USN | | | | | | | | | | |

13MCA41

Fourth Semester MCA Degree Examination, June/July 2017
Analysis and Design of Algorithms

Time: 3 hrs. Max. Marks:100

Note: *Answer any FIVE full questions.*

1 a. Define algorithm. Explain the steps involved in algorithm design and analysis process with neat diagram. **(10 Marks)**

b. Write formal definitions of asymptotic notations with graph representation. **(06 Marks)**

c. List the steps involved in analyzing the time efficiency of non-recursive algorithms. **(04 Marks)**

2 a. Solve the recurrence relation and draw a tree of recursive calls for tower of Hanoi problem. **(07 Marks)**

b. Write an algorithm for bubble sort and obtain an expression for number of times basic operation is executed. **(06 Marks)**

c. Sort the list E, X, A, M, P, L, E in an alphabetical order by using selection sort. Discuss whether selection sort satisfies stable and inplace properties of sorting algorithms. **(07 Marks)**

3 a. Write algorithm for merge sort. Find the time complexity of merge sort using master's theorem. **(08 Marks)**

b. Apply Strassen's algorithm to multiply the given two matrices:

$$A = \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} \text{ and } B = \begin{bmatrix} 8 & 7 \\ 1 & 2 \end{bmatrix}.$$ **(07 Marks)**

c. Discuss when best, worst and average case situations arise in binary search program with their time efficiencies. **(05 Marks)**

4 a. Write an algorithm for insertion sort, and find the time complexity of insertion sort in worst case situation. **(05 Marks)**

b. Discuss similarities and dissimilarities of Breadth-first-search and Depth-First-Search methods of traversing a graph. **(04 Marks)**

c. Apply Breadth-First-Search (BFS) traversal method for the following graph:

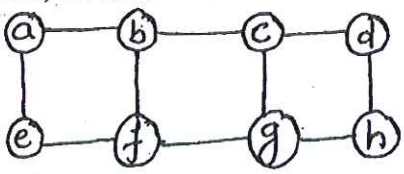


Fig. Q4 (c)

Consider 'a' as source node, find with what distance rest other nodes can be reached. **(06 Marks)**

d. What do you mean by topological order of a graph? Find the topological order of the given graph using source removal method. **(05 Marks)**

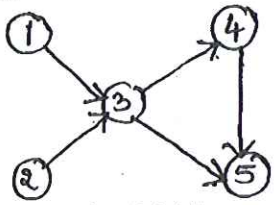


Fig. Q4 (d)

5 a. Write an algorithm for sorting by distribution counting. Apply the same algorithm to sort the elements :
12, 13, 10, 12, 10, 12, 11, 10, 13 **(10 Marks)**

b. Apply Horspool's algorithm to search a pattern PAPPAR in the text,
PAPPAPPAPPARRASSAN
Compare Bruteforce method and Horspool's algorithm of string matching. **(10 Marks)**

6 a. Write an algorithm for computing binomial co-efficient C(n, K). What is the time-efficiency of this algorithm? **(05 Marks)**

b. Find the transitive closure for the graph whose adjacency matrix is,

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

(05 Marks)

c. Apply the bottom up dynamic programming algorithm to the following instance of a knapsack problem. Capacity of knapsack W = 5. **(10 Marks)**

Item	1	2	3	4
Weight	2	1	3	2
Value	42	12	40	25

7 a. Write an algorithm to find minimum spanning tree using prim's algorithm. **(04 Marks)**

b. Obtain the shortest distance and shortest path from node a to all other nodes in a graph. **(08 Marks)**

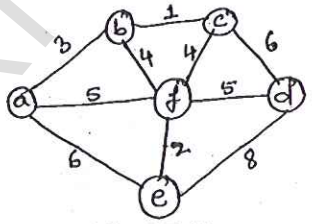


Fig.Q7 (b)

c. Construct Huffman code for the following data:

Character	A	B	C	D	- (underscore)
Probability	0.4	0.1	0.2	0.15	0.15

Encode the text ABCABC – AD
Decode the string whose encoding is 11111001010101 **(08 Marks)**

8 a. Write a note on NP-complete problems. **(03 Marks)**

b. Find the subsets from the given sum using backtracking method.
S = {3, 5, 6, 7} and d = 15 **(07 Marks)**

c. Write a problem statement for the assignment problem, and find the optimal solution for the following instance with the construction of state-space tree. **(08 Marks)**

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

d. Differentiate Branch-and-Bound and Back tracking techniques. **(02 Marks)**

* * * * *

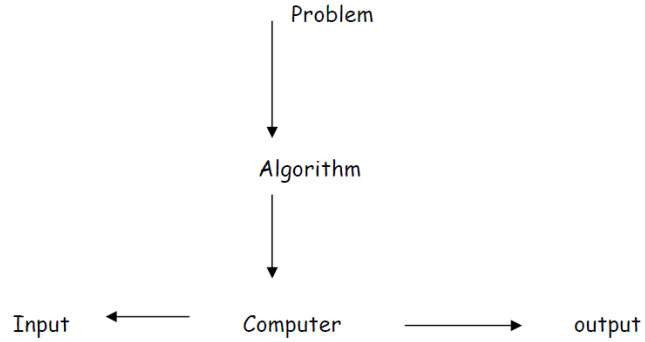
MCA - 13MCA41 - Analysis and Design of Algorithms

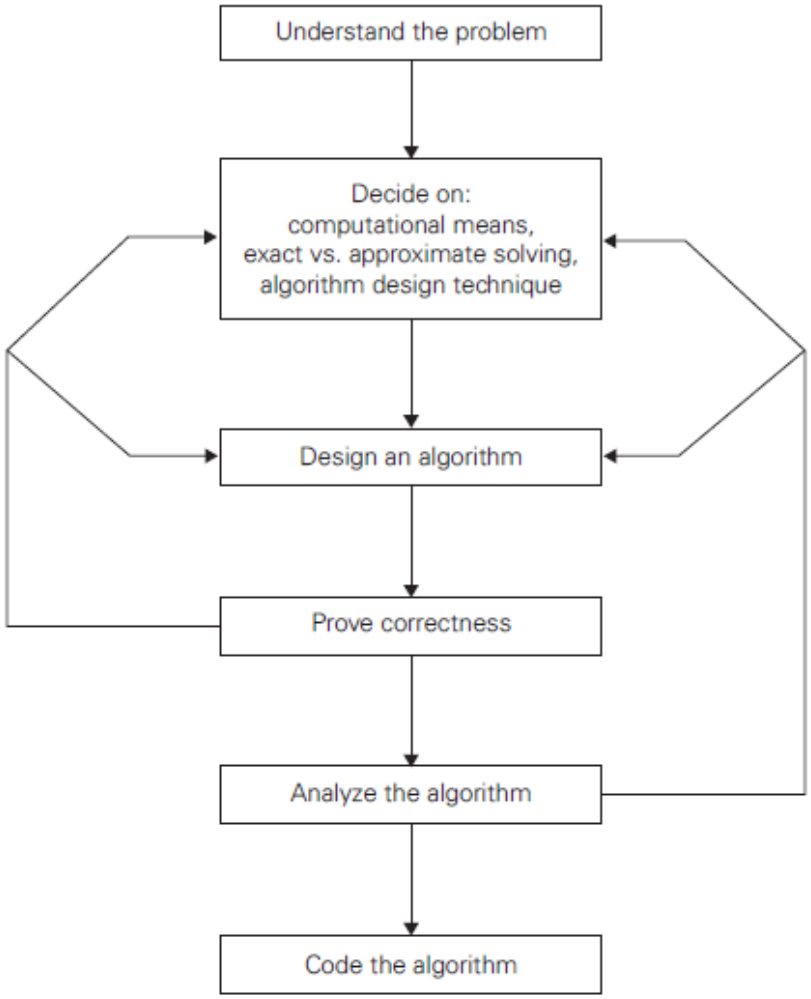
July 2017

Q1. (a) Define Algorithm, Explain the steps involed in algorithm designand analysis process with neat diagram.

