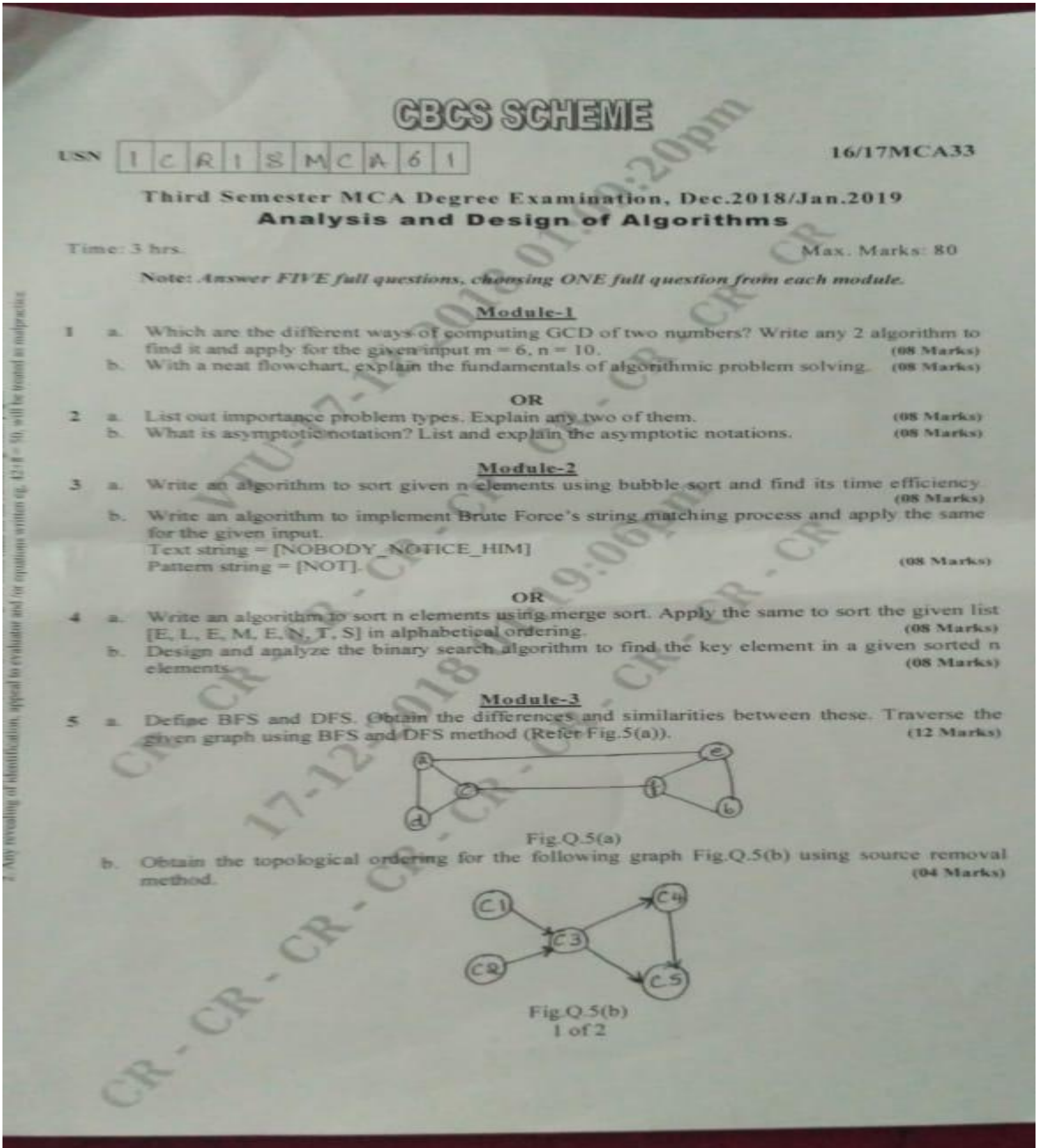


CBCS SCHEME

USN 1CR18MCA61 16/17MCA33

Third Semester MCA Degree Examination, Dec.2018/Jan.2019

Analysis and Design of Algorithms

Time: 3 hrs. Max. Marks: 80

Note: *Answer FIVE full questions, choosing ONE full question from each module.*

Module-1

1 a. Which are the different ways of computing GCD of two numbers? Write any 2 algorithm to find it and apply for the given input m = 6, n = 10. (08 Marks)

b. With a neat flowchart, explain the fundamentals of algorithmic problem solving. (08 Marks)

OR

2 a. List out importance problem types. Explain any two of them. (08 Marks)

b. What is asymptotic notation? List and explain the asymptotic notations. (08 Marks)

Module-2

3 a. Write an algorithm to sort given n elements using bubble sort and find its time efficiency. (08 Marks)

b. Write an algorithm to implement Brute Force's string matching process and apply the same for the given input.
Text string = [NOBODY_NOTICE_HIM]
Pattern string = [NOT]. (08 Marks)

OR

4 a. Write an algorithm to sort n elements using merge sort. Apply the same to sort the given list [E, L, E, M, E, N, T, S] in alphabetical ordering. (08 Marks)

b. Design and analyze the binary search algorithm to find the key element in a given sorted n elements. (08 Marks)

Module-3

5 a. Define BFS and DFS. Obtain the differences and similarities between these. Traverse the given graph using BFS and DFS method (Refer Fig.5(a)). (12 Marks)

Fig.Q.5(a)

b. Obtain the topological ordering for the following graph Fig.Q.5(b) using source removal method. (04 Marks)

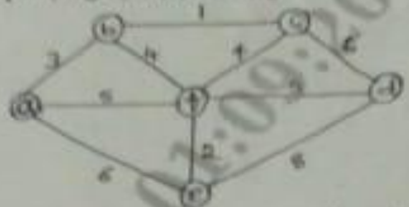
Fig.Q.5(b)

1 of 2

OR

6 a. Find the MST for the given graph (Fig.Q.6(a)) using Kruskal's algorithm. (05 Marks)

Fig.Q.6(a)



b. Write an algorithm to find the single-source shortest path using Dijkstra's algorithm. (05 Marks)

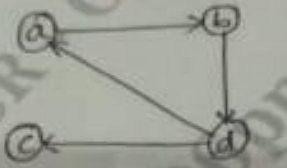
c. Find the Huffman code for the following data by obtaining Huffman tree: (06 Marks)

Character	A	B	C	D	E
Probability	0.35	0.1	0.2	0.2	0.15

Module-4

7 a. Write an algorithm to compute transitive closure / path matrix for the given graph. And obtain the transitive closure for the given graph shown in Fig.Q.7(a) using Warshall's algorithm. (08 Marks)

Fig.Q.7(a)



b. Find the optimal solution for the given Knapsack instance using 0/1 Knapsack method with capacity M = 5. (08 Marks)

Item	1	2	3	4
Weight	2	1	3	2
Value/profit	12	10	20	15

OR

8 a. Write an algorithm to sort given n elements using distribution counting method. Apply the same for the following input: [13, 11, 12, 13, 12, 12]. (08 Marks)

b. Explain Horspool's string matching algorithm with a suitable example. (08 Marks)

Module-5

9 a. What is decision tree? Obtain the decision tree to find minimum of 3 numbers. (08 Marks)

b. Explain N-Queens problem using back-tracking method. (08 Marks)

OR

10 a. Construct the state-space tree for the sum of subset problem for the given data: W={5, 10, 12, 13, 15, 18} , and M = 30 (08 Marks)

b. Find the optimal solution for the given assignment problem which is represented as a matrix as show below:

	J1	J2	J3	J4
a	9	2	7	8
b	6	4	3	7
c	5	8	1	8
d	7	6	9	4

(08 Marks)

* * * * *

CMR
INSTITUTE OF
TECHNOLOGY

Internal Assessment Test 1 – September 2018

Sub:	Design and Analysis of Algorithms							**Code:**	17MCA33
Date:	*14-12-2018*	*Duration:*	*3hrs*	*Max Marks:*	*100*	*Sem:*	**III Sem**	*Branch:*	*MCA*

Dr. Vakula Rani

Answer any five of the following **100 Marks**

Q1(a) Which are the different ways to compute GCD of two numbers ? Write any two algorithms to find it and apply for the given input m=6 , n=10

Def : An algorithm is a sequence of unambiguous instructions for solving a problem. i.e., for obtaining a required output for any legitimate input in a finite amount of time.

Figure : Notion of the Algorithm

Method -1

Below is given the psuedocode of the algorithm

Euclid's algorithm to find the GCD of two numbers is based on applying repeatedly the equality
gcd(m, n) = gcd(n, m mod n), where m>n
where m mod n is the remainder of the division of m by n, until m mod n is equal to 0. Since gcd(m, 0) = m , the last value of m is also the greatest commondivisor of the initial m and n.

For example, gcd(6, 10) can be computed as follows:
m<n hence swap the values of m and n
gcd(10,6) = gcd(6, 4) = gcd(4, 2) =gcd(2,0) = 2

Euclid's algorithm *for computing gcd(m, n)*
Step 1 *If n = 0, return the value of m as the answer and stop; otherwise, proceed to Step 2.*
Step 2 *Divide m by n and assign the value of the remainder to r.*
Step 3 *Assign the value of n to m and the value of r to n. Go to Step 1.*

```
ALGORITHM Euclid(m, n)
//Computes gcd(m, n) by Euclid's algorithm
//Input: Two nonnegative, not-both-zero integers m and n
//Output: Greatest common divisor of m and n
while n _= 0 do
r ←m mod n
m←n
n←r
return m
```

Method -2

Middle-school procedure for computing gcd*(m, n)*

Step 1 Find the prime factors of *m*.
Step 2 Find the prime factors of *n*.
Step 3 Identify all the common factors in the two prime expansions found in
Step 1 and Step 2. (If *p* is a common factor occurring *pm* and *pn* times in *m* and *n*, respectively, it should be repeated min{*pm*, *pn* } times.)
Step 4 Compute the product of all the common factors and return it as the
greatest common divisor of the numbers given.

Thus, for the numbers 6 and 10,
we get
6 = 2 * 3
10 = 2 * 5
gcd*(6, 10)* = 2

Q 1 (b) With a neat flow chart , explain the fundamentals of algorithmic problem Solving

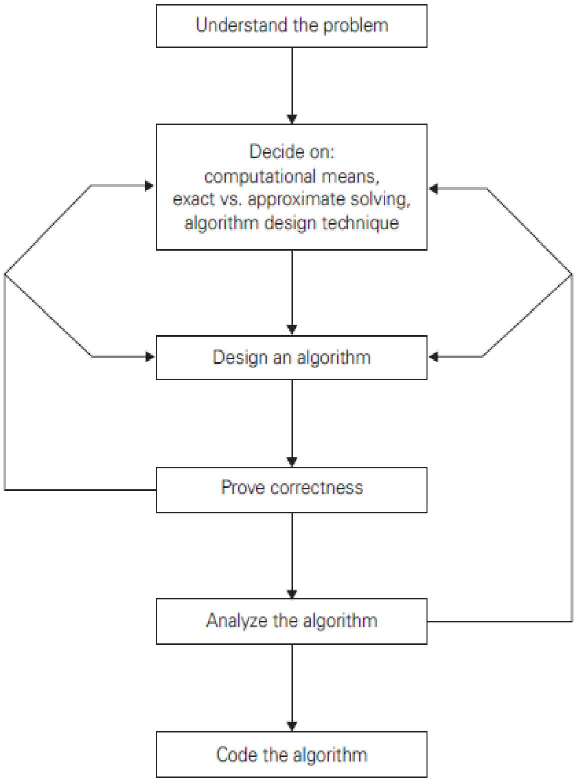


Fig : Algorithm Design and Analysis Process

1. **Understanding the problem:**

Before designing algorithm, one should understand the problem correctly. This may require the problem to be read multiple times, asking questions if required and working out smaller instances of problem by hand. Any input to an algorithm specifies an instance or event of the problem. So, it is very important to set the range of inputs so that the algorithm works for all legitimate inputs i.e work correctly under all circumstances..

2. **Ascertaining the capabilities of a computational Device:**

After understanding the problem, one must think of the machines that execute instructions. The machines that are capable of executing the instructions one after the other is known as sequential machines and algorithms which run on these machines are known as sequential algorithms Newer machines can run instructions concurrently re known as parallel machines and algorithms which have written for such machines are called parallel algorithms.

If we are dealing with the small problems, we need not worry about the time and memory requirements. But some complex problems which involve processing large amounts of data in real time are required to know about the time and memory requirements where the program is to be executed on the machine.

3. **Choosing between exact and approximate problem solving:**

   The algorithms which solves the problem and gives the exact solution is known as Exact Algorithm and one which gives approximate results is known as Approximation Algorithms.

   There are two situations in which we may have to go for approximate solution:

   i) If the quantity to be computed cannot be calculated exactly. For example finding square roots, solving non linear equations etc.

   ii) Complex algorithms may have solutions which take an unreasonably long amount of time if solved exactly. In such a case we may opt for going for a fast but approximate solution.

4. **Deciding on data structures:**

   Algorithms use different data structures for their implementation. Some use simple ones but some other may require complex ones. But, Data structures play a vital role in designing and analyzing the algorithms.

