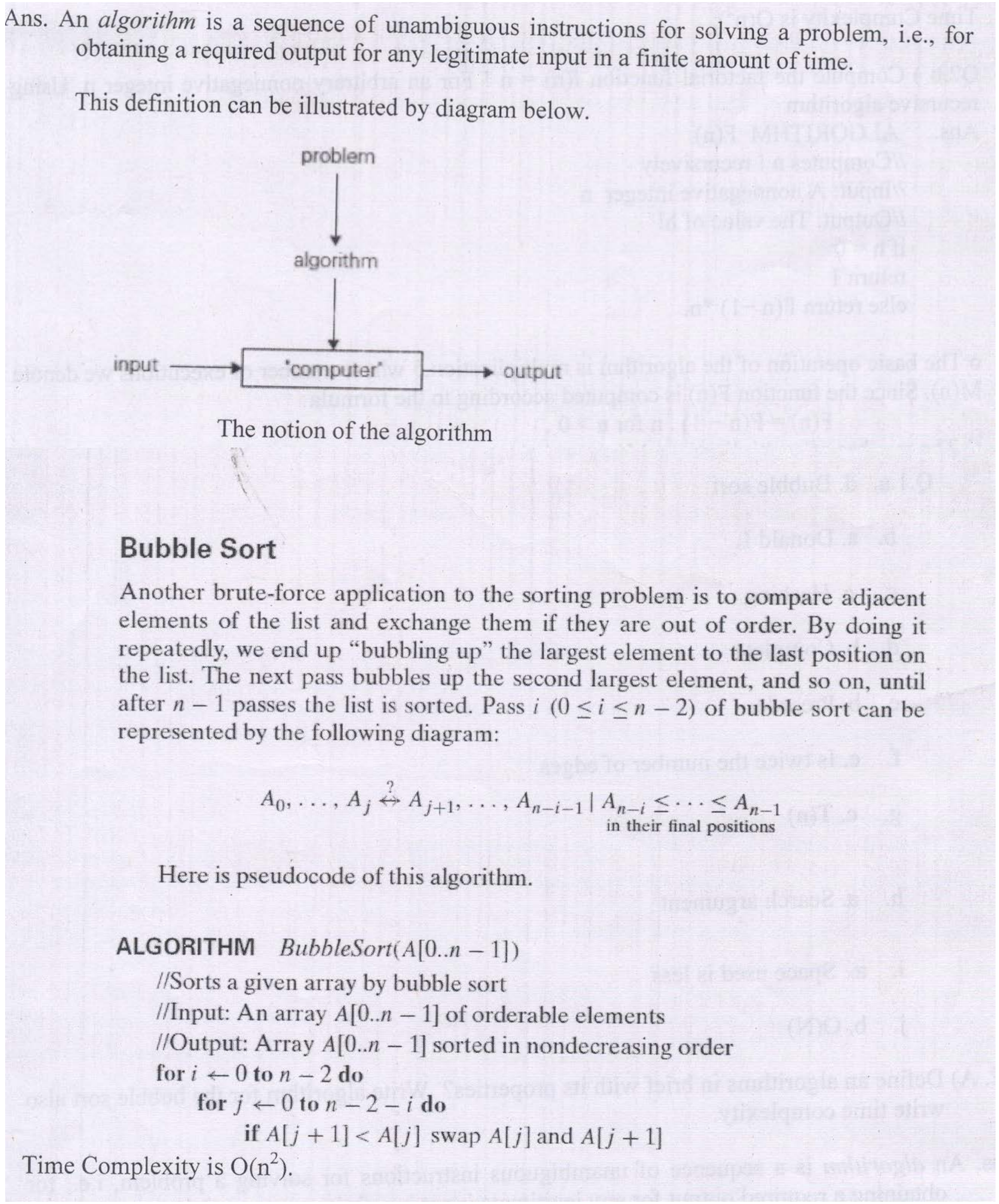
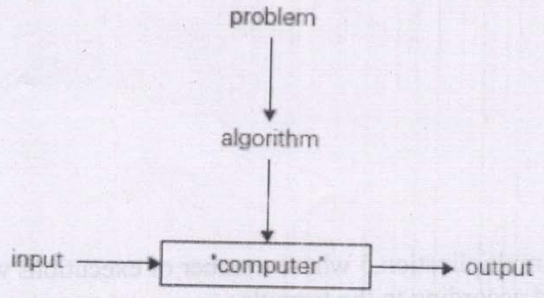
Q2 (a) Define an algorithm in brief with its properties. Write algorithm for the bubble sort.

Answer



Ans. 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.

This definition can be illustrated by diagram below.



The notion of the algorithm

Bubble Sort

Another brute-force application to the sorting problem is to compare adjacent elements of the list and exchange them if they are out of order. By doing it repeatedly, we end up "bubbling up" the largest element to the last position on the list. The next pass bubbles up the second largest element, and so on, until after $n - 1$ passes the list is sorted. Pass i $(0 \le i \le n - 2)$ of bubble sort can be represented by the following diagram:

$$A_0, \ldots, A_j \overset{?}{\leftrightarrow} A_{j+1}, \ldots, A_{n-i-1} \mid \underbrace{A_{n-i} \le \cdots \le A_{n-1}}_{\text{in their final positions}}$$

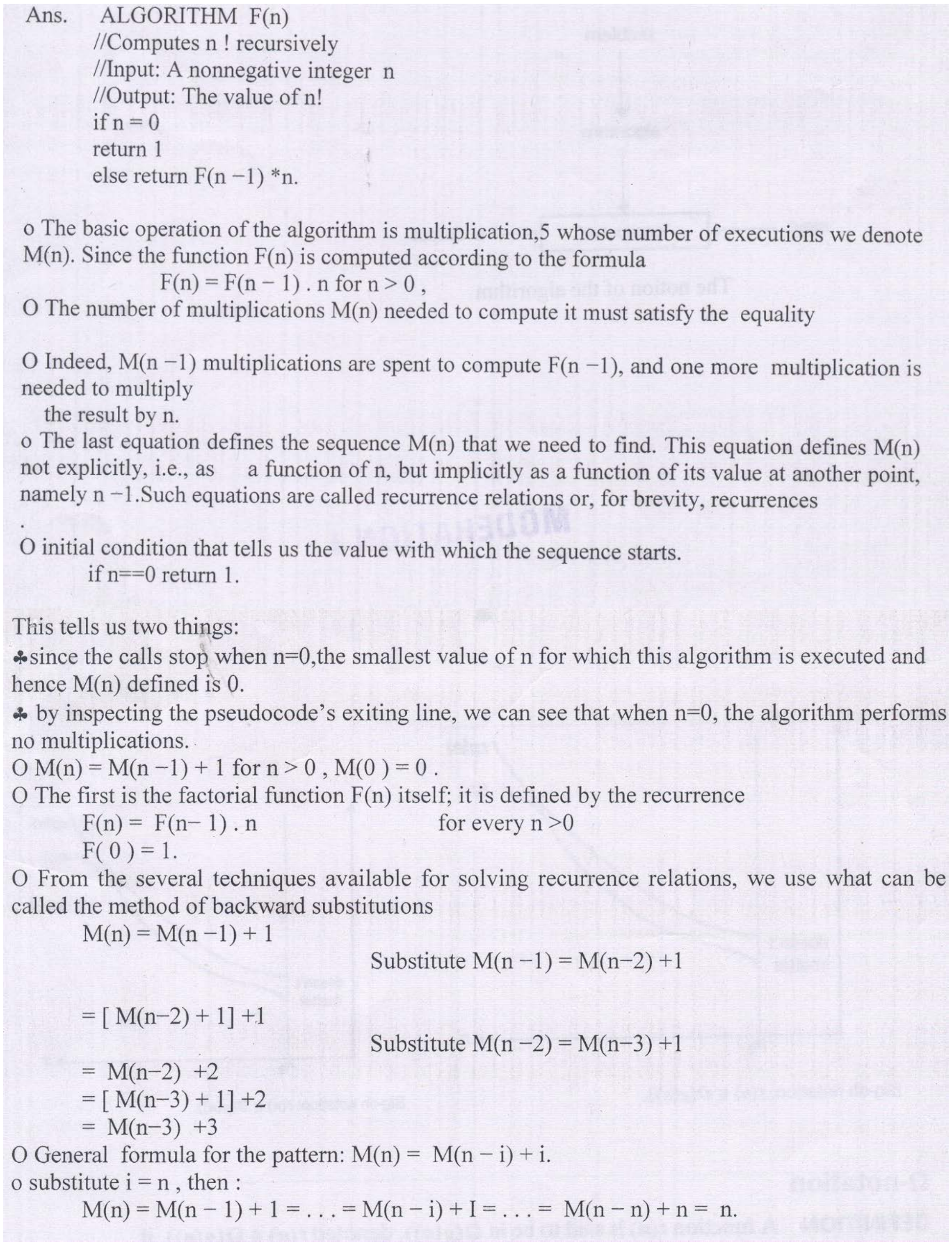
Here is pseudocode of this algorithm.

```
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]
```

Time Complexity is $O(n^2)$.

Q2 (b) Write a recursive algorithm to compute value of the factorial function f(n) = n!, for an arbitrary nonnegative integer n. Describe its time complexity.

Answer



Ans.

```
ALGORITHM F(n)
//Computes n ! recursively
//Input: A nonnegative integer n
//Output: The value of n!
if n = 0
return 1
else return F(n −1) *n.
```

o The basic operation of the algorithm is multiplication,5 whose number of executions we denote M(n). Since the function F(n) is computed according to the formula

$F(n) = F(n-1) . n$ for $n > 0$,

O The number of multiplications M(n) needed to compute it must satisfy the equality

O Indeed, M(n −1) multiplications are spent to compute F(n −1), and one more multiplication is needed to multiply
the result by n.

o The last equation defines the sequence M(n) that we need to find. This equation defines M(n) not explicitly, i.e., as a function of n, but implicitly as a function of its value at another point, namely n −1.Such equations are called recurrence relations or, for brevity, recurrences
.

O initial condition that tells us the value with which the sequence starts.
if n==0 return 1.

This tells us two things:

♣since the calls stop when n=0,the smallest value of n for which this algorithm is executed and hence M(n) defined is 0.

♣ by inspecting the pseudocode's exiting line, we can see that when n=0, the algorithm performs no multiplications.

O $M(n) = M(n-1) + 1$ for $n > 0$, $M(0) = 0$.

O The first is the factorial function F(n) itself; it is defined by the recurrence

$F(n) = F(n-1) . n$ for every $n > 0$

$F(0) = 1$.

O From the several techniques available for solving recurrence relations, we use what can be called the method of backward substitutions

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

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

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

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

$= M(n-2) + 2$

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

$= M(n-3) + 3$

O General formula for the pattern: $M(n) = M(n-i) + i$.

o substitute $i = n$, then :

$M(n) = M(n-1) + 1 = . . . = M(n-i) + I = . . . = M(n-n) + n = n$.

Q3 (a) Explain asymptotic notations and their basic efficiency classes.

Answer

Ans. • The efficiency analysis framework concentrates on the order of growth of an algorithm's basic operation count as the principal indicator of the algorithm's efficiency.

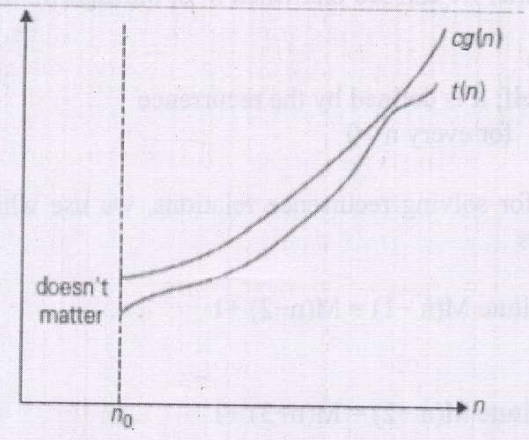
• To compare and rank such orders of growth, computer scientists use three notations: O (big oh), Ω (big omega) and Θ (big theta).

• t (n) and g(n) can be any nonnegative functions defined on the set of natural numbers.

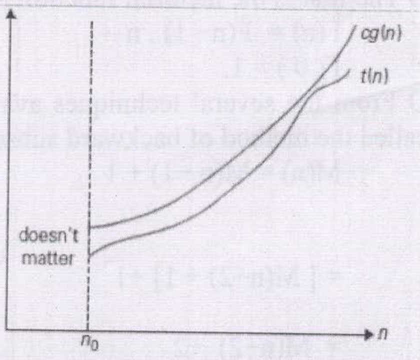
***O*-notation**

The Θ-notation asymptotically bounds a function from above and below. When we have only an ***asymptotic upper bound***, we use *O*-notation. For a given function $g(n)$, we denote by $O(g(n))$ (pronounced "big-oh of g of n" or sometimes just "oh of g of n") the set of functions

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



Big-oh notation: $t(n) \in O(g(n))$.