Sol:

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.





Understanding the problem:

- The first important thing to solve a problem to understand a problem sufficiently. This may require the problem to be read multiple times, asking questions if required and working out smaller instances of problem by hand.
- Sometimes the problem at hand might already have many algorithms to solve them. In such a case choose the algorithm that might be best.
- It is also important to specify the kind of inputs for the problem so that the algorithm can be designed to work correctly under all circumstances.

∗**Ascertaining the capabilities of a computational Device:**

- Machines that exexure instructions sequentially follow the random access model(RAM). Algorithms designed fir such machines are called sequential algorithms. Majority of algorithms are sequential algorithms.

- Newer machines can run instructions parallel and algorithms which have written for such machines are called parallel algorithms. RAM model doesn't support this.
- For complex problem which involve processing large amounts of data in real time it helps to be aware of the computational power and the memory available in the machine where the program is to be executed

∗ **Choosing between exact and approximate problem solving**:

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

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

∗**Deciding on data structures**:

Data structures play a vital role in designing and analyzing the algorithms. Some of the algorithm design techniques also depend on the structuring data specifying a problem's instance.

∗**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. Various design methods for algorithms exist, some of which are – divide and conquer, dynamic programming, greedy algorithms etc.

∗**Methods of specifying an Algorithm**:

Algorithm can be specified in using natural language, but due to the inherent ambiguity, a psuedocode representation which is closer to the computer language is used. Flowchart which is a diagrammatic representation of the flow of the algorithms was used in the earlier days but has decreased in popularity due to complexity for large programs. Thus the most prevelant method of specifying an algorithm is using psuedocode.

∗ **Proving an Algorithm's correctness**:

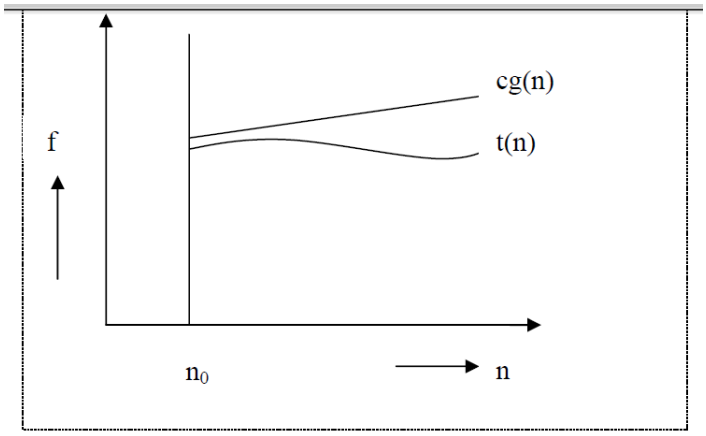
Correctness has to be proved for every algorithm. To prove that the algorithm gives the required result for every legitimate input in a finite amount of time. For some algorithms, a proof of correctness is quite easy; for others it can be quite complex. Mathematical Induction is normally used for proving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

(b) Write formal definition of asymptotic notations with graph representations.

Sol:

Sol: Definition: A function t(n) is said to be in O[g(n)]. Denoted t(n) ∈ 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) ≤ cg(n) for all n≥no.

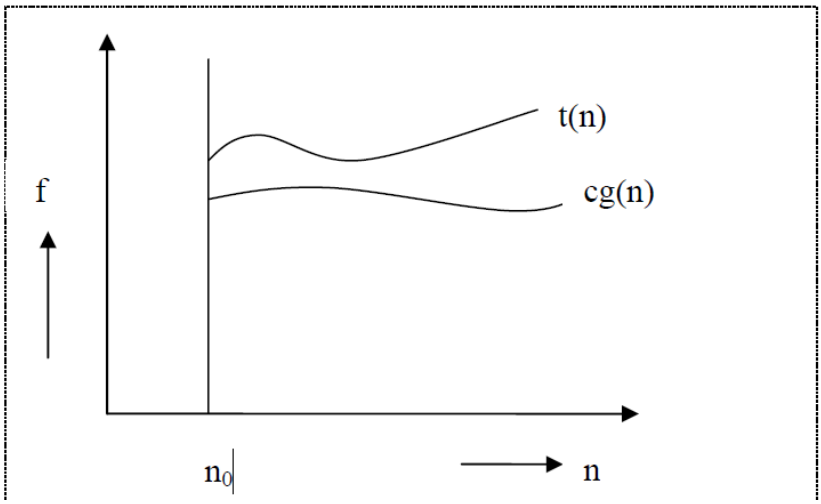
Eg. $100n+5 \in O(n^2)$



Ω -Notation:

Definition: A fn t(n) is said to be in Ω[g(n)], denoted t(n) ∈Ω[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 s.t. t(n) ≥ cg(n) for all n ≥ n0.

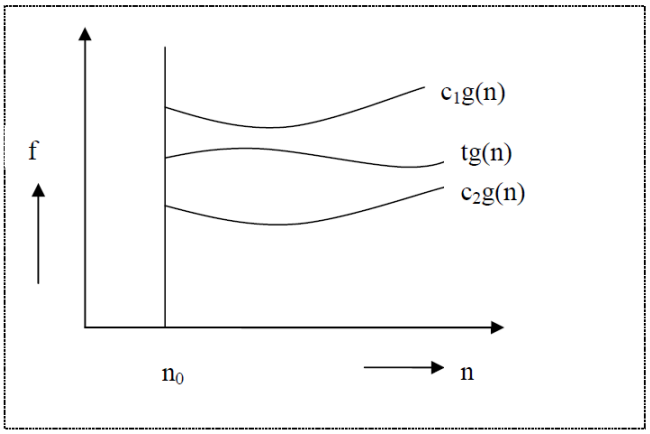
For example: $n^3 \in \Omega(n^2)$, Proof is $n^3 \geq n^2$ for all n ≥ n0. i.e., we can select c=1 and n0=0.



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: $n^2 \in \theta(n^2+4n+1)$, Proof is $7n^2 \geq n^2+4n+1$ for all n ≥ 0. i.e., we can select c=7 and n0=0.



(c) List the steps involved in analyzing the time efficiency of non recursive algorithms.

General Plan for Analyzing Efficiency of Nonrecursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation. (As a rule, it is located in its innermost loop.)
3. Check whether the number of times the basic operation is executed depends only on the size of an input. If it also depends on some additional property, the worst-case, average-case, and, if necessary, best-case efficiencies have to be investigated separately.
4. Set up a sum expressing the number of times the algorithm's basic operation is executed.

5. Using standard formulas and rules of sum manipulation either find a closed-form formula for the count or, at the very least, establish its order of growth.

For example Consider the **element uniqueness problem:** check whether all the elements in a given array are distinct. This problem can be solved by the following straightforward algorithm.

