 5: minimum 7/2

(iii) A visit to 4 that comes from neither 1 nor 3 and that leaves for neither 2 nor 3: minimum 3

(iv) A similar visit to 5: minimum 3

(v) A return to 1 from 4 or 5: minimum 11/2, which gives a total length of at least 24.

The most promising node is now (1,3,2). Its two children (1,3,2,4) and (1,3,2,5) are generated. This time, as node (1,3,2,4) for instance, corresponds to exactly one complete tour (1,3,2,4,5,1) and its length is calculated as 37. The length of the tour (1,3,2,5,4,1) is 31. Since to determine one optimal solution, we do not explore the nodes (1,2), (1,5) and (1,3,5) that cannot give a better solution. Even the node (1,3,4) is pointless.

There remains only node (1,4) to explore. The only child to offer interesting possibilities is (1,4,5). After looping at the two complete tours (1,4,5,2,3,1) and (1,4,5,3,2,1) find that the tour (1,4,5,2,3,1) of length 30 is optimal. This example demonstrates that although at one point (1,3) was the most promising node, the optimal solution does not come from there.

It is very difficult to program branch-and-bound since there is a need to keep a list of nodes that have been generated but not yet completely explored at all levels of the tree. Using the heap in this situation is a good solution to hold this list. Unlike depth-first search and its related techniques, no elegant recursive formulation of branch-an-bound is available to the programmer. In the worst case it may turn out that even an excellent bound does not allow us to cut any branches off the tree, and all the extra work done is wasted.

Let G = (V, E) be a directed graph used for the travelling salesperson problem. Let C_{ij} equal the cost of edge $<i, j>$, $C_{ij} = \infty$ if $<i, j> \notin E$, and let $|V| = n$.

We assume that every tour starts and ends at vertex 1. So, the solution space S is given by $S = \{1, \pi, 1/\pi)$ is a permutation of $(2, 3, \ldots, n)\}$. Then $|S| = (n - 1)!$. The size of S can be reduced by restricting S so that $(1, i_1, i_2, \ldots, i_{n-1}) \in S$ iff $<i_j, i_{j+1}> \in E$, $0 \le j \le n - 1$ and $i_o = i_n = 1$.

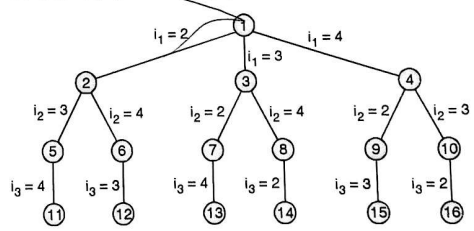


Fig. 7 : State space tree for the travelling salesperson problem with n = 4 and $i_0 = i_4 = 1$

S can be organized into state space tree similar to that of n-queens problem. Following Fig. 7 shows the tree organization for the case of a complete graph with |V| = 4. Each leaf node L is a solution node and represents the tour $i_0 = 1$, $i_1 = 3$, $i_2 = 4$, $i_3 = 2$ and $i_4 = 1$.

To use LCBB to search the travelling salesperson state space tree, we need to define a cost function c (·) and two other functions $\hat{c}$ (·) and u (·) such that $\hat{c}$ (r) ≤ c (r) ≤ u(r) for all nodes r. The cost c (·) is such that the solution node with least c (·) corresponds to the shortest four in G. One choice for c (·) is

$$c(A) = \begin{cases} \text{length of tour defined by the path from the root to A , if A is a leaf} \\ \text{cost of a minimum-cost leaf in the subtree A , if A is not a leaf} \end{cases}$$

A simple function $\hat{c}$ (·) such that $\hat{c}$ (A) ≤ c (A) for all A is obtained by defining $\hat{c}$ (A) to be the length of the path defined at node A.

□□□

Dec. 2012

Chapter 1 : Introduction to Analysis of Algorithm [Total Marks - 05]

Q. 1(a) Arrange the following functions in increasing order **(5 Marks)**
n, logn, n^3, n^2, nlogn, 2^n, n !

Ans. :

Typical growth rate :

log n	Logarithm
n	Linear
n log n	Linear f logarithmic
n^2	Quadratic
n^3	Cubic
2^n	Exponential
n!	Factorial

Increasing order : In analysis of algorithm, we always assume larger values of input.

Here for larger value of n, we have following order : logn, n, nlogn, n^2, n^3, 2^n, n!

Chapter 2 : Divide and Conquer [Total Marks - 30]

Q. 2(a) Sort the following numbers using Merge sort. Give the output of each pass. Write an algorithm for Merge sort.
27, 6, 18, 25, 48, 59, 98, 34 **(10 Marks)**

Ans. :

Merge sort algorithm :

(I) 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.

The key operation of the merge sort algorithm is the merging of two sorted sequences in the "combine" step. To perform the merging, use an auxiliary procedure MERGE (A, p, q, r), where A is an array and p, q, and r are indices numbering elements of the array such that p ≤ q ≤ r.

The procedure assumes that the subarrays A[p..q] and A[q+1..r] are in the current subarray A[p..r]. Now use the MERGE procedure as a subroutine in the merge sort algorithm. The procedure MERGE-SORT (A, p, r) sorts the elements in the subarray A[p..r]. If p ≥ r, the subarray has at most one element and is therefore already sorted. Otherwise, the divide step simply computes an index q that partitions A[p..r] into two subarrays : A[p..q], containing n/2 elements, and A[q+1..r], containing n/2 elements.

Algorithm : Mergesort

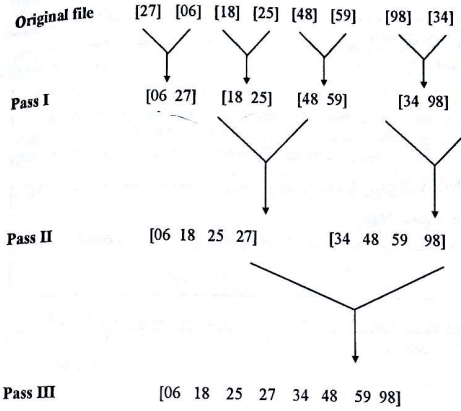
Input : An array A and indices p and r
Output: An sorted array A
MERGE-SORT(A,p,r)
1. if p < r
2. then q = (p + r) / 2
3. MERGE-SORT(A, p, q)
4. MERGE-SORT(A, q+1, r)
5. MERGE(A, p, q, r)

MERGE(A, p, q, r)
1. i = p, j = q + 1, n = r – p + 1
2. for k = 0 to n – 1 // for 1
3. if [(A[i] < A[j]) or (j > r)] and (i ≤ q)
4. B[k] = A[i]
5. i = i +1
6. Else
7. B[k] = A[j]
8. j = j +1
9. for k = 0 to n – 1
10. A[p + k] = B[k]

Consider following array *x* of size 8.

Array *x*[8] :

0	1	2	3	4	5	6	7
27	6	18	25	48	59	98	34



Q. 4(b) Write an algorithm for Binary Search. Derive its Best case and worst case complexities. **(10 Marks)**

Ans. :

Binary search :

A binary search algorithm is a technique for finding a particular value in a sorted list. It makes progressively better guesses, and closes in on the sought value by selecting the medium element in a list, comparing its value to the target value (key), and determining if the selected value is greater than, less than, or equal to the target value. A guess that turns out to be too high becomes the new top of the list, and a guess that is too low becomes the new bottom of the list. Pursuing this approach iteratively, it narrows the search by a factor of two each time, and finds the target value. The binary search consists of the following steps :

1. Search a sorted array by repeatedly dividing the search interval in half.
2. Begin with an interval covering the whole array.
3. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half.
4. Repeatedly check until the value is found or the interval is empty.

Recursive Binary Search : The most straight forward implementation is recursive, which recursively searches the subrange dictated by the comparison :

```
BinarySearch(A[0..N – 1], value, low, high)
{
  if (high < low)
       return – 1 // not found
    mid = (low + high) / 2
    if (A[mid] > value)
       return BinarySearch(A, value, low, mid – 1)
    else if (A[mid] < value)
       return BinarySearch(A, value, mid+1, high)
    else
       return mid // found
  }
```

Iterative Binary Search :

```
BinarySearch(A[0..N – 1], value)
{
    low = 0
    high = N – 1
    while (low <= high)
   {
       mid = (low + high) / 2
       if (A[mid] > value)
          high = mid – 1
       else if (A[mid] < value)
          low = mid + 1
       else
          return mid // found
    }
    return – 1 // not found
  }
```

For best case and worst case complexities :

Binary Search can be analyzed with the best, worst, and average case number of comparisons. These analyses are dependent upon the length of the array, so let N =|A| denote the length of the Array A. The numbers of comparisons for the recursive and iterative versions of Binary Search are the same, if comparison counting is relaxed slightly. For Recursive Binary Search, count each pass through the if-then-else block as one comparison. For Iterative Binary Search, count each pass through the while block as one comparison.

easy solution

***Best case* - O (1) comparisons**

In the best case, the item X is the middle in the array A. A constant number of comparisons (actually just 1) are required.

Worst case - O (log n) comparisons

In the worst case, the item X does not exist in the array A at all. Through each recursion or iteration of Binary Search, the size of the admissible range is halved. This halving can be done ceiling (lg n) times. Thus, ceiling(lg n) comparisons are required.

Q. 7(a) Implement the binary search, prove that the complexity of binary search is $O(\log_2 N)$. **(5 Marks)**

Ans. : **Time complexity analysis :** Time complexity can be written as a recurrence relation as,

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

The most popular and easiest way to solve recurrence relation is repeatedly make substitutions for each occurrence of function T in RHS until all such occurrences disappear.

Therefore,

$$\begin{aligned} T(n) &= T(n/1) + c \\ &= T(n/4) + 2c \\ &= T(n/8) + 3c \\ &\;\;\vdots \\ &= T(n/2^i) + ic \\ &\;\;\vdots \\ &= T(n/n) + kc = T(1) \text{ (after k steps)} \end{aligned}$$

Where $2^k = n$, hence $k = \log_2 n$

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

Q. 7(c) Write short note on : Strassen's Matrix Multiplication. **(5 Marks)**

Ans. : **Please refer Q. 5(a) of May 2012.**

Chapter 3 : Greedy Method [Total Marks - 30]

Q. 2(b) Write an algorithm for minimum spanning tree using Prim's method. **(10 Marks)**

Ans. : **Please refer Q. 2(b) of May 2012.**

Q. 5(b) Explain optimal storage on tapes with example. **(10 Marks)**

Ans. : **Please refer Q. 7(e) of May 2012.**

easy solution

Q. 7(b) Write short note on Job Sequencing with Deadlines. **(5 Marks)**

Ans. : **Please refer Q. 6(a) of May 2012.**

Q. 1(c) Compare greedy method and backtracking method. **(5 Marks)**

Ans. : **Comparison of Greedy Method with Backtracking Method :**

1. Greedy method takes decision on the basis of information immediately at hand without worrying about the effect these decisions may have in the future. In backtracking task is to determine algorithm to find solutions to specific problem not by following a fixed rule of computation, but by trial and error.
2. The greedy method suggests that one can devise an algorithm that works in stages, considering one input at a time. At each stage, a decision is made regarding whether a particular input is in an optimal solution.
3. This is done by considering the inputs in an order determined by some selection procedure. If the inclusion of the next input into the partially constructed optimal solution will result in an infeasible solution, then this input is not added to the partial solution. Otherwise, it is added.
4. In backtracking solution is constructed one component at a time and evaluate such partially constructed solutions as follows: if a partially constructed solution can be developed further without violating the problem's constraints, it is done by taking the first remaining legitimate option for the next component. If there is no legitimate option for the next component, no alternatives for any remaining component of the partially constructed solution with its next option.