Big-oh notation: $t(n) \in O(g(n))$.

Ω-notation

DEFINITION A function $t(n)$ is said to be in $\Omega(g(n))$, 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, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

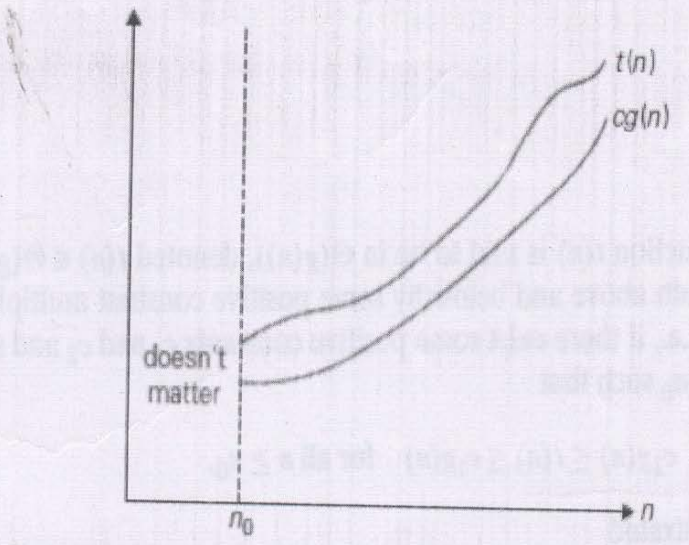
$$t(n) \ge cg(n) \quad \text{for all } n \ge n_0.$$

The definition is illustrated in

Here is an example of the formal proof that $n^3 \in \Omega(n^2)$:

$$n^3 \ge n^2 \quad \text{for all } n \ge 0,$$

i.e., we can select $c = 1$ and $n_0 = 0$.



Big-omega notation: $t(n) \in \Omega(g(n))$.

Θ-notation

DEFINITION A function $t(n)$ is said to be in $\Theta(g(n))$, denoted $t(n) \in \Theta(g(n))$, if $t(n)$ is bounded both above and below by some positive constant multiples of $g(n)$ for all large n, i.e., if there exist some positive constants c_1 and c_2 and some nonnegative integer n_0 such that

$$c_2 g(n) \le t(n) \le c_1 g(n) \quad \text{for all } n \ge n_0.$$

The definition is illustrated

For example, let us prove that $\frac{1}{2}n(n-1) \in \Theta(n^2)$. First, we prove the right inequality (the upper bound):

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \le \frac{1}{2}n^2 \quad \text{for all } n \ge 0.$$

Second, we prove the left inequality (the lower bound):

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \ge \frac{1}{2}n^2 - \frac{1}{2}n\frac{1}{2}n \text{ (for all } n \ge 2) = \frac{1}{4}n^2.$$

Hence, we can select $c_2 = \frac{1}{4}$, $c_1 = \frac{1}{2}$, and $n_0 = 2$.

Θ-notation

DEFINITION A function $t(n)$ is said to be in $\Theta(g(n))$, denoted $t(n) \in \Theta(g(n))$, if $t(n)$ is bounded both above and below by some positive constant multiples of $g(n)$ for all large n, i.e., if there exist some positive constants c_1 and c_2 and some nonnegative integer n_0 such that

$$c_2 g(n) \le t(n) \le c_1 g(n) \quad \text{for all } n \ge n_0.$$

The definition is illustrated

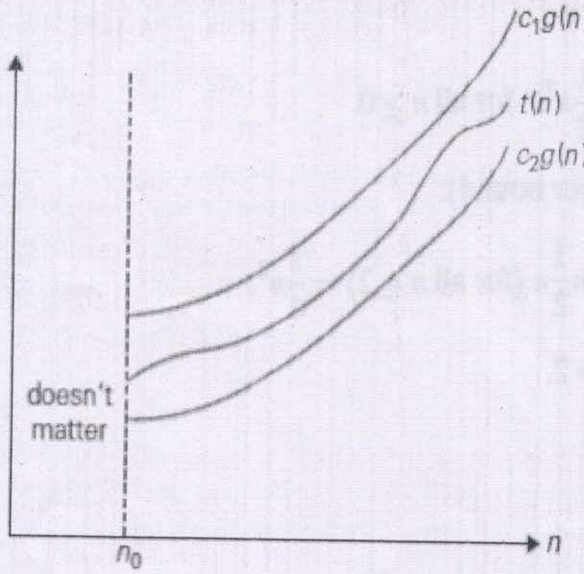
For example, let us prove that $\frac{1}{2}n(n-1) \in \Theta(n^2)$. First, we prove the right inequality (the upper bound):

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \le \frac{1}{2}n^2 \quad \text{for all } n \ge 0.$$

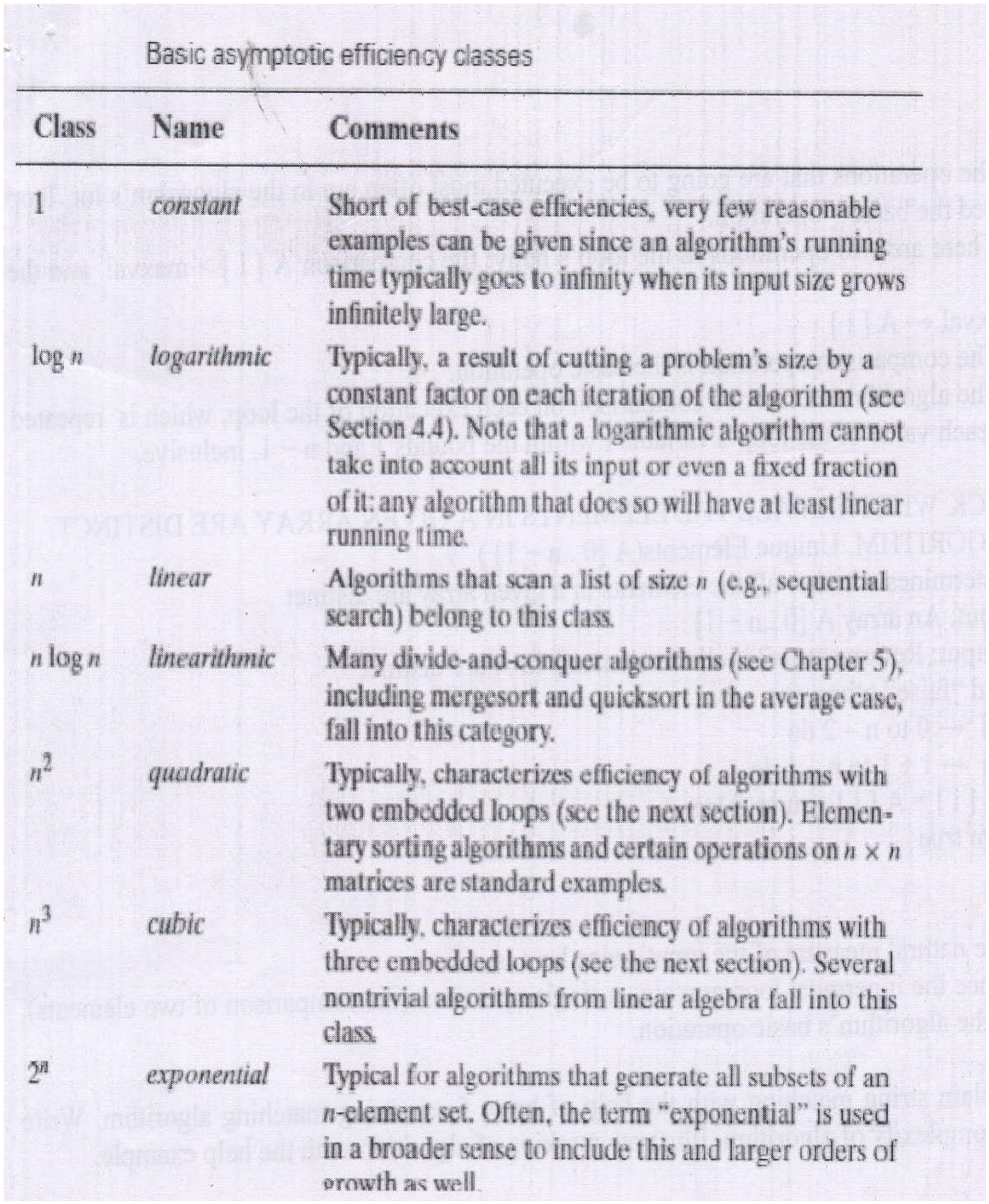
Second, we prove the left inequality (the lower bound):

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \ge \frac{1}{2}n^2 - \frac{1}{2}n\frac{1}{2}n \text{ (for all } n \ge 2) = \frac{1}{4}n^2.$$

Hence, we can select $c_2 = \frac{1}{4}$, $c_1 = \frac{1}{2}$, and $n_0 = 2$.



Big-theta notation: $t(n) \in \Theta(g(n))$.



Basic asymptotic efficiency classes

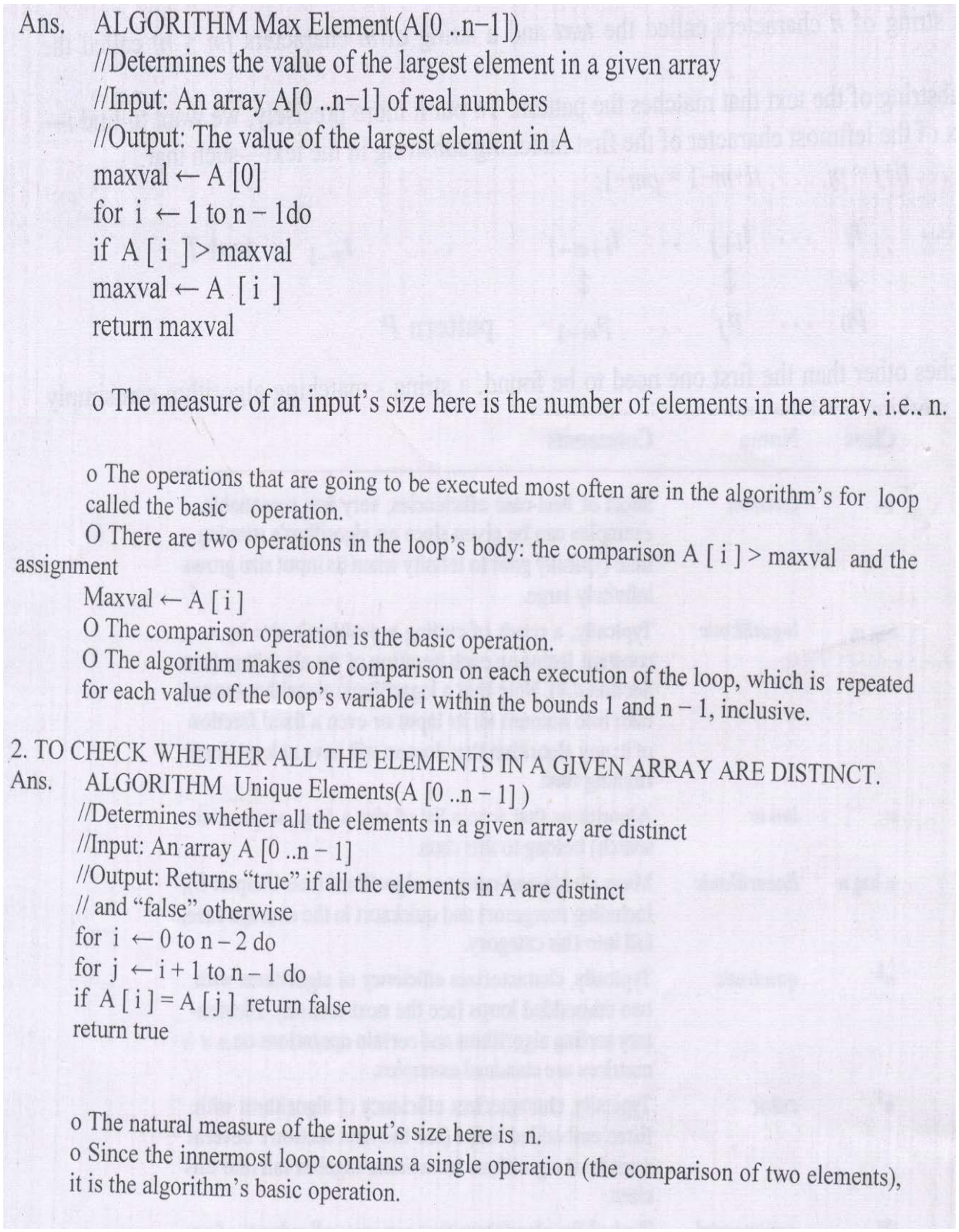
Class	Name	Comments
1	*constant*	Short of best-case efficiencies, very few reasonable examples can be given since an algorithm's running time typically goes to infinity when its input size grows infinitely large.
$\log n$	*logarithmic*	Typically, a result of cutting a problem's size by a constant factor on each iteration of the algorithm (see Section 4.4). Note that a logarithmic algorithm cannot take into account all its input or even a fixed fraction of it: any algorithm that does so will have at least linear running time.
n	*linear*	Algorithms that scan a list of size n (e.g., sequential search) belong to this class.
$n \log n$	*linearithmic*	Many divide-and-conquer algorithms (see Chapter 5), including mergesort and quicksort in the average case, fall into this category.
n^2	*quadratic*	Typically, characterizes efficiency of algorithms with two embedded loops (see the next section). Elementary sorting algorithms and certain operations on $n \times n$ matrices are standard examples.
n^3	*cubic*	Typically, characterizes efficiency of algorithms with three embedded loops (see the next section). Several nontrivial algorithms from linear algebra fall into this class.
2^n	*exponential*	Typical for algorithms that generate all subsets of an n-element set. Often, the term "exponential" is used in a broader sense to include this and larger orders of growth as well.

