DHANALAKSHMI COLLEGE OF ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CS6402 – DESIGN AND ANALYSIS OF ALGORITHMS

ACADEMIC YEAR 2016-2017 [EVEN SEMESTER]

QUESTION BANK – 2 &16 MARK QUESTIONS WITH ANSWERS

Year: II Year IV Semester **Batch: 2015-2019**

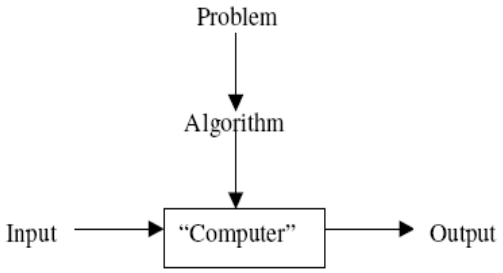
UNIT I INTRODUCTION

2 marks

1. What is an algorithm? [M – 13]

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

An algorithm is step by step procedure to solve a problem.



2. What are the types of algorithm efficiencies?

The two types of algorithm efficiencies are

- *Time efficiency:* indicates how fast the algorithm runs
- *Space efficiency:* indicates how much extra memory the algorithm needs

3. Mention some of the important problem types?

Some of the important problem types are as follows

- Sorting
- Searching
- String processing
- Graph problems
- Combinatorial problems
- Geometric problems
- Numerical problems

4. What is worst-case efficiency?

The worst-case efficiency of an algorithm is its efficiency for the worst-case input of size n, which is an input or inputs of size n for which the algorithm runs the longest among all possible inputs of that size.

5. What is best-case efficiency?

The best-case efficiency of an algorithm is its efficiency for the best-case input of size n, which is an

input or inputs for which the algorithm runs the fastest among all possible inputs of that size.

6. What is average case efficiency? [M-14]

The average case efficiency of an algorithm is its efficiency for an average case input of size n. It provides information about an algorithm behavior on a -typical‖ or -random‖ input.

7. Define O-notation? [M -13][M-15]

A function t[n] is said to be in O[g[n]], denoted by t[n] ε O[g[n]], if t[n] is bounded above by some constant multiple of g[n] for all large n, i.e., if there exists some positive constant c and some non-negative integer n_0 such that

$$T[n] <= cg[n] \text{ for all } n >= n_0$$

8. Define Ω-notation? [M – 14]

A function t[n] is said to be in Ω [g[n]], denoted by t[n] ε Ω [g[n]], if t[n] is bounded below by some constant multiple of g[n] for all large n, i.e., if there exists some positive constant c and some non-negative integer n_0 such that

$$T[n] >= cg[n] \text{ for all } n >= n_0$$

9. Define θ-notation? [N-14]

A function t[n] is said to be in θ [g[n]], denoted by t[n] ε θ [g[n]], if t[n] is bounded both above & below by some constant multiple of g[n] for all large n, i.e., if there exists some positive constants c1 & c2 and some nonnegative integer n0 such that $c_2 g[n] <= t[n] <= c_1 g[n]$ for all n >= n0

10. Give the Euclid's algorithm for computing gcd[m, n] [M – 16]

ALGORITHM Euclid_gcd[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
```

Example: gcd[60, 24] = gcd[24, 12] = gcd[12, 0] = 12.

11. Define the terms: pseudo code, flow chart

- A pseudocode is a mixture of a natural language and programming language like constructs. A pseudocode is usually more precise than natural language.
- A flowchart is a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps.

12. Compare the order of growth n[n-1]/2 and n^2. [M – 16]

n	n[n-1]/2	n^2
Polynomial	**Quadratic**	**Quadratic**
1	0	1
2	1	4
4	6	16
8	28	64
10	45	10^2
10^2	4950	10^4
Complexity	Low	High
Growth	Low	high

n[n-1]/2 is lesser than the half of n^2

13. State how binomial coefficient is computed through algorithm? Or Write an algorithm to find the number of binary digits in the binary representation of a positive decimal integer. [M-15]

ALGORITHM Binary[n]

//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
count ←1
while n >1do
count ←count +1
n← n/2
return count

14. What is the use of asymptotic notation? [M-15]

The notations describe different rate-of-growth relations between the defining function and the defined set of function. They are used to compare two function sizes. Ex: linear, logarithmic ,cubic, quadratic, exponential functions are used.

15. What is the substitution method? [M-14][N-14]

One of the methods for solving any such recurrence relation is called the substitution method.

16 marks

1. Explain the various Asymptotic Notations used in algorithm design? Or Discuss the properties of asymptotic notations. Or Explain the basic efficiency classes with notations.[N-14][M-15][M-16]

Asymptotic notation is a notation, which is used to take meaningful statement about the efficiency of

a program. 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, they are:

O - Big oh notation, Ω - Big omega notation, Θ - Big theta notation

Let t[n] and g[n] can be any nonnegative functions defined on the set of natural numbers. The algorithm's running time t[n] usually indicated by its basic operation count C[n], and g[n], some simple function to compare with the count.

There are 5 basic asymptotic notations used in the algorithm design.

- Big Oh: A function t[n] is said to be in O[g[n]], denoted by t[n] ε O[g[n]], if t[n] is bounded above by some constant multiple of g[n] for all large n, i.e., if there exists some positive constant c and some non-negative integer n0 such that T [n] <=cg [n] for all n >= n0
- Big Omega: A function t[n] is said to be in Ω [g[n]], denoted by t[n] ε Ω [g[n]], if t[n] is bounded below by some constant multiple of g[n] for all large n, i.e., if there exists some positive constant c and some non-negative integer n0 such that T [n] >=cg [n] for all n >=n0
- Big Theta: A function t[n] is said to be in θ [g[n]], denoted by t[n] ε θ [g[n]], if t[n] is bounded both above & below by some constant multiple of g[n] for all large n, i.e., if there exists some positive constants c1 & c2 and some nonnegative integer n0 such that c_2g [n] <= t [n] <= c_1g [n] for all n >= n0
- Little oh: The function f[n] = 0[g[n]] iff Lim f[n] = 0 n - ∝ g[n]
- Little Omega. :The function f[n] = ω [g[n]] iff Lim f[n] = 0 n - ∝ g[n]

t[n] ∈ O[g[n]] iff t[n] <=cg[n] for n > n_0

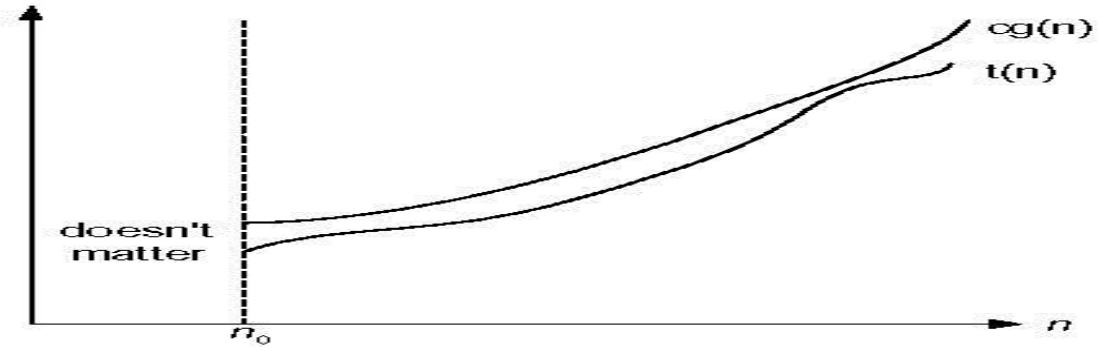


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

t[n] ∈ Ω[g[n]] iff t[n] >=cg[n] for n > n_0

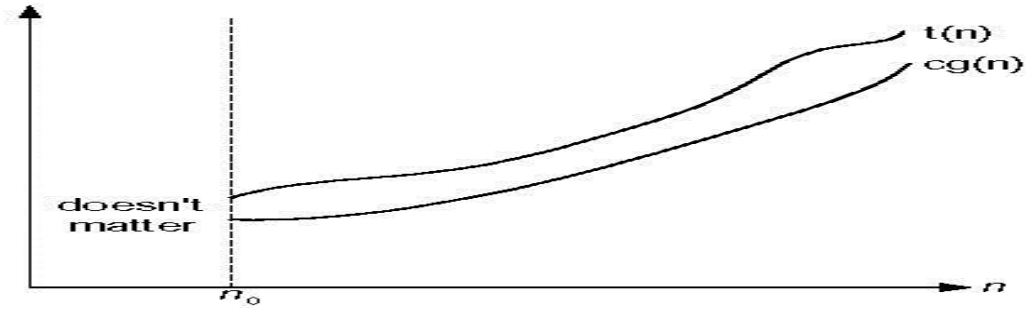


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

t[n]∈ Θ[g[n]] iff t[n]∈O[g[n]] and ∈Ω[g[n]]

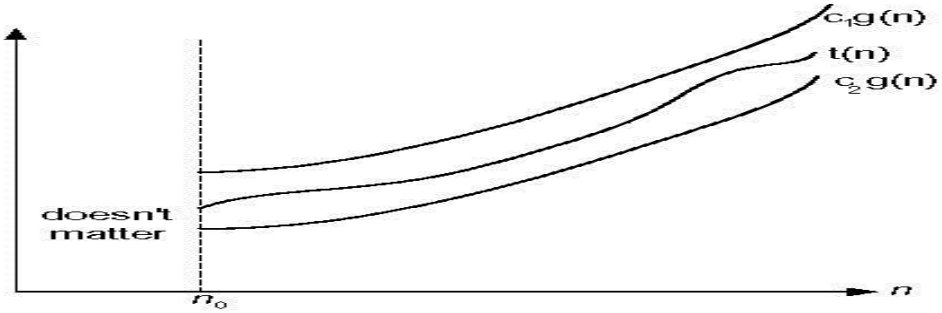


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

Informal Definitions: Big O, Ω, Θ

Some properties of asymptotic order of growth

- $f[n] \in O[f[n]]$
- $f[n] \in O[g[n]]$ iff $g[n] \in \Omega[f[n]]$
- If $f[n] \in O[g[n]]$ and $g[n] \in O[h[n]]$, then $f[n] \in O[h[n]]$ Note similarity with $a \leq b$
- If $f_1[n] \in O[g_1[n]]$ and $f_2[n] \in O[g_2[n]]$, then $f_1[n] + f_2[n] \in O[\max\{g_1[n], g_2[n]\}]$

Basic Efficiency classes:

1	constant	Best case
log *n*	logarithmic	Divide ignore part
n	linear	Examine each
n log *n*	*n*-log-*n* or linear logarithmic	Divide use all parts
n^2	quadratic	Nested loops
n^3	cubic	Nested loops
2^n	exponential	All subsets
n!	factorial	All permutations

2. Explain recursive and non-recursive algorithms with example. Or
With an example, explain how recurrence equations are solved. [N-14][M-14]

Mathematical Analysis of Recursive Algorithms

General Plan for Analysis

- Decide on a parameter indicating an input's size.
- Identify the algorithm's basic operation.
- Check whether the number of times the basic op. is executed may vary on different inputs of the same size. [If it may, the worst, average, and best cases must be investigated separately.]
- Set up a recurrence relation with an appropriate initial condition expressing the number of times the basic op. is executed.
- Solve the recurrence by backward substitutions or another method.

EXAMPLE Compute the factorial function $F[n] = n!$ for an arbitrary nonnegative integer n

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*

EXAMPLE 2: consider educational workhorse of recursive algorithms: the Tower of Hanoi puzzle. We have n disks of different sizes that can slide onto any of three pegs. Consider A (source), B (auxiliary), and C (Destination). Initially, all the disks are on the first peg in order of size, the largest on the bottom and the smallest on top. The goal is to move all the disks to the third peg, using the second one as an auxiliary.

ALGORITHM TOH(n, A, C, B)