ALGORITHM UniqueElements(A[0..n - 1])

```
//Checks whether all the elements in a given array are distinct
//Input: An array A[0..n - 1]
//Output: Returns "true" if all the elements in A are distinct
// and "false" otherwise.
for i «— 0 to n — 2 do
    for j' <- i + 1 to n - 1 do
        if A[i] = A[j]
          return false
return true
```

Since the innermost loop contains a single operation (the comparison of two elements), we should consider it as the algorithm's basic operation. There are two kinds of worst-case inputs (inputs for which the algorithm does not exit the loop prematurely): arrays with no equal elements and arrays in which the last two elements are the only pair of equal elements. For such inputs, one comparison is made for each repetition of the innermost loop, i.e., for each value of the loop's variable j between its limits i + 1 and n - 1; and this is repeated for each value of the outer loop, i.e., for each value of the loop's variable i between its limits 0 and n - 2. Accordingly, we get:

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

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

$$= (n-1)^2 - [(n-2)(n-1)]/2 = [(n-1)n]/2 \approx \tfrac{1}{2} n^2 \in \Theta(n^2)$$

Q2.(a) Solve the recurrence relation and draw a tree of recursive calls for the tower of Hanoi problem.

The Pseudocode for Tower of Hanoi is as follows -

```
1: TOH(n, src, int mdt, dest).
2:     if n = 1
3:         Transfer disk from src to dest
4:     else.
5:         TOH(n-1, src, dest, int mdt)
6:         Transfer disk n from src to dest
7:         TOH(n-1, ~~dest~~ int mdt, src, dest).
```

Analyzing the above pseudocode we observe that if there is only one disk then a constant amount of work needs to done (line 3). Hence

$$M(n) = 1 \text{ for } n = 1.$$

lines 5 and 7 take $M(n-1)$ amount of time. Line 6 takes constant amount of time. Thus the recurrence relation is

$$M(n) = \begin{cases} M(n-1) + 1 + M(n-1) & n > 1 \\ 1 & , \ n = 1. \end{cases}$$

$$M(n) = 2M(n-1) + 1.$$

Expanding this and using substitution method.

$$M(n) = 2M(n-1) + 1 = 2[2M(n-2) + 1] + 1$$

$$= 2^2 M(n-2) + 2 + 1.$$

$$= 2^2 [2M(n-3) + 1] + 2 + 1$$

$$= 2^3 M(n-3) + 2^2 + 2 + 1$$

Q2 (b) Write an algorithm for bubble sort and obtain an expression for the number of times the basic operation is executed.

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 $i \leftarrow 0$ **to** $n-2$ **do**
  for $j \leftarrow 0$ **to** $n-2-i$ **do**
    if $A[j+1] < A[j]$ swap $A[j]$ and $A[j+1]$

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 of decreasing arrays, is same as the no of key comparisons:

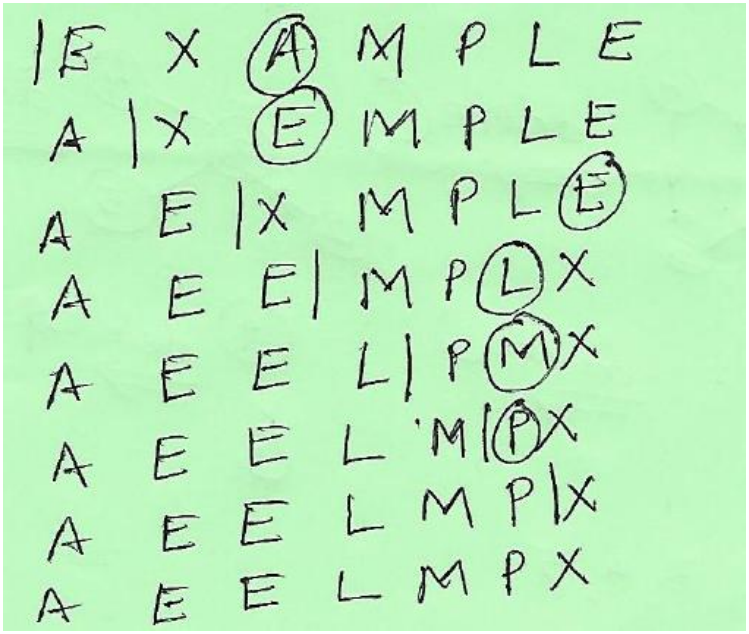
$$S_{worst}(n) = C(n) = [n(n-1)]/2 \in \theta(n^2)$$

Q2 (c) Sort the E,X,A,M,P,L,E in an alphabetical order by using selection sort. Discuss whether selection sort satisfies stable and in place properties of sorting algorithms.

Sol:

Selection sort is stable since the first minimum element from remaining elements is chosen at every stage and thus among two records having the same key value the one which occurs first always comes before the latter one.

Selection sort is in place since no extra memory is required for the sorting process.



Q3. (a) Write algorithm for merge sort. Find time complexity of merge sort using master's theorem.

Sol: 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

    // copying all elements from temp array to original array
    for i in l to u
        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 analyse the merge function used for mergesort. We notice that to merge an array with n elements at every step(in the first three loops) anelement 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 $a=b^d$ and thus case 2 of Master's method applies. Thus T(n) = O(nlgn).

Q3.(b) Apply strassen's method to multiply the matrices given below

A=[1 2 B=[8 7
5 6] 1 2]

Sol: First we find the 7 products

$[m_1 + m_4 - m_5 + m_7 \quad m_3 + m_5]$

$[m_2 + m_4 \quad m_1 + m_3 - m_2 + m_6]$

Where,

$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11})$

$m_2 = (a_{10} + a_{11})* b_{00}$

$m_3 = a_{00} * (b_{01} - b_{11})$

$m_4 = a_{11} * (b_{10} - b_{00})$

$m_5 = (a_{00} + a_{01}) * b_{11}$

$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01})$

$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11})$

m1= 70, m2= 88, m3=5, m4=-42, m5=6, m6=60, m7=-12

The product of matrices =

[70-42 – 6 -12 5+6]

[88-42 70 + 5 – 88 + 60]

= [10 11
46 47]

Q3. (c) Discuss when the best, worst and average case efficiency occurs in binary search program along with their time efficiency.

Sol:

Binary search algorithm is as follows:

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

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 $a=b^d$ and thus it is the second case.

Hence $C(n) = O(\lg n)$

Best Case: When the key is located as the middle element
Worst case: when key is not found
Average case: Other cases when key not in the middle but present in the array.

Q4. (a) Write an algorithm for insertion sort and find its time complexity in worst case.
Sol: Insertion sort is used for sorting an array. The decrease and conquer idea is to assume that the smaller problem of sorting the array $A[0..n-2]$ has already been solved to give
us a sorted array of size $n-1$: $A[0] \leq \ldots \leq A[n-2]$. We need is to find an appropriate position for $A[n-1]$ among the sorted elements and insert it there. This is done by scanning the sorted subarray from right to left until the first

element smaller than or equal to $A[n-1]$ is encountered to insert $A[n-1]$ right after that element.

Algorithm for insertion sort

ALGORITHM *InsertionSort*$(A[0..n-1])$
//Sorts a given array by insertion sort
//Input: An array $A[0..n-1]$ of n orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 1$ **to** $n-1$ **do**
    $v \leftarrow A[i]$
    $j \leftarrow i-1$
    while $j \geq 0$ **and** $A[j] > v$ **do**
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j-1$
    $A[j+1] \leftarrow v$

The operation of insertion sort is illustrated through the example shown below:

89 |	**45**	68	90	29	34	17
45	89 |	**68**	90	29	34	17
45	68	89 |	**90**	29	34	17
45	68	89	90 |	**29**	34	17
29	45	68	89	90 |	**34**	17
29	34	45	68	89	90 |	**17**
17	29	34	45	68	89	90

Analysis:

Since for this algorithm the time complexity is dependent on the kind of input we do worst case and best case analysis.