5. **Algorithm Design Techniques:**

   An algorithm design technique is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing. These techniques will provide guidance in designing algorithms for new problems. Various design methods for algorithms exist, some of which are – divide and conquer, dynamic programming, greedy algorithms etc.

6. **Methods of specifying an Algorithm:**

   Algorithm can be specified using natural language and psuedocode. Due to the inherent ambiguity of the natural language, the most prevelant method of specifying an algorithm is using psuedocode.

7. **Proving an Algorithm's correctness:**

Correctness has to be proved for every algorithm. To prove that the algorithm gives the required result for every legitimate input in a finite amount of time. For some algorithms, a proof of correctness is quite easy; for others it can be quite complex. Mathematical Induction is normally used for proving algorithm correctness.

8. **Analyzing an algorithm:**

Any Algorithm must be analysed for its efficiency time and space . Time efficiency indicates how fast the algorithm runs; space efficiency indicates how much extra memory the algorithm needs. Another desirable characteristic is simplicity. A code which is simple reduces the effort in understanding and writing it and thus leads to less chances of error. Another desirable characteristic is generality. An algorithm can be general if it addresses a more general form of the problem for which the algorithm is to be designed and is able to handle all legitimate inputs.

9. **Coding an algorithm:**

Programming the algorithm by using some programming language. Formal verification by proof is done for small programs. Validity of large and complex programs is done through testing and debugging.

(2)(a)List out important problem types. Explain any two of them .

The different problem types are :

1. ***Sorting***
2. Searching
3. String processing
4. Graph problems
5. Combinatorial problems
6. Geometric problems
7. Numerical problems.

1. Sorting:

Sorting problem is one which rearranges the items of a given list in ascending order. When sorting a record the field on which the sorting is performed is called a key. There are different types of sorting algorithms. There are some algorithms that sort an arbitrary of size n using nlog2n comparisons, On the other hand, no algorithm that sorts by key comparisons can do better than that. Algorithms for sorting vary from each other in their simplicity and efficiency. Although some algorithms are better than others, there is no algorithm that would be the best in all situations.

In addition there are two more desirable properties that a sorting algorithm may possess. The first is called stable, if it preserves the relative order of any two equal elements in its input. The second is said to be 'in place' if it does not require extra memory.

2. **Searching**:

The searching problem deals with finding a given value, called a search key, in a given set. The searching can be either linear search or binary search algorithm. These algorithms play a important role in real-life applications because they are used for storing and retrieving information from large databases. Searching, mainly deals with addition and deletion of records. In such cases, the data structures and algorithms are chosen to balance among the required set of operations.

3.String processing:

A String is a sequence of characters. Most common examples are text strings, which consists of letters, numbers and special characters. Bit strings consist of zeroes and ones. The most important problem is the string matching, which is used for searching a given word in a text. For e.g. Boyer-Moore algorithm and brute- force string matching algorithms.

4. Graph problems:

A graph is a collection of points called vertices which are connected by line segments called edges. Graphs are used for modeling a wide variety of real-life applications such as transportation and communication networks.

Graph problems include graph traversal, shortest-path and topological sorting algorithms. Some graph problems are very hard, only very small instances of the problems can be solved in realistic amount of time even with fastest computers such as traveling salesman problem, for finding the shortest tour through n cities that visits every city exactly once and the graph-coloring problem for assigning the smallest number of colors to vertices of a graph so that no two adjacent vertices are of the same color.

5.Combinatorial problems:

These are problems that ask us to find a combinatorial object such as permutation, combination or a subset that satisfies certain constraints and may have some desired properties (e.g. maximizes a value or minimizes a cost). E.g. TSP and finding subsets of a set. Most of these problems have algorithm which have a complexity having very high rate of growth and for most of them no efficient algorithm has been devised till date.

6.Geometric problems:

Geometric algorithms deal with geometric objects such as points, lines and polygons. triangles, circles etc. The applications for these algorithms are in computer graphic, robotics etc. E.g. closest-pair problem, given 'n' points in the plane, find the closest pair among them and convex-hull problem to find the smallest convex polygon that would include all the points of a given set.

7.Numerical problems:

These problems involve mathematical objects of continuous nature: solving equations computing definite integrals and evaluating functions and so on. These problems can be solved only approximately because of limited precision that can be offered by machine and can also lead to an accumulation of round-off errors. The algorithms designed are mainly used in scientific and engineering applications.

Q2.(b) What is asymptotic notation. List and Explain any two of them .

Asymptotic notations are the mathematical notations to express time and space complexity. It represents the runnning time of an algorithm.

Different Notations

1. Big oh Notation
2. Omega Notation
3. Theta Notation

1. Big oh (O) Notation : A function t(n) is said to be in O[g(n)], $t(n) \in O[g(n)]$, if t(n) is bounded above by some constant multiple of g(n) for all large n ie.., there exist some positive constant c and some non negative integer no such that $t(n) \leq cg(n)$ for all $n \geq no$.

Eg. t(n)=100n+5 express in O notation

$100n+5 <= 100n + n$ for all $n>=5$

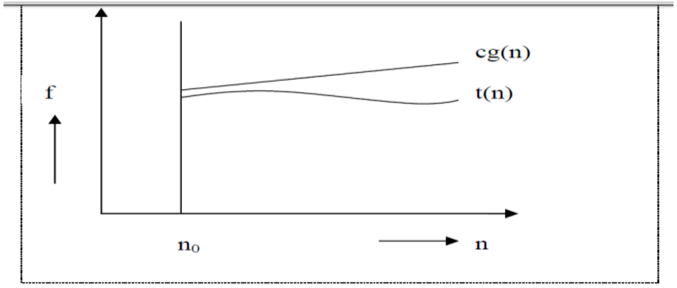
$<= 101 (n2)$

Let g(n)= n2 ; n0=5 ; c = 101

i.e $100n+5 <=101 n2$

$t(n) <= c* g(n)$ for all $n>=5$

There fore , $t(n) \in O(n2)$



2. Omega(Ω) -Notation:

Definition: A function t(n) is said to be in Ω[g(n)], denoted $t(n) \in \Omega[g(n)]$, if t(n) is bounded below by some positive constant multiple of g(n) for all large n, ie., there exist some positive constant c and some non negative integer n0 such that

$t(n) \geq cg(n)$ for all $n \geq n0$.

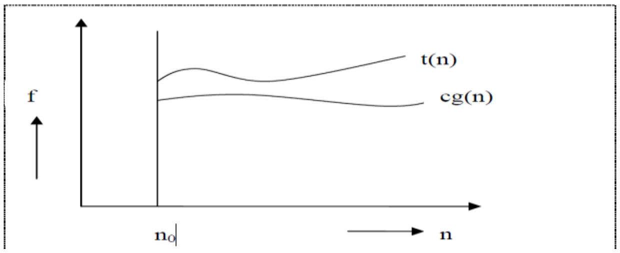
For example:

$t(n) = n3 \in \Omega(n2)$,

$n3 \geq n2$ for all $n \geq n0$.

we can select, g(n)= n3 , c=1 and n0=0

$t(n) \in \Omega(n2)$,



3. Theta (θ) - Notation:

Definition: A function t(n) is said to be in θ [g(n)], denoted t(n) ∈ θ (g(n)), if t(n) is bounded both above and below by some positive constant multiples of g(n) for all large n , ie., if there exist some positive constant c1 and c2 and some nonnegative integer n0 such that c2g(n) ≤ t(n) ≤ c1g(n) for all n ≥ n0.

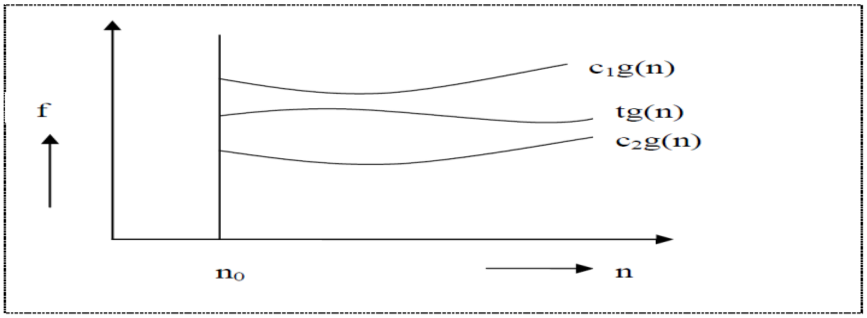
For example 1:

t(n)=100n+5 express in θ notation

100n <= 100n+5 <= 105n for all n>=1

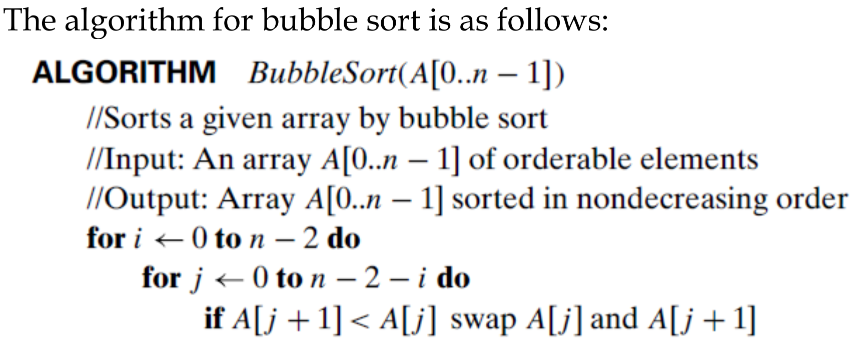
c1=100; c2=105; g(n) = n;

Therefore , t(n) ∈ θ (n)



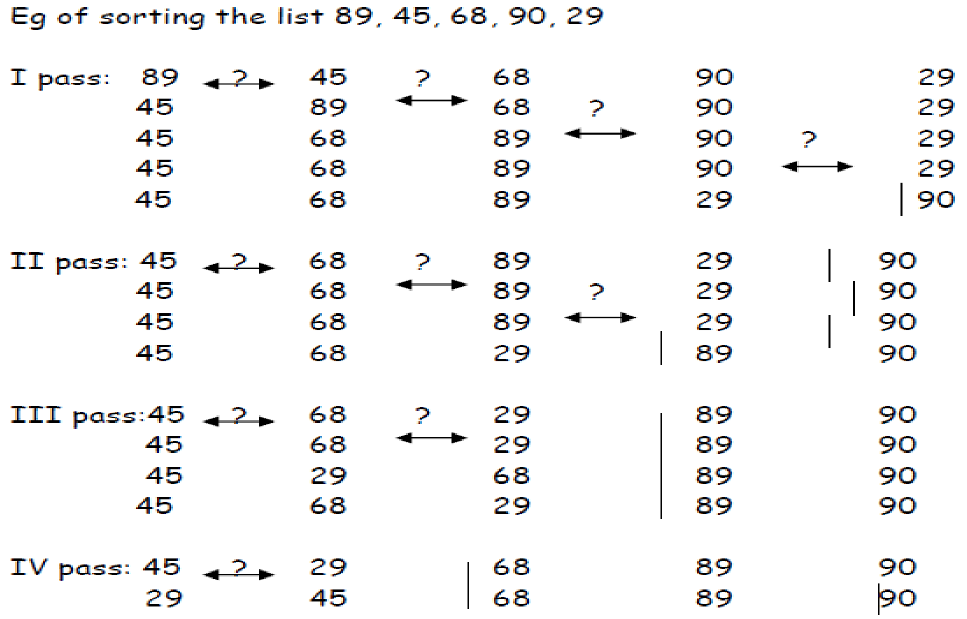
Q3.(a) Write an algorithm to sort given n elements using bubble sort and find its time complexity.

Sol :



The algorithm for bubble sort is as follows:

```
ALGORITHM BubbleSort(A[0..n − 1])
    //Sorts a given array by bubble sort
    //Input: An array A[0..n − 1] of orderable elements
    //Output: Array A[0..n − 1] sorted in nondecreasing order
    for i ← 0 to n − 2 do
        for j ← 0 to n − 2 − i do
            if A[j + 1] < A[j] swap A[j] and A[j + 1]
```

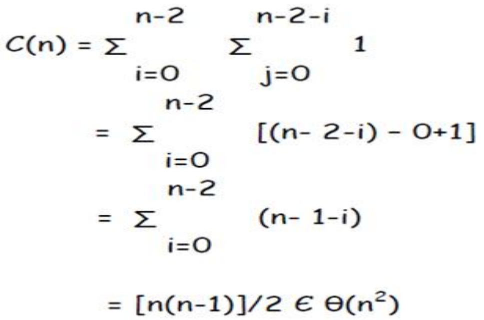


Eg of sorting the list 89, 45, 68, 90, 29

```
I pass:   89 <-?->  45   ?    68          90           29
          45        89  <-->  68    ?     90           29
          45        68        89  <-->    90     ?     29
          45        68        89          90   <-->    29
          45        68        89          29         | 90

II pass:  45 <-?->  68   ?    89          29     |   90
          45        68  <-->  89    ?     29       | 90
          45        68        89  <-->    29     |   90
          45        68        29        | 89         90

III pass: 45 <-?->  68   ?    29        | 89         90
          45        68  <-->  29        | 89         90
          45        29        68        | 89         90
          45        68        29        | 89         90

IV pass:  45 <-?->  29      | 68          89         90
          29        45      | 68          89        |90
```

Analysis:

The no of key comparisons is the same for all arrays of size n, it is obtained by a sum which is similar to selection sort.