```
//Move disks from source to destination recursively
//Input: n disks and 3 pegs A, B, and C
//Output: Disks moved to destination as in the source order.
if n=1
Move disk from A to C
else
Move top n-1 disks from A to B using C
TOH(n - 1, A, B, C)
Move top n-1 disks from B to C using A
TOH(n - 1, B, C, A)
```

Mathematical Analysis of Non-Recursive Algorithms

General Plan for Analysis

Decide on parameter *n* indicating *input size*

- Identify algorithm's *basic operation*
- Determine *worst*, *average*, and *best* cases for input of size *n*
- Set up a sum for the number of times the basic operation is executed

- Simplify the sum using standard formulas and rules [see Appendix A]

EXAMPLE Consider the problem of finding the value of the largest element in a list of n numbers

ALGORITHM MaxElement[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]
fori ←1 to n−1do
if A[i]> maxval
maxval←A[i]
return maxval
```

EXAMPLE 2 Given two n×n matrices A and B, find the time efficiency of the definition-based algorithm for computing their produc tC =AB. By definition,C is an n×n matrix whose elements are computed as the scalar [dot] products of the rows of matrix A and the columns of matrixB: whereC[i, j]=A[i, 0]B[0, j]+...+A[i, k]B[k, j]+...+A[i, n−1]B[n−1, j] for every pair of indices 0 ≤i, j ≤n−1.

ALGORITHM MatrixMultiplication[A[0..n−1, 0..n−1], B[0..n−1, 0..n−1]]

```
//Multiplies two square matrices of order n by the definition-based algorithm
//Input: Two n×n matrices A andB
//Output: MatrixC =AB
fori ←0 to n−1do
for j ←0 to n−1do
C[i, j]←0.0
for k←0 to n−1do
C[i, j]←C[i, j]+A[i, k]∗B[k, j]
return C
```