Chapter 4 : Dynamic Programming [Total Marks - 25]

Q. 1(b) Explain general method of Dynamic programming. List different examples of it. **(5 Marks)**

Ans. :

General Method of Dynamic Programming :

Dynamic programming is an algorithm design method that can be used when the solution to a problem can be viewed as the result of a sequence of decisions.

Example 1 : [Knapsack] :

The solution to the knapsack, problem can be viewed as the result of a sequence of decisions. We have to decide the values of x_i, $1 \le i \le n$. First we make a decision on x_1, then on x_2 then on x_3, and so on. An optimal sequence of decisions maximizes the objective function $\Sigma p_i x_i$. (It also satisfies the constraints $\Sigma w_i x_i \le m$ and $0 \le x_i \le 1$.)

Example 2 : [Shortest path] :

One way to find a shortest path from vertex i to vertex j in a directed graph G is to decide which vertex should be the second vertex, which the third, which the fourth, and so on, until vertex j is reached. An optimal sequence of decisions is one that results in a path of least length.

easy solution

One way to solve problems for which it is not possible to make a sequence of stepwise decisions leading to an optimal decision sequence is to try all possible decision sequences. We could enumerate all decision sequences and then pick out the best. But the time and space requirements may be prohibitive.

Dynamic programming often drastically reduces the amount of enumeration by avoiding the enumeration of some decision sequences that cannot possibly be optimal. In dynamic programming an optimal sequence of decisions is obtained by making explicit appeal to the principle of optimality.

Q. 5(a) For the following graph find all pair shortest path using dynamic programming. **(10 Marks)**

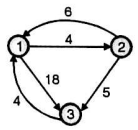


Fig. 1

Ans. :

The main motivation behind all - pairs shortest path problem is to determine a matrix A such that A(i,j) is the length of a shortest path from i to j. The only condition we consider to be satisfied by the graph G is that it should not contain any cycle with negative lengths. If cycles with negative lengths are allowed in the graph G, then the shortest path between any two vertices has length $-\infty$.

The shortest path from i to j in the given graph for $i \ne j$ is determined as follows :

The path begins at vertex i and terminates at vertex j through intermediate vertices. Let us assume that this path contains no cycles. If k is an intermediate vertex on this shortest path, then the subpaths from i to k and from k to j must be shortest paths from i to k and k to j respectively.

Let $A^k(i,j)$ represent length of a shortest path from i to j , going through no vertex of index greater than k, we obtain :

$$A(i,j) = \min \{ \min_{1 \le k \le n} \{A^{k-1}(i,k) + A^{k-1}(k,j)\}, \text{cost}(i,j) \}$$

$$A^0(i,j) = \text{cost}(i,j),\ 1 \le i \le n,\ 1 \le j \le n$$

Following tables are for matrices :

A^0 is graph matrix :

A^0	1	2	3
1	0	4	18
2	6	0	5
3	4	∞	0

(i.e. Cost of 2 to 1 is 6, but cost 1 to 1, 2 to 2,....is 0, no edge from 3 to 2 therefore 3 to 2 infinity)

easy solution

Now use following two cases :

Case 1 : If the shortest path p (from I to j going through vertices with indices ≤ k) does not go through the vertex k, then :

$$d_{ij}^{(k)} = d_{ij}^{(k-1)}$$

Case 2 : If the shortest path p goes through vertex k, then :

$$d_{ij}^{(k)} = d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$$

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

d_{ij} means matrix i=row j=column. K is the intermediate vertex.

For following matrix k=1,

A^1	1	2	3
1	0	4	18
2	6	0	5
3	4	8	0

Intermediate vertex is 2 i.e k=2 :

A^2	1	2	3
1	0	4	9
2	6	0	5
3	4	8	0

Intermediate vertex is 3 i.e k=3 :

A^3	1	2	3
1	0	4	9
2	6	0	5
3	4	8	0

Q. 6(b) Explain single source shortest path using Dynamic programming. Write an algorithm for same. **(10 Marks)**

Ans. :

Consider a directed graph G with some of the edges have negative length. When there is no cycles of negative length, there is a shortest path between any two vertices of an n vertex graph that has at most n – 1 edges on it. If a path has more than n – 1 edges, then it will repeat at least one vertex and thus it contains a cycle. Try some another path to eliminate the cycles from the results with the same source and destination vertices.

Now this path contains no cycle and length of path equals the original path. Use this observation on the maximum number of edges on a cycle free shortest path to obtain an algorithm to determine a

shortest path from a source vertex to all remaining vertices in the graph. Let $dist^l[u]$ be the length of a shortest path from the source vertex v to vertex u under the constrain that the shortest path contains at most *l* edges.

Then $dist^l[u] = cost[v, u] [v_1, \upsilon]$, $1 \le u \le n$. When there are no cycles of negative length, we can limit our search for shortest paths to paths with at must n–1 edges. Hence $dist^{n-1}[u]$ is the length of an unrestricted shortest path from v to u. Our goal is to compute $dist^{n-1}[u]$ for all u. This can be done using dynamic programming methodology.

1. Let the shortest path from v to u with at most k edges (where k > 1) has no more than k – 1 edges, than $dist^k[u] = dist^{k-1}[u]$.
2. Let the shortest path from v to u with at most k (where k > 1) edges has exactly k edges, then a shortest path from v to same vertex j followed by the edge (j,u).

   The path from v to j has k – 1 edges and its length is $dist^{k-1}[j]$. All vertices i such that the edge (i,u) is in the graph are candidates for j.

   We want to determine a shortest path, that minimizes $dist^{k-1}[i]$ + cost [i,u] is the correct value for j, then

   $dist^k[u] = \min \{ dist^{k-1}[u], \min\{ dist^{k-1}[i] + cost[i,u]\}\}$

   This recurrence can be used to compute $dist^k$ from $dist^{k-1}$, for k = 2, 3, …., n – 1

Algorithm :

```
1  Algorithm Bellman Ford (v,cost,dist,n)
2  // single source / all destinations shortest
3  // paths with negative edge costs
4  {
5  for i = 1 to n do // Initialize dist
6  dist [i] = cost [v,i] ;
7  for k = 2 to n – 1 do
8  for each u such that u ≠ v and u has
9  at least one incoming edge do
10 for each (i,u) in the graph do
11 if dist (u) > dist[i] + cost [i,u] then
12 dist (u) = dist[i] + cost [i,u] ;
13 }
```

Following are some algorithms that compute shortest paths :

1. Bellman Algorithm and 2. Ford Algorithm

The overall complexity is $O(n^3)$ when adjacency matrices are used.

Chapter 5 : Backtracking [Total Marks - 15]

Q. 1(d) State Graph coloring problem. State the strategy used to solve above problem. **(5 Marks)**

Ans. : Graph coloring algorithm :

Let G be graph and m be a given positive integer. With m colors we have to check whether the nodes of G can be coloured in such a way that no two adjacent nodes have the same color. The m colorability optimisation problem asks for the smallest integer m for which the graph G can be colored. This integer is referred as chromatic number of the graph. In above graph nodes can be colored with three colors 1, 2 and 3. The color of each node is indicated next to it. For above graph 3 colors are needed to color it and hence this graph's chromatic number is 3.

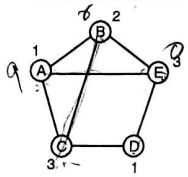


Fig. 2 : An example graph and its colouring

Suppose we represent a graph by its adjacency matrix G [1 : n, 1 : n] where

G [i, j] = 1 if [i, j] is an edge of G and

G [i, j] = 0 otherwise the colors are represented by integers 1, 2,……, m and solutions are given by n tuple $(x_1,\ldots,x_n)$ where x_i is the color of node i.

Algorithm : Finding all m-coloring of a graph

```
1. Algorithm m coloring(k)
2. // This algorithm was formed using the recursive backtracking schema. The graph is
3. //represented by its boolean adjacency matrix G[1 : n, 1 : n]. All assignments of
4. //1,2,..., m to the vertices of the graph such that adjacent vertices are assigned distinct
// 5 integers are printed, k is the index of the next vertex to color.
5. {
6.     repeat
7. {   // Generate all legal assignments for x[k].
8.     NextValue(k);    // Assign to x[k] a legal color.
9.     if (x[k] = 0) then return;   // No new color possible
10.    if (k = n) then     // At most m colors have been  used to color the n vertices.
11.    write (x[1 : n]);
12.    else mColoring (k + 1);
13.    } until (false);
14.}
```

easy solution

The state space tree used is a tree of degree m and height n + 1. Each node at le children corresponding to the m possible assignments to x_i, $1 \le i \le n$. Nodes at level n + 1 are leaf nodes. Following Fig. 3 shows the state space tree when n = 3 and m = 3.

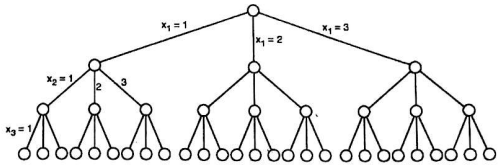


Fig. 3 : State space tree for mcoloring when n = 3 and m = 3

Function nextvalue produces the possible colors for x_k after x_1 through x_{k-1} have been defined. The main loop of mColoring repeatedly picks an element from the set of possibilities, assigns it to x_k and then calls mColoring recursively.

Q. 3(a) Explain N-Queen's problem using Backtracking with algorithm. **(10 Marks)**

Ans. : Please refer Q. 7(c) of May 2012.

Chapter 6 : String Matching Algorithms [Total Marks - 20]

Q. 3(b) Explain longest common subsequence with example. **(10 Marks)**

Ans. :

Longest Common Subsequence (LCS) Algorithm :

The longest common subsequence (LCS) is as implied by the phase is a subsequence of two strings; $X = x_0 x_1 x_2 \ldots x_{n-1}$ and $Y = y_0 y_1 \ldots y_{m-1}$ such it is the longest. The LCS is not necessarily unique. A current research topic in bio-informatics. DNA are represented as strings of the small alphabet, Sigma = {A, C, G, T}. DNA strands/strings can be changed by mutation by insertion of a character into the string. A longest common subsequence of two strings can represent the common ancestry of the two strings.

Finding the LCS :

We enumerate all the subsequences in X and then start with the longest check if it is a subsequence of Y. Because a character in X can either be a member or not a member of the subsequence the number of subsequence is 2n. Checking if the sequence is a subsequence of Y takes O(m). So worst case running time is $O(m2^n)$. This is called exponential time, to be avoided at all cost even at the cost of more space.

easy solution

Applying Dynamic Programming to the LCS Problem :

A technique for solving optimization problem that are exponential time using brute force algorithmic problems, which is polynomial time. Dynamic programming is similar to the greedy algorithm in that we globally optimize by locally optimizing a sub problem. Dynamic programming is different from the greedy algorithm in that it uses an additional data structure, a table as multidimensional array. Each entry in the table represents optimal value for a particular sub problem. The indices represent a particular subproblem. So the subproblem must be representable by integers given the global problem.

Dynamic programming for LCS :

L[i, j] is our table representing the length of the LCS for strings X[0..i] and Y[0..j]. Where represent the character in string X and j in Y. The goal is to construct the table using the rules:

$L[i, j] = L[i - 1, j - 1] + 1$ if $x_i = y_j$

$L[i, j] = \max\{ L[i - 1, j], L[i, j - 1]\}$ $x_i != y_j$

$L[i, -1] = L[-1, j] = 0$ initial condition

The table is built either row wise or column wise. L[i, j] represents the length of the LCS for $x_0x_1...x_i$ and $y_0y_1...y_j$, the sub problem. The LCS is built adding one match at a time search both the X and Y. In the finish table L[n, m] is the length of the LCS.