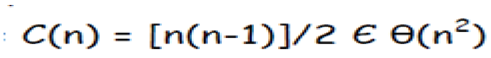
$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1$$

$$= \sum_{i=0}^{n-2} [(n-2-i) - 0+1]$$

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

$$= [n(n-1)]/2 \in \Theta(n^2)$$

The no.of key swaps depends on the input. The worst case is same as the no.of key comparisons.



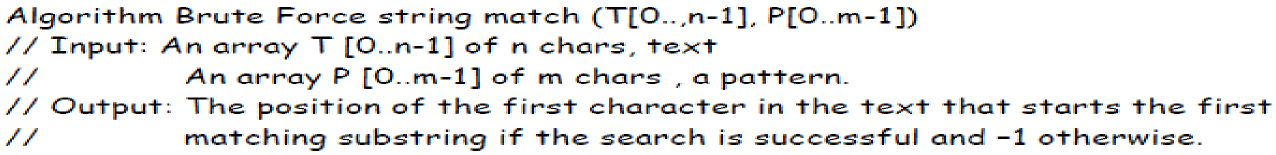
$$C(n) = [n(n-1)]/2 \in \Theta(n^2)$$

Q.3(b) Write algorithm for the brute force string matching process and Apply the same for the given input

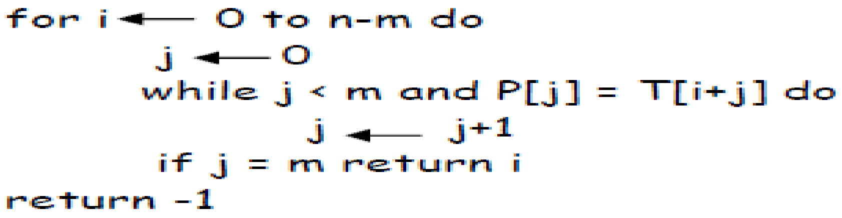
Text String = { NOBODY-NOTICED-HIM }

Pattern String ={ NOT }

Solution



```
Algorithm Brute Force string match (T[0..,n-1], P[0..m-1])
// Input: An array T [0..n-1] of n chars, text
//        An array P [0..m-1] of m chars , a pattern.
// Output: The position of the first character in the text that starts the first
//        matching substring if the search is successful and -1 otherwise.
```



```
for i ←— 0 to n-m do
        j ←— 0
        while j < m and P[j] = T[i+j] do
                j ←— j+1
        if j = m return i
return -1
```

N	**O**	**B**	**O**	**D**	**Y**	**_**	**N**	**O**	**T**	**I**	**C**	**E**	**D**	**_**	**H**	**I**	**M**
N	**O**	**T**															
	N	**O**	**T**														
		N	**O**	**T**													
			N	**O**	**T**												
				N	**O**	**T**											
					N	**O**	**T**										
						N	**O**	**T**									
							N	**O**	**T**								

String is matched return the starting Index -8

N	**O**	**B**	**O**	**D**	**Y**	**_**	**N**	**O**	**T**	**I**	**C**	**E**	**D**	**_**	**H**	**I**	**M**
							N	**O**	**T**								

The time complexity would be analyzed by finding the number of times the basic operation j=j+1 is executed.
The inner loop will be executed a maximum of m times (j=0 to m-1).

Therefore

T(n)= $\sum_{i=0}^{n-m}\sum_{j=0}^{m-1} 1 = \sum_{i=0}^{n-m} m$ = (n-m)*m = θ(mn).

Where m is the length of pattern and n is the length of text.

Q 4.(a) Write an algorithm to n elements using Merge sort. Apply the same to sort the given list { E, L , E , M ,E , N , T , S } in alphabetical ordering.

Mergesort is a perfect example of a successful application of the divide-and conquer technique. It sorts a given array A[0..n − 1] by dividing it into two halves A[0.._n/2_ − 1] and A[_n/2_..n − 1], sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one. The pseudocode for Merge sort is as follows:

Algorithm merge(arr,l,mid, u)

```
        Create a temporary array C[0..u]
        i<-- l
        j <-- mid+1
        k <-- l // index into temporary array
        while i <=mid and j <=u
                if arr[i] <= arr[j]
                        C[k] <-- arr[i]
                        i <-- i+1
                else
                        C[k] <-- arr[j]
                        j <-- j+1
        k <-- k+1
//copying rest of elements from first subarray
 while i<=mid
        C[k] <-- arr[i]
        i <-- i+1
        k <-- k+1
//copying rest of elements from second subarray
while j<=u
        C[k] <-- arr[j]
        j <-- j+1
        k <-- k+1
for i in l to u        // copying all elements from temp array to original array
arr[i] <-- C[i]
```

Algorithm mergesort(arr,l,u)

```
// only do it if the array contains atleast 2 elements
    if ( l < u  )
        mid = (l+u)/2
        mergesort(A,l,mid)
        mergesort(A,mid+1,u)
        Merge(A,l,mid,u)
```

Analysis

We first analyze the merge function used for mergesort. We notice that to merge an array with n elements at every step(in the first three loops) an element is always copied to the temporary array C. Since there are n elements to be copied the number of operations in the first three loops is n. Similarly in the last loop when the elements are copied from temporary array to the original array (arr) there are again "n" copies. Thus the total number of copy operations in the algorithm merge is O(n).

Analyzing the mergesort algorithm we find that each call involves two recursive calls to mergesort with the problem size half and a call to merge which takes O(n) time . Thus the recurrence can be wtitten as:

$$T(n) = 2\,T(n/2)+cn.$$

Applying the master's method,

$a=2$, $b=2$ and $d=1$.

Thus a=bd and thus case 2 of Master's method applies.

Thus T(n) = O(nlgn).

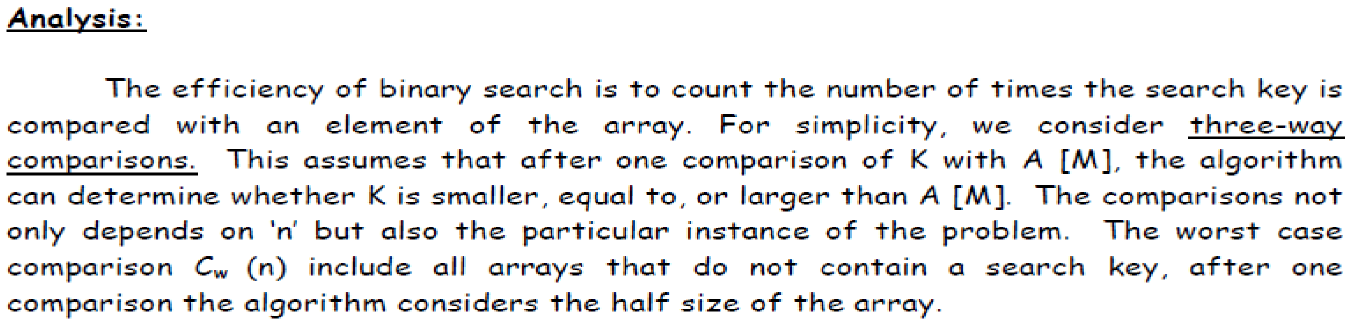
Q 4(b) Write an algorithm for binary search and analyze its time complexity

This search algorithm works on the principle of divide and conquer. For this algorithm, the data should be in the sorted order. Binary search stars comparing the middle element with the key value. If it matches, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

Algorithm binsearch(A[0..n-1],key,l,u)

```
Begin
```

```
   If  l>u
       Return -1
   While (l <=u )
       Mid ← (l+u)/2
       If A[mid] = key
            Return mid
      Else if A[mid] < key
             Return  binsearch(A, l,mid-1)
      Else
             Return binsearch(A,mid+1,u)
end
```



Analysis:

The efficiency of binary search is to count the number of times the search key is compared with an element of the array. For simplicity, we consider three-way comparisons. This assumes that after one comparison of K with A [M], the algorithm can determine whether K is smaller, equal to, or larger than A [M]. The comparisons not only depends on 'n' but also the particular instance of the problem. The worst case comparison C_w (n) include all arrays that do not contain a search key, after one comparison the algorithm considers the half size of the array.

Thus the recurrence would be C(n) = C(n/2)+1 , n>1 and C(1) = 0

The problem size at each step reduces by half each time. The additional amount in computing the mid element and comparison with the key is constant time operation and thus the 1 in the above expression.

Using master's method, we find a=1 , b=2 and d=0. The a=bd and thus it is the second case.

Hence C(n) = O(lgn)

Q5(b) Obtain the topological ordering for the following graph using source removal method.

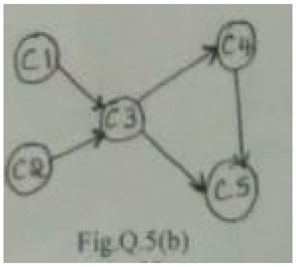


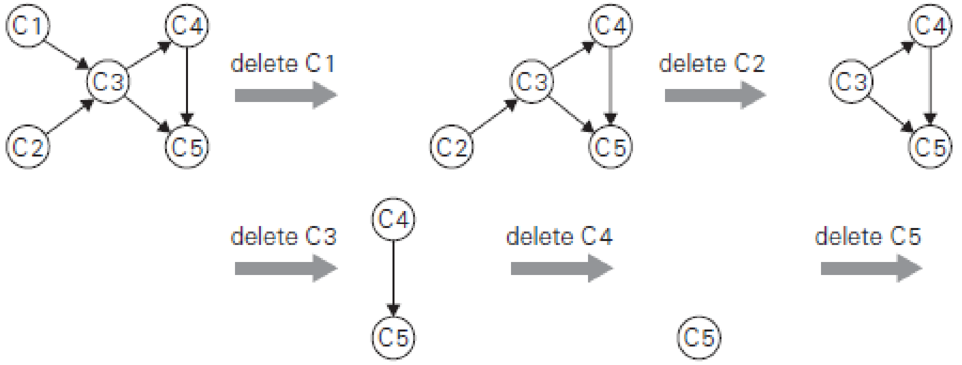
Fig Q.5(b)

For a Directed Acyclic Graph we can list its vertices in such an order that; for every edge in the graph, the vertex where the edge starts is listed before the vertex where the edge ends. This problem is called topological sorting. Topological sorting can be used to solve real world problems such as task scheduling where there is a dependency of a task on other tasks to be performed. Topological sorting is different from other sorting algorithms because the elements have a partial order and not a total order and thus the order of vertices arranged in topological order is not unique.

Topological sorting can be done in two ways:

1. Source Removal
2. Modified DFS.

Source Removal Method: In the source removal method at every step repeatedly, identify in a remaining digraph a ***source***, which is a vertex with no incoming edges, and delete it along with all the edges outgoing from it. The order in which the vertices are deleted yields a solution to the topological sorting problem. The following figure illustrates this method:

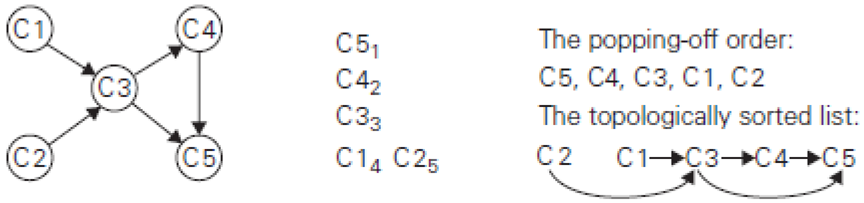


The solution obtained is C1, C2, C3, C4, C5

The algorithm for the same is given by:

```
Algorithm TopologicalSort(G,n)
        // G is a digraph G=(V,E), n is the number of vertices
        // create a array 'visit' to store if vertex is visited or not
        // array indeg stores the indegree of all vertices, to keep
        // track of vertices which can be removed from graph
        for u in V
                visit[u] <-- 0 // set it to not visited
                indeg[u] <-- 0
                for each vertex v which has an edge directed to u
                        indeg[u] <-- indeg[u] +1

        // since all vertices must be printed exactly once -
        // the loop goes on n times
        for i <-- 1 to n
                v <-- -1  // after the loop that follows v will be the
                          // vertex whose indegree is 0
                for u in V
                        if indeg[u] = 0
                                v <-- u
                                break
                if v = -1
                        print "Topological sort not possible, cycle in digraph"
                        break

visit[v] <-- 1
                print "Visiting vertex " , v
                for all vertices u adjacent to v
                        indeg[u] <-- indeg[u] -1 // removing v from graph
```

Modified DFS: Perform a DFS traversal and note the order in which vertices become dead-ends (i.e., popped off the traversal stack). Reversing this order yields a solution to the topological sorting problem, provided, of course, no back edge has been encountered during the traversal. If a back edge has been encountered, the digraph is not a dag, and topological sorting of its vertices is impossible. The figure below illustrates this method:



$C5_1$

$C4_2$

$C3_3$

$C1_4$ $C2_5$

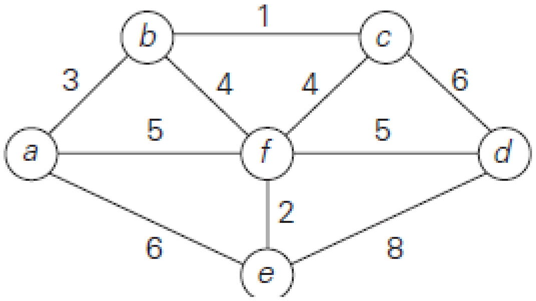
The popping-off order:

C5, C4, C3, C1, C2

The topologically sorted list:

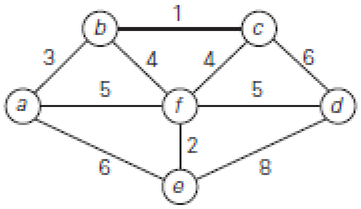
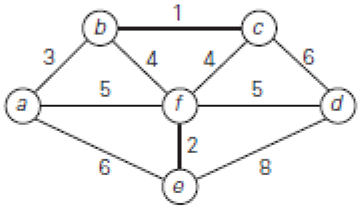
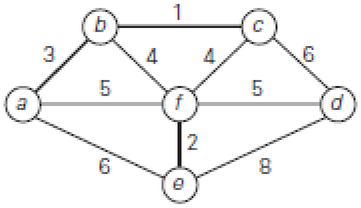
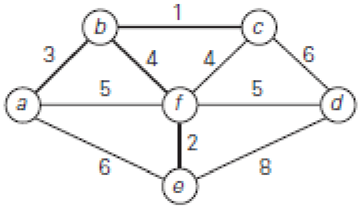
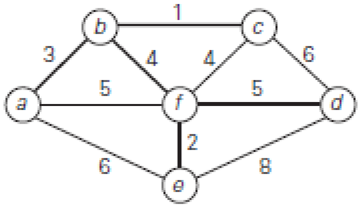
C2 C1→C3→C4→C5

Q6(a) Find the MST for the given graph using Kruskal's algorithm



Kruskal's algorithm is used for solving the minimal spanning tree problem. ***Spanning tree*** of an undirected connected graph is its connected acyclic subgraph(tree) that contains all the vertices of the graph. If such a graph has weights assigned to its edges, a ***minimum spanning tree*** is its spanning tree of the smallest weight, where the ***weight*** of a tree is defined as the sum of the weights on all its edges. The ***minimum spanning tree problem*** is the problem of finding a minimum spanning tree for a given weighted connected graph.

Kruskal's algorithm looks at a minimum spanning tree of a weighted connected graph $G = (V, E)$ as an acyclic subgraph with $|V| - 1$ edges for which the sum of the edge weights is the smallest. Consequently, the algorithm constructs a minimum spanning tree as an expanding sequence of subgraphs that are always acyclic but are not necessarily connected on the intermediate stages

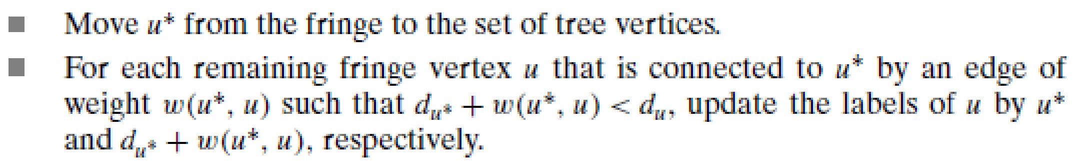
Tree edges	Sorted list of edges	Illustration
	bc ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	 1 b c 3 4 4 6 a 5 f 5 d 2 6 8 e
bc 1	bc **ef** ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	 1 b c 3 4 4 6 a 5 f 5 d 2 6 8 e
ef 2	bc ef **ab** bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	 1 b c 3 4 4 6 a 5 f 5 d 2 6 8 e
ab 3	bc ef ab **bf** cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	 1 b c 3 4 4 6 a 5 f 5 d 2 6 8 e
bf 4	bc ef ab bf cf af **df** ae cd de 1 2 3 4 4 5 5 6 6 8	 1 b c 3 4 4 6 a 5 f 5 d 2 6 8 e
df 5		

Q6(b) Write an algorithm to find single source shortest path using Dijkstra's algorithms

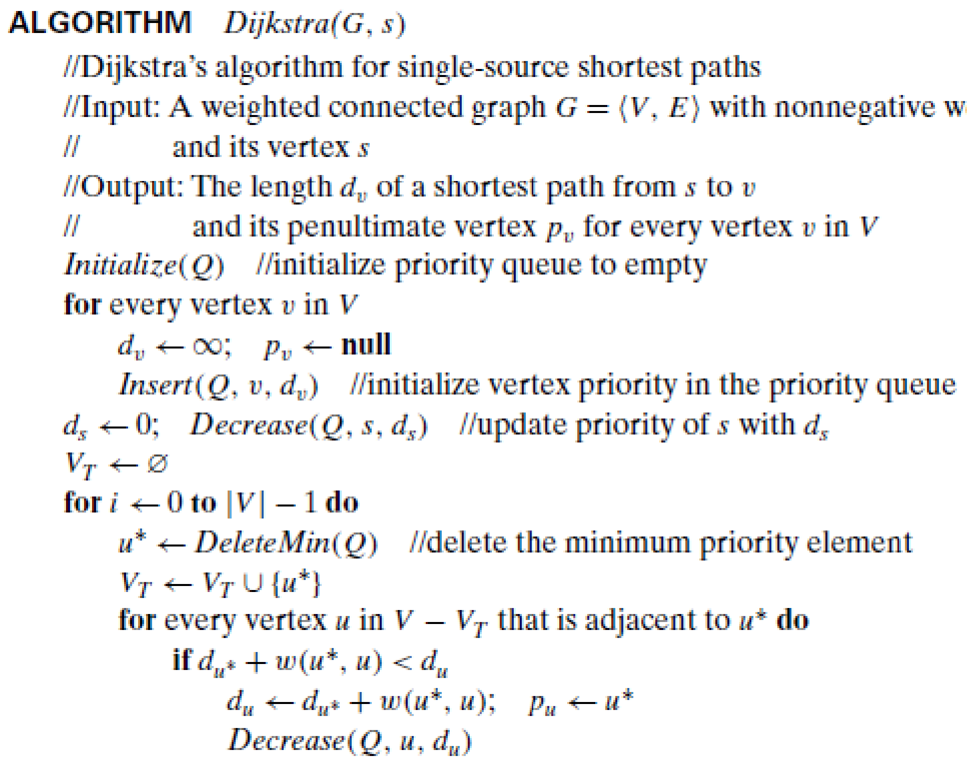
Explain the Dijkstra's single source shortest path algorithms and analyze its time complexity.

Sol: Dijkstra's algorithm is an algorithm for solving the single-source shortest-paths problem: for a given vertex called the source in a weighted connected graph with non negative edges, find shortest paths to all its other vertices. Some of the applications of the problem are transportation planning, packet routing in communication networks finding shortest paths in social networks, etc. First, it finds the shortest path from the source. to a vertex nearest to it, then to a second nearest, and so on. In general, before its ith iteration starts, the algorithm has already identified the shortest paths to i − 1 other vertices nearest to the source. These vertices, the source, and the edges of the shortest paths leading to them from the source form a subtree Ti of the given graph. The set of vertices adjacent to the vertices in T called "fringe vertices"; are the candidates from which Dijkstra's

algorithm selects the next vertex nearest to the source. To identify the ith nearest vertex, the algorithm computes, for every fringe vertex u, the sum of the distance to the nearest tree vertex v and the length dv of the shortest path from the source to v and then selects the vertex with the smallest such d value. d indicates the length of the shortest path from the source to that vertex till that point. We also associate a value p with each vertex which indicates the name of the next-to-last vertex on such a path, . After we have identified a vertex u* to be added to the tree, we need to perform two operations.
The psuedocode for Dijkstra's is as given below:



- Move u^* from the fringe to the set of tree vertices.
- For each remaining fringe vertex u that is connected to u^* by an edge of weight $w(u^*, u)$ such that $d_{u^*} + w(u^*, u) < d_u$, update the labels of u by u^* and $d_{u^*} + w(u^*, u)$, respectively.



```
ALGORITHM  Dijkstra(G, s)
    //Dijkstra's algorithm for single-source shortest paths
    //Input: A weighted connected graph $G = \langle V, E \rangle$ with nonnegative weights
    //        and its vertex $s$
    //Output: The length $d_v$ of a shortest path from $s$ to $v$
    //          and its penultimate vertex $p_v$ for every vertex $v$ in $V$
    Initialize(Q)   //initialize priority queue to empty
    for every vertex $v$ in $V$
        $d_v \leftarrow \infty$;   $p_v \leftarrow$ null
        Insert(Q, v, $d_v$)   //initialize vertex priority in the priority queue
    $d_s \leftarrow 0$;   Decrease(Q, s, $d_s$)   //update priority of $s$ with $d_s$
    $V_T \leftarrow \varnothing$
    for $i \leftarrow 0$ to $|V| - 1$ do
        $u^* \leftarrow$ DeleteMin(Q)   //delete the minimum priority element
        $V_T \leftarrow V_T \cup \{u^*\}$
        for every vertex $u$ in $V - V_T$ that is adjacent to $u^*$ do
            if $d_{u^*} + w(u^*, u) < d_u$
                $d_u \leftarrow d_{u^*} + w(u^*, u)$;   $p_u \leftarrow u^*$
                Decrease(Q, u, $d_u$)
```

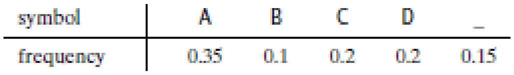
Analysis:
The time efficiency of Dijkstra's algorithm depends on the data structures used for implementing the priority queue and for representing an input graph itself.
Graph represented by adjacency matrix and priority queue by array:
In loop for initialization takes time |V| since the insertion into the queue would just involve appending the vertices at the end(since it is an array implementation). For the second loop, the loop runs |V| times. Each time the DeleteMin operation would take a maximum of θ(|V|) time since it would involve finding the vertex in the array with min d value, for a total time of |V|2. The for loop (for iupdating the neighbor vetices) would run |V| times again. However the Decrease would take θ(1) time because the index of the vertex would be known. Thus the total time complexity is θ(|V|2).
Graph represented by adjacency list and priority queue by binary heap:
All heap operations take θ(lg|V|) time. Thus the first loop runs |V| times and each time the Insert would take θ(lg|V|) time. The second loop runs |V| times and the DeleteMin would again take lg|V| time. Thus the total number of time DecreaseMin would run across all iterations is θ(Vlg|V|). In the second loop the basic operation is Decrease(Q,u,du) whoch is run the maximum number of times. Across all iterations using adjacency list,

since for each vertex Decrease is called for a maximum of all its adjacent vertices, the number of times Decrease is invoked |E| times. For each time it is onvoked , it takes O(lg|V|) time to execute. Thus the total time complexity is θ((|E|+|V|)lg|V|).

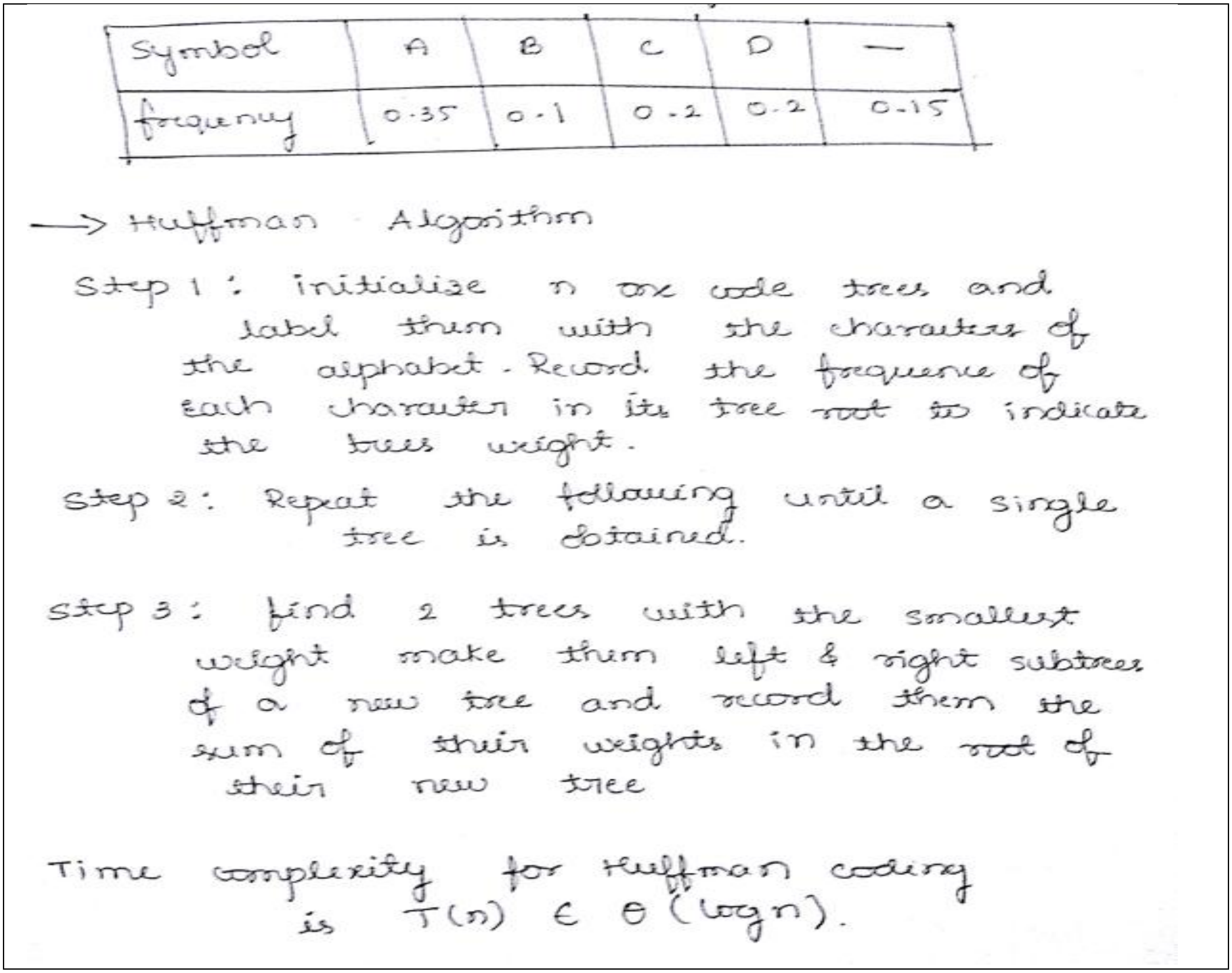
Graph represented by adjacency list and priority queue by fibonacci heap:

The time taken in this case θ(|E|+|V|lg|V|).