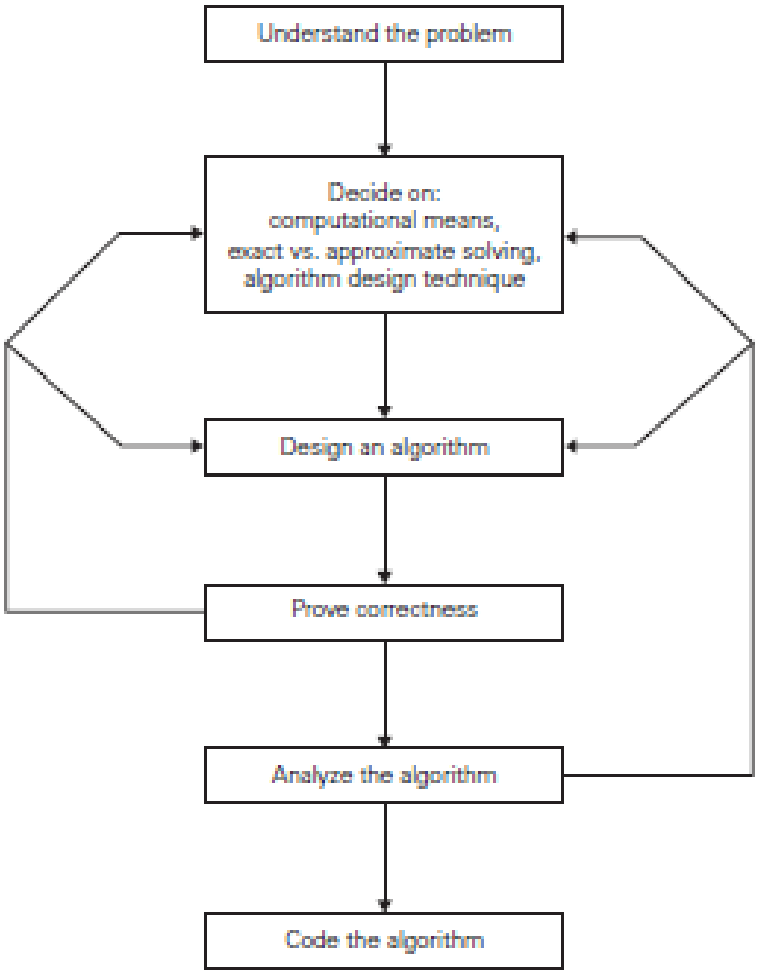
To measure an input's size by matrix order n. There are two arithmetical operations in the innermost loop here—multiplication and addition—that, in principle, can compete for designation as the algorithm's basic operation.

2. What are the fundamental steps to solve an algorithm? Explain. Or Describe in detail about the steps in analyzing and coding an algorithm. [M-15][M-14]

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

   - Understand the problem
   - Decision making
   - Design an algorithm
   - Proving correctness of an algorithm
   - Analyze the algorithm
   - Coding and implementation of an algorithm

Figure : Algorithm design and analysis process



3. **Understand the problem**
    a. Read the description carefully to understand the problem completely
    b. Identfiy the problem types and use existing algorithm to find solution
    c. Input (instance) to the problem and range of the input gets fixed.
4. **Decision making**
    a. Ascertaining the capabilities of computational device
        i. In Ram instructions are executed one after another, accordingly algorithms designed to be executed on such machines are executed sequential algorithms
        ii. In some computers operations are executed concurrently in parallel.
        iii. Choice of computational devices like processor and memory is mainly based on space and time efficiency.
    b. Choosing between exact versus approximate problem solving
        i. An algorithm used to solve the problem exactly and produce correct result is called exact algorithm
        ii. If the problem is to so complex and not able to get exact solution then it is called approximation algorithm.

c. Algorithm design strategies

i. Algorithms + data structures = programs, though algorithms and data structures are independent then they combined to produce programs.
ii. Implementation of an algorithm is possible with the help of algorithms and data structures.
iii. Algorithm design strategy techniques are brute force, dynamic programming, greedy technique, divide and conquer and so on.

iv. Methods for specifying an algorithm

1. **Natural language**: It is very simple and easy to specify an algorithm using natural language.
   Example // addition of 2 nos
   Read a
   Read b
   Add c=a+b
   Store and display the result in c
2. **Flow chart** – Flowchart is a diagrammatic and graphical representation of an algorithm. It is a method of expressing an algorithm by a collection of connected graphical shapes containing description of the algorithm's steps.
3. **Pseudo code** - It is a mixture of natural language and programming language constructs. It is usually more precise than natural language.
   Example // sum of 2 nos
   // input a and b
   // output c
   c← a+ b

d. Proving an algorithm's correctness

i. Once an algorithm has been specified then its correctness must be proved.
ii. An algorithm must yields a required result for every legitimate input in a finite amount of time.
iii. For example, the correctness of Euclid's algorithm for computing the greatest common divisor stems from the correctness of the equality gcd(m, n) = gcd(n, m mod n).
iv. A common technique for proving correctness is to use mathematical induction because an algorithm's iterations provide a natural sequence of steps needed for such proofs.
v. The notion of correctness for approximation algorithms is less straightforward than it is for exact algorithms. The error produced by the algorithm should not exceed a predefined limit.

e. Analyzing an algorithm

For an algorithm the most important is algorithm efficiency .There are two types of algorithm efficiencies are

- *Time efficiency:* indicates how fast the algorithm runs
- *Space efficiency:* indicates how much extra memory the algorithm needs

So the efficiency of an algorithm through analysis is based on both time and space efficiency. There are some factors to analyze an algorithm are:

- Simplicity of an algorithm
- Generality of an algorithm
- Time efficiency of an algorithm
- Space efficiency of an algorithm

f. Coding an algorithm

i. The coding / implementation of an algorithm is done by a suitable programming language like C, C++, JAVA.

ii. The transition from an algorithm to a program can be done either incorrectly or very inefficiently. Implementing an algorithm correctly is necessary. The Algorithm power should not reduce by inefficient implementation.

iii. Standard tricks like computing a loop's invariant outside the loop, collecting common sub expressions, replacing expensive operations by cheap ones, selection of programming language and so on should be known to the programmer.

5. What are the fundamental steps to solve an algorithm? Or
What are the steps for solving an efficient algorithm? [M-15]

Analysis of algorithm is the process of investigation of an algorithm's efficiency with respect to two resources: running time and memory space.

The reasons for selecting these two criteria are:

- The simplicity and generality measures of an algorithm estimate the efficiency
- The speed and memory are the efficiency considerations of modern computers.

That there are two kinds of efficiency: time efficiency and space efficiency.

- Time efficiency, also called time complexity, indicates how fast an algorithm in question runs.
- Space efficiency, also called space complexity, refers to the amount of memory units required by the algorithm in addition to the space needed for its input and output.

The steps for an efficient algorithm

1. Measuring an input's size
    a. The efficiency measure of an algorithm is directly proportional to the input size or range.
    b. The input given may be a square or a non-square matrix.
    c. Some algorithms require more than one parameter to indicate the size of their inputs.
2. Units for measuring time
    a. We can simply use some standard unit of time measurement-a second, a millisecond, and so on-to measure the running time of a program implementing the algorithm.
    b. There are obvious drawbacks to such an approach. They are
        - Dependence on the speed of a particular computer
        - Dependence on the quality of a program implementing the algorithm
        - The compiler used in generating the machine code
        - The difficulty of clocking the actual running time of the program.
    c. Since we are in need to measure algorithm efficiency, we should have a metric that does not depend on these extraneous factors.
    d. One possible approach is to count the number of times each of the algorithm's operations is executed. This approach is both difficult and unnecessary.
    e. The main objective is to identify the most important operation of the algorithm, called the basic operation, the operation contributing the most to the total running time, and compute the number of times the basic operation is executed.
3. Efficiency classes
    a. It is reasonable to measure an algorithm's efficiency as a function of a parameter

indicating the size of the algorithm's input.

b. But there are many algorithms for which running time depends not only on an input size but also on the specifics of a particular input.

4. **Example, sequential search**. This is a straightforward algorithm that searches for a given item (some search key K) in a list of n elements by checking successive elements of the list until either a match with the search key is found or the list is exhausted.