Q3 (b) Analyze the time complexity for the following using non recursive algorithms.

(i) Finding the value of the largest element in a list of n numbers

(ii) To check whether all the elements in a given array are distinct.

Answer



Ans.
```
ALGORITHM Max Element(A[0 ..n−1])
//Determines the value of the largest element in a given array
//Input: An array A[0 ..n−1] of real numbers
//Output: The value of the largest element in A
maxval ← A [0]
for i ← 1 to n − 1do
    if A [ i ] > maxval
        maxval ← A [ i ]
return maxval
```

o The measure of an input's size here is the number of elements in the array, i.e.. n.

o The operations that are going to be executed most often are in the algorithm's for loop called the basic operation.

O There are two operations in the loop's body: the comparison A [i] > maxval and the assignment

Maxval ← A [i]

O The comparison operation is the basic operation.

O The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable i within the bounds 1 and n − 1, inclusive.

2. TO CHECK WHETHER ALL THE ELEMENTS IN A GIVEN ARRAY ARE DISTINCT.

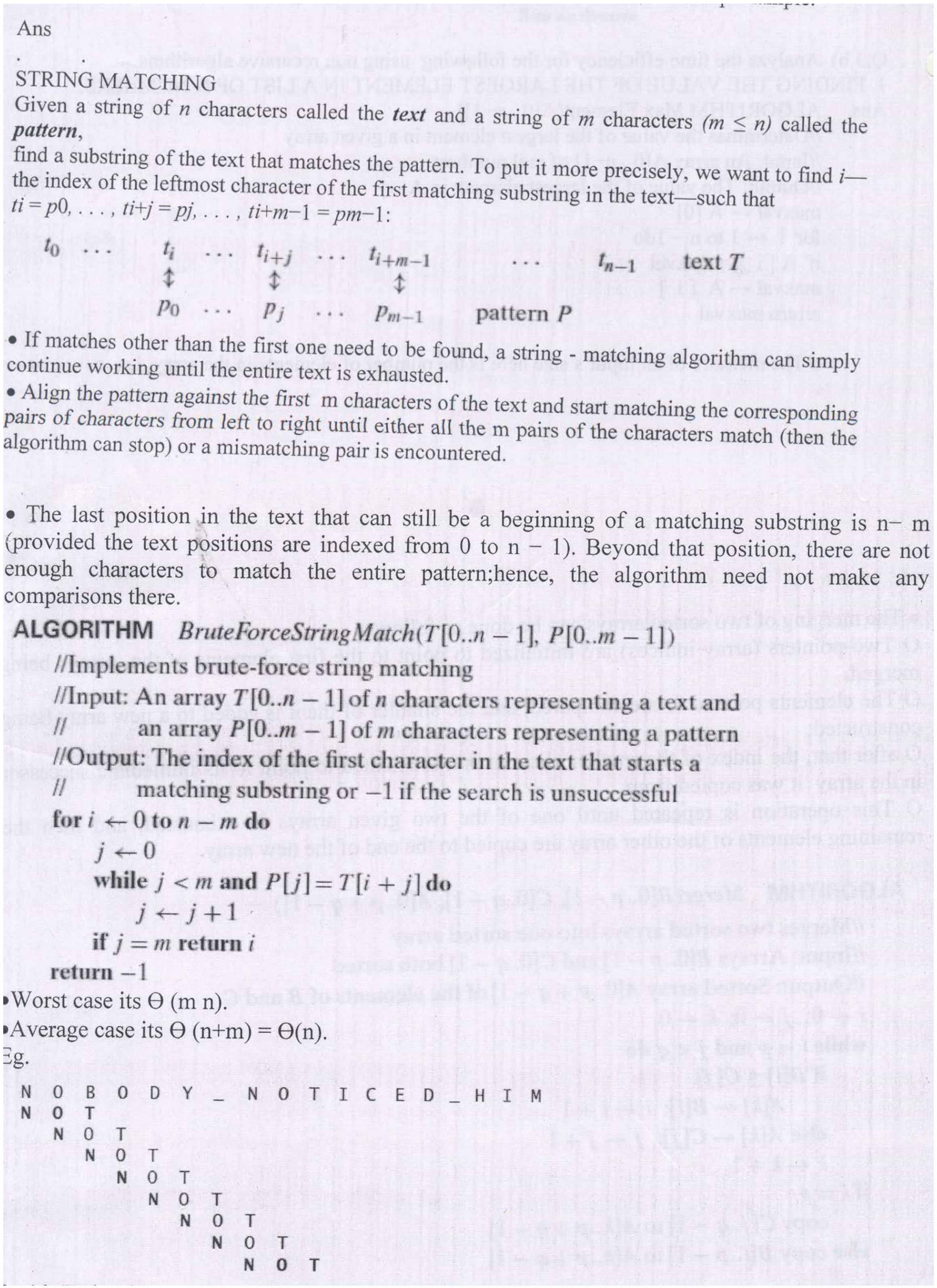
Ans.
```
ALGORITHM Unique Elements(A [0 ..n − 1] )
//Determines whether all the elements in a given array are distinct
//Input: An array A [0 ..n − 1]
//Output: Returns "true" if all the elements in A are distinct
// and "false" otherwise
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A [ i ] = A [ j ] return false
return true
```

o The natural measure of the input's size here is n.

o Since the innermost loop contains a single operation (the comparison of two elements), it is the algorithm's basic operation.

Q4 (a) Explain string matching with the help of brute force string matching algorithm. Write down time complexity of this algorithm. Illustrate working of this algorithm with the help of an example.

Answer



Ans

STRING MATCHING

Given a string of n characters called the ***text*** and a string of m characters $(m \le n)$ called the ***pattern***,
find a substring of the text that matches the pattern. To put it more precisely, we want to find i—the index of the leftmost character of the first matching substring in the text—such that
$t_i = p_0, \ldots, t_{i+j} = p_j, \ldots, t_{i+m-1} = p_{m-1}$:

$$\begin{array}{ccccccccc} t_0 & \ldots & t_i & \ldots & t_{i+j} & \ldots & t_{i+m-1} & \ldots & t_{n-1} \quad \textbf{text } T \\ & & \updownarrow & & \updownarrow & & \updownarrow & & \\ & & p_0 & \ldots & p_j & \ldots & p_{m-1} & & \textbf{pattern } P \end{array}$$

• If matches other than the first one need to be found, a string - matching algorithm can simply continue working until the entire text is exhausted.

• Align the pattern against the first m characters of the text and start matching the corresponding pairs of characters from left to right until either all the m pairs of the characters match (then the algorithm can stop) or a mismatching pair is encountered.

• The last position in the text that can still be a beginning of a matching substring is n− m (provided the text positions are indexed from 0 to n − 1). Beyond that position, there are not enough characters to match the entire pattern;hence, the algorithm need not make any comparisons there.

```
ALGORITHM  BruteForceStringMatch(T[0..n − 1], P[0..m − 1])
    //Implements brute-force string matching
    //Input: An array T[0..n − 1] of n characters representing a text and
    //       an array P[0..m − 1] of m characters representing a pattern
    //Output: The index of the first character in the text that starts a
    //        matching substring or −1 if the search is unsuccessful
    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
```

•Worst case its Θ (m n).

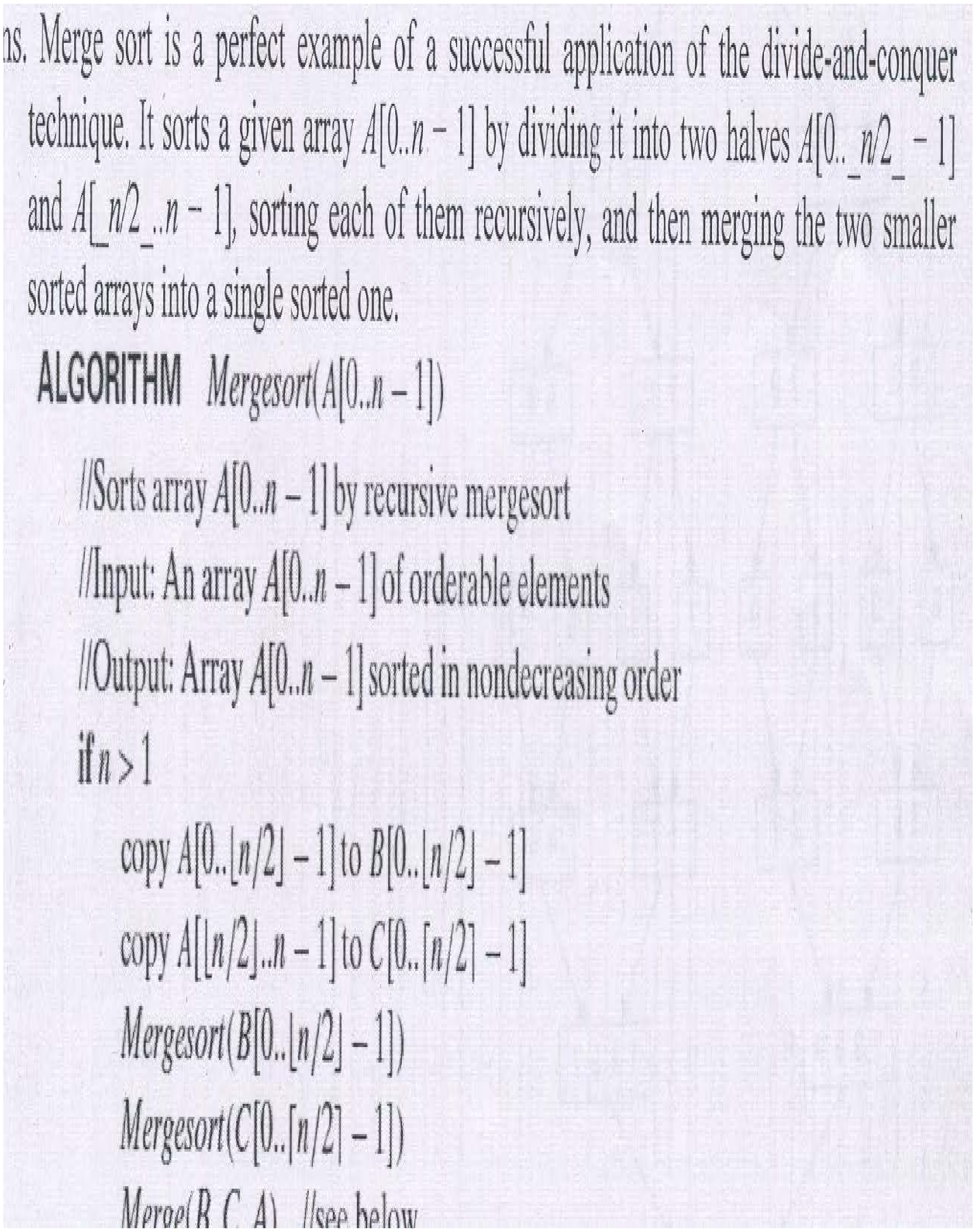
•Average case its Θ (n+m) = Θ(n).

Eg.

```
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
```

Q4 (b) Write down Merge sort algorithm. Explain functioning of this algorithm with the help of an example. What is the time complexity of this algorithm?