The LCS Algorithm : LCS(X, Y) :

```
Input: Strings X and Y with n and m elements
Output: The table L[i, j]  the LCS for the sub problems x0x1...xi and y0y1...yj,
for i = -1 to n-1 do
    L[i, -1] = 0
for j = 0 to m -1 do
    L[-1, j] = 0
for i = 0 to n-1 do
    for j = 0 to m-1 do
if xi == yj then
L[i, j]  = L[i-1, j-1] +1
else
L[i, j] = max{ L[i-1, j], L[i, j-1]}
return L
```


Example :

Find the longest common subsequence from the given two sequences :

P = (100101101101) Q = (0110)

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

∴ The longest common subsequence (LCS) is 0110

Q. 6(a) Find Longest common subsequence of given two strings
X = {B A T A}, Y = {T A T A} **(10 Marks)**

Ans. :

The longest common subsequence (LCS) is as implied by the phase is a subsequence of two strings; $X = x_0 x_1 x_2...x_{n-1}$ and $Y = y_0 y_1...y_{m-1}$ such it is the longest. The LCS is not necessarily unique. A longest common subsequence of two strings can represent the common ancestry of the two strings.

Step 1: no match of row character with column character therefore put 0's.

		B	A	T	A
	0	0	0	0	0
T	0				
A	0				
T	0				
A	0				

Now use following cases :

Case 1 : If X(i) =Y(j), then, the desired subsequence has to ignore one of X(i) or Y(j), so we have :

LCS [i , j] = max(LCS [i – 1,j], LCS [i, j – 1])

For e.g. LCS [1,1] :

X(1)=T and Y(1)=B

Here X(i)≠Y(j) so by applying Case 1 as above, we have :

LCS[1,1]= max (LCS[0,1],LCS[1,0])

Case 2 : If X(i)=Y(j), then, the LCS of X(i) and Y(j), may match as well. For example, if a common subsequence that matched X(i) to an earlier location in Y(j), we always match it to Y(j) instead.

For this case, we have :

LCS [i , j] =1 + LCS[i −1, j − 1].

LCS[1,3]= 1+ LCS[0,2] (Using Case 2)

LCS[1,4]= max(LCS[0,4], LCS[1,3]) (Using Case 1)

Indices		0	1	2	3	4
0			B	A	T	A
1		0	0	0	0	0
2	A	0	0	1	1	2
3	T	0	0	1	2	2
4	A	0	0	1	2	3

Thus LCS for the given problem is "ATA"

Chapter 7 : Branch and Bound [Total Marks - 15]

Q. 4(a) Describe 0/1 Knapsack problem. How to solve it using Branch and Bound? **(10 Marks)**

Ans. : **Please refer Q. 3(b) of May 2012.**

Q. 7(d) Write short note on : Branch and Bound Strategy. **(5 Marks)**

Ans. :

Branch and Bound Strategy :

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

The general idea of B and B is a BFS-like search for the optimal solution, but not all nodes get expanded (i.e., their children generated). Rather, a carefully selected criterion determines which node to expand and when, and another criterion tells the algorithm when an optimal solution has been found Search the tree using a breadth-first search (FIFO branch and bound).

Search the tree as in a BFS, but replace the FIFO queue with a stack (LIFO branch and bound). Replace the FIFO queue with a priority queue (least-cost (or max priority) branch and bound). The priority of a node p in the queue is based on an estimate of the likelihood that the answer node is in the subtree whose root is p. Space required is O(number of leaves).

For some problems, solutions are at different levels of the tree (e.g. 15 Puzzle).

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

(a) An arrangement

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

(b) Goal arrangement

Fig. 4 : 15 Puzzle

FIFO branch and bound finds solution closest to root. Backtracking may never find a solution because tree depth is infinite (unless repeating configurations are eliminated). Least-cost branch and bound directs the search to parts of the space most likely to contain the answer. So it could perform better than backtracking.

❑❑❑

May 2013

Chapter 1 : Introduction to Analysis of Algorithm [Total Marks - 10]

Q. 1(b) Explain Asymptotic notations. Explain time complexity and space complexity in detail. **(10 Marks)**

Ans. : Please refer Q. 1(a) of May 2012.

Chapter 2 : Divide and Conquer [Total Marks - 25]

Q. 1(a) Explain Divide and Conquer strategy. Write control abstraction (General method) for it. List any Four examples that can be solved by divide and conquer. **(10 Marks)**

Ans. :

Divide and Conquer :

A divide and conquer algorithm repeatedly reduces an instance of a problem to one or smaller instances of the same problem (usually recursively) until the instances are small enough to solve easily. One such example of divide and conquer is merge sorting. Sorting can be done on each segment of data after dividing data into segments and sorting of entire data can be obtained in the conquer phase by merging the segments. A simpler variant of divide and conquer is called a decrease and conquer algorithm, that solves the identical subproblems and uses the solution of these subproblems to solve the bigger problem.

Divide and conquer divides the problem into multiple subproblems and so conquer stage will be more complex than decrease and conquer algorithms. An example of decrease and conquer algorithm is the binary search algorithm.

General Method (Divide and Conquer) :

Following are the steps in solving a problem in divide and conquer strategy :

1. Divide the problem into number of subproblms.
2. And then solve these subproblems recursively.
3. Integrate all the subproblems to give a result.

These three steps are coordinated together by the algorithm to compute the final output.

Many algorithms call themselves recursively one or more times to handle closely related subproblems. All of these algorithms follow a divide-and-conquer approach to get the output for the computation. They decompose the larger problem into number of subproblems or the small solvable unit and solve recursively and finally integrate the results of all these subproblems to obtain the final result.

In short the divide-and-conquer approach has basic three steps at each level of the recursion :

1. Divide the problem into a number of smaller units called subproblems.
2. Conquer (i.e solve) the subproblems recursively.
3. Combine the solutions of all the subproblems into a sloution for the original problem.

Algorithm : Divide and Conquer algorithm

```
1  Algorithm DAndC (P)
2  {
3  if Small(P) then return S(P);
4  else
5  {
6  divide P into smaller instances P1, P2,..., Pk , k ≥ 1;
7  Apply DAndC to each of these subproblems;
8  return Combtne(DAndC(P1),DAndC(P2),.. .,DAndC(Pk ));
9  }
10 }
```

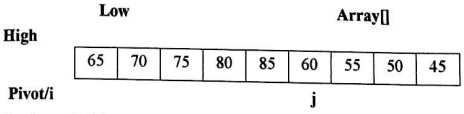

Where P is the problem to be solved and Small(P) is a Boolean-valued function that determines whether the input size is small or not, so that answer can be computed without splitting. If it is small, then the function S is invoked, otherwise the problem P is divided into smaller subproblems. Then these subproblems P_1 , P_2 ,.....,P_k are solved by recursive applications of DAndC (P). And finally with the help of function Combine, the solution to P is obtained.

For Merge Sort : Please refer Q. 2(a) of Dec. 2012.

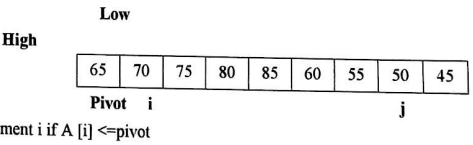
Q. 4(a) Sort following numbers using Quicksort algorithm. Show all passes of execution. **(10 Marks)**

65, 70, 75, 80, 85, 60, 55, 50, 45

Ans. :



Split the Array A [] in two parts. The left sublist will contain the elements less than pivot=65 and right sublist will contain the elements greater than pivot = 65. For that matter, let us partition the array using following procedure.

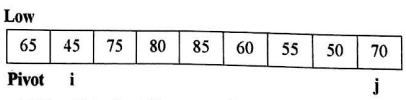


Increment i if A [i] <=pivot

And decrement j if A [j] >= pivot

As now A [i] > pivot, stop incrementing. Now check if (A [j] > = pivot)? No. So stop decrementing j.

Swap A [i] and A [j]



Increment i, now A [i] = 75 > pivot. Hence stop incrementing i. Decrement j, now

easy solution

= min { 67, 273 }

= 5

= min { [(2,4) + a

A [j] = 50 < 65. Hence stop decrementing j.

Low								
65	45	75	80	85	60	55	50	70
Pivot	i					j		

Swap A [i] and A [j], then increment i and decrement j.

Low								
65	45	50	80	85	60	55	75	70
Pivot			i		j			

Again A [i] > pivot , stop increment i.

Again A [i] < pivot , stop decrementing j.

Swap A [i] and A [j] and increment i and decrement j

Low								
65	45	50	55	85	60	80	75	70
Pivot				i	j			

Again A [i] > pivot , stop increment i.

Again A [i] < pivot , stop decrementing j.

Swap A [i] and A [j]

Low								
65	45	50	55	60	85	80	75	70
Pivot				i	j			

As A [i] < pivot increment i, further i cannot be incremented because 85 > 65.

Now A [j] > pivot, decrement

Low								
65	45	50	55	60	85	80	75	70
Pivot				j	i			

Swap A [j] and A [low]

Low								
60	45	50	55	65	85	80	75	70
Left Sublist				pivot		right sublist		

This is an end of pass 1. Now sort left sublist by considering first element as pivot

60	45	50	55
Pivot	i		j

At end of pass2

55	45	50	60
Pivot	i	j	

At end of pass 3

50	45	50	60
Pivot	i	j	

Left sublist is sorted

45	50	55	60

Similarly right sublist

85	80	75	70

Right sublist after sorting is as follows

70	75	80	85

Q. 7(c) Write short note on : Advantages of divide and conquer. **(5 Marks)**

Ans. :

Advantages of divide and conquer :

1. **Solving difficult problems :** Divide and conquer technique is a tool for solving difficult problems conceptually. e.g, Tower of Hanoi puzzle. It requires a way of breaking the problem into sub-problems, and solving all of them as an individual cases and then combining sub-problems to the original problem.
2. **Algorithm efficiency :** The divide-and-conquer paradigm often helps in the discovery of efficient algorithms. It was the key, for example to the Quicksort and Mergesort algorithms, and fast Fourier transforms. In all these examples, the D and C approach led to an improvement in the asymptotic cost of the solution. In particular, if the base cases have constant-bounded size, the work of splitting the problem and combining the partial solutions is proportional to the problem's size n, and there are a bounded number p of subproblems of size ~ n/p at each stage, then the cost of the divide-and-conquer algorithm will be O(n log n).
3. **Parallelism :** Normally DAC algorithms are used in multi-processor machines having shared-memory systems where the communication of data between processors does not need to be planned in advance, because distinct sub-problems can be executed on different processors.
4. **Memory access :** These algorithms naturally make an efficient use of memory caches. Since the subproblems are small enough to be solved in cache without using the main memory that is slower one. Any algorithm that uses cache efficiently is called cache oblivious.
5. **Roundoff control :** In computations with rounded arithmetic, e.g. with floating point numbers, a divide-and-conquer algorithm may yield more accurate results than a superficially equivalent iterative method. For example, one can add N numbers either by a simple loop that adds each datum to a single variable, or by a D And C algorithm that breaks the data set into two halves, recursively computes the sum of each half, and then adds the two sums. While the second method performs the same number of additions as the first, and pays the overhead of the recursive calls, it is usually more accurate.