ALGORITHM Sequential Search(A[0..n -1], K)

//Searches for a given value in a given array by sequential search

//Input: An array A[0..n -1] and a search key K

//Output: Returns the index of the first element of A that matches K

// or -1 ifthere are no matching elements

i←0

while i < n and A[i] ≠ K do

i←i+1

if i < n return i

else return -1

Worst case efficiency

- The worst-case efficiency of an algorithm is its efficiency for the worst-case input of size n, which is an input (or inputs) of size n for which the algorithm runs the longest among all possible inputs of that size.
- In the worst case, when there are no matching elements or the first matching element happens to be the last one on the list, the algorithm makes the largest number of key comparisons among all possible inputs of size n: **Cworst (n) = n.**

Best case Efficiency

- The best-case efficiency of an algorithm is its efficiency for the best-case input of size n, which is an input (or inputs) of size n for which the algorithm runs the fastest among all possible inputs of that size.
- First, determine the kind of inputs for which the count C (n) will be the smallest among all possible inputs of size n. (Note that the best case does not mean the smallest input; it means the input of size n for which the algorithm runs the fastest.)
- Then ascertain the value of C (n) on these most convenient inputs. Example- for sequential search, best-case inputs will be lists of size n with their first elements equal to a search key; accordingly, **Cbest(n) = 1.**

Average case efficiency

- The average number of key comparisions Cavg(n) can be computed as follows,
  - let us consider again sequential search. The standard assumptions are,

In the case of a successful search, the probability of the first match occurring in the ith position of the list is pin for every i, and the number of comparisons made by the algorithm in such a situation is obviously i. **Cavg(n) =(n+ 1)/2**

5. Orders of growth

Big oh , Big omega and Big Theta notations described above 2 Question.

UNIT II DIVIDE AND CONQUER METHOD AND GREEDY METHOD

2 marks

1. What is brute force algorithm?

A straightforward approach, usually based directly on the problem‘s statement and definitions of the concepts involved.

2. What is exhaustive search?

A brute force solution to a problem involving search for an element with a special property, usually among combinatorial objects such as permutations, combinations, or subsets of a set.

Examples or Techniques used:

- Traveling Salesman Problem (TSP)
- Knapsack Problem (KP)
- Assignment Problem (AP)

3. Give the general plan for divide-and-conquer algorithms. [N – 16]

The general plan is as follows

- A problems instance is divided into several smaller instances of the same problem, ideally about the same size
- The smaller instances are solved, typically recursively
- If necessary the solutions obtained are combined to get the solution of the original problem Given a function to compute on _n‘ inputs the divide-and-conquer strategy suggests splitting the inputs in to‘k‘ distinct sub sets, 1<k <n, yielding _k‘ sub problems. The sub problems must be solved, and then a method must be found to combine sub solutions into a solution of the whole. If the sub problems are still relatively large, then the divide-and conquer strategy can possibly be reapplied.

4. Define – Feasibility [M -14]

A feasible set [of candidates] is promising if it can be extended to produce not merely a solution, but an optimal solution to the problem.

5. Define - Hamiltonian circuit [N-14][M -13]

A Hamiltonian circuit is defined as a cycle that passes through all the vertices of the graph exactly once.

6. Define- Merge sort and list the steps also

Merge sort 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 then merging the two smaller sorted arrays into a single sorted one.

Steps

1. Divide Step: If given array A has zero or one element, return S; it is already sorted. Otherwise, divide A into two arrays, A1 and A2, each containing about half of the elements of A.

2. Recursion Step: Recursively sort array A1 and A2.

3. Conquer Step: Combine the elements back in A by merging the sorted arrays A1 and A2 into a sorted sequence

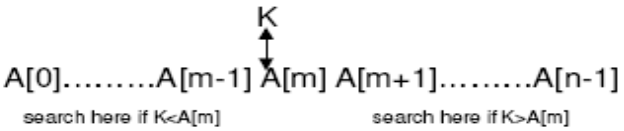
7. Define -Quick Sort

Quick sort is an algorithm of choice in many situations because it is not difficult to implement, it is a good\"general purpose\" sort and it consumes relatively fewer resources during execution.

Analysis: O[n log n] in average and best cases and $O[n^2]$ in worst case

8. What is binary search? [M -15]

Binary search is a remarkably efficient algorithm for searching in a sorted array. It works by comparing a search key K with the arrays middle element A[m]. If they match the algorithm stops; otherwise the same operation is repeated recursively for the first half of the array if K < A[m] and the second half if K > A[m]. Binary Search Time Complexity: O[n log n]



9. Define - Dijkstra's Algorithm

Dijkstra's algorithm solves the single source shortest path problem of finding shortest paths from a given vertex[the source], to all the other vertices of a weighted graph or digraph.
Dijkstra's algorithm provides a correct solution for a graph with non-negative weights.

10. Define - Huffman trees

A Huffman tree is binary tree that minimizes the weighted path length from the root to the leaves containing a set of predefined weights. The most important application of Huffman trees are Huffman code.

11. What do you mean by optimal solution? [N-14][M -14]

Given a problem with n inputs, we obtain a subset that satisfies come constraints. Any subset that satisfies these constraints is called a feasible solution. A feasible solution, which maximizes or minimizes a given objective function, is called optimal solution.

12. What is Closest-Pair Problem? [M – 16]

The closest-pair problem finds the two closest points in a set of n points. It is the simplest of a variety of problems in computational geometry that deals with proximity of points in the plane or higher-dimensional spaces.

13. Distinguish between BFS and DFS. [M-13]

DFS follows the procedure:

a. Select an unvisited node x, visit it, and treat as the current node
b. Find an unvisited neighbor of the current node, visit it, and make it the new current node;

c. If the current node has no unvisited neighbors, backtrack to the its parent, and make that parent the new current node;
d. Repeat steps 3 and 4 until no more nodes can be visited.
e. If there are still unvisited nodes, repeat from step 1.

BFS follows the procedure:

f. Select an unvisited node x, visit it, have it be the root in a BFS tree being formed. Its level is called the current level.
g. From each node z in the current level, in the order in which the level nodes were visited, visit all the unvisited neighbors of z. The newly visited nodes from this level form a new level that becomes the next current level.
h. Repeat step 2 until no more nodes can be visited.
i. If there are still unvisited nodes, repeat from Step 1

14. What are applications or examples of Brute force techniques?

- Exhaustive searching techniques [TSP,KP,AP]
- Finding closest pair and convex hull problems
- Sorting: selection and bubble sort
- Searching: Brute force string match and sequential search.
- Computing n!

15. What are the applications of divide and conquer techniques?

- Sorting : Merge sort and quick sort
- Search: Binary search
- Strassen's Matrix multiplication
- Finding closest pair and convex hull problems
- Multiplication of large integers

16 marks

1. Write the algorithm to perform Binary Search and compute its time complexity. Or Explain binary search algorithm with an example.[N-14][N-15]

BINARY SEARCH ALGORITHM

Very efficient algorithm for searching in sorted array:

K

vs

A[0] . . . A[m] . . . A[n-1]

If K = A[m], stop [successful search]; otherwise, continue searching by the same method in A[0..m-1] if K < A[m] and in A[m+1..n-1] if K > A[m]

// Input: An Array A[0…n-1] sorted in ascending orger and a search key K
//Output: An index of the array's element that is equal to K or -1 if there is no such element.

```
l ← 0;   r ← n-1
while l ≤ r do
    m ← ⌊[l+r]/2⌋
     if  K = A[m]  return m
     else if K < A[m]  r ← m-1
     else l ← m+1
return -1
```

Time complexity: $C_{Worst}[n]=1, C_{avg}[n]=\log_2 n, C_{best}[n]=\log_2 n+1$

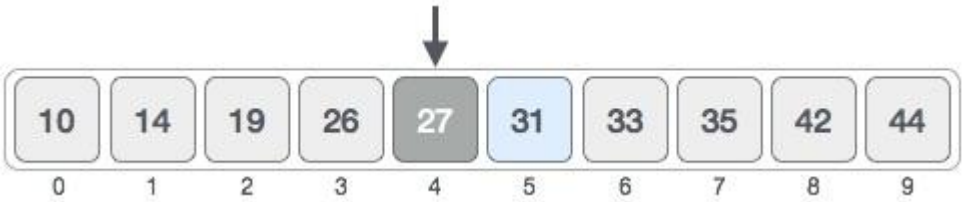
For Example

The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.