Answer



ns. Merge sort 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.

```
ALGORITHM   Mergesort(A[0..n − 1])
    //Sorts array A[0..n − 1] by recursive mergesort
    //Input: An array A[0..n − 1] of orderable elements
    //Output: Array A[0..n − 1] sorted in nondecreasing order
    if n > 1
        copy A[0..⌊n/2⌋ − 1] to B[0..⌊n/2⌋ − 1]
        copy A[⌊n/2⌋..n − 1] to C[0..⌈n/2⌉ − 1]
        Mergesort(B[0..⌊n/2⌋ − 1])
        Mergesort(C[0..⌈n/2⌉ − 1])
        Merge(B, C, A)   //see below
```

• The merging of two sorted arrays can be done as follows:

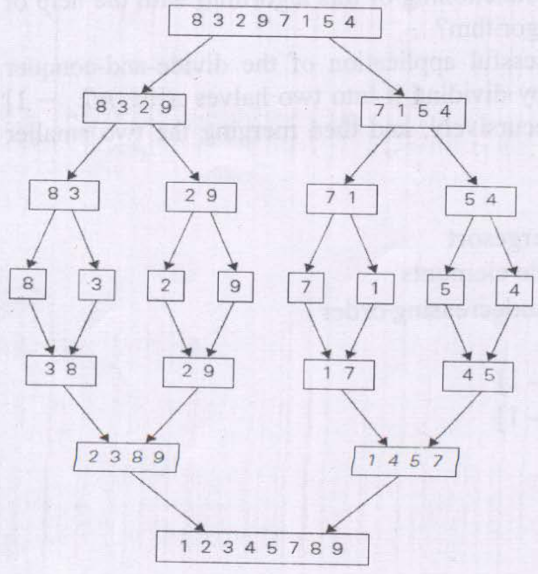
O Two pointers (array indices) are initialized to point to the first elements of the arrays being merged.

O The elements pointed to are compared, and the smaller of them is added to a new array being constructed;

O after that, the index of the smaller element is incremented to point to its immediate successor in the array it was copied from.

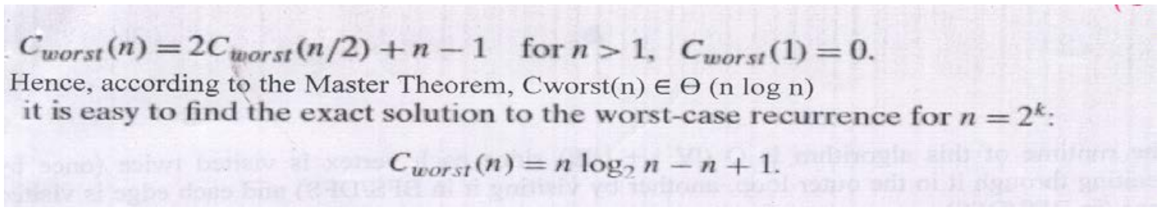
O This operation is repeated until one of the two given arrays is exhausted, and then the remaining elements of the other array are copied to the end of the new array

```
ALGORITHM  Merge(B[0..p − 1], C[0..q − 1], A[0..p + q − 1])
    //Merges two sorted arrays into one sorted array
    //Input: Arrays B[0..p − 1] and C[0..q − 1] both sorted
    //Output: Sorted array A[0..p + q − 1] of the elements of B and C
    i ← 0;  j ← 0;  k ← 0
    while i < p and j < q do
        if B[i] ≤ C[j]
            A[k] ← B[i];  i ← i + 1
        else A[k] ← C[j];  j ← j + 1
        k ← k + 1
    if i = p
        copy C[j..q − 1] to A[k..p + q − 1]
    else copy B[i..p − 1] to A[k..p + q − 1]
```



Example of mergesort operation.

$$C(n) = 2C(n/2) + C_{merge}(n) \quad \text{for } n > 1, \quad C(1) = 0.$$



$$C_{worst}(n) = 2C_{worst}(n/2) + n - 1 \quad \text{for } n > 1,\ C_{worst}(1) = 0.$$

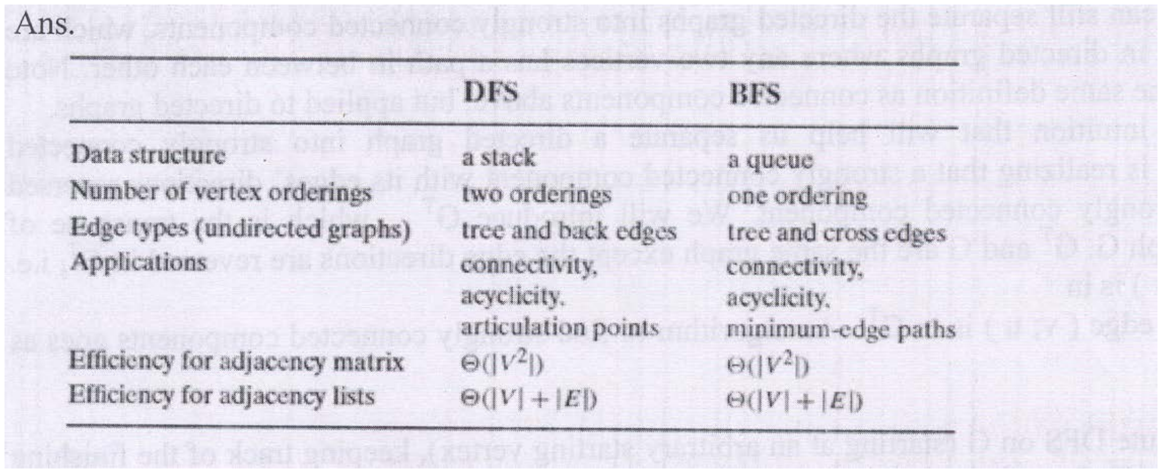
Hence, according to the Master Theorem, Cworst(n) ∈ Θ (n log n)

it is easy to find the exact solution to the worst-case recurrence for $n = 2^k$:

$$C_{worst}(n) = n \log_2 n - n + 1.$$

Q5 (a) What are the main difference of BFS over DFS? Which one is preferable and when?

Answer

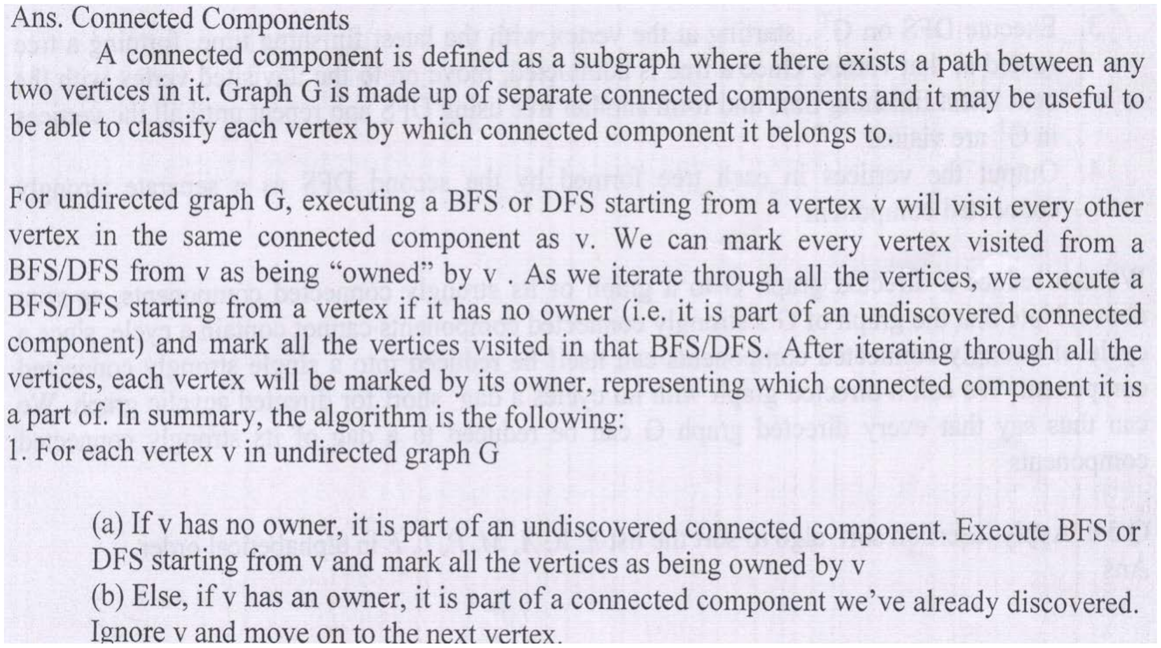


Ans.

	DFS	BFS
Data structure	a stack	a queue
Number of vertex orderings	two orderings	one ordering
Edge types (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 lists	$\Theta(\|V\| + \|E\|)$	$\Theta(\|V\| + \|E\|)$

Q5 (b) Explain how one can identify Connected & Strongly Connected Components of a graph by using DFS & BFS.

Answer



Ans. Connected Components

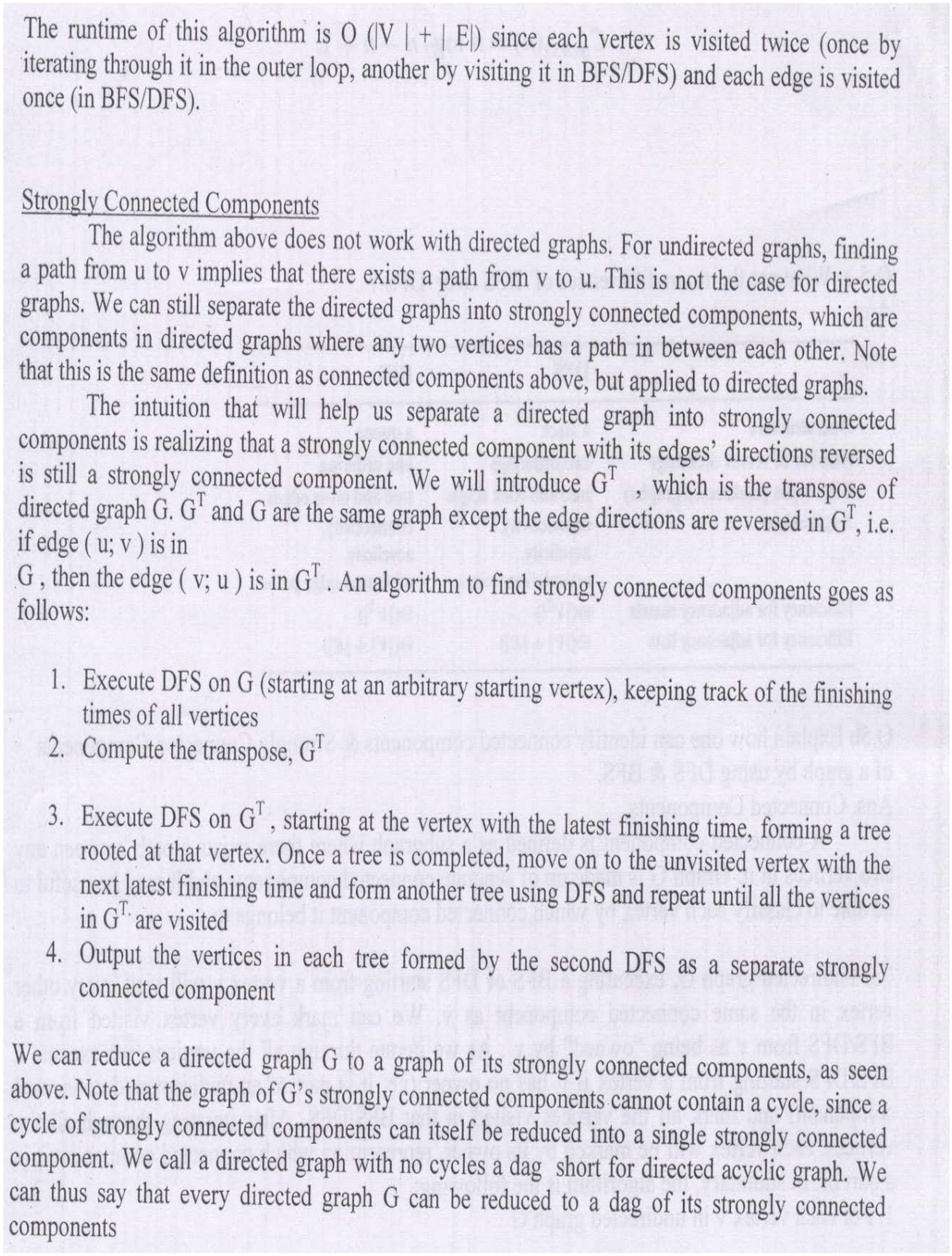
A connected component is defined as a subgraph where there exists a path between any two vertices in it. Graph G is made up of separate connected components and it may be useful to be able to classify each vertex by which connected component it belongs to.