Q6(c) Find the Huffman code for the following data by obtaining Huffman tree .



symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15



Symbol	A	B	C	D	—
frequeny	0.35	0.1	0.2	0.2	0.15

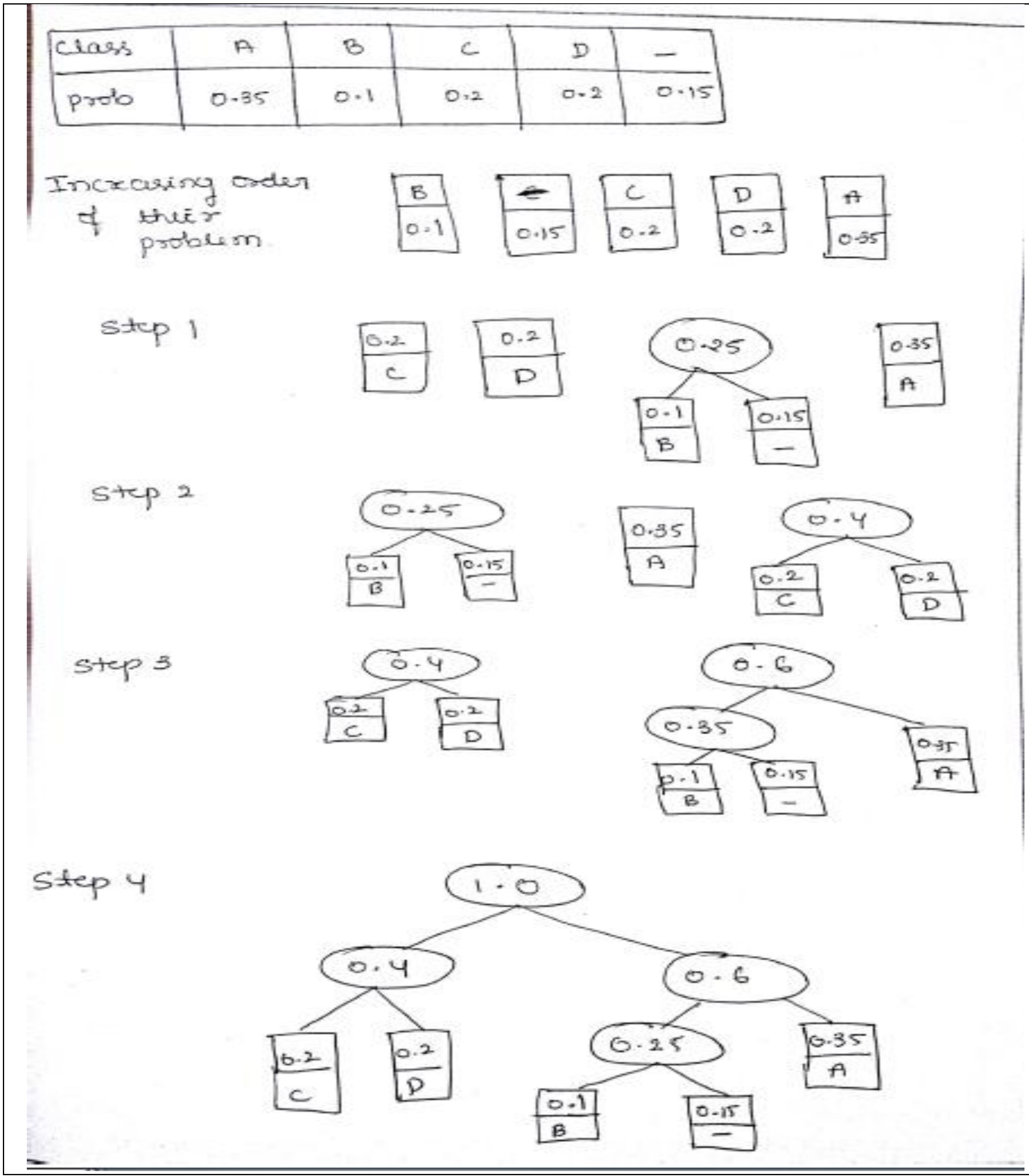
→ Huffman Algorithm

Step 1 : initialize n one code trees and label them with the charaters of the alphabet. Record the frequence of each character in its tree root to indicate the trees weight.

Step 2 : Repeat the following until a single tree is obtained.

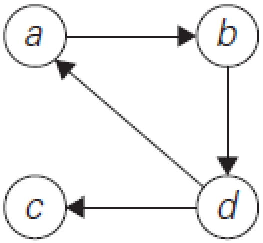
Step 3 : find 2 trees with the smallest weight make them left & right subtrees of a new tree and record them the sum of their weights in the root of their new tree

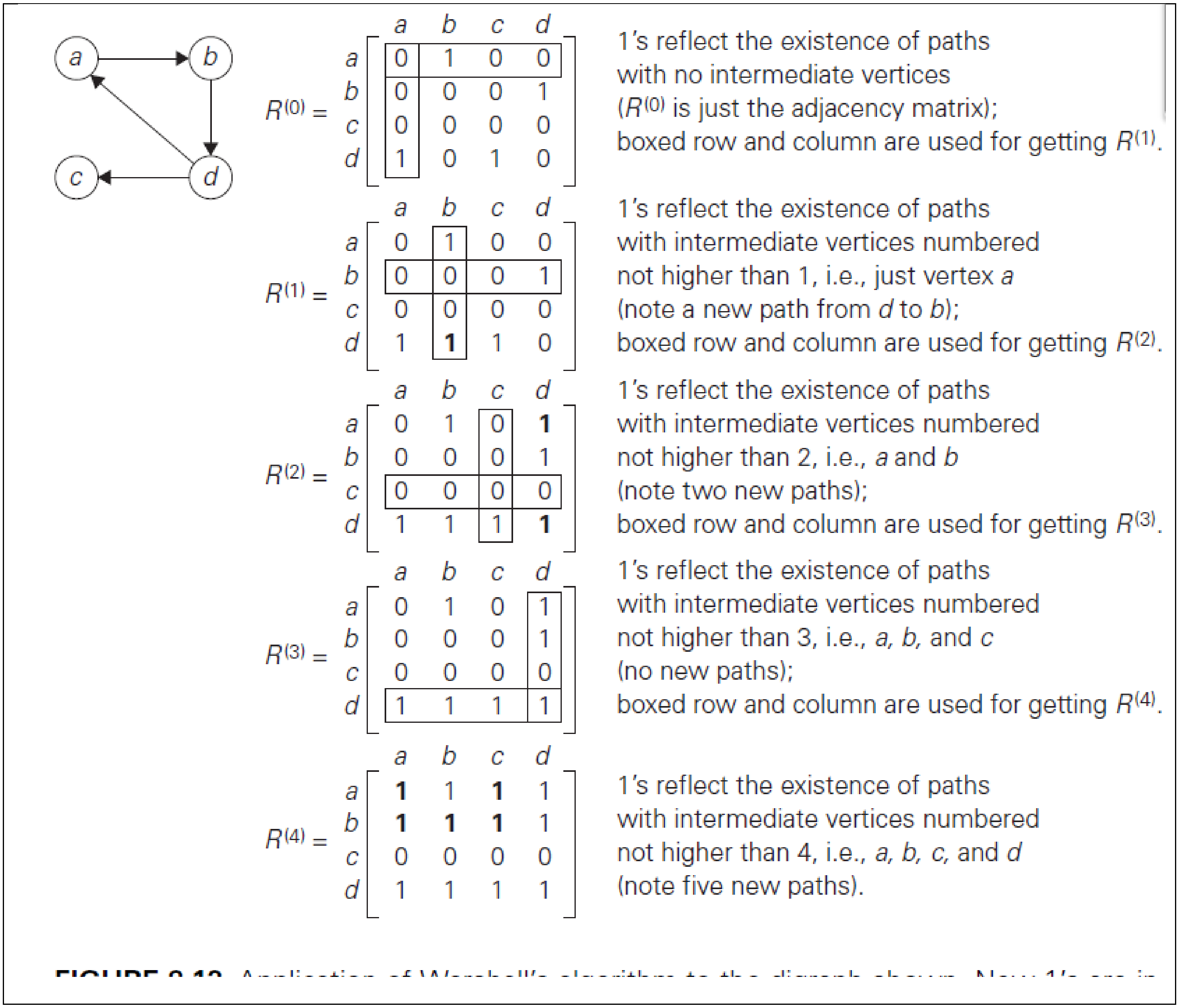
Time complexity for Huffman coding is $T(n) \in \Theta(\log n)$.



Class	A	B	C	D	—
prob	0.35	0.1	0.2	0.2	0.15

Q 7 (a) Write an algorithm to compute transitive closure/ path matrix for the given graph and obtain the transitive closure for the given graph :





$$R^{(0)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 0 & 1 & 0 \end{array}$$

1's reflect the existence of paths with no intermediate vertices ($R^{(0)}$ is just the adjacency matrix); boxed row and column are used for getting $R^{(1)}$.

$$R^{(1)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & \mathbf{1} & 1 & 0 \end{array}$$

1's reflect the existence of paths with intermediate vertices numbered not higher than 1, i.e., just vertex *a* (note a new path from *d* to *b*); boxed row and column are used for getting $R^{(2)}$.

$$R^{(2)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & \mathbf{1} \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & \mathbf{1} \end{array}$$

1's reflect the existence of paths with intermediate vertices numbered not higher than 2, i.e., *a* and *b* (note two new paths); boxed row and column are used for getting $R^{(3)}$.

$$R^{(3)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 1 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

1's reflect the existence of paths with intermediate vertices numbered not higher than 3, i.e., *a*, *b*, and *c* (no new paths); boxed row and column are used for getting $R^{(4)}$.

$$R^{(4)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & \mathbf{1} & 1 & \mathbf{1} & 1 \\ b & \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

1's reflect the existence of paths with intermediate vertices numbered not higher than 4, i.e., *a*, *b*, *c*, and *d* (note five new paths).

FIGURE 8.13 Application of Warshall's algorithm to the digraph shown. New 1's are in

Q 7(b) Find the optimal solution for the given Knapsack instance using 0/1 Knapsack method with capacity M=5

Item	**1**	**2**	**3**	**4**
Weight	**2**	**1**	**3**	**2**
Value/Profit	**12**	**10**	**20**	**15**

knapsack problem: given n items of known weights w1, . . . , wn and values v1, . . . , vn and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack

Consider an instance defined by the first i items, 1≤ i ≤ n, with weights w1, . . . , wi, values v1, . . . , vi , and knapsack capacity j, 1 ≤ j ≤ W. Let F(i, j) be the value of an optimal solution to this instance, i.e., the value of the most valuable subset of the first i items that fit into the knapsack of capacity j.We can divide all the subsets of the first i items that fit the knapsack of capacity j into two categories: those that do not include the ith item and those that do. Note the following:

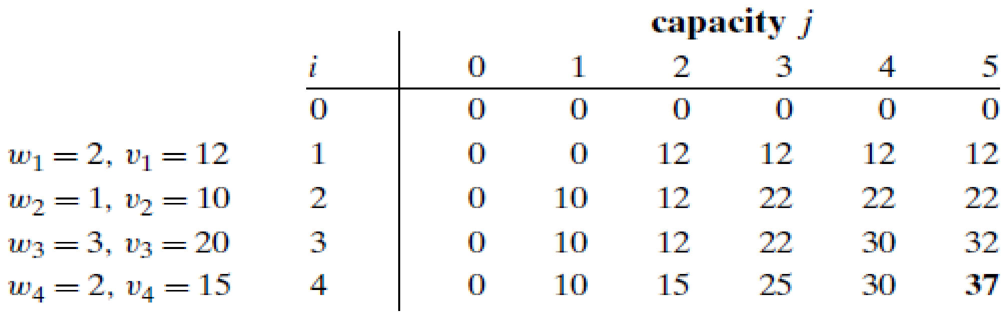
1. Among the subsets that do not include the ith item, the value of an optimal subset is, by definition, F(i − 1, j).