= min { 6+1, 2+3 }

= 5

A[2,3] = min { [(2,4) + a

Chapter 3 : Greedy Method [Total Marks - 30]

Q. 2(b) Find minimum cost spanning tree for the graph shown below using prim's algorithm. **(10 Marks)**

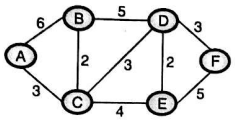
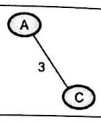
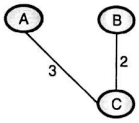
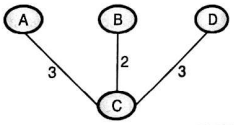
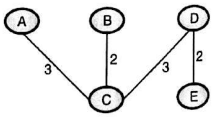


Fig. 1

Ans. :

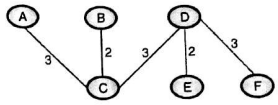
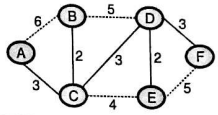
Table 1 showing run-steps for above algorithm

Sr. No.	Step No. executed	Edges Considered	Edges Selected	Cost	Spanning tree
1	1	-	-	-	
2	1	(A,B),(A,C)	(A,C)	3	
3	2 & 3-1st iteration	(A,B),(C,B), (C,D),(C,E)	(C,B)	2	
4	2 & 3-2nd iteration	(A,B),(B,D), (C,D),(C,E)	(C,D)	3	
5	2 & 3-3rd iteration	(A,B),(B,D), (C,E),(D,E), (D,F).	(D,C)	2	

easy solution

= min { 6+7, 2+3 }

= 5

Sr. No.	Step No. executed	Edges Considered	Edges Selected	Cost	Spanning tree
6	2 & 3-4th iteration	(A,B),(B,D), (C,E),(D,F), (E,F).	(D,F)	3	
7	5	Minimum Spanning Tree in the graph with total cost is 13. 			
8	6	Stop			

Q. 5(a) Explain job-sequencing with deadlines. Solve the following instance : **(10 Marks)**

n = 5.

$(P_1, P_2, P_3, P_4, P_5) = (20, 15, 10, 5, 1)$

$(d_1, d_2, d_3, d_4, d_5) = (2, 2, 1, 3, 3)$

Ans. : **Please refer Q. 6(a) of May 2012.**

Q. 7(b) Write short note on : Optimal storage on tapes. **(5 Marks)**

Ans. : **Please refer Q. 7(e) of May 2012.**

Q. 7(d) Write short note on : Minimum spanning Tree using Kruskal's Algorithm. **(5 Marks)**

Ans. : Minimum spanning tree using Kruskal's Algorithm :

In Kruskal's minimum spanning tree algorithm, the tree is started with the empty graph and then it selects edges from E according to the following rule :

(i) Always add the next edge that has a minimum weight and that doesn't produce a cycle. Thus the tree is constructed edge by edge and cycles are avoided.

(ii) Always select that edge which is cheapest at the moment.

(iii) This is a greedy algorithm approach in which every decision it makes is based on most obvious immediate advantage.

Algorithm :

```
1. Algorithm Kruskal (E, cost, n, t)
2. // E is the set of edges in G. n vertices in G.
3. // cost[u, v] is cost of edge (u, v)
4. // t is set of edges in minimum-cost spanning tree
5. // {
6. Construct a heap out of the edge costs
```

easy solution

C(2,3) = min { C(2,4) + C

```
7.  for i = 1 to n do parent [i]  = -1
8.  // Each vertex is in a different set
9.  i = 0; mincost = 0.0;
10. while((i < n - 1) and (heap not empty)) do
11. {
12. // Delete a minimum cost edge (u, v) from heap 13. and reheapify j
13. j = Find (u); k = Find (v)
14. if(j ≠ k) then
15. {
16. i = i + 1;
17. t[i, 1] = u;   t[i, 2] = v;
18. mincost = mincost + cost [u, v];
19. Union(j, k)
20. }
21. }
22. if(i ≠ n - 1) then write ("No spanning tree");
23. else return mincost;
24. }
```

In line 6, initial heap of edges is constructed. In line 7 each vertex is assigned to a distinct set t. The set t is the set of edges to be added in the minimum-cost spanning tree and i is the number of edges in t. In the while loop of line 10, edges are removed from the heap one by one in non-decreasing order of cost. Line 14 determines the sets containing u and v. If j ≠ k then vertices u and v are in different sets and edge (u, v) in included into t.

The sets containing u and v are combined (line 20). If u = v, the edge(u, v) is discarded as its inclusion into t would create a cycle. Line 23 determines whether a spanning tree was found. It follows that i ≠ n – 1 iff the graph G is not connected. One can verify that computing time is O(| E | log | E |) where E is the edge set of G.

Chapter 4 : Dynamic Programming [Total Marks - 15]

Q. 6(a) Solve shortest path from source 1 for following graph using dynamic programming. **(10 Marks)**

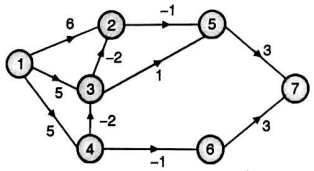


Fig. 2

easy solution

= min { 6+7, 2+3 }

Ans. :

For finding shortest path from source 1 for the given graph using dynamic programming, use forward approach. By using forward approach for the given graph, use the following formula :

Cost (1, 7) = min {6 + cost(2, 7), 5 + cost (3, 7), 5 + cost (4, 7)} ...(1)

Now, solve cost (2, 7), cost (3, 7) and cost (4, 7) individually as follows :

(1) cost (2, 7) = – 1 + 3 = 2

(2) cost (3, 7) = 1 + 3 = 4

(3) cost (4, 7) = min {– 1 + 3, – 2 + 1 + 3, – 2 – 2 – 1 + 3} = min {2, – 2, – 2}

Put these individual values in Equation (1) :

Equation (1) becomes,

cost (1, 7) = min {6 + 2, 5 + 4, 5 + (– 2)} = min {8, 9, 3}

cost (1, 7) = 3

The shortest path from source 1 is 3.

Q. 7(a) Write short note on : Differentiate between greedy approach and dynamic programming. **(5 Marks)**

Ans. :

Difference between greedy approach and dynamic programming :

Sr. No.	Greedy approach	Dynamic programming
1.	Greedy method is used for obtaining optimum solution.	Dynamic programming is also for obtaining optimum solution.
2.	In greedy method a set of feasible solutions and the picks up the optimum solution.	There is no special set of feasible solutions in this method
3.	In this method the optimum selection is without revising previously generated solutions.	It considers all possible sequences in order to obtain the optimum solution.
4.	There is no as such guarantee of getting optimum solution.	It is guaranteed that the dynamic programming will generate optimal solution using principle of optimality.
5.	In this method only one decision sequence is generated.	In this method many decision sequences may be generated.

Chapter 5 : Backtracking [Total Marks - 20]

Q. 2(a) Explain Graph coloring problem using backtracking. Write algorithm for same. **(10 Marks)**

Ans. : **Please refer Q. 1(d) of Dec. 2012.**

Q. 5(b) Solve following sum of subset problem using backtracking : **(10 Marks)**

w = { 1, 3, 4, 5 }

m = 8

Find all possible subsets of 'w' that sum to 'm'.

easy solution

= min { (2,4) + ...

Ans. :

w = {1, 3, 4, 5}
m = 8

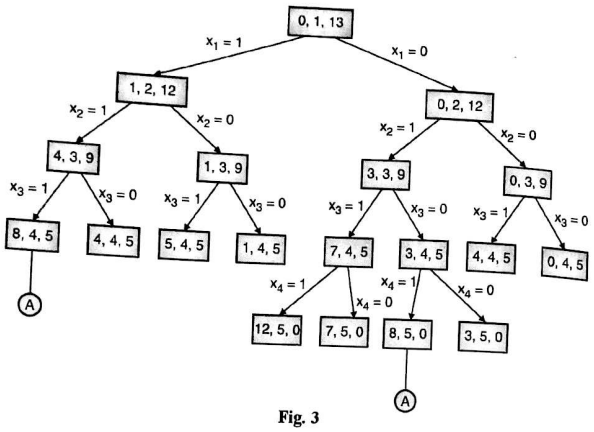


Fig. 3

The nodes marked with are possible subset of 'w' that sum to 'm' i.e. 8.

Chapter 6 : String Matching Algorithms [Total Marks - 20]

Q. 3(a) Find the longest common subsequence from the given two sequences : **(10 Marks)**

P = (100101101101)

Q = (0110)

Ans. : **Please refer Q. 3(b) of Dec. 2012.**

Q. 4(b) Explain and write Knunth –Morris Pratt algorithm. Explain with an example. **(10 Marks)**

Ans. : **Please refer Q. 5(b) of May 2012.**

Chapter 7 : Branch and Bound [Total Marks - 20]

Q. 3(b) Explain 15-puzzle problem using branch and bound. **(10 Marks)**

Ans. : The 15-puzzle Problem using branch and bound :

The 15-puzzle consists of 15 numbered tiles on a square frame with a capacity of 16 tiles (Fig. 4). We are given an initial arrangement of the tiles, and the objective is to transform the arrangement into the goal arrangement of Fig. 4(b) through a series of legal moves. The only legal moves are ones in which a tile adjacent to the empty spot (ES) is moved to ES. Thus from the initial arrangement of Fig. 4(a), four moves are possible.

= min{ 647, 243 }
= 5.

We can move any one of the tiles numbered 2, 3, 5, or 6 to the empty spot. Depending on these moves, other moves can be made. Every new move creates a new arrangement of the tiles. These arrangements of the tiles are called as the states of the puzzle. The initial arrangements and final arrangements are called as the initial states and goal states respectively. A state is reachable from the initial state if and only if there is a sequence of legal moves from the initial state to this state.

The state space of an initial state consists of all states that can be-reached from the initial state. The most straightforward way to solve the puzzle would be to search the state space for the goal state and use the path from the initial state to the goal state as the answer. It is easy to see that there are 16! ($16! \approx 20.9 \times 10^{12}$) different arrangements of the tiles on the frame. Of these only one-half are reachable from any given initial state. Indeed, the state space for the problem is very large. Before attempting to search this state space for the goal state, it would be worthwhile to determine whether the goal state is reachable from the initial state. There is a very simple way to do this.

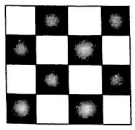
Let us number the frame positions 1 to 16. Position *i* is the frame position containing tile numbered *i* in the goal arrangement of Fig. 4(b). Position 16 is the empty spot. Let position(i).be the position number in the initial state of the tile numbered i. Then position(16) will denote the position of the empty spot.

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

(a) An arrangement

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

(b) Goal arrangement



(c)

Fig. 4 : 15-puzzle arrangements

For any state let less(i) be the number of tiles j such that j < i and position(j) > position{4). For the state of Fig. 4(a) we have, for example, less(l) = 0, less(4) = 1, and less(12) = 6. Let x.=l if in the initial state the empty spot is at one of the shaded positions of Fig. 4(c) and x = 0 if it is at one of the remaining positions.

Theorem :

The goal state of Fig. 4(b) is reachable from the initial state iff $\sum_{i=1}^{16}$ = i less(i) + x is even.

Theorem can be used to determine whether the goal state is in the state space of the initial state. If it is, then proceed to determine a sequence of moves leading to the goal state.

To carry out this search, the state space can be organized into a tree. The children of each node x in this tree represent the states reachable from state x by one legal move. It is convenient to think of a move as involving a move of the empty space rather than a move of a tile. The empty space, on each move, moves either up, right, down, or left.

Fig. 5 shows the first three levels of the state space tree of the 15-puzzle beginning with the initial state shown in the root. Parts of levels 4 and 5 of the tree are also shown. The tree has been pruned a little. No node p has a child state that is the same as p 's parent. The subtree eliminated in this way is already present in the tree and has root parent(p). As can be seen, there is an answer node at level 4.