For undirected graph G, executing a BFS or DFS starting from a vertex v will visit every other vertex in the same connected component as v. We can mark every vertex visited from a BFS/DFS from v as being "owned" by v . As we iterate through all the vertices, we execute a BFS/DFS starting from a vertex if it has no owner (i.e. it is part of an undiscovered connected component) and mark all the vertices visited in that BFS/DFS. After iterating through all the vertices, each vertex will be marked by its owner, representing which connected component it is a part of. In summary, the algorithm is the following:

1. For each vertex v in undirected graph G

   (a) If v has no owner, it is part of an undiscovered connected component. Execute BFS or DFS starting from v and mark all the vertices as being owned by v

   (b) Else, if v has an owner, it is part of a connected component we've already discovered. Ignore v and move on to the next vertex.



The runtime of this algorithm is O (|V | + | E|) since each vertex is visited twice (once by iterating through it in the outer loop, another by visiting it in BFS/DFS) and each edge is visited once (in BFS/DFS).

Strongly Connected Components

The algorithm above does not work with directed graphs. For undirected graphs, finding a path from u to v implies that there exists a path from v to u. This is not the case for directed graphs. We can still separate the directed graphs into strongly connected components, which are components in directed graphs where any two vertices has a path in between each other. Note that this is the same definition as connected components above, but applied to directed graphs.

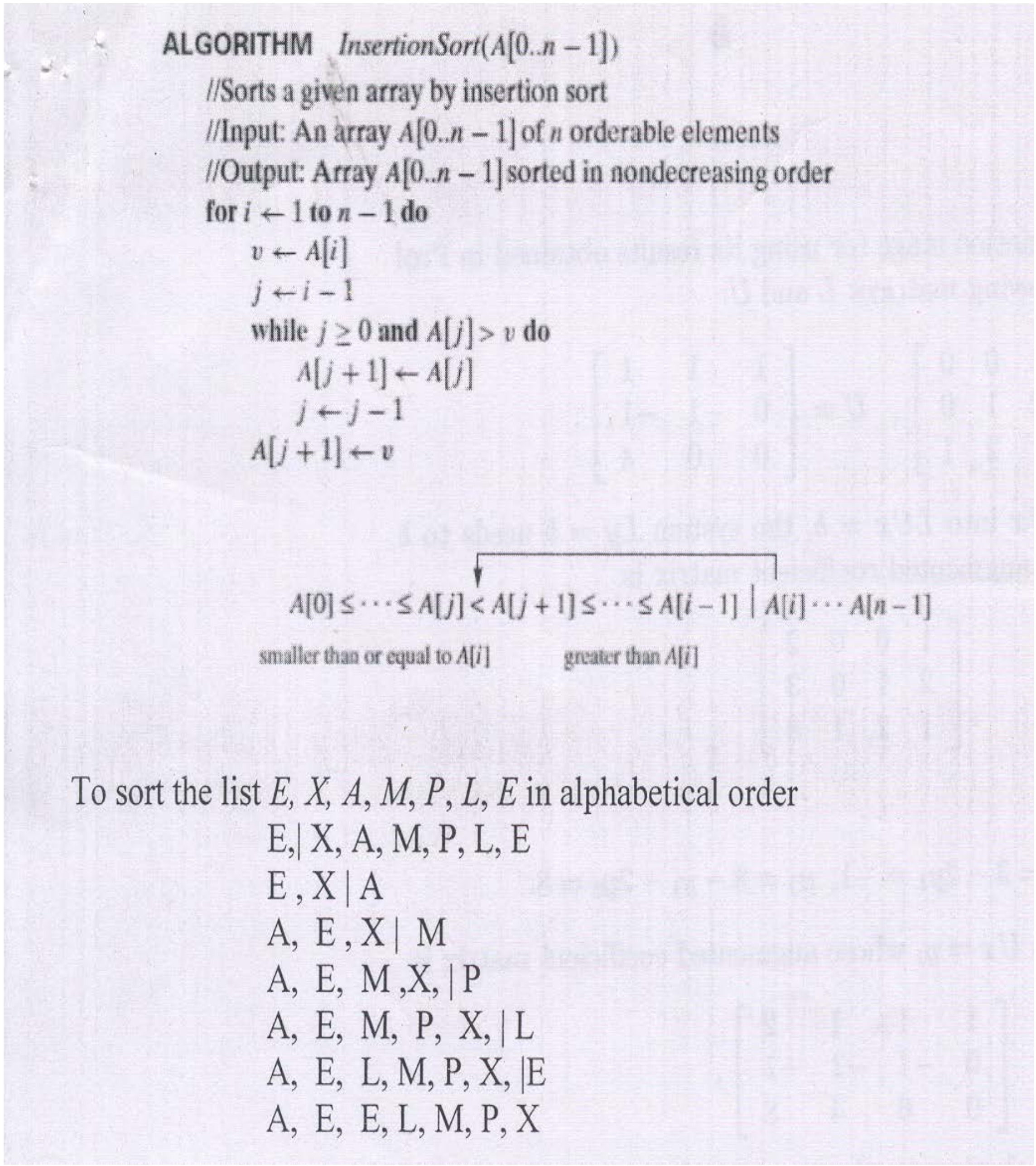
The intuition that will help us separate a directed graph into strongly connected components is realizing that a strongly connected component with its edges' directions reversed is still a strongly connected component. We will introduce G^T , which is the transpose of directed graph G. G^T and G are the same graph except the edge directions are reversed in G^T, i.e. if edge (u; v) is in
G , then the edge (v; u) is in G^T. An algorithm to find strongly connected components goes as follows:

1. Execute DFS on G (starting at an arbitrary starting vertex), keeping track of the finishing times of all vertices
2. Compute the transpose, G^T
3. Execute DFS on G^T , starting at the vertex with the latest finishing time, forming a tree rooted at that vertex. Once a tree is completed, move on to the unvisited vertex with the next latest finishing time and form another tree using DFS and repeat until all the vertices in $G^{T.}$ are visited
4. Output the vertices in each tree formed by the second DFS as a separate strongly connected component

We can reduce a directed graph G to a graph of its strongly connected components, as seen above. Note that the graph of G's strongly connected components cannot contain a cycle, since a cycle of strongly connected components can itself be reduced into a single strongly connected component. We call a directed graph with no cycles a dag short for directed acyclic graph. We can thus say that every directed graph G can be reduced to a dag of its strongly connected components

Q5 (c) Apply insertion sort algorithm to sort the list *E, X, A, M, P, L, E* in alphabetical order.

Answer



```
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 ← 1 to n − 1 do
        v ← A[i]
        j ← i − 1
        while j ≥ 0 and A[j] > v do
            A[j + 1] ← A[j]
            j ← j − 1
        A[j + 1] ← v
```

$$\underbrace{A[0] \le \cdots \le A[j]}_{\text{smaller than or equal to } A[i]} < \underbrace{A[j+1] \le \cdots \le A[i-1]}_{\text{greater than } A[i]} \mid A[i] \cdots A[n-1]$$

To sort the list *E, X, A, M, P, L, E* in alphabetical order

E,| X, A, M, P, L, E

E , X | A

A, E , X | M

A, E, M ,X, | P

A, E, M, P, X, | L

A, E, L, M, P, X, |E

A, E, E, L, M, P, X

Q6 (a) Solve the following system by the *LU* decomposition method and by Gaussian elimination.

$$x1 + x2 + x3 = 2$$

$$2x1 + x2 + x3 = 3$$

$$x1 - x2 + 3x3 = 8$$

Answer

a. Repeating the elimination stage (or using its results obtained in Prol
lem 1), we get the following matrices L and U:

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & 2 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 1 & 1 & 1 \\ 0 & -1 & -1 \\ 0 & 0 & 4 \end{bmatrix}.$$

On substituting $y = Ux$ into $LUx = b$, the system $Ly = b$ needs to b
solved first. Here, the augmented coefficient matrix is:

$$\begin{bmatrix} 1 & 0 & 0 & 2 \\ 2 & 1 & 0 & 3 \\ 1 & 2 & 1 & 8 \end{bmatrix}$$

Its solution is

$$y_1 = 2, \quad y_2 = 3 - 2y_1 = -1, \quad y_3 = 8 - y_1 - 2y_2 = 8.$$

Solving now the system $Ux = y$, whose augmented coefficient matrix is

$$\begin{bmatrix} 1 & 1 & 1 & 2 \\ 0 & -1 & -1 & -1 \\ 0 & 0 & 4 & 8 \end{bmatrix},$$

yields the following solution to the system given:

$$x_3 = 2, \quad x_2 = (-1 + x_3)/(-1) = -1, \quad x_1 = 2 - x_3 - x_2 = 1.$$

by Gaussian elimination

$$\begin{bmatrix} 1 & 1 & 1 & 2 \\ 2 & 1 & 1 & 3 \\ 1 & -1 & 3 & 8 \end{bmatrix} \begin{array}{l} \\ \text{row 2 - } \frac{2}{1}\text{row 1} \\ \text{row 3 - } \frac{1}{1}\text{row 1} \end{array}$$

$$\begin{bmatrix} 1 & 1 & 1 & 2 \\ 0 & -1 & -1 & -1 \\ 0 & -2 & 2 & 6 \end{bmatrix} \begin{array}{l} \\ \\ \text{row 3 - } \frac{-2}{-1}\text{row 2} \end{array}$$

$$\begin{bmatrix} 1 & 1 & 1 & 2 \\ 0 & -1 & -1 & -1 \\ 0 & 0 & 4 & 8 \end{bmatrix}$$

Then, by backward substitutions, we obtain the solution as follows:

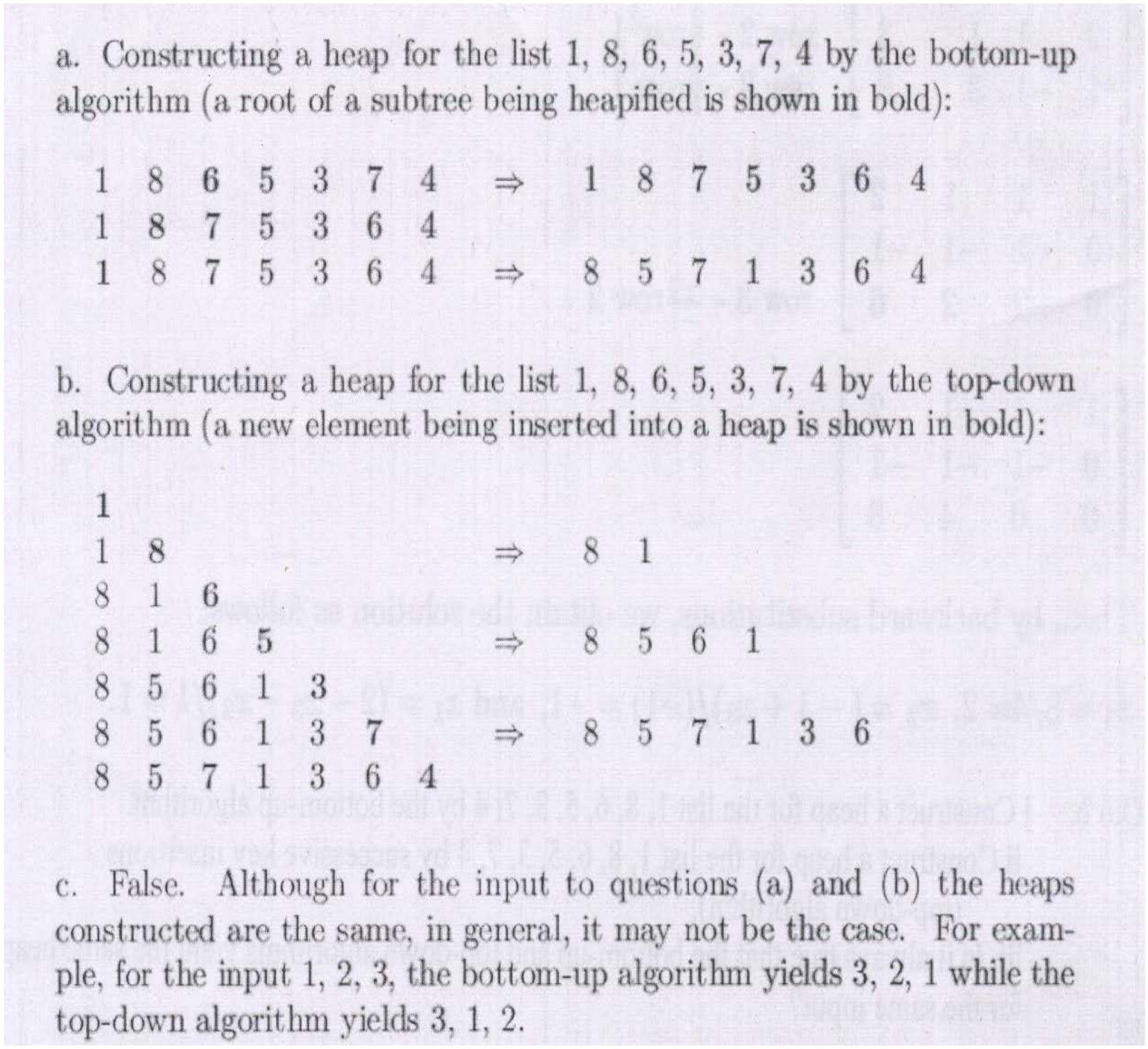
$$x_3 = 8/4 = 2, \; x_2 = (-1 + x_3)/(-1) = -1, \text{ and } x_1 = (2 - x_3 - x_2)/1 = 1.$$

Q6 (b) (i) Construct a heap for the list 1, 8, 6, 5, 3, 7, 4 by the bottom-up algorithm.