2. Among the subsets that do include the ith item (hence, j – wi ≥ 0), an optimal subset is made up of this item and an optimal subset of the first i − 1 items that fits into the knapsack of capacity j − wi . The value of such an optimal subset is vi + F(i − 1, j − wi).

Thus, the value of an optimal solution among all feasible subsets of the first i items is the maximum of these two values. Of course, if the ith item does not fit into the knapsack, the value of an optimal subset selected from the first i items is the same as the value of an optimal subset selected from the first i − 1 items.
These observations lead to the following recurrence:

F(i, j) = max**{F(i − 1, j), vi + F(i − 1, j − wi)} if j – wi ≥ 0,**
F(i − 1, j) if j − wi < 0

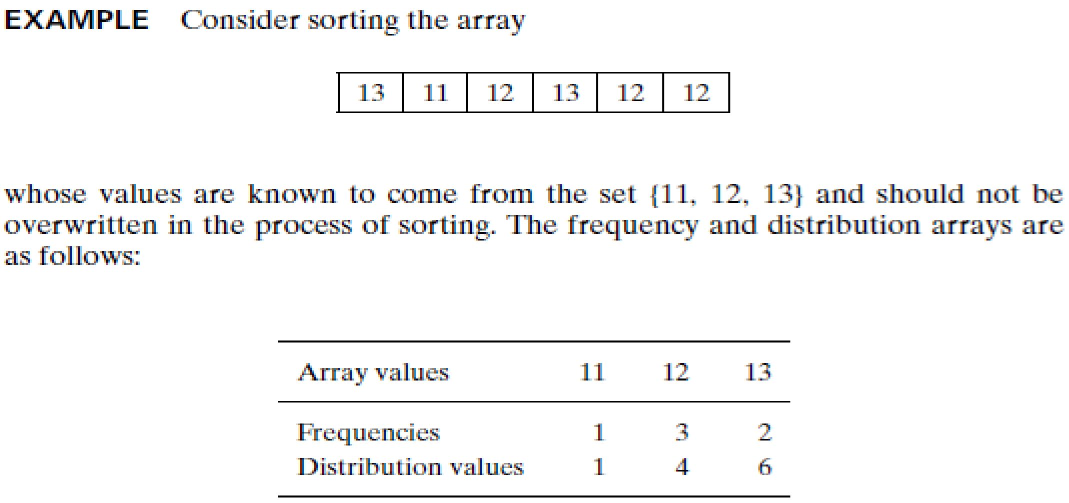
It is convenient to define the initial conditions as follows:
F(0, j) = 0 for j ≥ 0 and F(i, 0) = 0 for i ≥ 0.
The goal is to find F(n, W), the maximal value of a subset of the n given items that fit into the knapsack of capacity W, and an optimal subset itself.



		capacity *j*					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2,\ v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1,\ v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3,\ v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2,\ v_4 = 15$	4	0	10	15	25	30	**37**

Thus, the maximal value is *F(*4, 5*)* = $37. We can find the composition of an optimal subset by backtracing the computations of this entry in the table. Since *F(*4, 5*)* > *F(*3, 5*)*, item 4 has to be included in an optimal solution along with an optimal subset for filling 5 − 2 = 3 remaining units of the knapsack capacity. The value of the latter is *F(*3, 3*)*. Since *F(*3, 3*)* = *F(*2, 3*)*, item 3 need not be in an optimal subset. Since *F(*2, 3*)* > *F(*1, 3*)*, item 2 is a part of an optimal selection, which leaves element *F(*1, 3 − 1*)* to specify its remaining composition. Similarly,since *F(*1, 2*)* > *F(*0, 2*)*, item 1 is the final part of the optimal solution {item 1, item 2, item 4}.

8(a) Write an algorithm to sort given elements using counting method. Apply the same for the following input { 13,11,12,13,12,12}



EXAMPLE Consider sorting the array

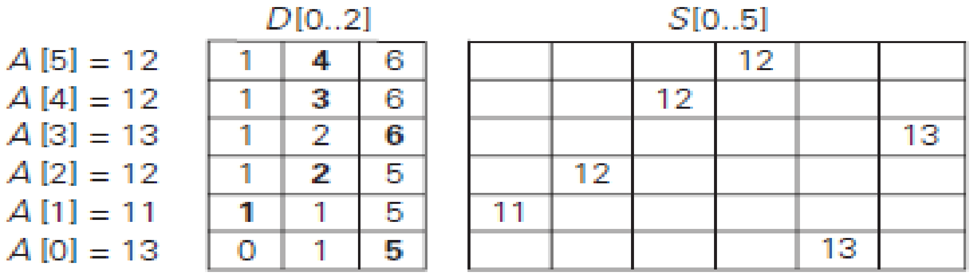
13	11	12	13	12	12

whose values are known to come from the set {11, 12, 13} and should not be overwritten in the process of sorting. The frequency and distribution arrays are as follows:

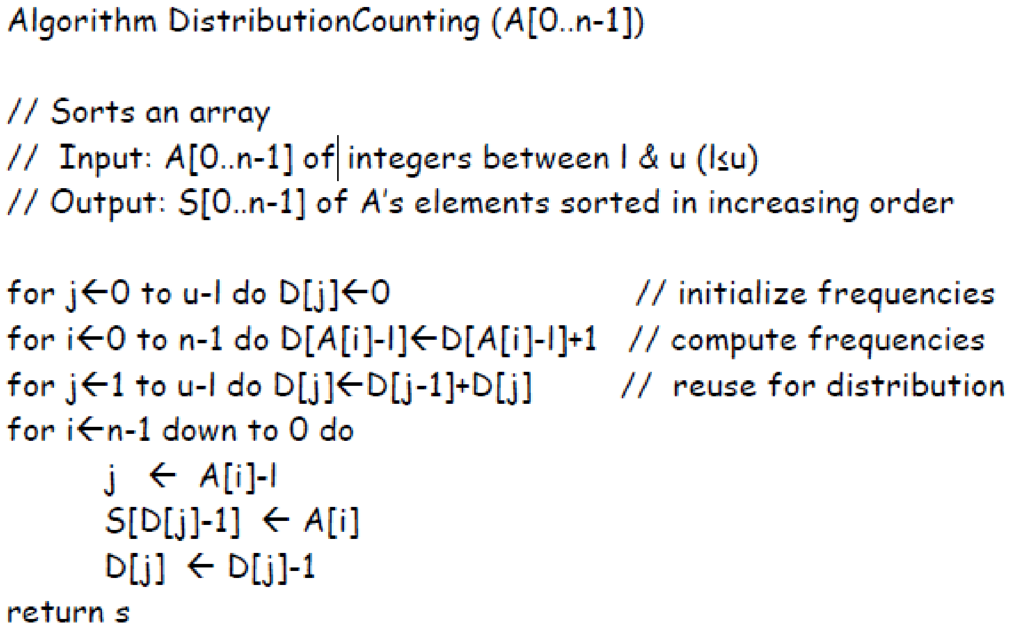
Array values	11	12	13
Frequencies	1	3	2
Distribution values	1	4	6

Note that the distribution values indicate the proper positions for the last occurrences of their elements in the final sorted array. If we index array positions from 0 to *n* − 1, the distribution values must be reduced by 1 to get corresponding element positions.

It is more convenient to process the input array right to left. For the example, the last element is 12, and, since its distribution value is 4, we place this 12 in position 4 − 1= 3 of the array *S* that will hold the sorted list. Then we decrease the 12's distribution value by 1 and proceed to the next (from the right) element in the given array.



	D[0..2]			*S*[0..5]					
A [5] = 12	1	**4**	6				12		
A [4] = 12	1	**3**	6			12			
A [3] = 13	1	2	**6**						13
A [2] = 12	1	**2**	5		12				
A [1] = 11	**1**	1	5	11					
A [0] = 13	0	1	**5**					13	



```
Algorithm DistributionCounting (A[0..n-1])

// Sorts an array
// Input: A[0..n-1] of integers between l & u (l≤u)
// Output: S[0..n-1] of A's elements sorted in increasing order

for j←0 to u-l do D[j]←0                    // initialize frequencies
for i←0 to n-1 do D[A[i]-l]←D[A[i]-l]+1   // compute frequencies
for j←1 to u-l do D[j]←D[j-1]+D[j]        //  reuse for distribution
for i←n-1 down to 0 do
        j  ←  A[i]-l
        S[D[j]-1]  ← A[i]
        D[j]  ← D[j]-1
return s
```

Assuming that the range of array values is fixed, this is obviously a linear algorithm because it makes just two consecutive passes through its input array *A*. This is a better time-efficiency class than that of the most efficient sorting algorithms—mergesort, quicksort, and heapsort—we have encountered

Q8(b) Explain the Horspool's string matching algorithm with a suitable example

Sol :

Horspool's algorithm is used for string matching and performs better than the brute force string matching by attempting the largest possible shift after every mismatch. this however, is done at the cost of extra storage which is a shift table maintained. While matching a string with the pattern the following four cases occur

Case 4: If C happens to be the last character in the pattern and there are other C's among its first n-1 characters the shift in same as case 2.

S_0…………….……O R…………S_{n-1}

≠ ||

REORD E R

RE O R DER

Input enhancement makes repetitive comparisons unnecessary. Shift sizes are precomputed and stored in a table. The shift value is calculated by the formula:

t(c)=
- the pattern's length m, if c is not among the first m-1 characters of the pattern
- the distance from the rightmost c among the 1st m-1 characters of the pattern to its last character, otherwise

Case 1: If there are no C's in the pattern, shift the pattern by its entire length to right.

e.g : S_0…………S…………S_{n-1}

≠

BARBER

BARBER

Case 2: If there are occurrences of character 'c' in the pattern but the last one, then shift should align the right most occurrence of c in the pattern

S_0……….……B…………S_{n-1}

≠

BARBE R

BARBER

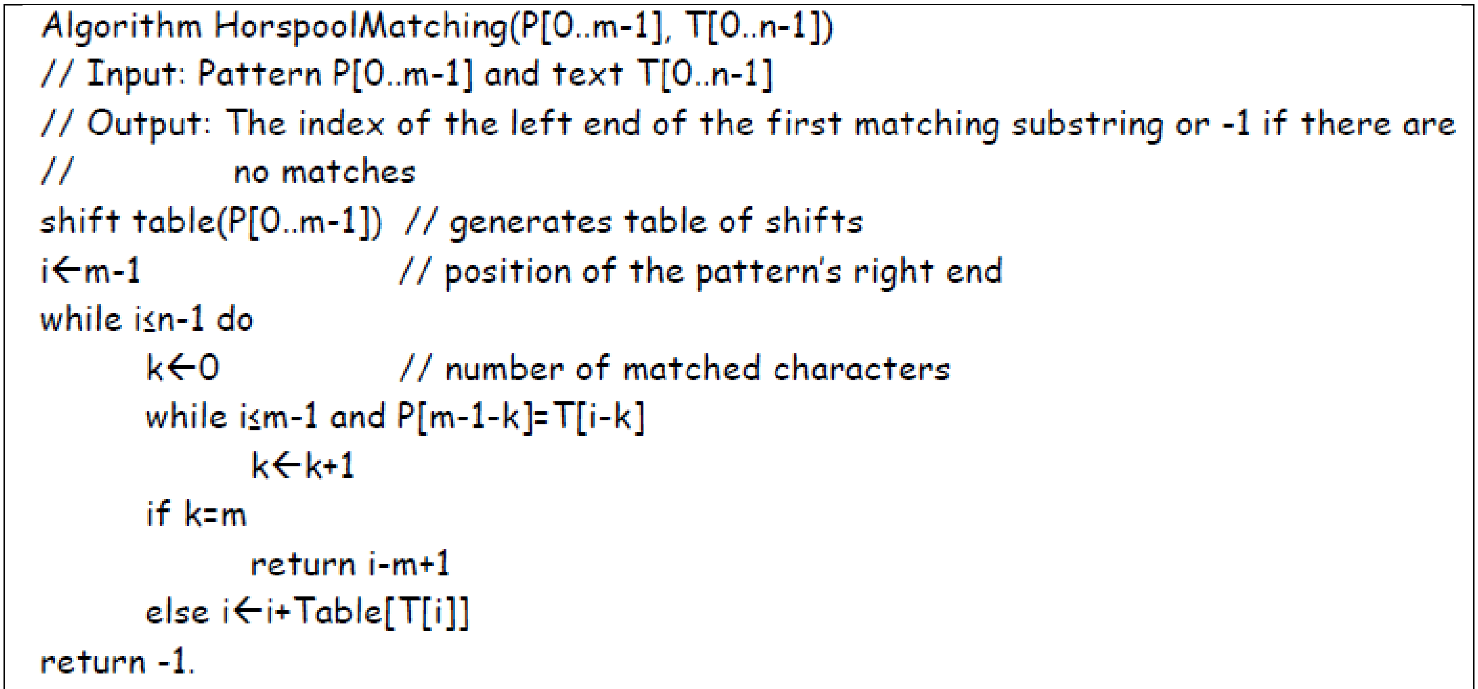
Case 3: If C happens to be the last character in the pattern then there are no C's among its m-1 character, shift same as case 1.