= min{ L(2,4) + C

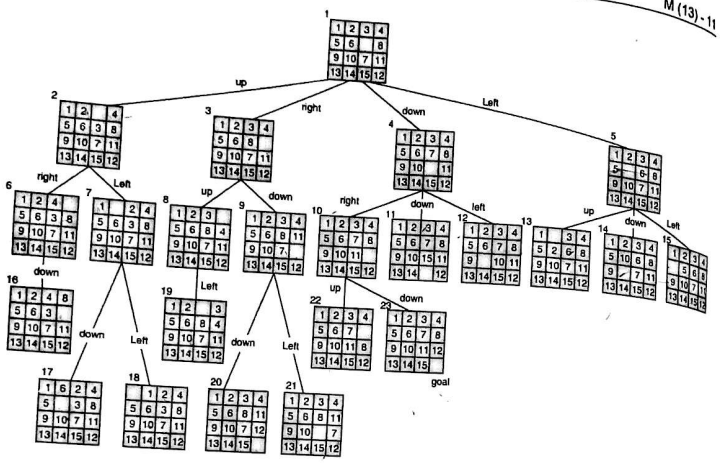


Fig. 5 : Part of the state space tree for the 15-puzzle

Q. 6(b) Explain travelling salesperson problem using branch and bound method. **(10 Marks)**

Ans. : **Please refer Q. 4(a) of May 2012.**

□□□

= min { 67, 243 {

Dec. 2013

Chapter 1 : Introduction to Analysis of Algorithm [Total Marks - 15]

Q. 1(b) Explain asymptotic notations. Explain time complexity and space complexity in detail. **(10 Marks)**

Ans. : **Please refer Q. 1(a) of May 2012.**

Q. 7(a)(iii) Write short note on : Randomised Algorithm. **(5 Marks)**

Ans. : **Please refer Q. 7(b) of May 2012.**

Chapter 2 : Divide and Conquer [Total Marks - 30]

Q. 1(a) Explain divide and conquer strategy. Write control abstraction (General Method) for it. List any four problems that can be solved using divide and conquer. **(10 Marks)**

Ans. : **Please refer Q. 1(a) of May 2013.**

Q. 5(a) Explain Binary search. Derive its best case and worst case complexity. **(10 Marks)**

Ans. : **Please refer Q. 4(b) of Dec. 2012.**

Q. 7(a)(i) Write short note on : Advantages of Divide and Conquer. **(5 Marks)**

Ans. : **Please refer Q. 7(c) of May 2013.**

Q. 7(a)(iv) Write short note on : Strassen's matrix multiplication. **(5 Marks)**

Ans. : **Please refer Q. 5(a) of May 2012.**

Chapter 3 : Greedy Method [Total Marks - 25]

Q. 3(b) Consider the graph given in following Fig. 1, construct minimum spanning tree using Kruskal's algorithm. **(10 Marks)**

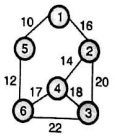


Fig. 1

Ans. :

The step wise procedure for construction of minimum spanning tree using Kruskal's algorithm is given below :

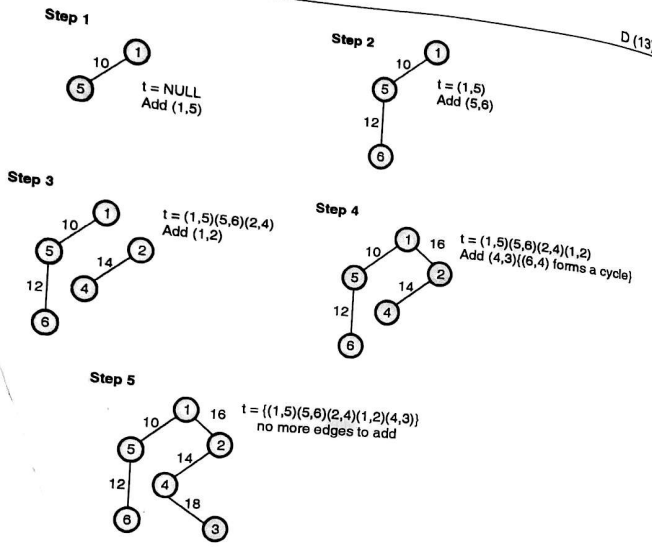


Fig. 1(a)

Q. 6(b) Explain job sequencing with dead lines along with example. **(10 Marks)**

Ans. : **Please refer Q. 6(a) of May 2012.**

Q. 7(a)(ii) Difference between Prim's Algorithm and Kruskal's Algorithm. **(5 Marks)**

Ans. : **Please refer Q. 7(a) of May 2012.**

Chapter 4 : Dynamic Programming [Total Marks - 20]

Q. 2(a) Construct the optimal Binary search tree for identifier set **(10 Marks)**

(a_1, a_2, a_3, a_4) = (out, float, if, while)

with p (1 : 4) = $\left(\frac{1}{20}, \frac{1}{5}, \frac{1}{10}, \frac{1}{20}\right)$ and q (0 : 4) = $\left(\frac{1}{5}, \frac{1}{10}, \frac{1}{5}, \frac{1}{20}, \frac{1}{20}\right)$

Ans. : **Please refer Q. 1(b) of May 2012.**

Q. 3(a) Explain flow shop scheduling with the help of example. **(10 Marks)**

Ans. : **Please refer Q. 2(a) of May 2012.**

easy solution

= min { 6+7, 2+3 }

Chapter 5 : Backtracking [Total Marks - 20]

Q. 4(a) State Graph coloring algorithm. Explain strategy used for solving it along with example. **(10 Marks)**

Ans. : **Please refer Q. 1(d) of Dec. 2012.**

Q. 6(a) Explain 8 Queen problem and strategy used to solve it. **(10 Marks)**

Ans. : **8-Queens Problem :**

Let the chessboard squares is numbered as the indices of the two-dimensional array a [1 : n, 1 : n], then every elements on the same diagonal that runs from the upper left to lower right has the same row – column value.

Also every element on the same diagonal that goes from the upper right to lower left has the same row + column value. Suppose two queens are placed at positions (i, j) and (k, l), then by the above they are on the same diagonal only if i – j = k – l or i + j = k + l

The first equation implies

j – l = i – k

The second implies

j – l = k – i

Thus the two queens lie on the same diagonal if and only if | j – l | = | i – k |

Algorithm :

```
1.  Algorithm Place(k, i)
2.  // Returns true if a queen can be placed in k^th row and i^th column. Otherwise it
    // returns
3.  false. x[ ] is a global array whose first (k – 1) values have been set.
4.  // Abs(r) returns the absolute value of r.
5.  {
6.  for j:= 1 to k – 1 do
7.      if (x[j] = i)   // Two in the same column
8.          or (Abs(x[j] – i) = Abs(j – k))
9.              // or in the same diagonal
10.     then return false;
11. return true;
12. }
```

Above algorithm place (k, i) returns a Boolean value that is true if the k^{th} queen can be placed in column i. It tests both whether i is distinct from all previous values x [1 ....., x[k – 1] and whether there is no other queen on the same diagonal. Its computing time is O (k – 1).

The array x [ ] is a global. The algorithm is invoked by NQueens (1, n).

easy solution

Algorithm : All solutions to the n-queens problem.

```
1.  Algorithm NQueens(k, n)
2.  // Using backtracking, this procedure prints all possible placements of n queens on
3.  // an n × n chessboard so that they are nonattacking.
4.  {
5.      for i := 1 to n do
6.      {
7.
8.          if Place(k,i) then
9.          {
10.             x[k] := i;
11.             if (k = n) then write (x[l : n]);
12.             else NQueens(k + l, n);
13.     }
14.     }
15. }
```

For an 8 × 8 chessboard there are $\binom{64}{8}$ possible ways to place 8 pieces or approximately 4.4 billion 8-tuples to examine using brute force approach. However by allowing only placements of queens on distinct rows and columns, we require the examination of atmost 8! or only 40, 320, 8 tuples.

Chapter 6 : String Matching Algorithms [Total Marks - 10]

Q. 5(b) Find the longest common subsequence of given two strings. **(10 Marks)**

X = {B A T A}, Y = {T A T A}

Ans. : **Please refer Q. 6(a) of Dec. 2012.**

Chapter 7 : Branch and Bound [Total Marks - 20]

Q. 2(b) Explain 0/1 knapsack problem using Branch and Bound method. **(10 Marks)**

Ans. : **Please refer Q. 3(b) of May 2012.**

Q. 4(b) Explain 15-puzzle problem using branch and bound. **(10 Marks)**

Ans. : **Please refer Q. 3(b) of May 2013.**

□□□

easy solution

= min { 647, 243 }

= 5.

Analysis of Algorithm

Statistical Analysis

Chapter No.	May 14
Chapter 1	15 Marks
Chapter 2	15 Marks
Chapter 3	20 Marks
Chapter 4	30 Marks
Chapter 5	15 Marks
Chapter 6	25 Marks
Repeated Questions	-

May 2014

Chapter 1 : Introduction to Analysis of Algorithm [Total Marks - 15]

Q. 1(a) Explain Big-oh(O), Omega(Ω) and Theta(θ) Notations with the help of Graph. And represent the following function using above notations. **(10 Marks)**

(i) $T(n) = 3n + 2$ (ii) $T(n) = 10n^2 + 2n + 1$

Ans. : Big-oh(O) :

This notation bounds a function to within constant factors. That $f(n) = \theta(g(n))$ if there exist positive constants n_0, c_1 and c_2 such that to the right of n_0 the value of $f(n)$ always lies between $c_1 g(n)$ and $c_2 g(n)$ inclusive.

$$f(n) = \theta(g(n))$$

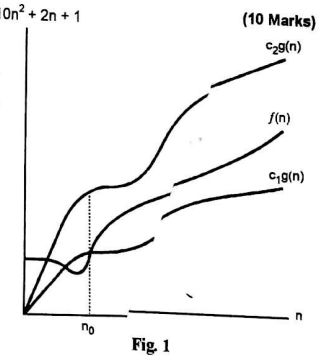


Fig. 1

Omega notations(Ω) :

This notation gives an upper bound for a function to within a constant factor. Then write $f(n) = O(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or below $cg(n)$.

Following functions are used in this estimation:

1, log (n), n, n log (n), n^2, 2^n, n!

The Big-O notation has two main application areas :

Analysis of the complexity of algorithms.

In mathematics, for an infinite series like an asymptotic series. Big-O gives us a formal way of expressing asymptotic upper bounds, a way of bounding from above the growth of a function.

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

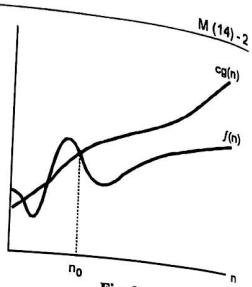


Fig. 2

Theta notation :

This notation gives a lower bound for a function to within a constant factor. Then write $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of f(n) always lies on or above $cg(n)$.

g(n) is an asymptotic lower bound for f (n).

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

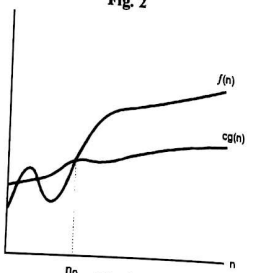


Fig. 3

(i) **T(n) = 3n + 2**

Big-oh notation :

As per definition of Big O, "The function f(n) = O(g(n)) (i.e. f of n is big oh of g of n) iff (if and only if) there exist positive constants c and n_0 such that $f(n) <= c * g(n)$ for all n, $n >= n_0$.