First, we shall determine half of the array by using this formula −

mid = low + [high - low] / 2

Here it is, 0 + [9 - 0] / 2 = 4 [integer value of 4.5]. So, 4 is the mid of the array.



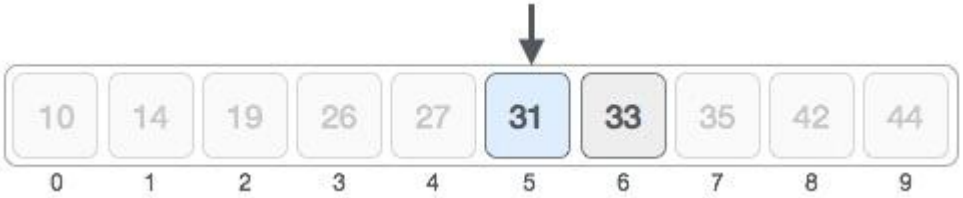
Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array,

to change our low to mid + 1 and find the new mid value again. low = mid + 1, mid = low + [high - low] / 2

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

The value stored at location 7 is not a match, rather it is less than what we are looking for. So, the value must be in the lower part from this location.

Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.

We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

2. Write down the algorithm to construct a convex hull based on divide and conquer strategy. Or Explain the convex hull problem and the solution involved behind it.[N-15]

CONVEX HULL OR QUICK HULL PROBLEM

Convex hull: smallest convex set that includes given points. An O[n^3] brute force time. Assume points are sorted by *x*-coordinate values

- Identify *extreme points* P_1 and P_2 [leftmost and rightmost]
- Compute *upper hull* recursively:
    1. find point P_{max} that is farthest away from line P_1P_2
    2. compute the upper hull of the points to the left of line P_1P_{max}
    3. compute the upper hull of the points to the left of line $P_{max}P_2$
- Compute *lower hull* in a similar manner
- Finding point farthest away from line P_1P_2 can be done in linear time
- Time efficiency: T[n] = T[x] + T[y] + T[z] + T[v] + O[n], where x + y + z +v <= n.
  worst case: $\Theta[n^2]$ T[n] = T[n-1] + O[n]
  average case: $\Theta[n]$

If points are not initially sorted by *x*-coordinate value, this can be accomplished in O[*n* log *n*] time. Several O[*n* log *n*] algorithms for convex hull are known.

CLOSEST PAIR PROBLEM

1. Sort the points by x [list one] and then by y [list two].
2. Divide the points given into two subsets S_1 and S_2 by a vertical line $x = c$ so that half the points lie to the left or on the line and half the points lie to the right or on the line.
3. Find recursively the closest pairs for the left and right subsets.
4. Set $d = \min\{d_1, d_2\}$, We can limit our attention to the points in the symmetric vertical strip of width $2d$ as possible closest pair. Let $C_1$and C_2 be the subsets of points in the left subset S_1 and of the right subset S_2, respectively, that lie in this vertical strip. The points in C_1 and C_2 are stored in increasing order of their y coordinates, taken from the second list.
5. For every point $P[x,y]$ in C_1, we inspect points in C_2 that may be closer to P than d. There can be no more than 6 such points [because $d \le d_2$]! Running time of the algorithm [without sorting] is: $T[n] = 2T[n/2] + M[n]$, where $M[n] \in \Theta[n]$ By the Master Theorem [with $a = 2$, $b = 2$, $d = 1$] $T[n] \in \Theta[n \log n]$ So the total time is $\Theta[n \log n]$.

3. Explain the closest pair and convex hull problem in brute force technique. [N-15]

CLOSEST-PAIR AND CONVEX-HULL PROBLEMS

Find the two closest points in a set of n points [in the two-dimensional Cartesian plane].

Brute-force algorithm

- Compute the distance between every pair of distinct points
- And return the indexes of the points for which the distance is the smallest.

ALGORITHM BruteForceClosestPair[P]

//Finds distance between two closest points in the plane by brute force

//Input: A list P of n [n ≥ 2] points p1[x1, y1], . . . , pn[xn, yn]

//Output: The distance between the closest pair of points

d←∞

for i ←1 to n − 1 do

for j ←i + 1 to n do

d ←min[d, sqrt[[xi− xj]2 + [yi− yj]2]] //sqrt is square root

return d

CONVEX HULL THEOREM The convex hull of any set S of n>2 points not all on the same line is a convex polygon with the vertices at some of the points of S. [If all the points do lie on the same line, the polygon degenerates to a line segment but still with the endpoints at two points of S.]

The convex-hull problem is the problem of constructing the convex hull for a given set S of n points. To solve it, we need to find the points that will serve as the vertices of the polygon in question. Mathematicians call the vertices of such a polygon “extreme points.”

4. Develop a pseudo code for divide and conquer algorithm for merge two sorted arrays into a single sorted one – explain with example. or Write down the algorithm for merge sorting. Explain how the following elements get sorted [310,285,179,652,351,423,861,254,450,520] or Sort the following set of elements using merge sort:12,24,8,71,4,23,6,89,56. Or State and Explain Merge sort algorithm and give the recurrence relation and efficiency [M-14] [N-15][M-15][M-16]

MERGE SORT

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.

Procedure:

Merge sort 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 then merging the two smaller sorted arrays into a single sorted one.

- Divide Step: If given array A has zero or one element, return S; it is already sorted. Otherwise, divide A into two arrays, A1 and A2, each containing about half of the elements of A.
- Recursion Step: Recursively sort array A1 and A2.
- Conquer Step: Combine the elements back in A by merging the sorted arrays A1 and A2 into a sorted sequence

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]

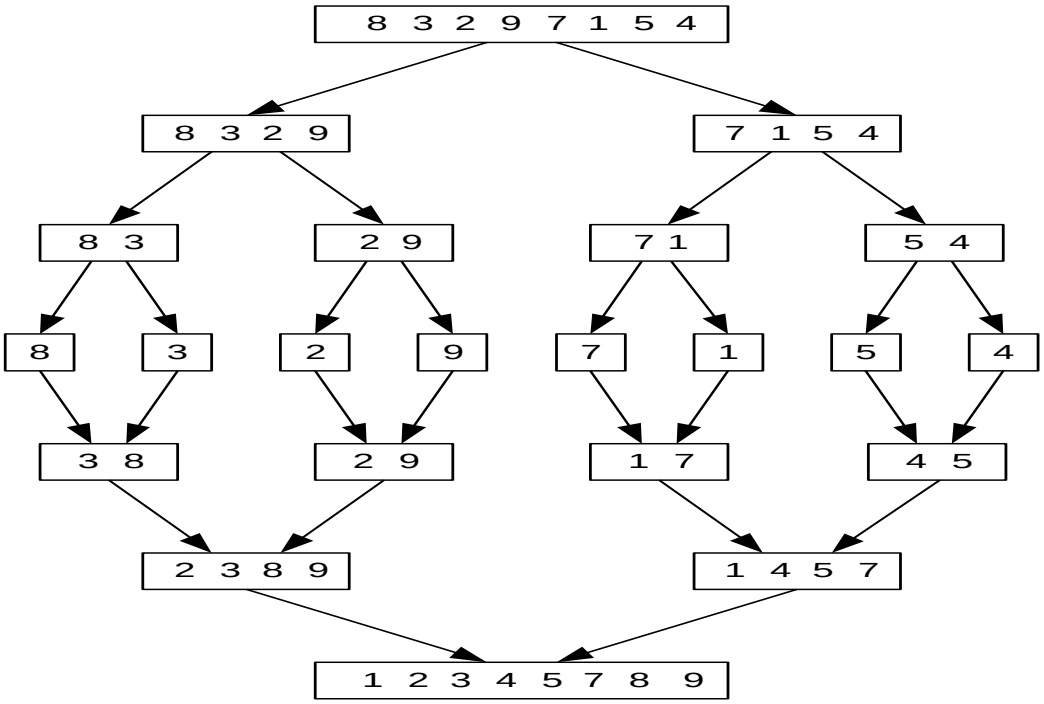
The non-recursive version of Mergesort starts from merging single elements into sorted pairs.