Worst Case:
In the worst case, $A[j\ ] > v$ is executed the largest number of times, i.e., for every $j = i-1, \ldots, 0$. Since $v = A[i]$, it happens if and only if $A[j\ ] > A[i]$ for $j = i-1, \ldots, 0$ which means that the array is sorted in decreasing order. The number of key comparisons for such an input is:

$$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \Theta(n^2).$$

Q4 (b) Write the similarities and dissimilarities of BFS and DFS.
Sol:
Given below is the comparison between DFS and BFS:

Comparison of DFS and BFS:

	DFS	BFS
Data Structure	Stack	Queue
No. of Vertex Orderings	2 Orderings	1 Ordering
Edge Type (undirected graphs)	Tree and back edges	Tree and cross edges
Applications	Connectivity, acyclicity, articulation points	Connectivity, acyclicity, minimum edge paths
Efficiency for adjacency matrix	$\Theta(\|v\|^2)$	$\Theta(\|v\|^2)$
Efficiency for adjacency Linked List	$\Theta(\|V\|+\|E\|)$	$\Theta(\|V\|+\|E\|)$

Similarities are:

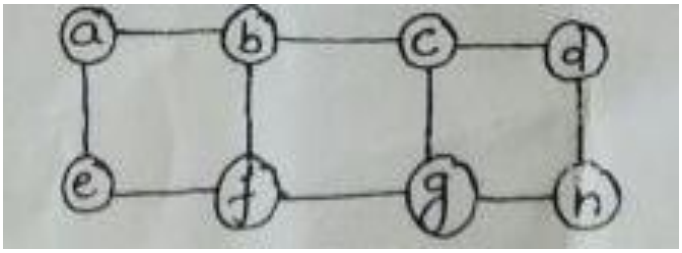
- Both have the same time complexity for any graph representation.
- Both can be used for checking for graph connectedness and acyclicity.
- Both algorithms can be traced by drawing a Traversal forests .

Differences:

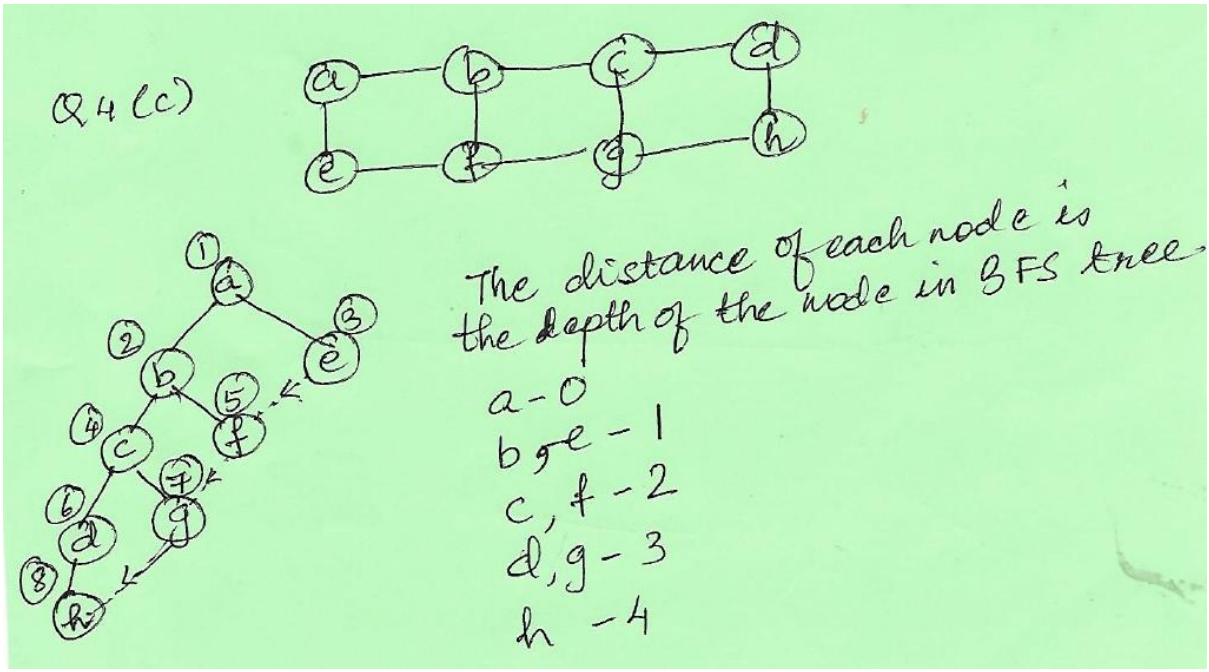
- DFS inherently uses a stack whereas BFS uses a queue
- The traversal forest for DFS has only back edges(for undirected graph) whereas that for BFS has cross edges.

BFS can be used to find minimum edge paths whereas DFS can be used for finding articulation points.

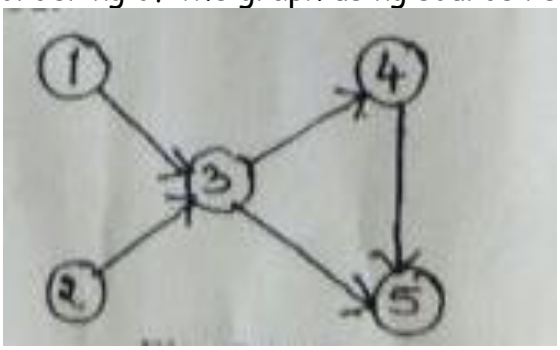
Q4 (c) Apply BFS on the following graph. Find for each vertex the distance from source. Consider the source as 'a'.



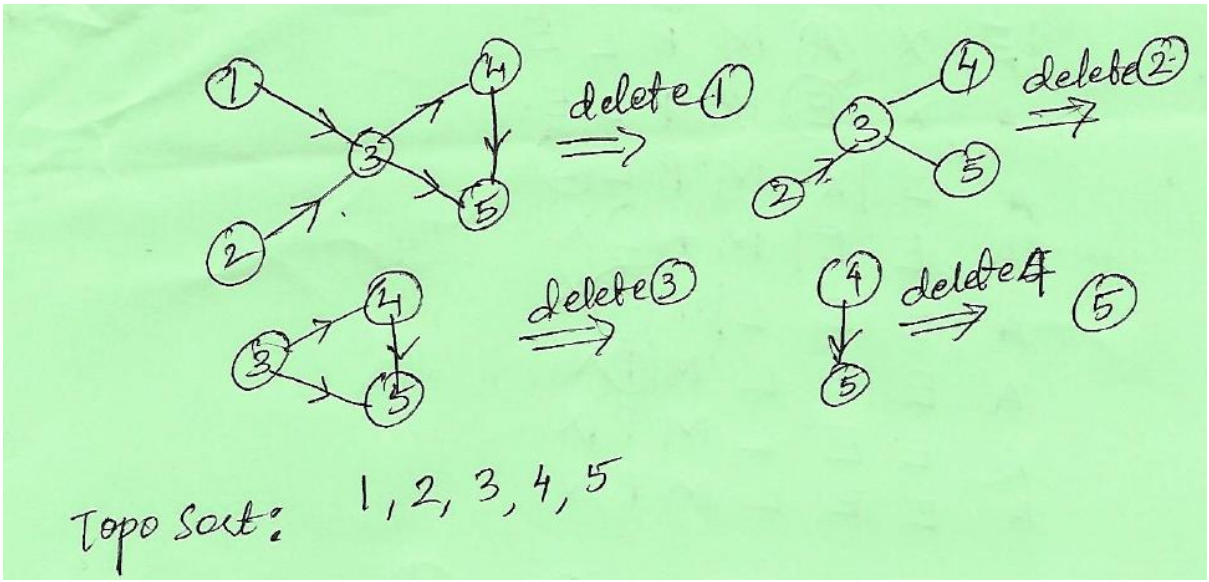
Sol:



Q4. (d) What do you mean by topological order of a graph? Find the topological ordering of the graph using source removal method.