In this Example f(n) = 3n + 2

$3n + 2 <= 4n$ is true for all $n >= 2$

So, we have $c = 4$, $n_0 = 2$ and g(n) = n

Hence we can say $3n + 2 = O(n)$

Omega notation :

Here in this example, $3n + 2 \geq 3n$ for $n \geq 1$

$3n + 2 = \Omega(n)$ as

$3n + 2 \geq 3n$ for $n \geq 1$ i.e. $c=3$ and $n_0=1$

Theta notation :

The function $3n + 2 = \theta(n)$ as

$3n+ 2 \geq 3n$ for all $n \geq 2$ and

$3n + 2 \leq 4n$ for all $n \geq 2$, so $c_1 = 3$, $c_2 = 4$ and $n_0 = 2$

(ii) ***T(n) = 10n² + 2n + 1***

Big-oh notation :

In this Example $f(n) = 10n^2 + 2n + 1$

$10n^2 + 2n + 1 <= 11n^2$ is true for all $n >= 3$

= min { 677, 273 }

= 5.

So, we have $c = 11$, $n_0 = 3$ and $g(n) = n^2$

Hence we can say $10n^2 + 2n + 1 = O(n^2)$

Omega notation :

In this Example $f(n) = 10n^2 + 2n + 1$

$10n^2 + 2n + 1 >= 10n^2$ is true for all $n >= 1$

So, we have $c = 10$, $n_0 = 1$ and $g(n) = n^2$

Hence we can say $10n^2 + 2n + 1 = \Omega(n^2)$

Theta notation :

The function $10n^2 + 2n + 1 = \theta(n)$ as

$10n^2 + 2n + 1 \geq 11n^2$ for all $n \geq 2$ and

$10n^2 + 2n + 1 \leq 10n^2$ for all

$n \geq 2$, so $c_1 = 11$, $c_2 = 10$ and $n_0 = 3$

Q. 6(b) Write note on : Randomized algorithm **(5 Marks)**

Ans. :

Randomized Algorithm :

A randomized algorithm are also called as probabilistic algorithm. The algorithm typically uses uniformly random bits as an input to guide its behavior and it tries to achieve good performance in the "average case". Basically, the algorithm's performance will be a random variable and thus either the running time, or the output are random variables. In common practice, randomized algorithms are approximated using a pseudorandom number generator in place of a true source of random bits; such an implementation may deviate from the expected theoretical behavior.

Consider the following example :

The problem of finding 'a' in an array of *n* elements :

In a given array of n elements, let half the elements are 'a' and that of half are 'b' . One of the approach is to look at each element of the array and this is an expensive one and will take n/2 operations, if the array were ordered as 'b's first followed by 'a's. With this approach, can not guarantee that the algorithm will complete quickly. On the other hand, if look randomly then can find 'a' quickly with high probability.Randomized algorithms are particularly useful when faced with a malicious "adversary" or attacker who deliberately tries to feed a bad input to the algorithm (see worst-case-complexity and competitive analysis (online algorithm)). It is for this reason that randomness is ubiquitous in cryptography. In cryptographic applications, pseudo-random numbers cannot be used, since the adversary can predict them, making the algorithm effectively deterministic. Therefore either a source of truly random numbers or a cryptographically secure pseudo-random number generator is required. Another area in which randomness is inherent is quantum computing.

In the example above, the randomized algorithm always outputs the correct answer, but its running time is a random variable. Sometimes the algorithm to complete in a fixed amount of time (as a function of the input size), but allow a small probability of error. The former types are called Las Vegas algorithms, and the latter are Monte Carlo algorithms (related to the Monte Carlo method for simulation). Observe that any Las Vegas algorithm can be converted into a Monte Carlo algorithm (via Markov's inequality), by having its output, an arbitrary, possibly incorrect answer if it fails to complete within a specified time. Conversely, if an efficient verification procedure exists to check whether an answer is correct, then a Monte Carlo algorithm can be converted into a Las Vegas algorithm by running the Monte Carlo algorithm repeatedly till a correct answer is obtained.

Chapter 2 : Divide and Conquer [Total Marks - 15]

Q. 4(a) Write a Min Max function to find minimum and maximum value from given set of values using divide and conquer. Also drive its complexities. **(10 Marks)**

Ans. : Finding Min Max function :

Algorithm 1 : StraightMaxMin(a,n,max,min)

```
//set max to maximum and min to minimum of a[1:n]
{
max = min = a[1];
for i = 2 to n do
{
if(a[i]>max) then max = a[i];
if(a[i]<min) then min = a[i];
}
}
```

Algorithm 2 : MaxMin(i,j,max,min)

```
//a[1:n] is globalarray.i,j, are integers
// 1<= i <= j <= n, the effect is to set max and min to the largest and smallest
// values in a[i:j] respectively.
{
   if( i = j ) then max = min = a[i];
   else if( i = j - 1) then
   {
       if (a[i] <a[ j]) then
       {
           max = a[j]; min = a[ i];
       }
       else
       {
           max = a[i] ; min = a[j];
       }
   }
   else
   {  // if P is not small , divide P into subproblems
       // find where to split the set
       mid = (i + j) /2;
       // solve the subproblems
       MaxMin(i, mid, max, min);
```



```
       MaxMin(mid+1, j, max1, min1);
       //combine the solutions
       if (max < max1) then max = max1;
       if(min > min1) then min = min1;
   }
}
```

MaxMin is a recursive algorithm that finds the maximum and minimum of the set of elements {a[i],a[i+1],...,a[j]}. The situation of set sizes one (i = = j) and two (i = j – 1) are handled separately. For sets containing more than two elements, the midpoint is determined maxima and minima of these subproblems are determined, the two maxima are compared and the two minima are compared to achieve the solution for the entire set. The procedure is initially invoked by the statement.

MaxMin (1, n, x,y)

Suppose simulate MaxMin on the following nine elements

a	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	–5	33	78	60	10	8	–9	15	50

A good way of keeping track of recursive calls is to build a tree by adding node each time a new cal is made. For this algorithm each node has four items of information: i, j, max and min. On the array a[ ] above the tree of Fig. 4 is produced.

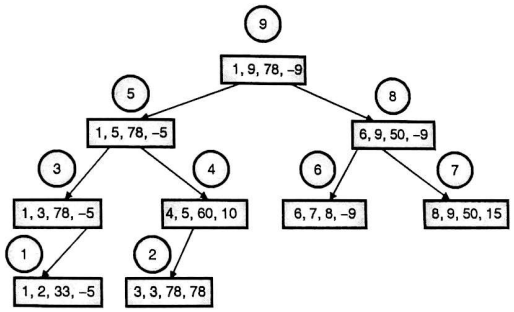


Fig. 4

Root node contains 1 and 9 as the value of i and j corresponding to the initial call to MaxMin. The execution produces two new calls to MaxMin where i and j have the values 1, 5 and 6, 9 respectively and thus split the set into two subsets of approximately the same size. Here maximum depth of recursion is four.

The numbers written in circle represent the orders in which max and min are assigned values. If T(n) represents the number of element comparisons, then the resulting recurrence relation is :

$$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + T(\lfloor n/2 \rfloor) + n & n > 2 \\ 1 & n = 2 \\ 0 & n = 1 \end{cases}$$

Where n is a power of two, $n = 2^k$ for some positive integer k then

$$T(n) = 2T(n/2) + 2$$
$$= 2(2T(n/4) + 2) + 2$$
$$= \ldots$$
$$= 2^{k-1} T(2) + \sum_{1 \le i \le k-1} 2^i$$
$$= 2^{k-1} + 2^k - 2 = 3n/2 - 2$$

$\left\lceil \frac{3}{2}n - 2 \right\rceil$ is the best average and worst-case number of comparisons when n is power of two. In terms of storage MaxMin is worse than the Straightforward algorithm because it requires stack space for i, j, max,min, max1, min1. The n elements, there will be $\log_2^{n+1}$ levels of recursion and to save seven values for each recursion calls.

Q. 4(b) Comment on model of computation. **(5 Marks)**

Ans. : Divide and conquer method :

Following are the steps in solving a problem in divide and conquer strategy :

1. Divide the problem into number of subproblms.
2. And then solve these subproblems recursively.
3. Integrate all the subproblems to give a result.

These three steps are coordinated together by the algorithm to compute the final output.

Many algorithms call themselves recursively one or more times to handle closely related subproblems. All of these algorithms follow a divide-and-conquer approach to get the output for the computation. They decompose the larger problem into number of subproblems or the small solvable unit and solve recursively and finally integrate the results of all these subproblems to obtain the final result. In short we conclude that the divide-and-conquer approach has basic three steps at each level of the recursion :

1. **Divide** the problem into a number of smaller units called subproblems.
2. **Conquer** (i.e solve) the subproblems recursively.
3. **Combine** the solutions of all the subproblems into a solution for the original problem.

Algorithm : Divide and Conquer algorithm

```
Algorithm DAndC (P)
{
if Small(P) then return S(P);
else
{
divide P into smaller instances P1, P2,..., Pk, k ≥ 1;
Apply DAndC to each of these subproblems;
return Combtne(DAndC(P1),DAndC(P2),.. .,DAndC(Pk ));
} }
```

Where P is the problem to be solved and Small(P) is a Boolean-valued function that determines whether the input size is small or not, so that answer can be computed without splitting. If it is small,

then the function S is invoked, otherwise the problem P is divided into smaller subproblems. Then these subproblems P_1 , P_2 ,.....,P_k are solved by recursive applications of DAndC (P). And finally with the help of function Combine, the solution to P is obtained.

Chapter 3 : Greedy Method [Total Marks - 20]

Q. 4(b) Comment on model of computation. **(5 Marks)**

Ans. : General Method (Greedy Method) :

The greedy strategy works in the form of stages i.e we can devise an algorithm that works in stages by considering only one input at a time. At the end of every stage it is checked that the given input is an optimal solution or not. This is done by taking all the inputs in some order. If the next input is included to the partially constructed optimal solution, then it will result in an infeasible solution. Hence this input is not added to the partial solution else it can be added. Thus Greedy algorithms build up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. Although such an approach can be disastrous for some computational tasks, there are many for which it is optimal.

Algorithm : Algorithm Greedy(c,n)

```
// c[1:n] contains the n inputs
{
    solution=0; //initialize the solution
    for i = 1 to n do
    {
        x = Select(c);
        if(feasible(solution, x) then
            solution = Union(solution, x);
    }
    return solution;
}
```

The function 'Select' selects as input from array c and assign to x. The feasible function checks whether x can be added to solution or not. 'Union' function adds x to solution.

Q. 5(a) To find Dijkstra's shortest path from vertex 1 to vertex 4 for following graph. **(10 Marks)**

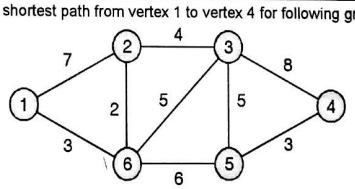


Fig. 5

Ans. :

The iterations of Dijkstra's algorithm are described in the following table.

S	L(1)	L(2)	L(6)	L(3)	L(5)	L(4)
ϕ	0	∞	∞	∞	∞	∞
{1}		7	3	∞	∞	∞
{1,2}			3	11	∞	∞
{1,2,6}				8	9	∞
{1,2,6,5}				14		12
{1,2,6,3}					13	16
{1,2,6,5,3}					13	16
{1,2,6,5,3,4}						

At the last iteration, $4 \in S$ and $L(4) = 12$. That the shortest path from 1 to 4 has a cost of 12.