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]

Analysis of Merge sort algorithm

The recurrence relation for the number of key comparisons

C[n] is C[n] = 2C[n/2] + Cmerge[n] for n > 1, C[1] = 0.
In the worst case, Cmerge[n] = n − 1, and we have the recurrence C
worst[n] = 2Cworst[n/2] + n − 1 for n > 1, Cworst[1] = 0.
By Master Theorem, Cworst[n] ∈ Θ[n log n] the exact solution to the worst-case recurrence for n = 2 k
Cworst[n] = n log2 n − n + 1.
For large n, the number of comparisons made by this algorithm in the average case turns out to be about 0.25n less and hence is also in Θ [n log n].
For example



QUICK SORT

Quicksort is the other important sorting algorithm that is based on the divide-and-conquer approach. quicksort divides input elements according to their value. A partition is an arrangement of the array's elements so that all the elements to the left of some element A[s] are less than or equal to A[s], and all the elements to the right of A[s] are greater than or equal to it:
Sort the two subarrays to the left and to the right of A[s] independently. No work required to combine the solutions to the subproblems.
Here is pseudocode of quicksort: call Quicksort[A[0..n − 1]] where As a partition algorithm use the HoarePartition

ALGORITHM Quicksort[A[l..r]]
//Sorts a subarray by quicksort
//Input: Subarray of array A[0..n − 1], defined by its left and right
// indices l and r
//Output: Subarray A[l..r] sorted in nondecreasing order
if l < r
s ←Partition[A[l..r]] //s is a split position

Quicksort[A[l..s − 1]]
Quicksort[A[s + 1..r]]

Algorithm *Partition*$(A[l..r])$
//Partitions a subarray by using its first element as a pivot
//Input: A subarray $A[l..r]$ of $A[0..n-1]$, defined by its left and right
// indices l and r $(l < r)$
//Output: A partition of $A[l..r]$, with the split position returned as
// this function's value
$p \leftarrow A[l]$
$i \leftarrow l$; $j \leftarrow r+1$
repeat
  repeat $i \leftarrow i+1$ **until** $A[i] \geq p$
  repeat $j \leftarrow j-1$ **until** $A[j] \leq p$
  swap$(A[i], A[j])$
until $i \geq j$
swap$(A[i], A[j])$ //undo last swap when $i \geq j$
swap$(A[l], A[j])$
return j

Time Efficiency analysis
Best case: split in the middle — Θ[*n* log *n*]
Worst case: sorted array! — Θ[n^2]
Average case: random arrays — Θ[*n* log *n*]

Example: 5 3 1 9 8 2 4 7
2 3 1 4 5 8 9 7
1 2 3 4 5 7 8 9
1 2 3 4 5 7 8 9
1 2 3 4 5 7 8 9
1 2 3 4 5 7 8 9

5. Explain the method used for performing Multiplication of two large integers. Explain how divide and conquer method can be used to solve the same. [M-16]

Some applications like modern cryptography require manipulation of integers that are over 100 decimal digits long. Since such integers are too long to fit in a single word of a modern computer, they require special treatment.In the conventional pen-and-pencil algorithm for multiplying two n-digit integers, each ofthe n digits of the first number is multiplied by each of the n digits of the second number for thetotal of n2 digit multiplications.
The divide-and-conquer method does the above multiplication in less than n2 digit multiplications.
For any pair of two-digit numbers *a* = *a*1*a*0 and *b* = *b*1*b*0, their product *c* can be

computed by the formula $c = a * b = c2102 + c1101 + c0$, where
$c2 = a1 * b1$ is the product of their first digits,
$c0 = a0 * b0$ is the product of their second digits,
$c1 = [a1 + a0] * [b1 + b0] - [c2 + c0]$ is the product of the sum of the
a's digits and the sum of the *b*'s digits minus the sum of *c*2 and *c*0.
$c = a * b = [a110n/2 + a0] * [b110n/2 + b0] = [a1 * b1]10n + [a1 * b0 + a0 * b1]10n/2 + [a0 * b0]$
$= c210n + c110n/2 + c0,$
where
$c2 = a1 * b1$ is the product of their first halves,
$c0 = a0 * b0$ is the product of their second halves,
$c1 = [a1 + a0] * [b1 + b0] - [c2 + c0]$ is the product of the sum of the
a's halves and the sum of the *b*'s halves minus the sum of *c*2 and *c*0.

Analysis of Time Complexity: By using Master Theorem, we obtain $\boldsymbol{A[n] \in \Theta[n\log 23],}$
Example: For instance: a = 2345, b = 6137, i.e., n=4.
Then *C* = a * b = [23*102+45]*[61*102+37]
$C = a * b = [a110n/2 + a0] * [b110n/2 + b0]$
$= [a1 * b1]10n + [a1 * b0 + a0 * b1]10n/2 + [a0 * b0]$
= [23 * 61]104 + [23 * 37 + 45 * 61]102 + [45 * 37]
= 1403•104 + 3596•102 + 1665
= 14391265

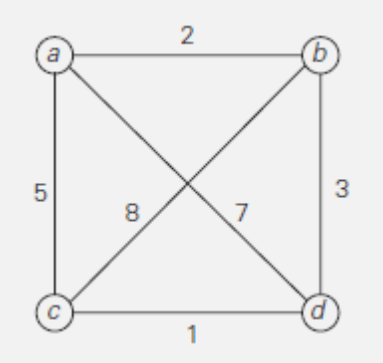
6. Find all the solution to the traveling salesman problem [cities and distance shown below] by exhaustive search. Give the optimal solutions. Or Explain exhaustive searching techniques with example. Or Find the optimal solution to the fractional knapsack problem with example. Or Solve the given knapsack problem un=3,m=20,[p1,p2,p3]=[25,24,15],[w1,w2,w3]=[18,15,10][M-15][N-14][M-14][N-15] [M-16]

TRAVELING SALESMAN PROBLEM

The ***traveling salesman problem [TSP]*** is one of the combinatorial problems. The problem asks to find the shortest tour through a given set of *n* cities that visits each city exactly once before returning to the city where it started.The problem can be conveniently modeled by a weighted graph, with the graph's vertices representing the cities and the edge weights specifying the distances. Then the problem can bestated as the problem of finding the shortest ***Hamiltonian circuit*** of the graph.
[A Hamiltoniancircuit is defined as a cycle that passes through all the vertices of the graph exactly once].
A Hamiltonian circuit can also be defined as a sequence of $n + 1$ adjacent vertices
$vi0, vi1, \ldots, vin-1, vi0,$ where the first vertex of the sequence is the same as the last one and all the other $n - 1$ vertices are distinct. All circuits start and end at one particular vertex.