Sol: 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.



Q5. (a)Write an algorithm for sorting by distribution counting . Apply the same algorithm to sort the element: 12 13 10, 12, 10, 12, 11, 10, 13.

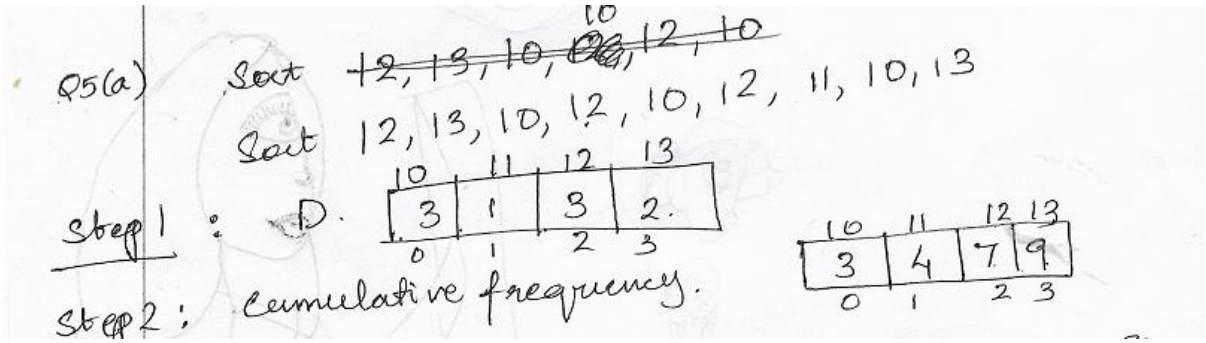
Sol:

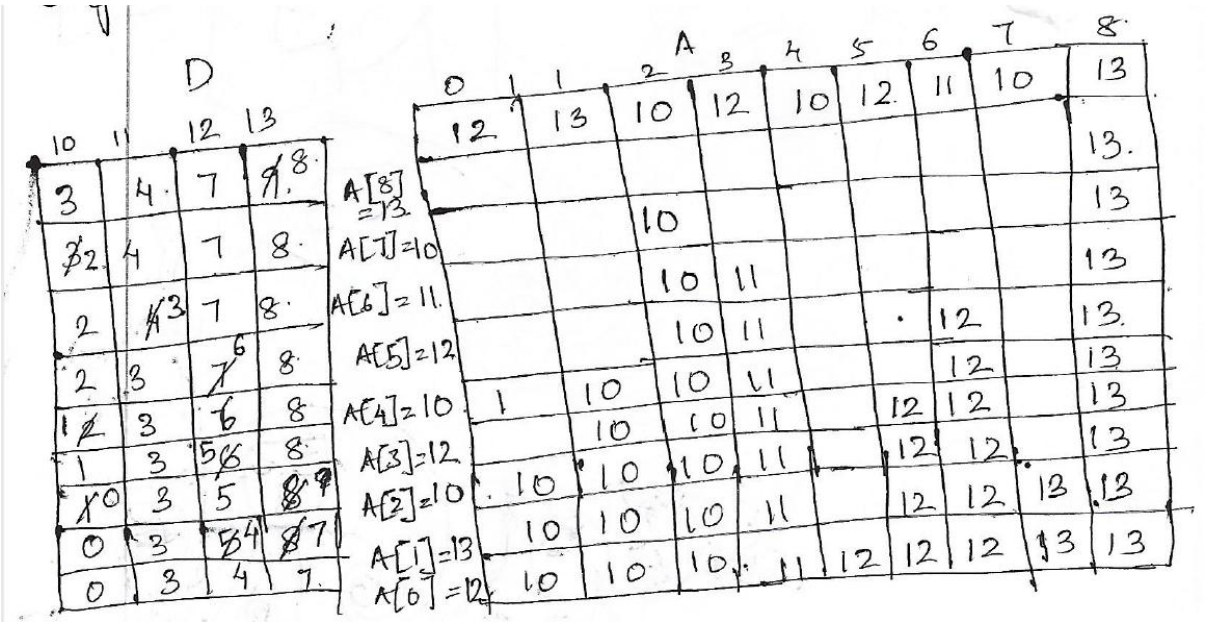
Counting sort algorithm is an example of a situation where using extra memory results in a efficient sorting technique. The idea is to count the frequencies of occurrence of each value in an array. Thus if the set of elements belong to a small range of numbers [l..u] then we can initially maintain a count of number of times each number occurs in an array F. Thus F[0] would store the frequency of occurance of l , F[1] would store the same of l+2 and F[u-l] would store the frequency of u occurring. The next step is to find the cumulative sum of elements in F. Thus F[i] would store the number of values in the original array which are less than the element associated with i, i.e. l+i. The elements of *A* whose values are equal to the lowest possible value *l* are copied into the first *F*[0]elements of *S*, i.e., positions 0 through *F*[0]– 1;the elements of value *l* + 1 are copied to positions from *F*[0] to (*F*[0]+ *F*[1]) – 1; and so on.. The algorithm for the same is given below:

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

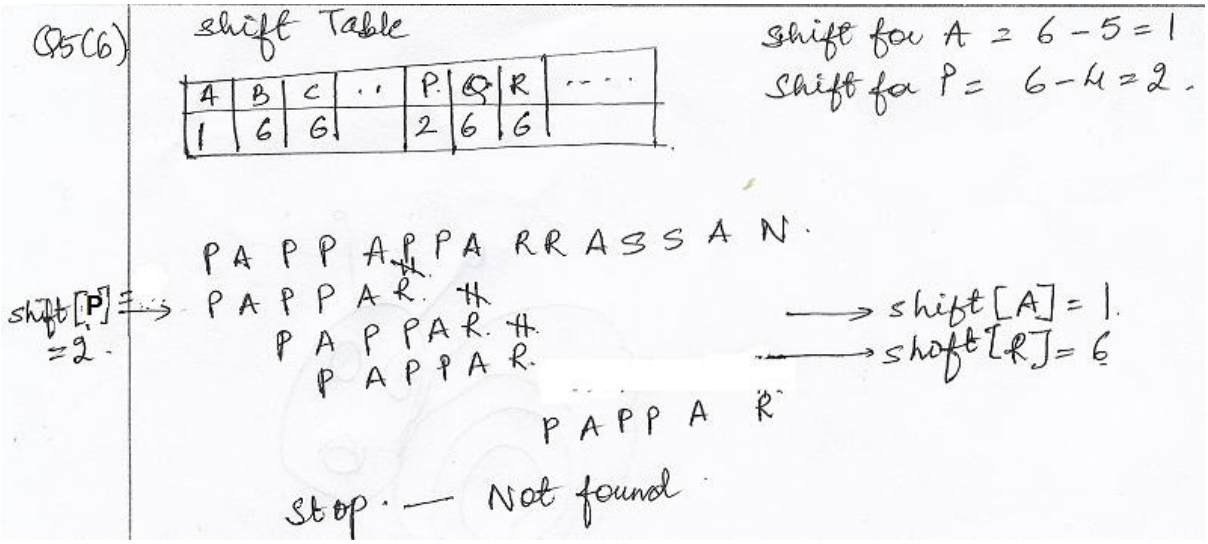
```
for j←0 to u-l do D[j]←0                    // initialize frequencies
for i←0 to n-1 do D[A[i]-l]←D[A[i]-l]+1     // compute frequencies
for j←1 to u-l do D[j]←D[j-1]+D[j]          //  reuse for distribution
for i←n-1 down to 0 do
        j  ←  A[i]-l
        S[D[j]-1]  ←  A[i]
        D[j]  ←  D[j]-1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





(Q5(b) Apply Horspool's algorithm to search a pattern PAPPAR in the text PAPPAPPAPPARRASSAN

Sol:



Q6((a) Write an algorithm for computing binomial coefficient C(n,K). What is the time efficiency of this algorithm?

Sol:

The formula for finding Binomial coefficient is :

C(n,r) = C(n-1,r)+C(n-1,r-1) , if n > r

= 1, if n=r or r=0

The formula for binomial coefficient is recursive and gives rise to overlapping subproblems and thus is naturally solved using dynamic programming approach. as follows:

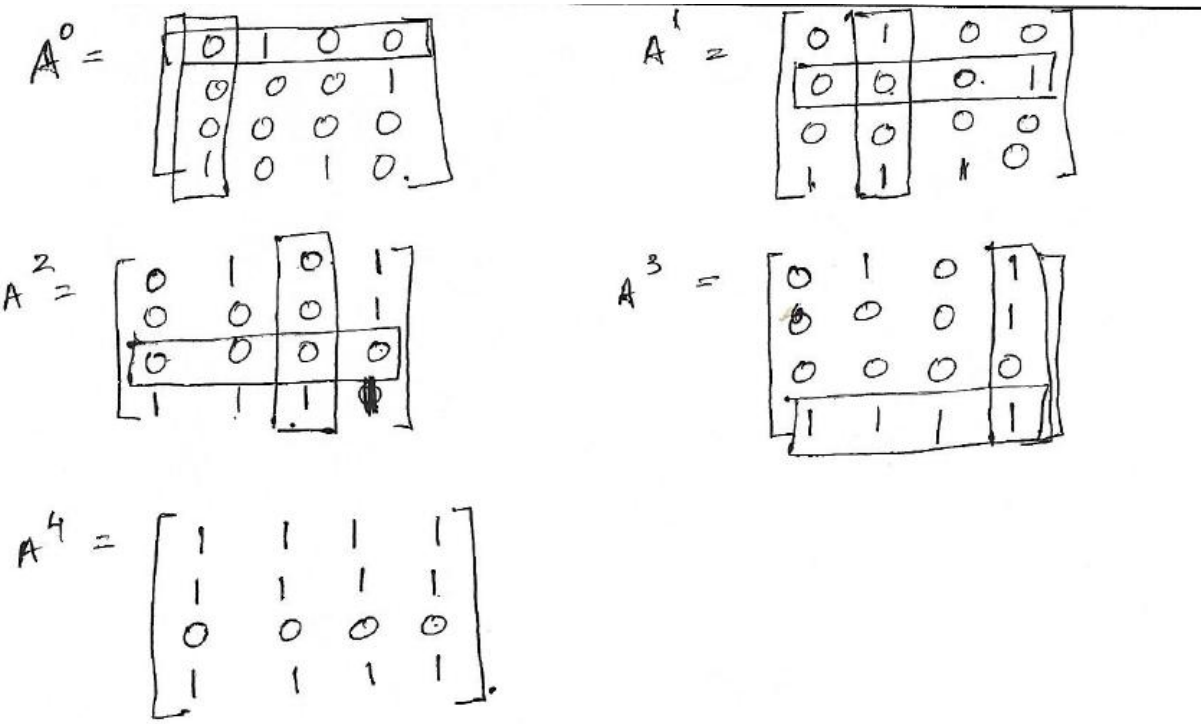
```
Algo BinCoeff(n,r)
        Create a matrix C(n+1,r+1)
        for i <-- 0 to n
                C(i,0 ) <-- 1
        for i<-- 1 to r
                C(i,i) <-- 1
        for i<-- 2 to n
                for j<-- 1 to min(r,i-1)
                        C(i,j) <-- C(i-1,j)+C(i-1,j-1)
```

This algorithm takes time O(nr) by avoiding repeated computation of overlapping terms.

Q6(b) Find the transitive closure for the graph whose adjacency matrix is given by:

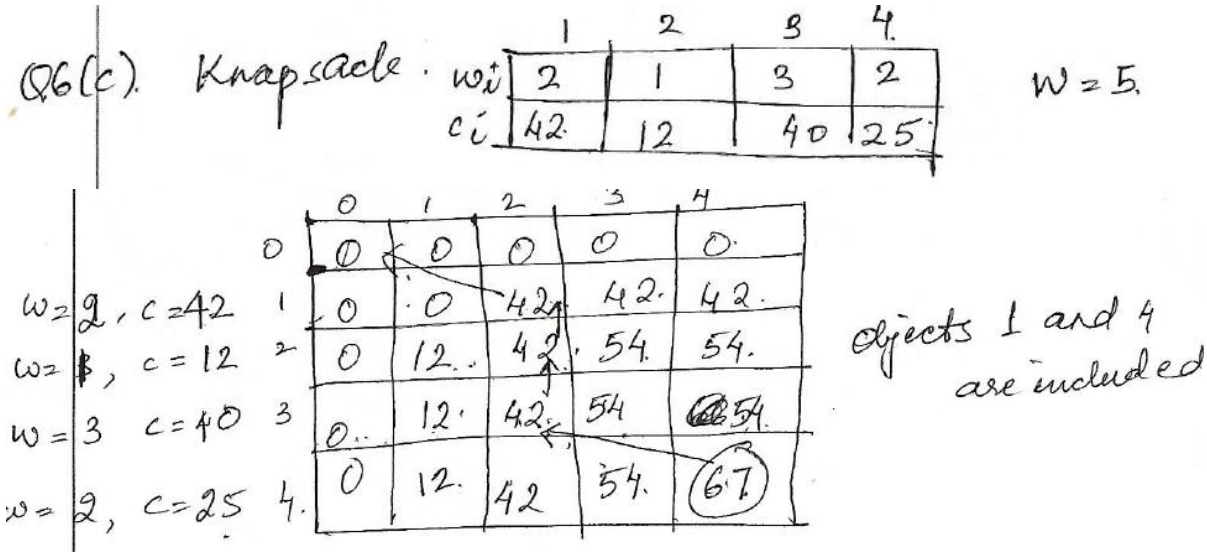
0 1 0 0
0 0 0 1
0 0 0 0
1 0 1 0

Sol:



(c) Apply bottom up dynamic programming to the following instance of knpsacl problem. Capacity of knapsack W=5

Sol:



Q7(a) Write an algorithm for Prim's minimal spanning tree algorithm.

Sol:

Prim'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.

Prim's algorithm constructs a minimum spanning tree through a sequence of expanding subtrees. The initial subtree in such a sequence consists of a single vertex selected arbitrarily from the set *V* of the graph's vertices. On each iteration, the algorithm expands the current tree in the greedy manner by simply attaching to it the nearest vertex(i.e. connected using the min weight) not in that tree. The algorithm stops after all the graph's vertices have been included in the tree being constructed. Since the algorithm expands a tree by exactly one vertex on each of its iterations, the total number of such iterations is *n* – 1, where *n* is the number of vertices in the graph. The pseudocode of this algorithm is as follows.

ALGORITHM *Prim(G)*

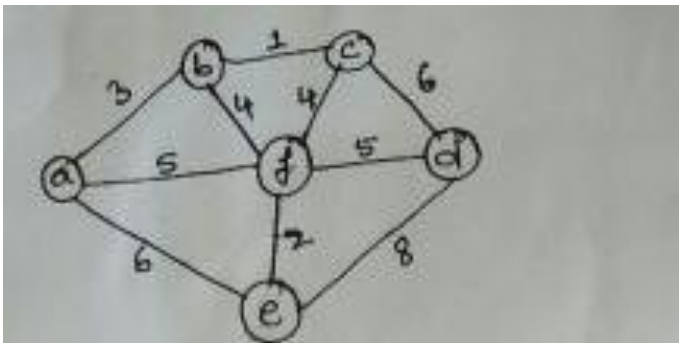