(ii) Construct a heap for the list 1, 8, 6, 5, 3, 7, 4 by successive key insertions (top-down algorithm).

(iii) Is it always true that the bottom-up and top-down algorithms yield the Same heap for the same input?

Answer



a. Constructing a heap for the list 1, 8, 6, 5, 3, 7, 4 by the bottom-up algorithm (a root of a subtree being heapified is shown in bold):

```
1  8  6  5  3  7  4   ⇒   1  8  7  5  3  6  4
1  8  7  5  3  6  4
1  8  7  5  3  6  4   ⇒   8  5  7  1  3  6  4
```

b. Constructing a heap for the list 1, 8, 6, 5, 3, 7, 4 by the top-down algorithm (a new element being inserted into a heap is shown in bold):

```
1
1  8                  ⇒   8  1
8  1  6
8  1  6  5            ⇒   8  5  6  1
8  5  6  1  3
8  5  6  1  3  7      ⇒   8  5  7  1  3  6
8  5  7  1  3  6  4
```

c. False. Although for the input to questions (a) and (b) the heaps constructed are the same, in general, it may not be the case. For example, for the input 1, 2, 3, the bottom-up algorithm yields 3, 2, 1 while the top-down algorithm yields 3, 1, 2.

Q7 (a) Define MST with the help of Kruskal's algorithm & explain the algo with suitable example.

Answer

Ans. **DEFINITION** A***spanning tree*** of an undirected connected graph is its connected acyclic subgraph (i.e., a 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

•The algorithm constructs a minimum spanning tree as an expanding sequence of sub graphs that are always acyclic but are not necessarily connected on the intermediate stages of the algorithm.

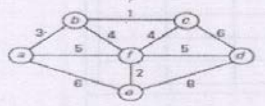
•The algorithm begins by sorting the graph's edges in non decreasing order of their weights. Then, starting with the empty sub graph, it scans this sorted list, adding the next edge on the list to the current sub graph if such an inclusion does not create a cycle and simply skipping the edge otherwise.

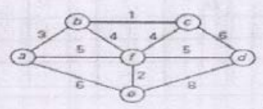
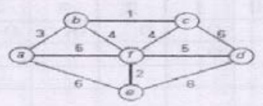
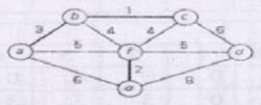
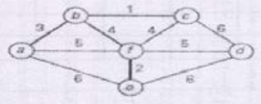
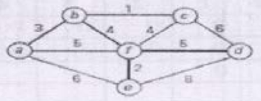
ALGORITHM Kruskal(G)

```
//Kruskal'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 $G$
sort $E$ in nondecreasing order of the edge weights $w(e_{i_1}) \le \cdots \le w(e_{i_{|E|}})$
$E_T \leftarrow \varnothing$;   $ecounter \leftarrow 0$      //initialize the set of tree edges and its size
$k \leftarrow 0$                                  //initialize the number of processed edges
while $ecounter < |V| - 1$ do
    $k \leftarrow k + 1$
    if $E_T \cup \{e_{i_k}\}$ is acyclic
        $E_T \leftarrow E_T \cup \{e_{i_k}\}$;    $ecounter \leftarrow ecounter + 1$
return $E_T$
```

•The time efficiency of Kruskal's algorithm will be inO(| E | log | E|).

Eg.

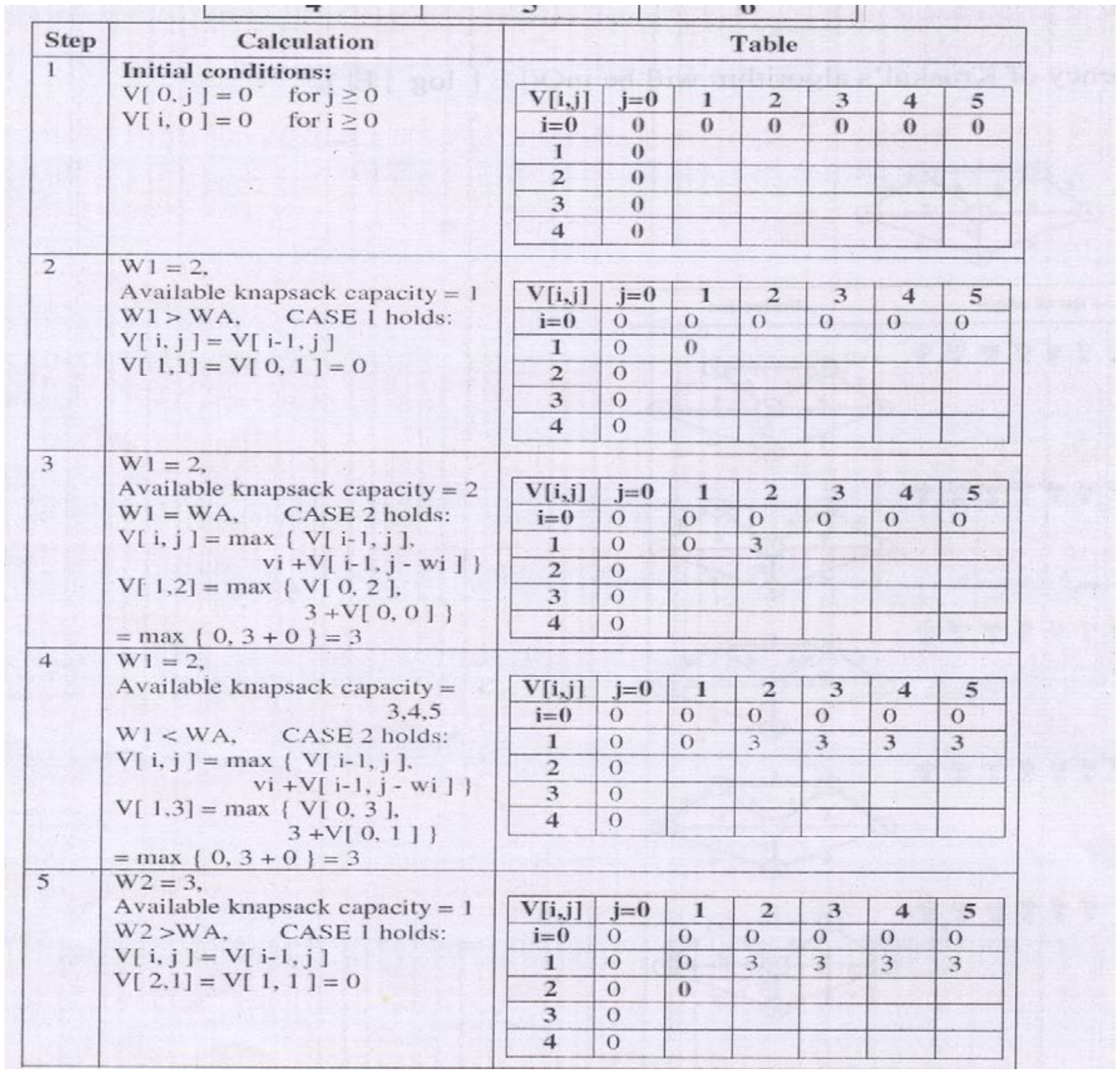


Tree edges	Sorted list of edges	Illustration
	bc 1, ef 2, ab 3, bf 4, cf 4, af 5, df 5, ae 6, cd 6, de 8	
bc 1	bc 1, ef 2, ab 3, bf 4, cf 4, af 5, df 5, ae 6, cd 6, de 8	
ef 2	bc 1, ef 2, ab 3, bf 4, cf 4, af 5, df 5, ae 6, cd 6, de 8	
ab 3	bc 1, ef 2, ab 3, bf 4, cf 4, af 5, df 5, ae 6, cd 6, de 8	
bf 4	bc 1, ef 2, ab 3, bf 4, cf 4, af 5, df 5, ae 6, cd 6, de 8	
df 5		

Q7 (b) Apply bottom-up dynamic programming algorithm to the following instance of the knapsack problem (Capacity W= 5)

Item #	Weight (Kg)	Value (Rs.)
1	2	3
2	3	4
3	4	5
4	5	6

Answer



Step	Calculation
1	**Initial conditions:** V[0, j] = 0 for j ≥ 0 V[i, 0] = 0 for i ≥ 0

V[i,j]	j=0	1	2	3	4	5
i=0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

Step	Calculation
2	W1 = 2, Available knapsack capacity = 1 W1 > WA, CASE 1 holds: V[i, j] = V[i-1, j] V[1,1] = V[0, 1] = 0

V[i,j]	j=0	1	2	3	4	5
i=0	0	0	0	0	0	0
1	0	0				
2	0					
3	0					
4	0					

Step	Calculation

3	W1 = 2,
	Available knapsack capacity = 2
	W1 = WA, CASE 2 holds:
	V[i, j] = max { V[i-1, j],
	vi +V[i-1, j - wi] }
	V[1,2] = max { V[0, 2],
	3 +V[0, 0] }
	= max { 0, 3 + 0 } = 3

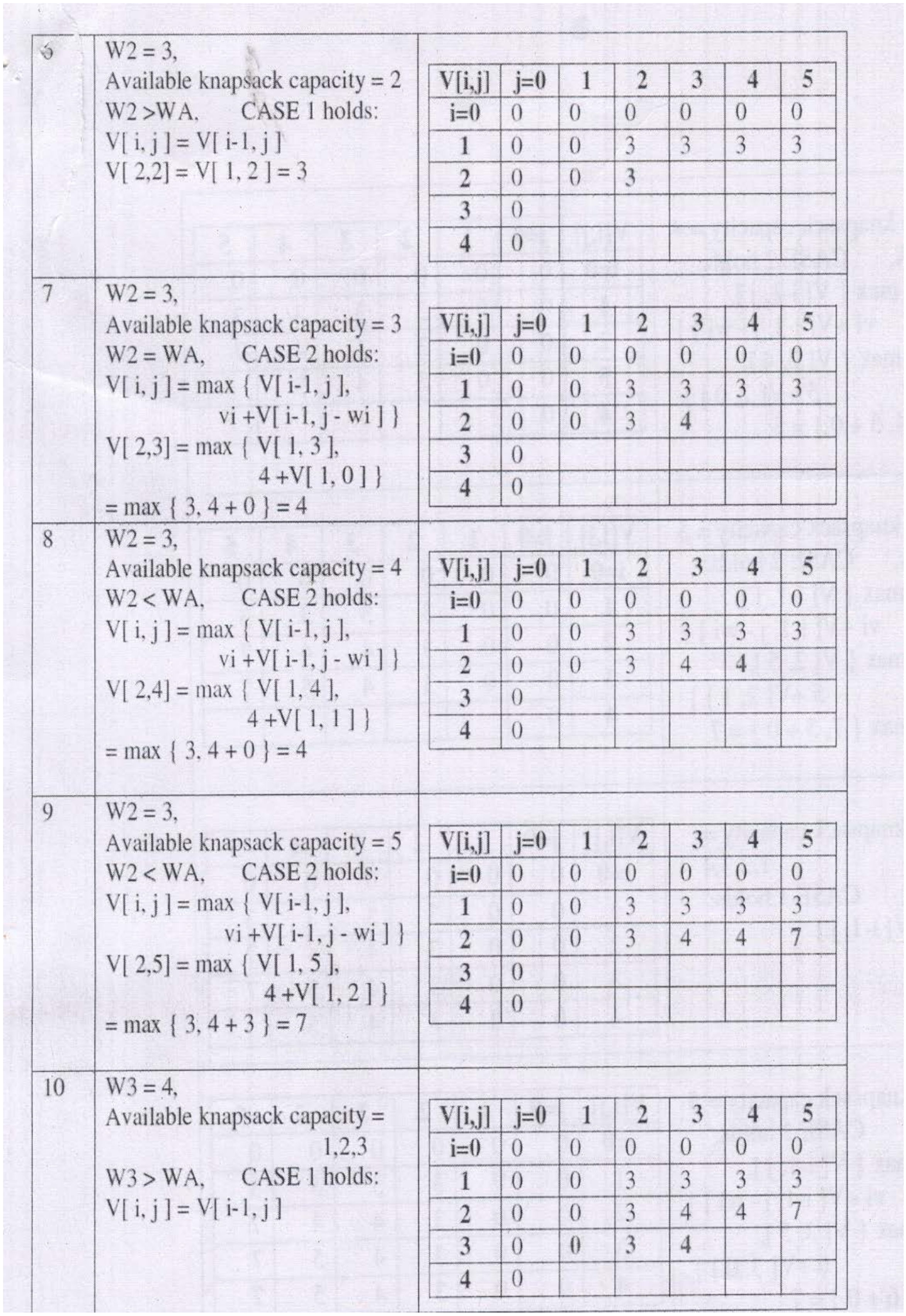
V[i,j]	j=0	1	2	3	4	5
i=0	0	0	0	0	0	0
1	0	0	3			
2	0					
3	0					
4	0					