Figure presents a small instance of the problem and its solution by this method.

For example,



Tour Length

a ---> b ---> c ---> d ---> a l = 2 + 8 + 1 + 7 = 18

a ---> b ---> d ---> c ---> a **l = 2 + 3 + 1 + 5 = 11 optimal**

a ---> c ---> b ---> d ---> a l = 5 + 8 + 3 + 7 = 23

a ---> c ---> d ---> b ---> a **l = 5 + 1 + 3 + 2 = 11 optimal**

a ---> d ---> b ---> c ---> a l = 7 + 3 + 8 + 5 = 23

a ---> d ---> c ---> b ---> a l = 7 + 1 + 8 + 2 = 18

FIGURE Solution to a small instance of the traveling salesman problem by exhaustive search.

Time Complexity of TSP:O[n-1!]

KNAPSACK PROBLEM

Given *n* items of known weights *w*1, *w*2, . . . , *wn* and values *v*1, *v*2, . . . , *vn* and a knapsack of capacity *W*, find the most valuable subset of the items that fit into the knapsack.

Real time examples:

- A Thief who wants to steal the most valuable loot that fits into his knapsack,
- A transport plane that has to deliver the most valuable set of items to a remote location without exceeding the plane's capacity.

The exhaustive-search approach to this problem leads to generating all the subsets of the set of *n* items given, computing the total weight of each subset in order to identify feasible subsets [i.e., the ones with the total weight not exceeding the knapsack capacity], and finding a subset of the largest value among them.

Time Complexity of KP: O[2^n]

Given *n* items:

- weights: w_1 w_2 … w_n
- values: v_1 v_2 … v_n
- a knapsack of capacity *W*

Find most valuable subset of the items that fit into the knapsack

Example: Knapsack capacity W=16

item	weight	value
1	2	$20
2	5	$30
3	10	$50

4 5 $10

Subset	Total weight	Total value
{1}	2	$20
{2}	5	$30
{3}	10	$50
{4}	5	$10
{1,2}	7	$50
{1,3}	12	$70
{1,4}	7	$30
{2,3}	15	$80
{2,4}	10	$40
{3,4}	15	$60
{1,2,3}	17	not feasible
{1,2,4}	12	$60
{1,3,4}	17	not feasible
{2,3,4}	20	not feasible
{1,2,3,4}	22	not feasible

ASSIGNMENT PROBLEM

There are n people who need to be assigned to n jobs, one person per job. The cost of assigning person i to job j is C[i,j]. Find an assignment that minimizes the total cost.

For Example,

	Job 0	Job 1	Job 2	Job 3
Person 0	9	2	7	8
Person 1	6	4	3	7
Person 2	5	8	1	8
Person 3	7	6	9	4

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

Assignment [col.#s]	Total Cost
1, 2, 3, 4	9+4+1+4=18
1, 2, 4, 3	9+4+8+9=30
1, 3, 2, 4	9+3+8+4=24
1, 3, 4, 2	9+3+8+6=26

1, 4, 2, 3 **9+7+8+9=33**

1, 4, 3, 2 **9+7+1+6=23**

etc.

[For this particular instance, the optimal assignment can be found by exploiting the specific features of the number given. It is: 2,1,3,4]

Time Complexity of TSP: O[n!]

7. Explain about Strassen's Matrix Multiplication with example.

The divide-and-conquer approach can reduce the number of one-digit multiplications in multiplying two integers by using strassen's logic. Thus, to multiply two 2 × 2 matrices, Strassen's algorithm makes 7 multiplications and 18 additions/subtractions, whereas the brute-force algorithm requires 8 multiplications and 4 additions. These numbers should not lead us to multiplying 2 × 2 matrices by Strassen's algorithm. Itsimportance stems from its asymptotic superiority as matrix order n goes to infinity.

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

where

$$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11}),$$
$$m_2 = (a_{10} + a_{11}) * b_{00},$$
$$m_3 = a_{00} * (b_{01} - b_{11}),$$
$$m_4 = a_{11} * (b_{10} - b_{00}),$$
$$m_5 = (a_{00} + a_{01}) * b_{11},$$
$$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01}),$$
$$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11}).$$

Example: Multiply the following two matrices by Strassen's matrix multiplication algorithm.

$$A = \begin{bmatrix} 1 & 0 & 2 & 1 \\ 4 & 1 & 1 & 0 \\ 0 & 1 & 3 & 0 \\ 5 & 0 & 2 & 1 \end{bmatrix} \qquad B = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 2 & 1 & 0 & 4 \\ 2 & 0 & 1 & 1 \\ 1 & 3 & 5 & 0 \end{bmatrix}$$

Answer:

$$C = \begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} x \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

Where $A_{00} = \begin{bmatrix} 1 & 0 \\ 4 & 1 \end{bmatrix}$ $A_{01} = \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix}$ $A_{10} = \begin{bmatrix} 0 & 1 \\ 5 & 0 \end{bmatrix}$ $A_{11} = \begin{bmatrix} 3 & 0 \\ 2 & 1 \end{bmatrix}$

$B_{00} = \begin{bmatrix} 0 & 1 \\ 2 & 1 \end{bmatrix}$ $B_{01} = \begin{bmatrix} 0 & 1 \\ 0 & 4 \end{bmatrix}$ $B_{10} = \begin{bmatrix} 2 & 0 \\ 1 & 3 \end{bmatrix}$ $B_{11} = \begin{bmatrix} 1 & 1 \\ 5 & 0 \end{bmatrix}$

$$M_1 = (A_{00}+A_{11})*(B_{00}+B_{11}) = \left(\begin{bmatrix} 1 & 0 \\ 4 & 1 \end{bmatrix} + \begin{bmatrix} 3 & 0 \\ 2 & 1 \end{bmatrix}\right) * \left(\begin{bmatrix} 0 & 1 \\ 2 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 1 \\ 5 & 0 \end{bmatrix}\right) = \begin{bmatrix} 4 & 0 \\ 6 & 2 \end{bmatrix} * \begin{bmatrix} 1 & 2 \\ 7 & 1 \end{bmatrix} = \begin{bmatrix} 4 & 8 \\ 20 & 14 \end{bmatrix}$$

Similarly apply Strassen's matrix multiplication algorithm to find the following.

$M_2 = \begin{bmatrix} 2 & 4 \\ 2 & 8 \end{bmatrix}$, $M_3 = \begin{bmatrix} -1 & 0 \\ -9 & 4 \end{bmatrix}$, $M_4 = \begin{bmatrix} 6 & -3 \\ 3 & 0 \end{bmatrix}$, $M_5 = \begin{bmatrix} 8 & 3 \\ 10 & 5 \end{bmatrix}$, $M_6 = \begin{bmatrix} 2 & -3 \\ -2 & -3 \end{bmatrix}$, $M_7 = \begin{bmatrix} 3 & 2 \\ -9 & -4 \end{bmatrix}$

$C_{00} = \begin{bmatrix} 5 & 4 \\ 4 & 5 \end{bmatrix}$, $C_{01} = \begin{bmatrix} -7 & 3 \\ 1 & 9 \end{bmatrix}$, $C_{10} = \begin{bmatrix} 8 & 1 \\ 5 & 8 \end{bmatrix}$, $C_{11} = \begin{bmatrix} 3 & 7 \\ 7 & 7 \end{bmatrix}$

$$C = \begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} 5 & 4 & 7 & 3 \\ 4 & 5 & 1 & 9 \\ 8 & 1 & 3 & 7 \\ 5 & 8 & 7 & 7 \end{bmatrix}$$

UNIT – III DYNAMIC PROGRAMMING AND GREEDY TECHNIQUE

2 marks

1. Define - Dynamic programming.

Dynamic programming is a technique for solving problems with overlapping sub problems.