//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
$V_T \leftarrow \{v_0\}$ //the set of tree vertices can be initialized with any vertex
$E_T \leftarrow \varnothing$
for $i \leftarrow 1$ **to** $|V| - 1$ **do**
    find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges (v, u)
    such that v is in V_T and u is in $V - V_T$
    $V_T \leftarrow V_T \cup \{u^*\}$
    $E_T \leftarrow E_T \cup \{e^*\}$
return E_T

To implement Prim's algorithm we attach two labels to a vertex: the name of the nearest tree vertex and the length (the weight) of the corresponding edge. Vertices that are not adjacent to any of the tree vertices can be given the ∞ label indicating their "infinite" distance to the tree vertices and a null label for the name of the nearest tree vertex. With such labels,

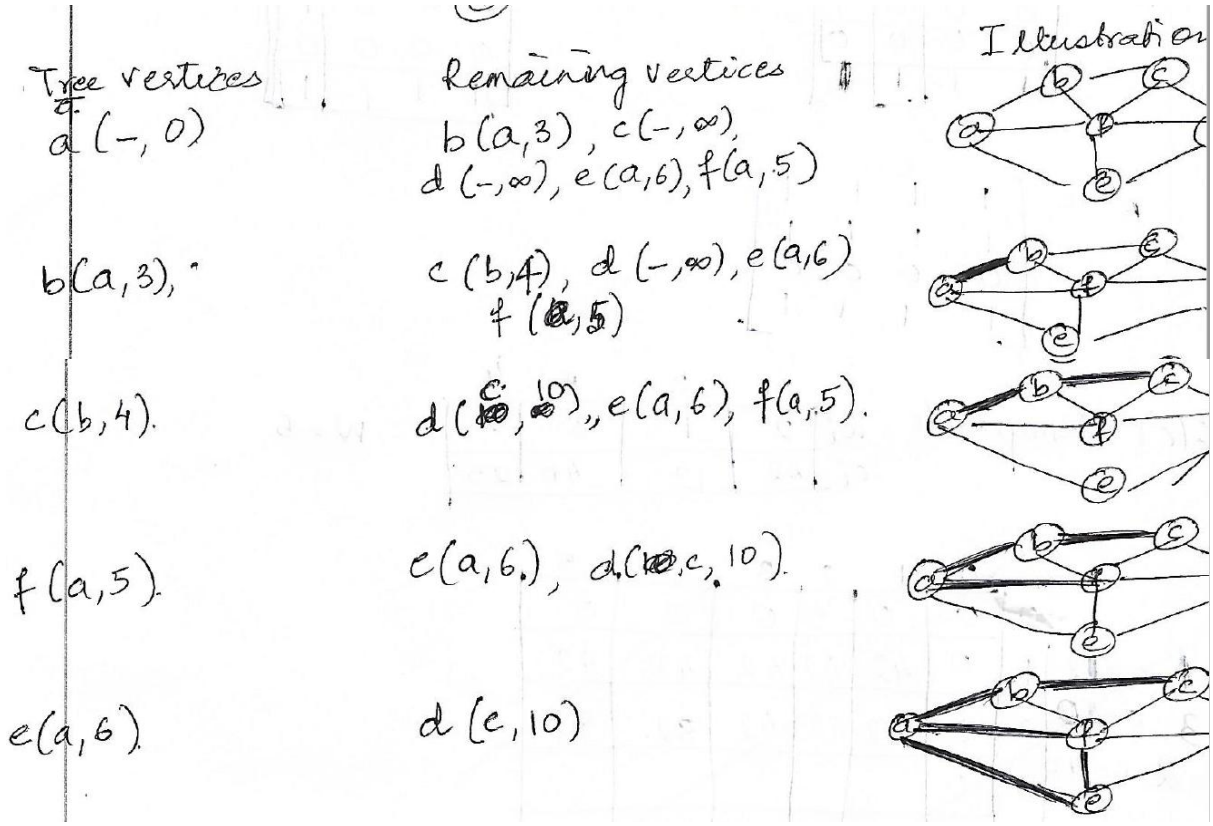
finding the next vertex to be added to the current tree $T = (V_T, E_T)$ becomes a simple task of finding a vertex with the smallest distance label in the set *V – VT*. Afterwehave identified a vertex u* to be added to the tree, we need to perform two operations:

- Move u^* from the set $V - V_T$ to the set of tree vertices V_T.
- For each remaining vertex u in $V - V_T$ that is connected to u^* by a shorter edge than the u's current distance label, update its labels by u^* and the weight of the edge between u^* and u, respectively.[2]

(b)Find shortest distance and shortest path from 'a' to all vertices in the graph.

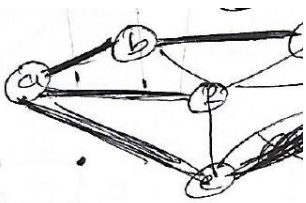


Sol:



d(e, 10)

	Cost	Route
a	0	—
b	3	a–b
c	4	a–b–c
d	10	a–b–c–d
e	6	a–e
f	5	a–f



(c) Construct Huffman code for the following data:

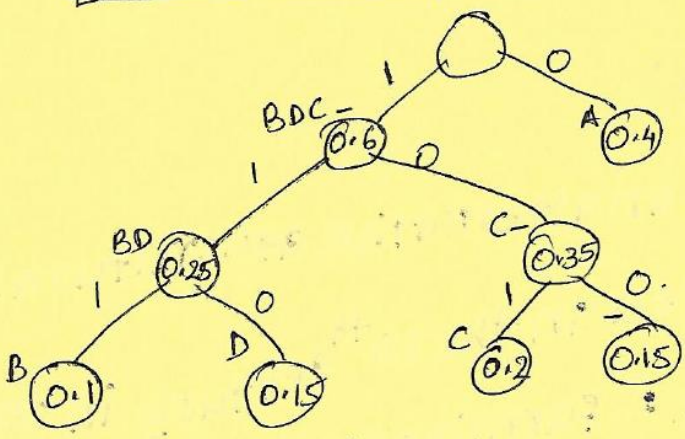
Character	A	B	C	D	- (underscore)
Probability	0.4	0.1	0.2	0.15	0.15

Encode the text ABCABC-AD

Decode the string whose encoding is 11111001010101

Sol:

A	B	C	D	–
0.4	0.1	0.2	0.15	0.15



A – 0
B – 111
C – 101
D – 110
E – – 100

Code for text – ABCABC–AD

0 111 101 111 101 100 0 110

Decode – 111 100 1 0 1 0 1 – BDACAC.

B D A C A C.

Q8 (a) Write a note on NP-Complete problems.

Sol:

Class NP is the class of decision problems that can be solved by nondeterministic polynomial algorithms. This class of problems is called **nondeterministic polynomial**.

The problems in class P are in NP because the polynomial time solution can be used for guessing and the result of verification can be ignored and hence . But in addition P also contains decision problems which currently don't have a polynomial time solution e.g. Hamiltonian circuit problem, knapsack, graph coloring etc. The question as to whether P = NP remains unanswered.