Q. 6(a) Write note on : Job sequencing with deadlines **(5 Marks)**

Ans. : A set of n jobs. Associated with job i is an integer deadline $d_i > 0$ and a profit $p_i > 0$. For any job i the profit p_i is earned if the job is completed by its deadline. To complete a job, one has to process the job on a machine for one unit of time. Only one machine is available for processing jobs. A feasible solution for this problem is a subset J of jobs such that each job in this subset can be completed by its deadline. The value of a feasible solution J is the sum of the profits of the jobs in J, or $\sum_{i \in j} p_i$. An optimal solution is a feasible solution with maximum value. Here again, since the problem involves the identification of a subset, it fits the subset paradigm.

Example :

$n = 4$, $(p_1,p_2,p_3,p_4) = (100,10,15,27)$ and $(d_1,d_2,d_3,d_4) = (2,1,2,1)$. The feasible solutions and their values are:

Sr. No.	Feasible solution	Processing sequence	value
1.	(1,2)	2,1	110
2.	(1,3)	1, 3 or 3, 1	115
3.	(1, 4)	4,1	127
4.	(2,3)	2,3	25
5.	(3,4)	4,3	42
6.	(D)	1	100
7.	(2)	2	10
8.	(3)	3	15
9.	(4)	4	27

In the solutions, the solution 3 is an optimal one. In this solution only jobs 1 and 4 are processed and the value is 127. These jobs must be processed in the order i.e. "job 4 followed by job 1". Thus the processing of job 4 begins at time zero and that of job 1 is completed at time 2. To choose the next job, formulate an optimization measure.

It can choose the objective function $\sum_{i \in j} p_i$ as our optimization measure. Pairs are in non-decreasing order. Start adding the job with $J = 0$ and $\sum_{i \in j} p_i = 0$. Job 1 is added to J as it has the largest

easy solution

$= \min\{67, 2+3\}$

profit and thus $J = \{1\}$ is a feasible solution. Next, job 4 is considered. The solution $J = \{1,4\}$ is also feasible. The next job, Job 3 is considered and it is discarded as $J = \{1,3,4\}$ is not feasible. Finally, job 2 is added into J and it is discarded as $J = \{1,2,4\}$ is not feasible. That have the solution $J = \{1,4\}$ with value 127. This is the optimal solution for the given problem instance.

Algorithm 1 :

```
Algorithm GreedyJob(d, J, n)
// J is a set of jobs that can be completed by their deadlines.
{
J := {1};
for i := 2 to n do
{
if (all jobs in J ∪ {i} can be completed
by their deadlines) then J := J ∪ {i};
}
}
```

J be a set of k jobs and $\sigma = i_1, i_2 \ldots\ldots, i_k$ a permutation of jobs in J such that $d_{i1} \leq d_{i2} \ldots \leq d_{ik}$. Then J is feasible solution if and only if the jobs in J can be processed in the order σ without violating any deadline. This theorem determine whether all jobs in J U{i} can be completed by their deadlines or not. This is use an array d[1 : n] to store the deadlines of the jobs in the order of their p-values. The set J can be represented as J[1 : k] such that $J[r]$, $1 \leq r \leq k$ are jobs in J and $d[J[1]] \leq d[J[2]] \leq \ldots \leq d[J][k]]$. To test whether J U{i} is feasible that have to just insert i into J preserving the deadline ordering and then verify that $d[[r]] \leq r$, $1 \leq r \leq k+1$. The insertion of i into J is simplified by the use of fictitious job 0 with $d[0] = 0$ and $J[0] = 0$. If job i is to be inserted at position 9, then only the positions of jobs J[9], J[9 + 1], ...J[k] are changed after the insertion. Hence it is necessary to verify only that these jobs do not violate their deadlines following the insertion.

Algorithm 2 :

```
Algorithm JS(d, j, n)
// d[i] ≥ 1, 1 ≤ i ≤ n are the deadlines, n ≥ 1.
//jobs are ordered P[1] ≥ P[2] ≥ .....[n].
// J[i] is the ith job in the optimal solution.
// 1 ≤ i ≤ k at termination d[J[i]] ≤ d[J[i + 1]], 1 ≤ i < k
{
d[0] = J[0] = 0;
J[1] = 1;
k = 1;
for i = 2 to n do
{
```

easy solution

$C(2,5) + 1$

```
r = k
while((d[J[r]] > d[i]) and (d[J[r]] ≠ r)) do r = r – 1;
if((d [J[r]]) ≤ d[i]) and (d[i] > r)) then
{
// insert i into J[]
for q = k to (r + 1) step - 1 do J[q + 1] = J[q];
J[r + 1] = i;
k = k + 1;
}
}
return k;
}
```

Chapter 4 : Dynamic Programming [Total Marks - 30]

Q. 1(b) Explain 0/1 Knapsack problem with example. **(10 Marks)**

Ans. ; 0/1 Knapsack problem :

A knapsack with maximum capacity W, and a set S consisting of n items Each item i has some weight w_i and benefit value b_i (all w_i , b_i and W are integer values).

Problem : How to pack the knapsack to achieve maximum total value of packed items?

0/1 Knapsack problem: A picture.

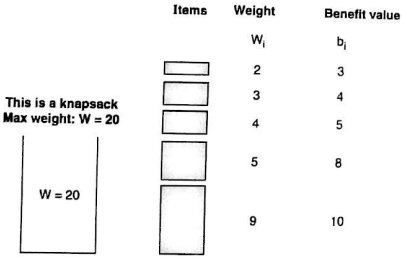


Fig. 6

Problem, in other words, is to find $\max \sum_{i \in T} b_i$ subject to $\sum_{i \in T} w_i \leq W$

The problem is called a "0/1" problem, because each item must be entirely accepted or rejected. In the "Fractional Knapsack Problem," take fractions of items.

0/1 Knapsack problem : Brute-force approach

Solve this problem with a straightforward algorithm. Since there are *n* items, there are 2^n possible combinations of items. Through all combinations and find the one with maximum value and with total

easy solution

= min { 6+7, 2+3 }

weight less or equal to W. Running time will be $O(2^n)$. With an algorithm based on dynamic programming. This need to carefully identify the subproblems. If items are labeled 1..n, then a subproblem would be to find an optimal solution for S_k = {items labeled 1, 2, .. k}

Defining a subproblem :

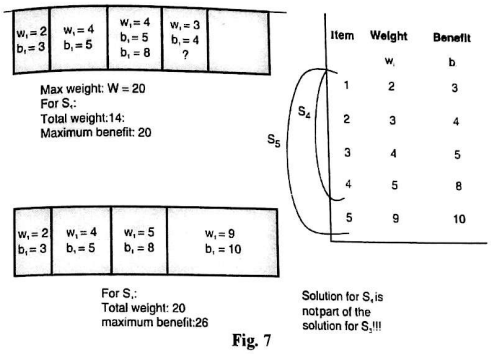


Fig. 7

If items are labeled 1..n, then a subproblem would be to find an optimal solution for S_k = {items labeled 1, 2, .. k}. This is a reasonable subproblem definition. The question is : that can describe the final solution (S_n) in terms of subproblems (S_k). Unfortunately, cannot do that.

The solution for S_4 is not part of the solution for S_5 So definition of a subproblem is flawed and need another one! Add another parameter: w, which will represent the exact weight for each subset of items. The subproblem then will be to compute B[k,w].

Recursive formula for subproblems :

$$B[k, w] = \begin{cases} B[k-1, w] & \text{if } w_k > w \\ \max \{B[k-1, w], [k-1, w-w_k] + b_k\} & \text{otherwise,} \end{cases}$$

It means, that the best subset of S_k that has total weight w is:

(1) The best subset of S_{k-1} that has total weight w, **or**

(2) The best subset of S_{k-1} that has total weight w-w_k plus the item k

The best subset of S_k that has the total weight w, either contains item k or not.

First case : $w_k > w$. Item k cannot be part of the solution , since if it was, the total weight would be w, which is unacceptable.

Second case : $w_k \leq w$. Then the item k can be in the solution, and choose the case with greater.

Algorithm : Knapsack algorithm

```
for w = 0 to W
B[0,w] = 0
for i = 1 to n
```

easy solution

```
B[i,0] = 0
for i = 1 to n
for w =  0 to W
if wi < = w // item i can be part of the solution
if b i + B[i–1,w-w i] > B[i–1,w]
B[i,w] = bi + B[i–1,w– w i ]
else
B[i,w] = B[i–1,w]
else B[i,w] = B[i–1,w] // w i > w
```

Running time

```
for w = 0 to W  O(W)
B[0,w] = 0
for i = 1 to n
B[i,0] = 0
for i = 1 to n Repeat n times
for w = 0 to W O(W)
< the rest of the code >
```

Running time of this algorithm?

$O(n*W)$

The brute-force algorithm takes $O(2^n)$

Example : The Knapsack Problem : Greedy Vs. Dynamic. The fractional problem can be solved greedily. The 0–1 problem can be solved with a dynamic programming approach.

Q. 3(a) Explain all pair shortest path algorithm with suitable example. **(10 Marks)**

Ans. : All pair shortest path algorithm :

Consider a directed graph G = (U,E) with n vertices. The cost is considered as the cost adjacency matrix for G such that cost (i,i) = 0 for $1 \le i \le n$. Then cost (i,j) is the length of edge $(i,j) \in E(G)$ and cost (i,j) = ∞, for $i \ne j$ and $(i,j) \notin E(G)$.

The main motivation behind all - pairs shortest path problem is to determine a matrix A such that A(i,j) is the length of a shortest path from i to j. The only condition to be satisfied by the graph G is that it should not contain any cycle with negative lengths. If cycles with negative lengths are allowed in the graph G, then the shortest path between any two vertices has length $-\infty$.

The shortest path from i to j in G for $i \ne j$ is determined as follows :

The path begins at vertex i and terminates at vertex j through intermediate vertices. Let us assume that this path contains no cycles. If k is an intermediate vertex on this shortest path, then the subpaths from i to k and from k to j must be shortest paths from i to k and k to j respectively. Let $A^k(i,j)$ represent length of a shortest path from i to j , going through no vertex of index greater than k, obtain :

$$A(i,j) = \min\{ \min_{1 \le k \le n}\{A^{k-1}(i,k) + A^{k-1}(k,j)\}, \text{cost}(i,j) \}$$

$$A^0(i,j) = \text{cost}(i,j),\ 1 \le i \le n,\ 1 \le j \le n$$

It can obtain a recurrence for $A^k(i,j)$. A shortest path from i to j going through no vertex higher than k either goes through vertex k or it does not. If it does $A^k(i,j) = A^{k-1}(i,k) + A^{k-1}(k,j)$. If it does not, then no intermediate vertex has index greater than k – 1.

easy solution

= min { 67, 2+3 }

Hence $A^k(i,j) = A^{k-1}(i,j)$ combining then get :

$$A^k(i,j) = \min\{ A^{k-1}(i,j), A^{k-1}(i,k) + A^{k-1}(k,j) \},\ k \ge 1$$

Above equation is not true for graphs with cycles of negative length.