Step	Calculation
4	W1 = 2, Available knapsack capacity = 3,4,5 W1 < WA, CASE 2 holds: V[i, j] = max { V[i-1, j], vi +V[i-1, j - wi] } V[1,3] = max { V[0, 3], 3 +V[0, 1] } = max { 0, 3 + 0 } = 3

V[i,j]	j=0	1	2	3	4	5
i=0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0					
3	0					
4	0					

Step	Calculation
5	W2 = 3, Available knapsack capacity = 1 W2 >WA, CASE 1 holds: V[i, j] = V[i-1, j] V[2,1] = V[1, 1] = 0

V[i,j]	j=0	1	2	3	4	5
i=0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0				
3	0					
4	0					



6

W2 = 3,
Available knapsack capacity = 2
W2 >WA, CASE 1 holds:
V[i, j] = V[i-1, j]
V[2,2] = V[1, 2] = 3

V[i,j]	j=0	1	2	3	4	5
i=0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3			
3	0					
4	0					

7

W2 = 3,
Available knapsack capacity = 3
W2 = WA, CASE 2 holds:
V[i, j] = max { V[i-1, j],
vi +V[i-1, j - wi] }
V[2,3] = max { V[1, 3],
4 +V[1, 0] }
= max { 3, 4 + 0 } = 4

V[i,j]	j=0	1	2	3	4	5
i=0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4		
3	0					
4	0					

8

W2 = 3,
Available knapsack capacity = 4
W2 < WA, CASE 2 holds:
V[i, j] = max { V[i-1, j],
vi +V[i-1, j - wi] }
V[2,4] = max { V[1, 4],
4 +V[1, 1] }
= max { 3, 4 + 0 } = 4

V[i,j]	j=0	1	2	3	4	5
i=0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	
3	0					
4	0					

9

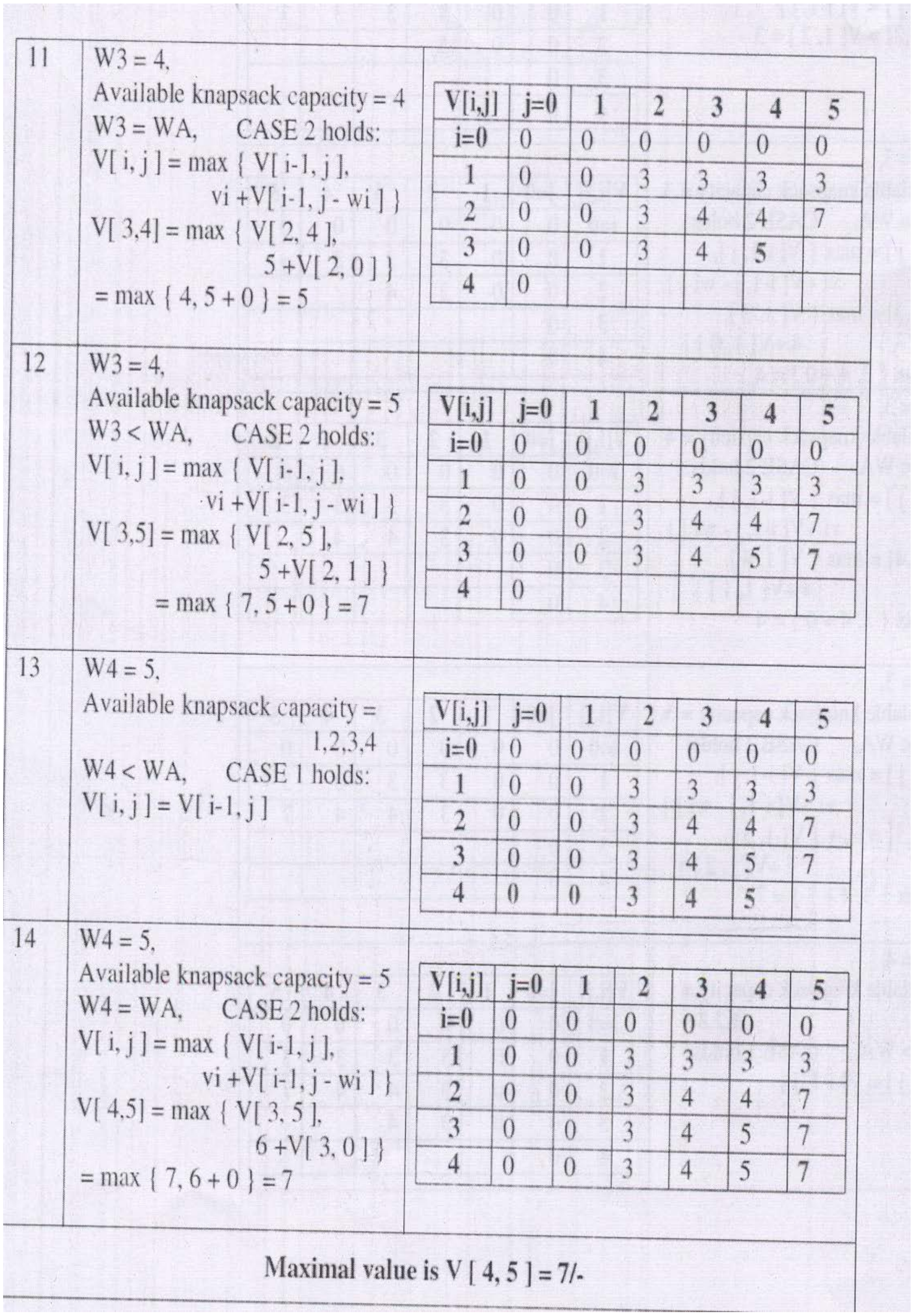
W2 = 3,
Available knapsack capacity = 5
W2 < WA, CASE 2 holds:
V[i, j] = max { V[i-1, j],
vi +V[i-1, j - wi] }
V[2,5] = max { V[1, 5],
4 +V[1, 2] }
= max { 3, 4 + 3 } = 7

V[i,j]	j=0	1	2	3	4	5
i=0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0					
4	0					

10

W3 = 4,
Available knapsack capacity =
1,2,3
W3 > WA, CASE 1 holds:
V[i, j] = V[i-1, j]

V[i,j]	j=0	1	2	3	4	5
i=0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4		
4	0					



11

W3 = 4,
Available knapsack capacity = 4
W3 = WA, CASE 2 holds:
V[i, j] = max { V[i-1, j],
vi +V[i-1, j - wi] }
V[3,4] = max { V[2, 4],
5 +V[2, 0] }
= max { 4, 5 + 0 } = 5

V[i,j]	j=0	1	2	3	4	5
i=0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	
4	0					

12

W3 = 4,
Available knapsack capacity = 5
W3 < WA, CASE 2 holds:
V[i, j] = max { V[i-1, j],
vi +V[i-1, j - wi] }
V[3,5] = max { V[2, 5],
5 +V[2, 1] }
= max { 7, 5 + 0 } = 7

V[i,j]	j=0	1	2	3	4	5
i=0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0					

13

W4 = 5,
Available knapsack capacity =
1,2,3,4
W4 < WA, CASE 1 holds:
V[i, j] = V[i-1, j]

V[i,j]	j=0	1	2	3	4	5
i=0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	

14

W4 = 5,
Available knapsack capacity = 5
W4 = WA, CASE 2 holds:
V[i, j] = max { V[i-1, j],
vi +V[i-1, j - wi] }
V[4,5] = max { V[3, 5],
6 +V[3, 0] }
= max { 7, 6 + 0 } = 7

V[i,j]	j=0	1	2	3	4	5
i=0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

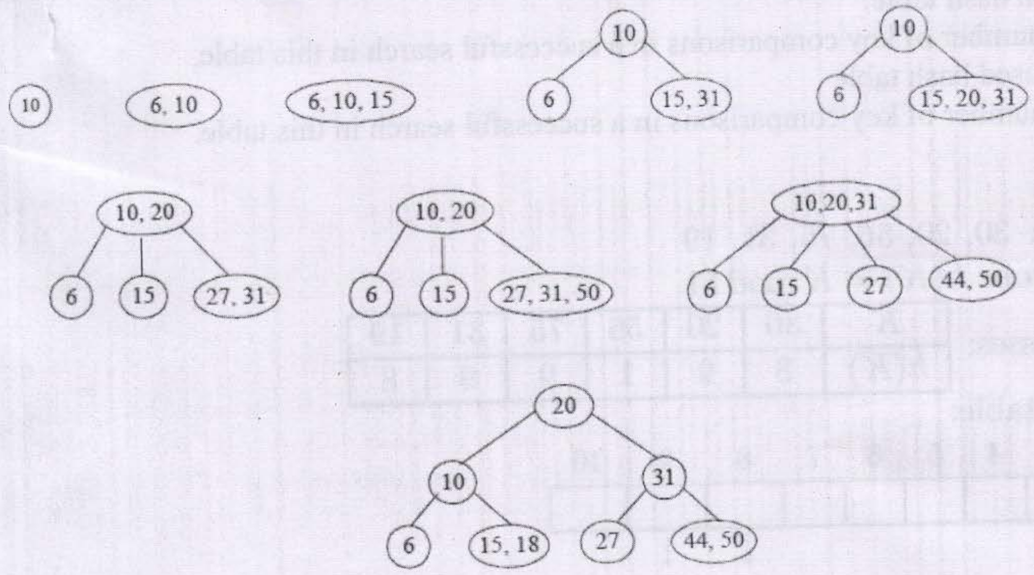
Maximal value is V [4, 5] = 7/-

Q8 (a) Construct a top-down 2-3-4 tree by inserting the following list of keys in the initially empty tree: 10, 6, 15, 31, 20, 27, 50, 44, 18. What is the principal advantage of this insertion procedure & what is its disadvantage?

Answer

a. Constructing a top-down 2-3-4 tree by inserting the following list of keys in the initially empty tree:

10, 6, 15, 31, 20, 27, 50, 44, 18.



b. The principal advantage of splitting full nodes (4-nodes with 3 keys) on a way down during insertion of a new key lies in the fact that if the appropriate leaf turns out to be full, its split will never cause a chain reaction of splits because the leaf's parent will always have a room for an extra key. (If the parent is full before the insertion, it is split before the leaf is reached.) This is not the case for the insertion algorithm employed for 2-3 trees (see Section 6.3).

The disadvantage of splitting full nodes on the way down lies in the fact that it can lead to a taller tree that necessary. For the list of part (a), for example, the tree before the last one had a room for key 18 in the leaf containing key 15 and therefore didn't require a split executed by the top-down insertion.