S_0…………M E R…………S_{n-1}

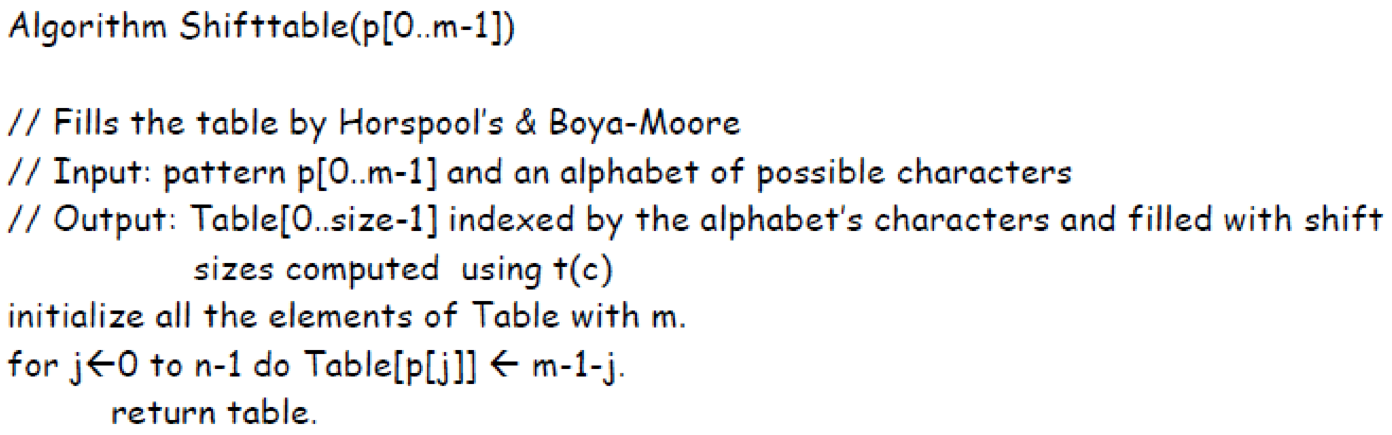
≠ || ||

LEA D E R

LEADER



```
Algorithm HorspoolMatching(P[0..m-1], T[0..n-1])
// Input: Pattern P[0..m-1] and text T[0..n-1]
// Output: The index of the left end of the first matching substring or -1 if there are
//              no matches
shift table(P[0..m-1])  // generates table of shifts
i←m-1                   // position of the pattern's right end
while i≤n-1 do
        k←0             // number of matched characters
        while i≤m-1 and P[m-1-k]=T[i-k]
                k←k+1
        if k=m
                return i-m+1
        else i←i+Table[T[i]]
return -1.
```



```
Algorithm Shifttable(p[0..m-1])

// Fills the table by Horspool's & Boya-Moore
// Input: pattern p[0..m-1] and an alphabet of possible characters
// Output: Table[0..size-1] indexed by the alphabet's characters and filled with shift
            sizes computed  using t(c)
initialize all the elements of Table with m.
for j←0 to n-1 do Table[p[j]] ← m-1-j.
        return table.
```

To search for the pattern DEMO in the text THIS IS A DEMO FOR STRING MATCHING, we first find the shift table for DEMO

Here n(length of string)=34 and m=4

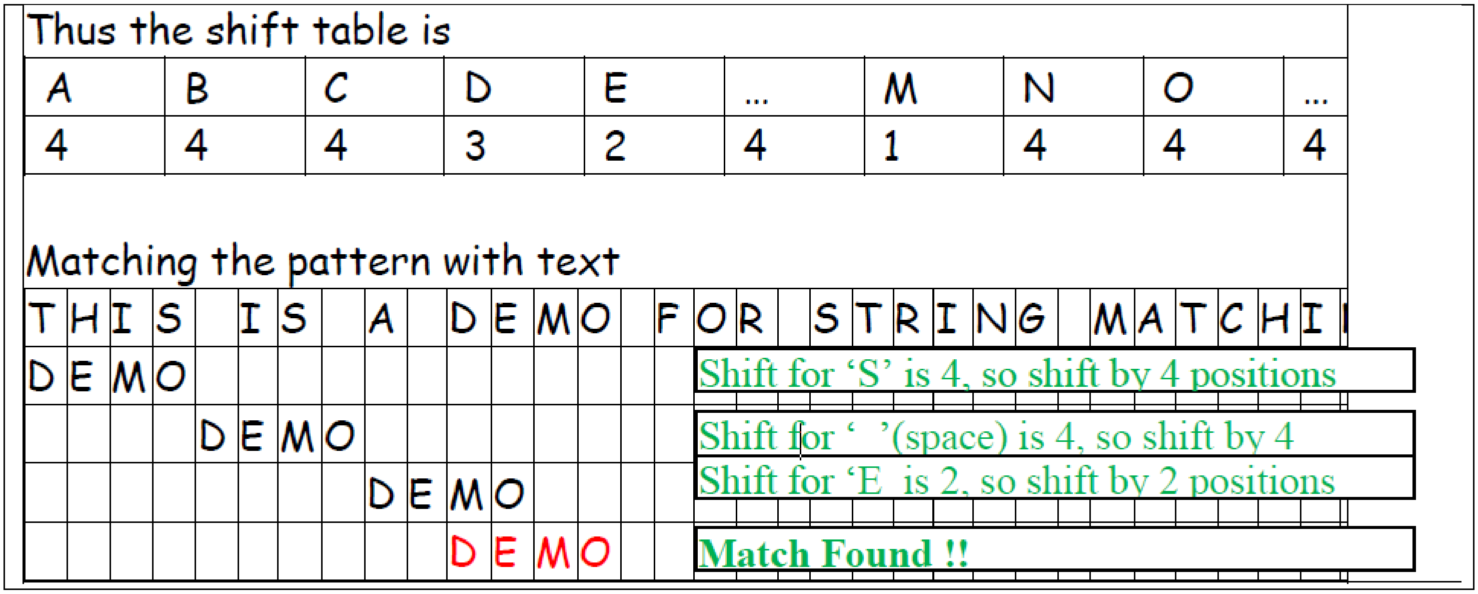
Calculating the shift only for the first 3 characters:

Shift for D = m – 1 – I = 4-1-0=3

Shift for E = 4-1-1=2

Shift for M = 4-1-2=1

For all characters the shift table will have entries 4.



Thus the shift table is

A	B	C	D	E	...	M	N	O	...
4	4	4	3	2	4	1	4	4	4

Matching the pattern with text

T	H	I	S		I	S		A		D	E	M	O		F	O	R		S	T	R	I	N	G		M	A	T	C	H	I
D	E	M	O																												
				D	E	M	O																								
								D	E	M	O																				
										D	E	M	O																		

Shift for ‘S’ is 4, so shift by 4 positions

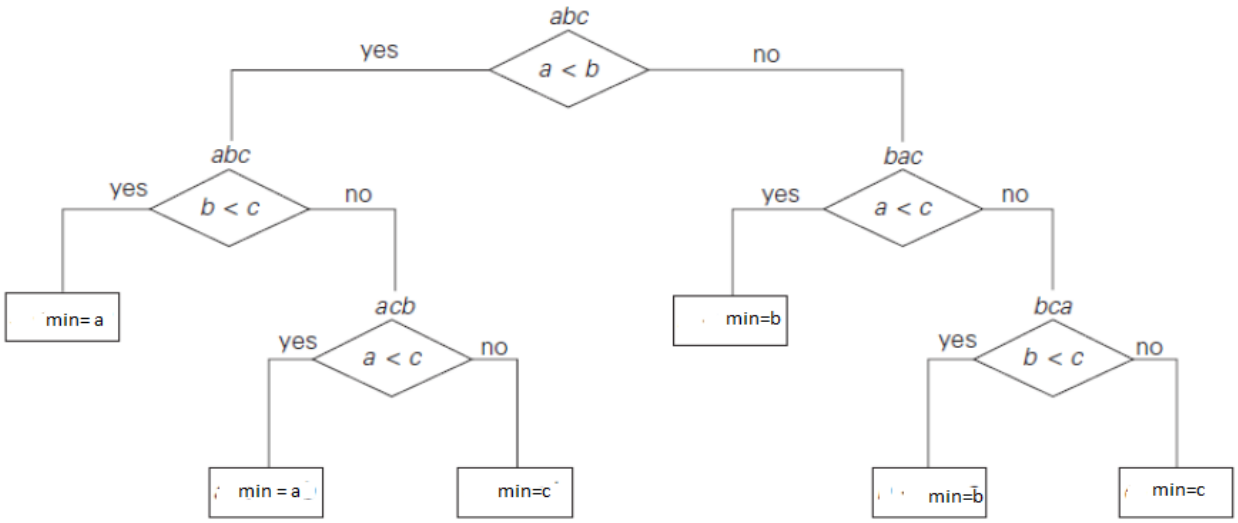
Shift for ‘ ’(space) is 4, so shift by 4

Shift for ‘E is 2, so shift by 2 positions

Match Found !!

Q9.(a) What is Decision Tree ? Obtain the decision tree to find minimum of 3 numbers.

Decision tree of an algorithm for finding a minimum of three numbers. Each internal node of a binary decision tree represents a key comparison indicated in the node, e.g., k < k'.The node’ s left subtree contains the information about subsequent comparisons made if , k < k', and its right subtree does the same for the case of k > k'. (For the sake of simplicity, we assume throughout this section that all input items are distinct.) Each leaf represents a possible outcome of the algorithm’ s run on some input of size n. Note that the number of leaves can be greater than the number of outcomes because, for some algorithms, the same outcome can be arrived at through a different chain of comparisons. The algorithm’s work on a particular input of size n can be traced by a path from the root to a leaf in its decision tree, and the number of comparisons made by the algorithm on such a run is equal to the length of this path. Hence, the number of comparisons in the worst case is equal to the height of the algorithm’s decision tree. The central idea behind this model lies in the observation that a tree with a given number of leaves, which is dictated by the number of possible outcomes, hasto be tall enough to have that many leaves. We use the lemma that for any binary tree with l leaves and height h, $h \geq \log_2 l$. Indeed, a binary tree of height h with the largest number of leaves has all its leaves on the last level. Hence, the largest number of leaves in such a tree is 2_h. In other words, $2_h \geq l$, which immediately implies, $h \geq \log_2 l$.



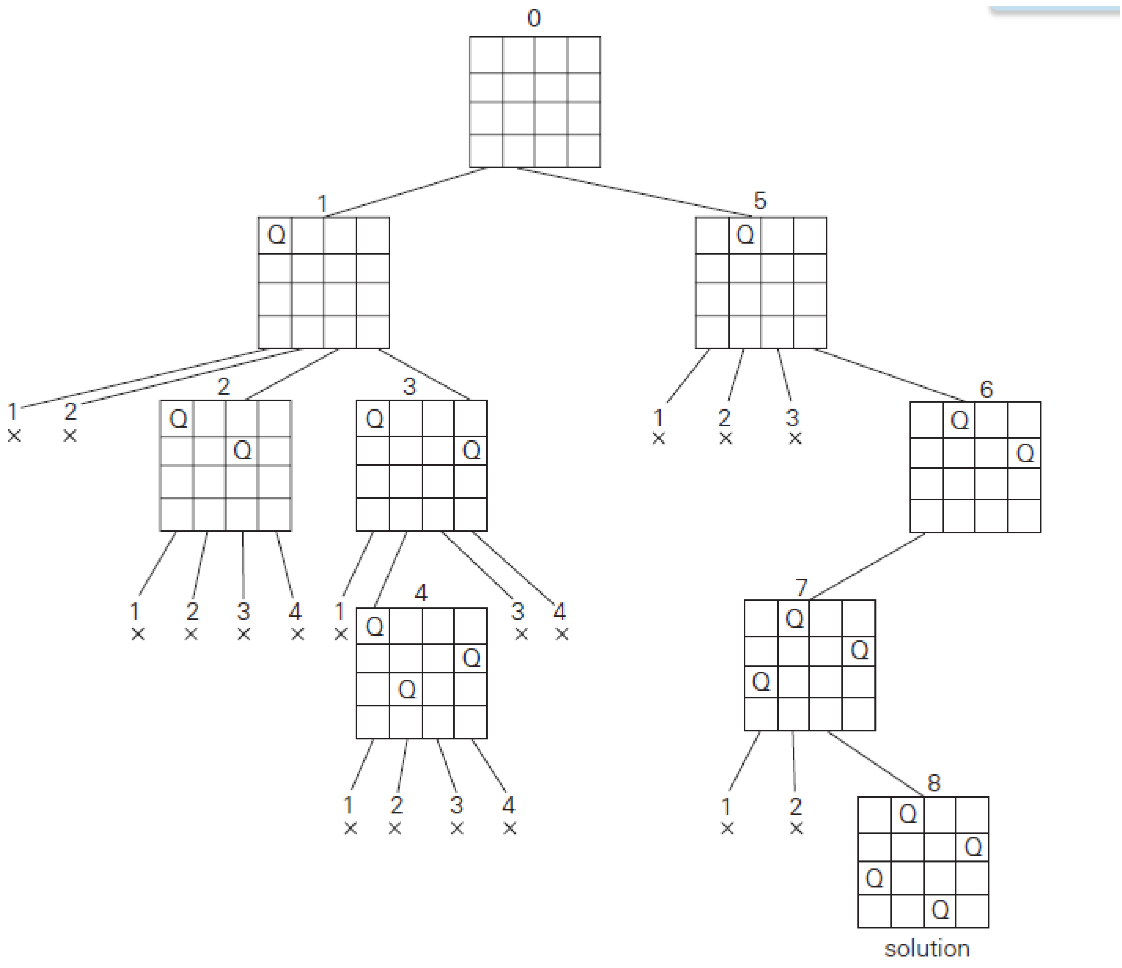
Q9.(b) Explain N-Queens problem using Back Tracking.