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

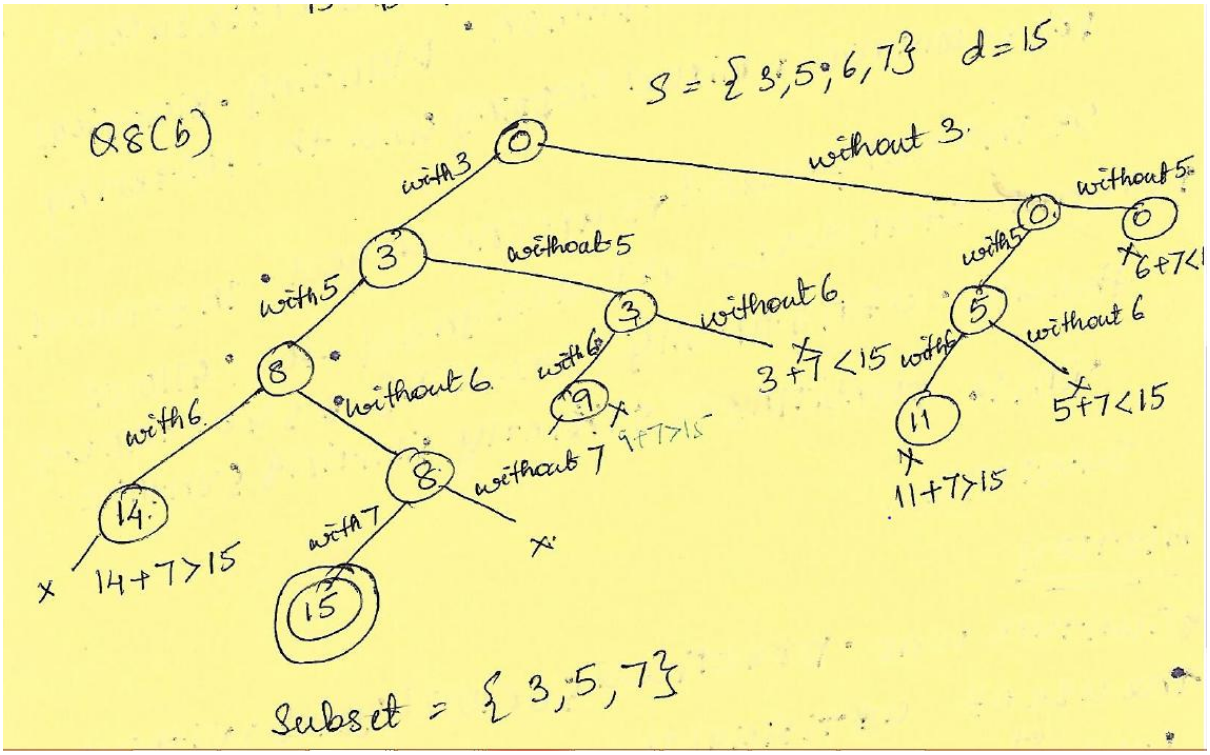
1. it belongs to class NP

2. every problem in NP(Q) is polynomially reducible to D i.e. it should be possible to change an instance of Q to an instance of D and get the answer of Q from the output of D in polynomial time.

The first example of NP complete problem(proved by Cook) is **CNF-satisfiability problem which is to determine given a Boolean expression** in CNF form whether or not one can assign values true and false to variables to make the entire expression true. Other examples of NP-Complete problems are : Hamiltonian circuit, traveling salesman, partition, bin packing,
and graph coloring etc.NP complete problems are very important because even if one of the problems are solvable in polynomial time then a wide variety of important problems would have a polynomial time solution.

(b) Find the subsets to form the given sum using backtracking
S={3,5,6,7} and d=15

Sol:



(c) Write a problem statement for the assignment problem and find the optimal solution for the instance with construction of state space tree

	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.

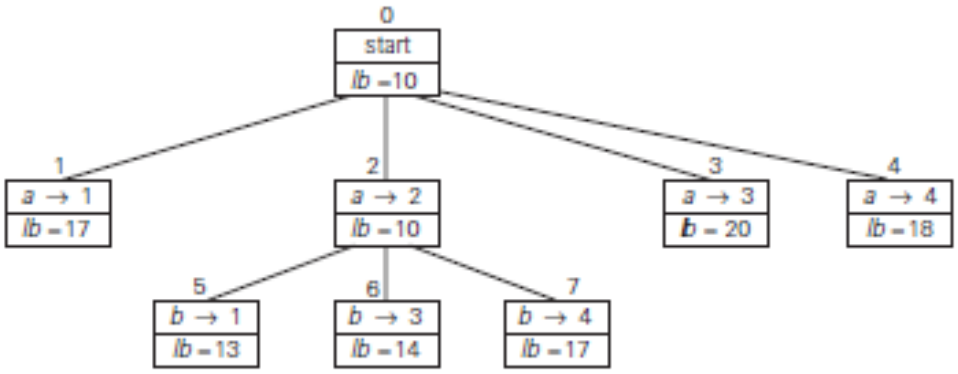
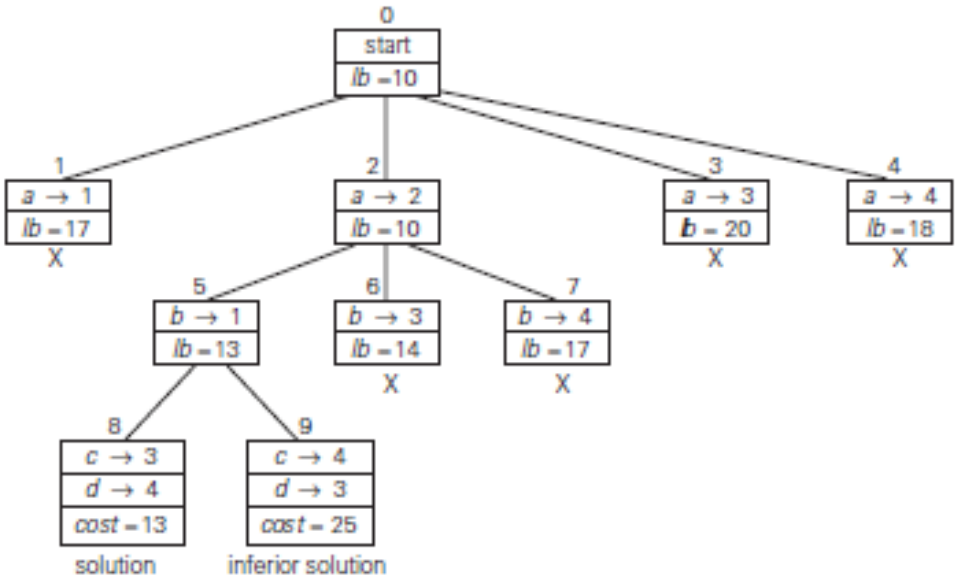


FIGURE 12.6 Levels 0, 1, and 2 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm.



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

(d) Differentiate branch and bound and backtracking techniques.

Sol: Backtracking and branch and bound are similar to each other because they are normally used to solve problems whose state space grows exponential. To save time in searching in the huge search space a method is employed to prune some states which would not lead to a solution. Backtracking checks for a partially constructed solution satisfying constraints whereas branch and bound also uses an additional bound per state which decides if a state is promising or not. If a state is unpromising then the state is not explored/expanded.

Backtracking	Branch and bound
Used for nonoptimization problems	Used for solving optimization problems
Uses a DFS	Sometimes uses BFS to implement because all states at a level are produced to check which is the most promising.
Does not use any bound. Checks if partial solution follows the constraints	Defines a lower/upper bound for every state which is used for 2 purposes: To determine the next most promising state. To use the bound to prune a state by comparing its lower(upper bound) of the already found solution.