Q8 (b) For the input 30, 20, 56, 75, 31, 19 and hash function *h(K)* = *K* mod 11

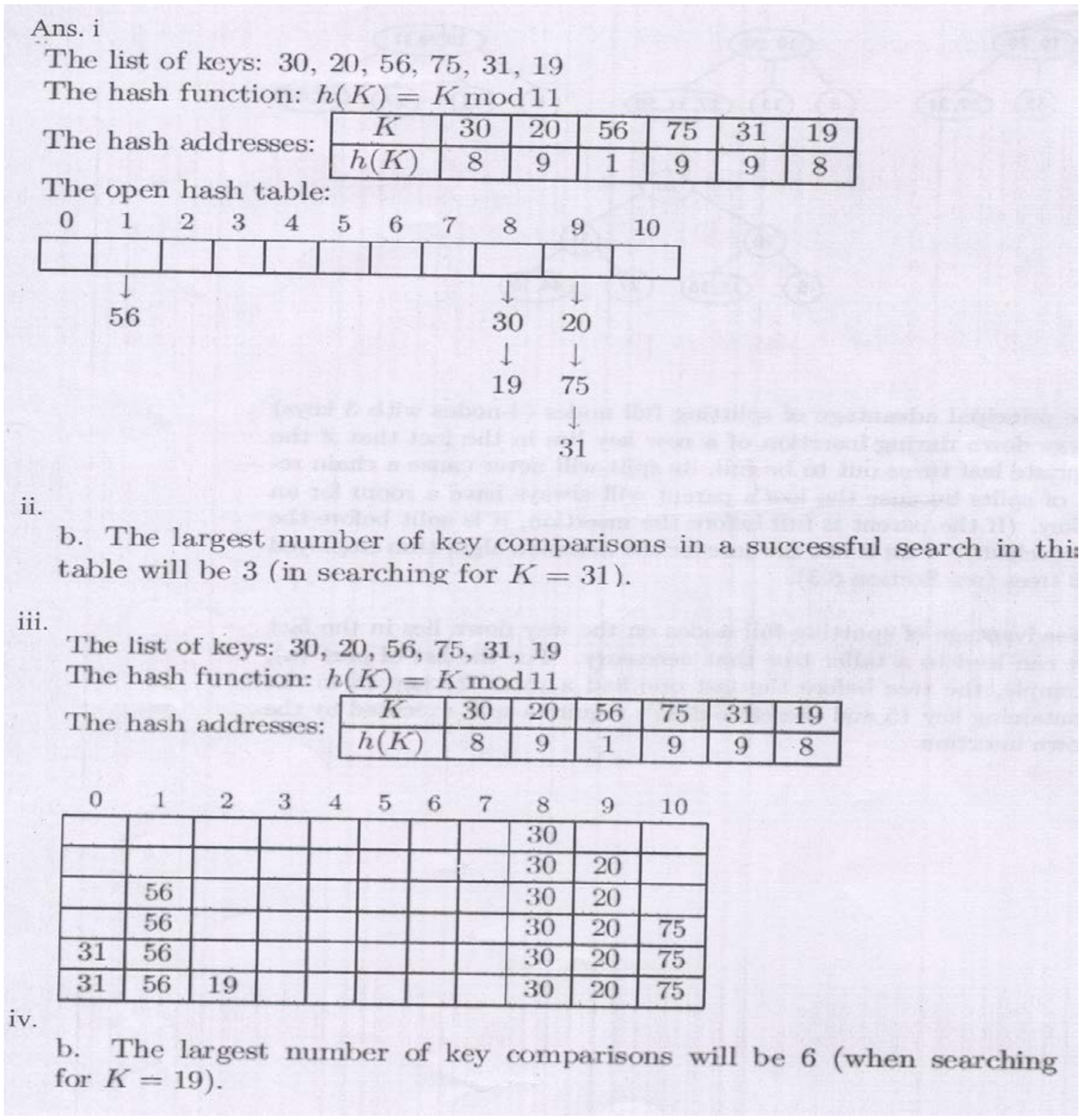
(i) Construct the open hash table.

(ii) Find the largest number of key comparisons in a successful search in this table.

(iii) Construct the closed hash table.

(iv) Find the largest number of key comparisons in a successful search in this table.

Answer



Ans. i

The list of keys: 30, 20, 56, 75, 31, 19

The hash function: $h(K) = K \bmod 11$

The hash addresses:

K	30	20	56	75	31	19
$h(K)$	8	9	1	9	9	8

The open hash table:

0	1	2	3	4	5	6	7	8	9	10
	↓ 56							↓ 30 ↓ 19	↓ 20 ↓ 75 ↓ 31	

ii.

b. The largest number of key comparisons in a successful search in thi
table will be 3 (in searching for $K = 31$).

iii.

The list of keys: 30, 20, 56, 75, 31, 19

The hash function: $h(K) = K \bmod 11$

The hash addresses:

K	30	20	56	75	31	19
$h(K)$	8	9	1	9	9	8

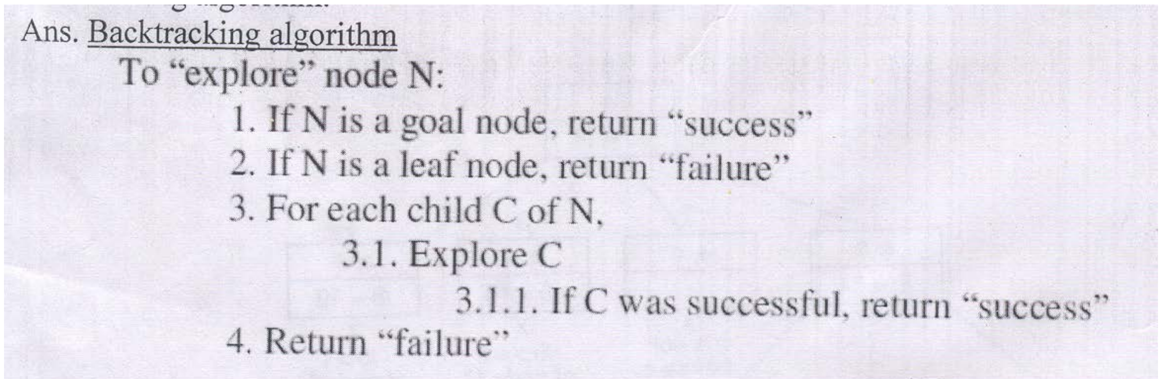
0	1	2	3	4	5	6	7	8	9	10
								30		
								30	20	
	56							30	20	
	56							30	20	75
31	56							30	20	75
31	56	19						30	20	75

iv.

b. The largest number of key comparisons will be 6 (when searching for $K = 19$).

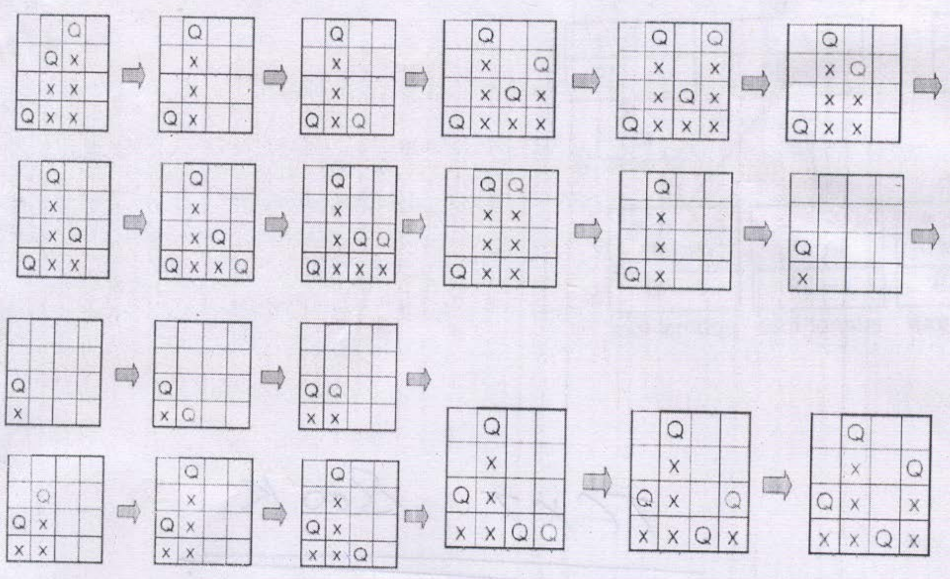
Q9 (a) Which is the *last* solution to the five-queens problem found by the backtracking algorithm? Use the board's symmetry to find at least four other solutions to the problem. Explore backtracking algorithm.

Answer

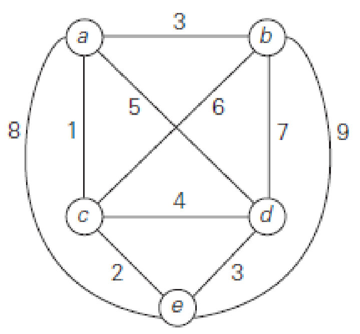


Ans. Backtracking algorithm

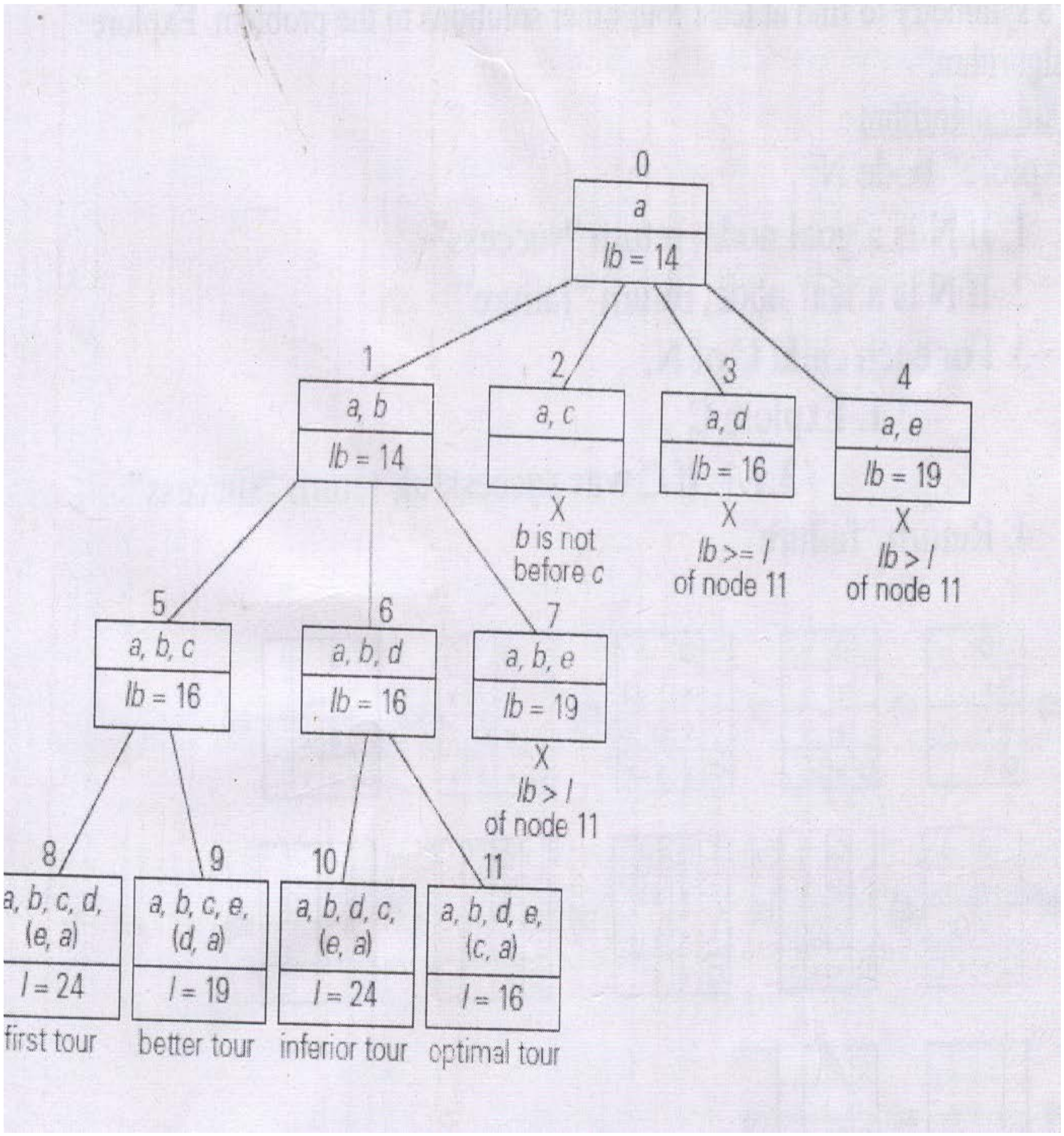
```
To "explore" node N:
    1. If N is a goal node, return "success"
    2. If N is a leaf node, return "failure"
    3. For each child C of N,
        3.1. Explore C
            3.1.1. If C was successful, return "success"
    4. Return "failure"
```



Q9 (b) Find the shortest Hamiltonian circuit for the given graph using branch & bound algorithm



Answer



<u>Text Book</u>

Introduction to the Design & Analysis of algorithms, Anany Levitin, 2nd Edition Pearson Education, 2007.