Algorithm:

```
Algorithm All Paths (cost, A, n)
// cost [1 : n, 1 : n] is the cost adjacency matrix of a graph with
// n vertices, A [i,j] is the cost of a shortest path from vertex
// i to vertex j. cost [i,j] = 0.0, for 1 ≤ i ≤ n
{
        for i = 1 to n do
        for j = 1 to n do
        A [i,j] = cost [i,j];       // copy cost into A
        for k = 1 to n do
        for i = 1 to n do
        for j = 1 to n do
        A [i,j] = min (A[i,j], A[i,k] + A[k,j]) ;
}
```

Time required for above path is $\theta(n3)$

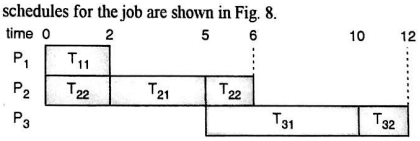
Q. 5(b) Explain flow shop scheduling with example **(10 Marks)**

Ans. : Flow shop scheduling :

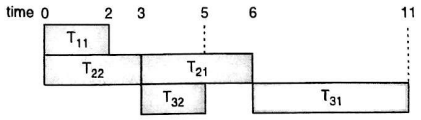
There are use n jobs each requiring m tasks $T_{1i}, T_{2i}, \ldots, T_{mi}$, $1 \le i \le n$ to be performed. Task T_{ji} is to be performed on processes P_j, $1 \le j \le m$. The time required to complete task T_{ji} is t_{ji}. No processor may have more than one task assigned to it in any time interval. Processing of T_{ji}, $j > 1$ cannot be started until task $T_j - 1$, i has been completed. For example, two jobs have to be scheduled on three processors. The task times are given by the matrix J.

$$J = \begin{bmatrix} 2 & 0 \\ 3 & 3 \\ 5 & 2 \end{bmatrix}$$

Two possible schedules for the job are shown in Fig. 8.



(a) Preemptive schedule



(b) Non-preemptive schedule

Fig. 8

easy solution

Chapter 5 : Backtracking [Total Marks - 15]

Q. 2(a) Write an algorithm of sum of subsets. Solve following problem and draw portion of state space tree. M = 35, W = (5, 7, 10, 12, 15, 18, 20). **(10 Marks)**

Ans. :

Algorithms of sum of subset :

Algorithm :

```
Algorithm sumofsub (s, k, r)
{
    //generate left child,
if (s + w [k] = m) then write (x(1 : k)); //subset found
else if (s + w [k] + w [k + 1] ≤ m)
then sumofsub (s + w [k], k + 1, r – w[k]);
//generate right child and evaluate Bk
If ((s + r – w[k] ≥ m) and (s + w [k + 1)] ≤ m)) then
{
    x[k] = 0;
    sumofsub (s, k + 1, r – w [k]);
}
}
```

The values of s, k and r are listed by the rectangular nodes on each of the calls to sumofsub. Circular nodes represent points at which subsets with sums m are printed out.

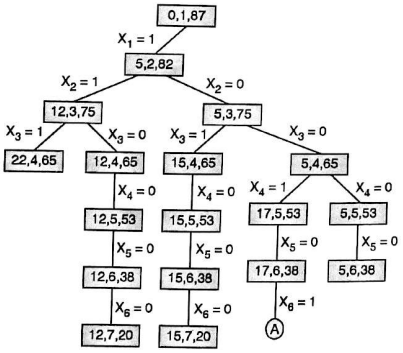


Fig. 9(a) : Left subtree of tree

= min { 67, 243 }

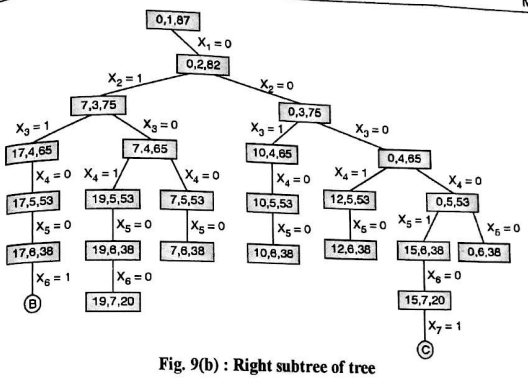


Fig. 9(b) : Right subtree of tree

At nodes A, B and C the output is respectively.

A (1, 0, 0, 1, 0, 1) , B (0, 1, 1, 0, 0, 1), C (0, 0, 0, 0, 1, 0, 1)

Q. 6(d) Write note on : N-queen problem. **(5 Masks)**

Ans. :

N-queen problem :

Solve the 8-queens problem via a backtracking solution.

Let the chessboard squares is numbered as the indices of the two-dimensional array a[1 : n, 1 : n], then every elements on the same diagonal that runs from the upper left to lower right has the same row – column value.

Also every element on the same diagonal that goes from the upper right to lower left has the same row + column value. Suppose two queens are placed at positions (i, j) and (k, l), then by the above they are on the same diagonal only if $i - j = k - l$ or $i + j = k + l$

The first equation implies

$j - l = i - k$

The second implies

$j - l = k - i$

Thus the two queens lie on the same diagonal if and only if $|j - l| = |i - k|$

Algorithm 1 : Can a new queen be placed?

```
Algorithm Place(k, i)
// Returns true if a queen can be placed in k^th row and i^th column. Otherwise it
// returns
false. x[ ] is a global array whose first (k – 1) values have been set.
// Abs(r) returns the absolute value of r.
{
```

easy solution

```
    for j:= 1 to k – 1 do
        if (x[j] = i)   // Two in the same column
            or (Abs(x[j] – i) = Abs(j – k))
            // or in the same diagonal
        then return false;
    return true;
    }
```

Above algorithm place (k, i) returns a Boolean value that is true if the k^{th} queen can be placed in column i. It tests both whether i is distinct from all previous values x [1],……,x[k – 1] and whether there is no other queen on the same diagonal. Its computing time is O (k – 1).

The array x [ ] is a global. The algorithm is invoked by NQueens (1, n).

Algorithm 2 : All solutions to the n-queens problem.

```
Algorithm NQueens(k, n)
// Using backtracking, this procedure prints all possible placements of n queens on
// an n × n chessboard so that they are nonattacking.
{
for i := 1 to n do
{
if Place(k,i) then
{
        x[k] := i;
        if (k = n) then write (x[l : n]);
        else NQueens(k + l, n);
} } }
```

Chapter 6 : String Matching Algorithms [Total Marks – 25]

Q. 2(b) Explain longest common subsequence with example. **(10 Marks)**

Ans. :

Longest Common Subsequence :

The longest common subsequence (LCS) is as implied by the phase is a subsequence of two strings; $X = x_0\, x_1\, x_2 \ldots x_{n-1}$ and $Y = y_0\, y_1 \ldots y_{m-1}$ such it is the longest. Note the LCS is not necessarily unique. A current research topic in bio-informatics. DNA are represented as strings of the small alphabet, Sigma = {A, C, G, T}. DNA strands/strings can be changed by mutation by insertion of a character into the string. A longest common subsequence of two strings can represent the common ancestry of the two strings.

Finding the LCS :

This is enumerate all the subsequences in X and then start with the longest check if it is a subsequence of Y. Because a character in X can either be a member or not a member of the subsequence the number of subsequence is 2n. Checking if the sequence is a subsequence of Y takes O(m). So worst case running time is $O(m2^n)$. This is called exponential time, to be avoided at all cost even at the cost of more space.

easy solution

The LCS algorithm :

Algorithm: LCS(X, Y) :

```
Input: Strings X and Y with n and m elements
Output: The table L[i, j]  the LCS for the sub problems x0x1...xi and y0y1...yj,
for i = -1 to n-1 do
    L[i, -1] = 0
for j = 0 to m -1 do
    L[-1, j] = 0
for i = 0 to n-1 do
    for j = 0 to m-1 do
if xi = yj then
L[i, j]  = L[i-1, j-1] +1
else
L[i, j] = max{ L[i-1, j], L[i, j-1]}
return L
```

Q. 3(b) Explain different string matching algorithms. **(10 Marks)**

Ans. : The Naive String Matching Algorithm :

Algorithm : Naive_String_Matcher (T, P)

```
    n • length [T]
    m • length [P]
for s • 0 to n - m do
if P[1 . . m] = T[s +1 . . s + m]
    then return valid shift s
```

The naive string-matching procedure can be interpreted graphically as a sliding a pattern P[1 . . m] over the text T[1 . . n] and noting for which shift all of the characters in the pattern match the corresponding characters in the text. In other to analysis the time of naïve matching, it would like to implement above algorithm to understand the test involves in line 4. Note that in this implementation, use notation P[1 . . j] to denote the substring of P from index i to index j.

That is, P[1 . . j] = P[i] P[i +1] . . . P[j].

Algorithm: Naive_String_Matcher (T, P)

```
    n • length [T]
    m • length [P]
    for s • 0 to n-m do
    j • 1
    while j • m and T[s + j] = P[j] do
    j • j +1
```

easy solution

```
    If j > m then
        return valid shift s
    return no valid shift exist    // i.e., there is no substring of T matching P.
```

The Rabin Karp algorithm :

Algorithm: Rabin-Karp String Matching

```
    Compute hp (for pattern p)
    Compute ht (for the first substring of t with m length)
    for i= 1 to n – m
    if hp = ht
    match t[i ....i + m] with p, if matched return 1
    else
    ht =(d(ht – t[i+1].dm-1) + t[m+i+1]) mod q
    end
```

Suppose t = 2359023141526739921 and p= 31415,

Now, hp = 7(31415 = 7(mod 13))

Substring beginning at position 7 = valid match

The Knuth-Morris-Pratt algorithm :

Knuth-Morris-Pratt Failure (P) :Pattern with m characters

```
Input   :   Pattern with m characters
Output  :   Failure function f for P[i . . j]
            i ← 1
            j ← 0
            f(0) ← 0
            while i < m do
            if P[j] = P[i]
                    f(i) ← j + 1
                i ← i + 1
                j← j + 1
            else if
                j ← f(j - 1)
            else
                f(i) ← 0
                i ← i + 1
```


Chapter 7 : Branch and Bound [Total Marks – 10]

Q. 6(c) Write note on : The 15 puzzle problem **(10 Marks)**

Ans. : The 15-puzzle problem :

The 15-puzzle consists of 15 numbered tiles on a square frame with a capacity of 16 tiles (Fig. 10). This are given an initial arrangement of the tiles, and the objective is to transform the arrangement into the goal arrangement of Fig. 10(b) through a series of legal moves The only legal moves are ones in which a tile adjacent to the empty spot (ES) is moved to ES. Thus from the initial arrangement of Fig. 10(a), four moves are possible. It can move any one of the tiles numbered 2, 3, 5, or 6 to the empty spot. Depending on these moves, other moves can be made. Every new move creates a new arrangement of the tiles. These arrangements of the tiles are called as the states of the puzzle.

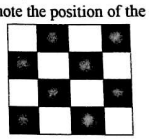
The initial arrangements and final arrangements are called as the initial states and goal states respectively. A state is reachable from the initial state if and only if there is a sequence of legal moves from the initial state to this state. The state space of an initial state consists of all states that can be-reached from the initial state. The most straightforward way to solve the puzzle would be to search the state space for the goal state and use the path from the initial state to the goal state as the answer. It is easy to see that there are 16! (16! ≈ 20.9 x 10^{12}) different arrangements of the tiles on the frame. Of these only one-half are reachable from any given initial state. Indeed, the state space for the problem is very large. Before attempting to search this state space for the goal state, it would be worthwhile to determine whether the goal state is reachable from the initial state. There is a very simple way to do this. Let us number the frame positions 1 to 16. Position *i* is the frame position containing tile numbered *i* in the goal arrangement of Fig. 10(b). Position 16 is the empty spot. Let position(i).be the position number in the initial state of the tile numbered *i*. Then position(16) will denote the position of the empty spot.

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

(a) An arrangement

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

(b) Goal arrangement



(c)

Fig. 10 : 15-puzzle arrangements

For any state let less(i) be the number of tiles j such that j < i and position(j) > position{4). For the state of Fig. 10(a), for example, less(l) = 0, less(4) = 1, and less(12) = 6. Let x:=l if in the initial state the empty spot is at one of the shaded positions of Fig. 10(c) and x = 0 if it is at one of the remaining positions.

❑❑❑