N Queen's problem:

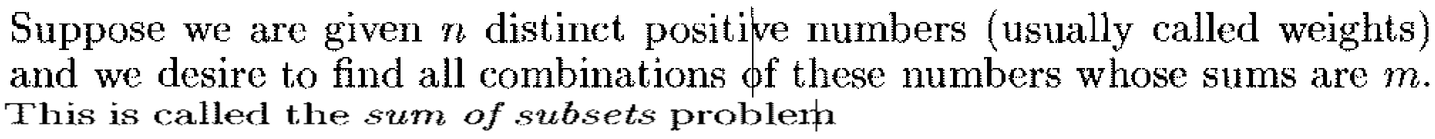
The n-queens problem. is to place n queens on an n × n chessboard so that no two queens attack each other by being in the same row or in the same column or on the same diagonal. To solve this using backtracing we use the following strategy. Consider the 4 Queens problem:

We start with the empty board and then place queen 1 in the first possible position of its row, which is in column 1 of row 1. Then we place queen 2, after trying unsuccessfully columns 1 and 2, in the first acceptable position for it, which is square (2, 3), the square in row 2 and column 3. This proves to be a dead end because there is no acceptable position for queen 3. So, the algorithm backtracks and puts queen 2 in the next possible position at (2, 4). Then queen 3 is placed at (3, 2), which proves to be another dead end. The algorithm then backtracks all the way to queen 1 and moves it to (1, 2). Queen 2 then goes to (2, 4), queen 3 to (3, 1), and queen 4 to (4, 3), which is a solution to the problem.

The state space tree is shown below:



Q10(a) Construct the state –space tree for the Sum of subsets problem for the given data W = { 5,10,12,13,15,18} and M = 30



Suppose we are given n distinct positive numbers (usually called weights) and we desire to find all combinations of these numbers whose sums are m. This is called the *sum of subsets* problem

A simple choice for the bounding functions is $B_k(x_1, \ldots, x_k)$ = true iff

$$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m$$

Clearly $x_1, \ldots, x_k$ cannot lead to an answer node if this condition is not satisfied. The bounding functions can be strengthened if we assume the w_i's are initially in nondecreasing order. In this case $x_1, \ldots, x_k$ cannot lead to an answer node if

$$\sum_{i=1}^{k} w_i x_i + w_{k+1} > m$$

The bounding functions we use are therefore

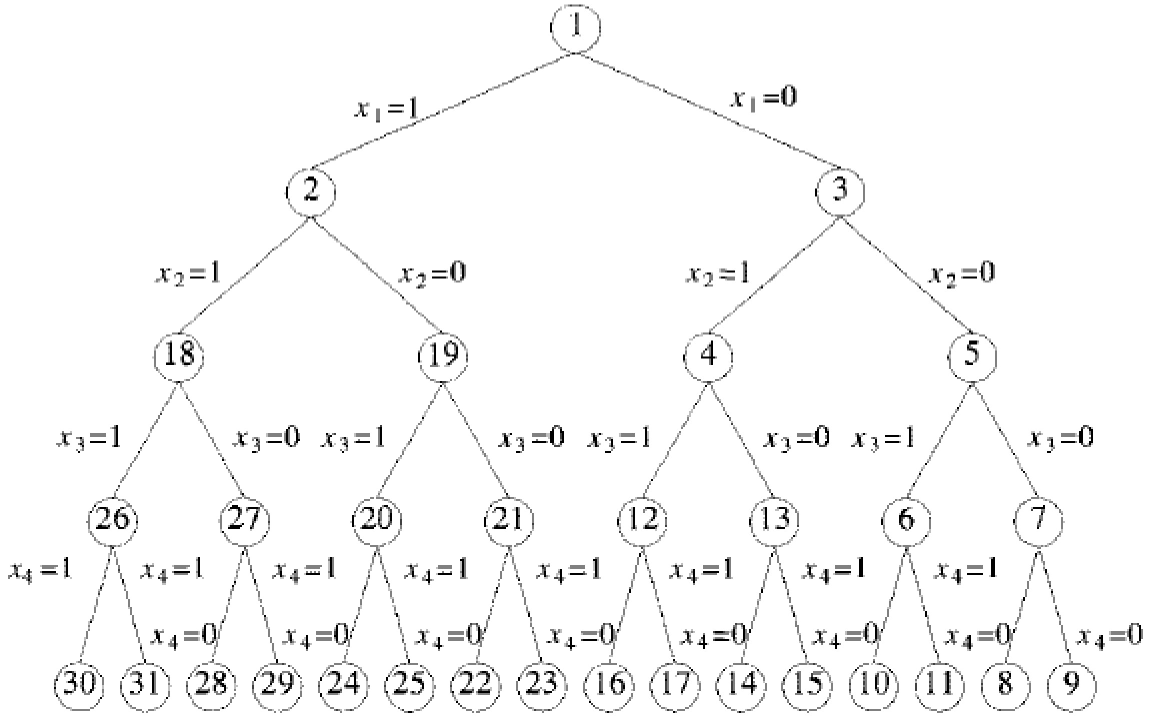
This simplification results from the realization that if $x_k = 1$, then

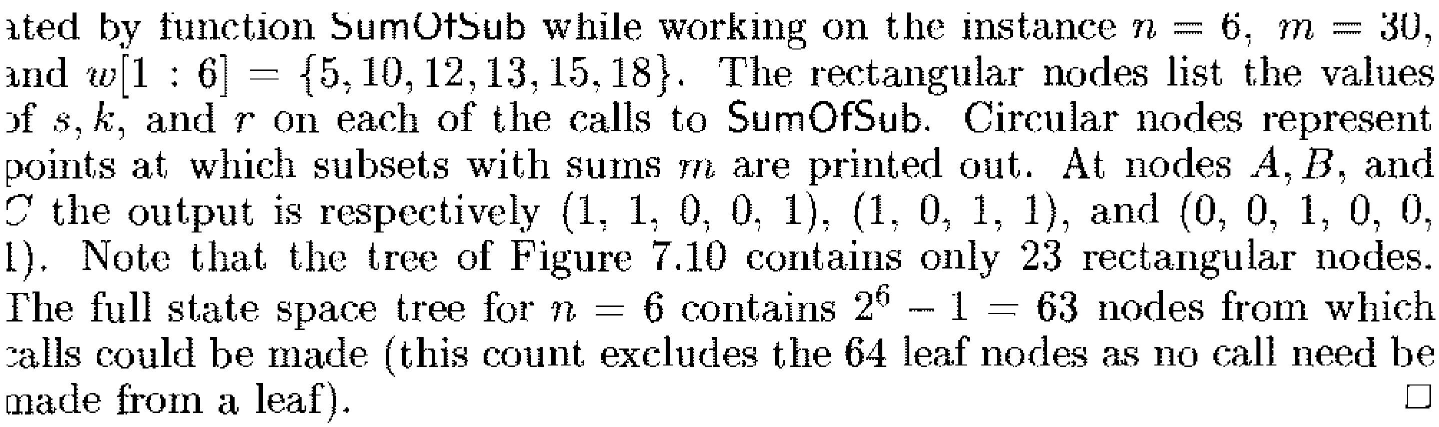
$$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i > m$$

$$B_k(x_1, \ldots, x_k) = true \text{ iff } \sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m$$

$$\text{and} \quad \sum_{i=1}^{k} w_i x_i + w_{k+1} \leq m$$

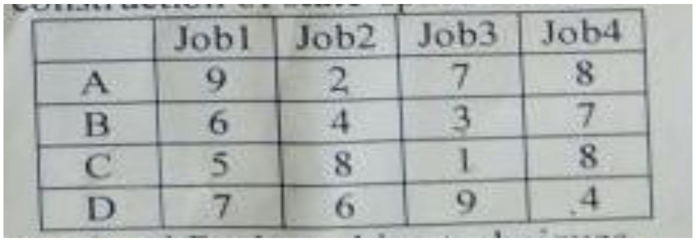
possible organization for the sum of subsets problems.





ated by function SumOfSub while working on the instance $n = 6$, $m = 30$, and $w[1:6] = \{5, 10, 12, 13, 15, 18\}$. The rectangular nodes list the values of s, k, and r on each of the calls to SumOfSub. Circular nodes represent points at which subsets with sums m are printed out. At nodes A, B, and C the output is respectively (1, 1, 0, 0, 1), (1, 0, 1, 1), and (0, 0, 1, 0, 0, 1). Note that the tree of Figure 7.10 contains only 23 rectangular nodes. The full state space tree for $n = 6$ contains $2^6 - 1 = 63$ nodes from which calls could be made (this count excludes the 64 leaf nodes as no call need be made from a leaf). □

Q10 (b) Find the optimal solution for the given assignment problem which is represented aqs the matrix as shown below:



	Job1	Job2	Job3	Job4
A	9	2	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	9	4

.

Sol:

Problem Statement : There are n people who need to be assigned to execute n jobs, one person per job. (That is, each person is assigned to exactly one job and each job is assigned to exactly one person.) The cost that would accrue if the ith person is assigned to the jth job is a known quantity C[i, j] for each pair i, j = 1, 2, . . . , n. The problem is to find an assignment with the minimum total cost.

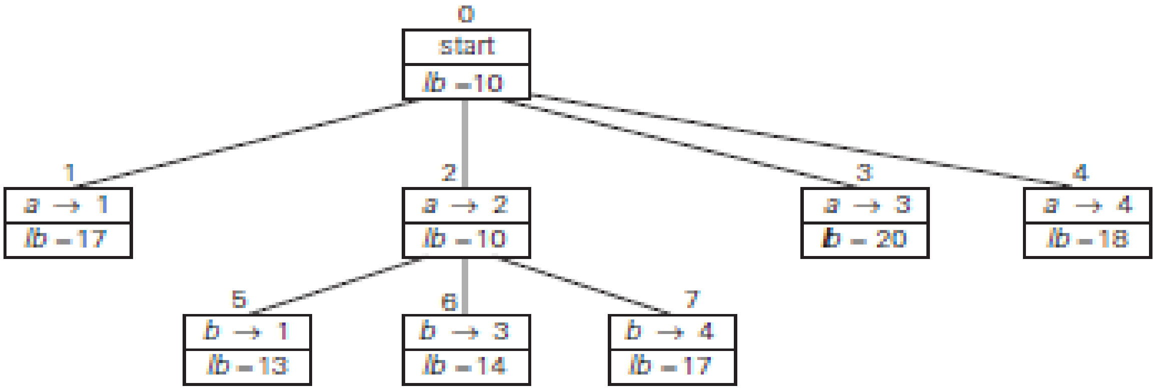
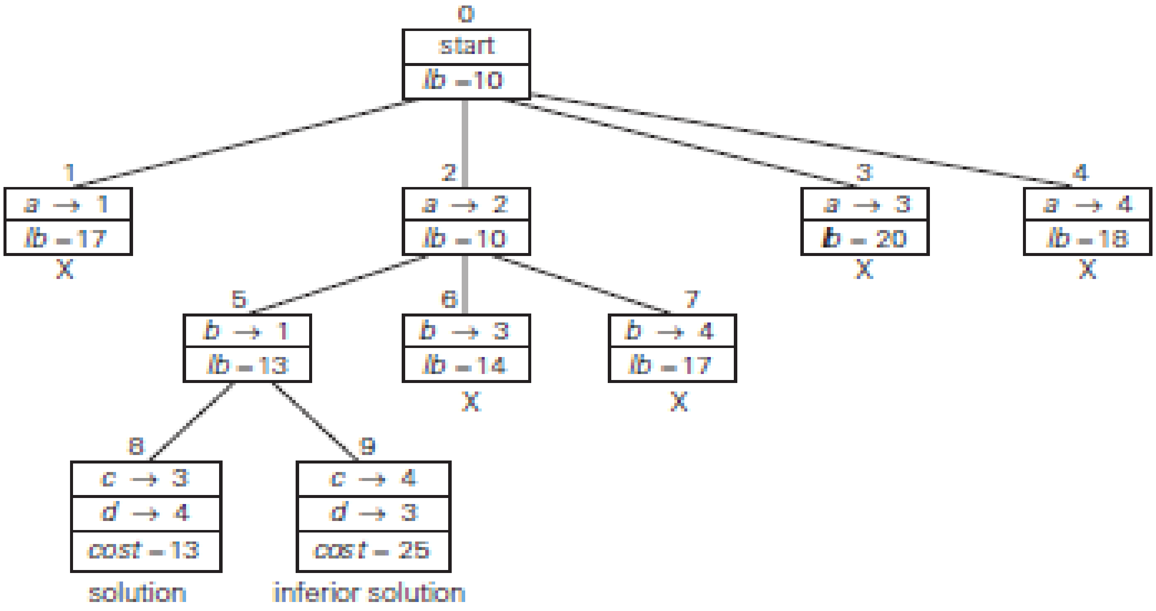


FIGURE 12.6 Levels 0, 1, and 2 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm.



Best solution : a->2, b->1 c->3,d->4 Min Cost = 13