2. What are the features of dynamic programming? [M-14] [N – 14] [M - 15]

- Optimal solutions to sub problems are retained so as to avoid re-computing their values.
- Decision sequences containing subsequences that are sub optimal are not considered.
- It definitely gives the optimal solution always.

3. Define - Principle of optimality [N-14] [M – 14]

It states that an optimal sequence of decisions has the property that whenever the initial stage or decisions must constitute an optimal sequence with regard to stage resulting from the first decision.

4. Write the difference between the Greedy method and Dynamic programming.

Greedy method	Dynamic programming
Only one sequence of decision is generated.	Many number of decisions are generated.
It does not guarantee to give an optimal solution always.	It definitely gives an optimal solution always.

5. What is greedy technique?

Greedy technique suggests a greedy grab of the best alternative available in the hope that a sequence of locally optimal choices will yield a globally optimal solution to the entire problem. The choice must be made as follows

- *Feasible : It has to satisfy the problem's constraints*
- *Locally optimal: It has to be the best local choice among all feasible choices available on that step.*
- *Irrevocable : Once made, it cannot be changed on a subsequent step of the algorithm*

6. What is the use of Dijksra's algorithm? [M – 16]

Dijkstra's algorithm is used to solve the single-source shortest-paths problem: for a given vertex called the source in a weighted connected graph, find the shortest path to all its other vertices. The single-source shortest-paths problem asks for a family of paths, each leading from the source to a different vertex in the graph, though some paths may have edges in common.

7. Define - Spanning tree and MST.

Spanning tree of a connected graph *G*: a connected acyclic subgraph of *G* that includes all of *G*'s vertices.*Minimum spanning tree* of a weighted, connected graph *G*: a spanning tree of *G* of the minimum total weight.

8. Define -Warshall's algorithm [M – 16]

Warshall's algorithm is an application of dynamic programming technique, which is used to find the transitive closure of a directed graph.

9. Define the single source shortest path problem. [M – 16]

Dijkstra's algorithm solves the single source shortest path problem of finding shortest paths from a given vertex[the source], to all the other vertices of a weighted graph or digraph.
Dijkstra's algorithm provides a correct solution for a graph with non-negative weights.

10. State Assignment problem. [M – 16]

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 $[i, j]$ for each pair $i, j = 1, 2, \ldots, n$. The problem is to find an assignment with the minimum total cost.

11. What are the applications or examples of dynamic programming?

- Computing a binomial co-efficient
- Finding shortest path of Floyd's algorithm
- Finding the transitive closure of a graph using Warshall's algorithm
- Optimal Binary search tree

12. Write the Warshall's algorithm to find the transitive closure of a digraph.

ALGORITHM Warshall(A[1..n, 1..n])
//ImplementsWarshall's algorithm for computing the transitive closure
//Input: The adjacency matrix A of a digraph with n vertices
//Output: The transitive closure of the digraph
R(0) ←A
for k←1 to n do
for i ←1 to n do
for j ←1 to n do
R(k)[i, j]←R(k−1)[i, j] or (R(k−1)[i, k] and R(k−1)[k, j])
return R(n)

13. What are the applications or examples of greedy techniques?

- Finding Minimum spanning tree of using Prim's and Kruskal's algorithm
- Finding all pairs shortest path problem of using Dijkstra's algorithm
- Huffman trees

16 marks

1. Write and analyze the Prim's Algorithm. How do you construct a MST using Kruskal's Algorithm? Explain. Define Spanning tree. Discuss the design steps in Kruskal's algorithm tp construct MST with example.[M-14][M-15][N-15]

MINIMUM SPANNING TREE

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.

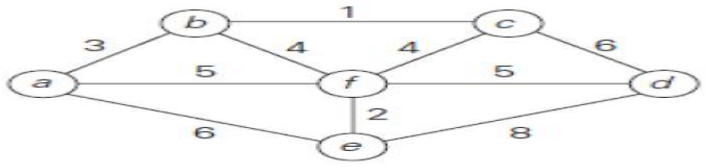
PRIM'S ALGORITHM

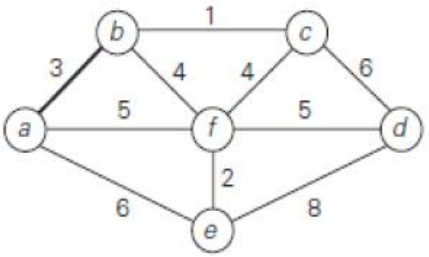
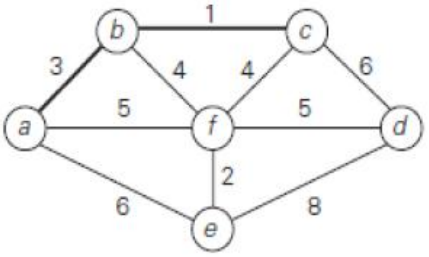
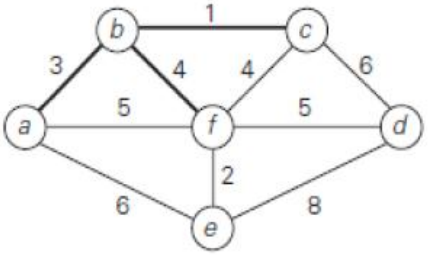
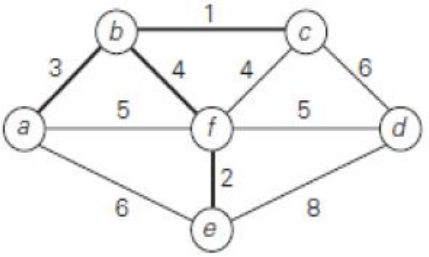
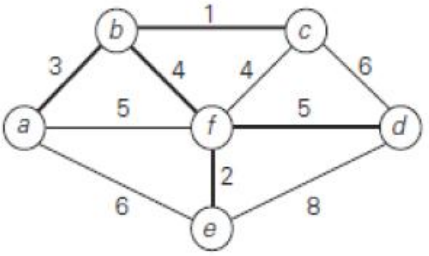
Prim's algorithm constructs a minimum spanning tree through a sequence of expanding sub trees. The initial sub tree in such a sequence consists of a single vertex selected arbitrarily from the set V of the graph's vertices. On each iteration, the algorithm expands the current tree in the greedy manner by simply attaching to it the nearest vertex not in that tree.

ALGORITHM Prim[G]

//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph G = V, E
//Output: ET , the set of edges composing a minimum spanning tree of G
VT←{v0} //the set of tree vertices can be initialized with any vertex
ET←Ø
for i ←1 to |V| − 1 do
find a minimum-weight edge e∗ = [v∗, u∗] among all the edges [v, u]
such that v is in VT and u is in V − VT
VT←VT∪ {u∗}
ET←ET∪ {e∗}
return ET

For example,



Tree vertices	Remaining vertices	Illustration
a(−, −)	**b(a, 3)** c(−, ∞) d(−, ∞) e(a, 6) f(a, 5)	
b(a, 3)	**c(b, 1)** d(−, ∞) e(a, 6) f(b, 4)	
c(b, 1)	d(c, 6) e(a, 6) **f(b, 4)**	
f(b, 4)	d(f, 5) **e(f, 2)**	
e(f, 2)	**d(f, 5)**	
d(f, 5)		

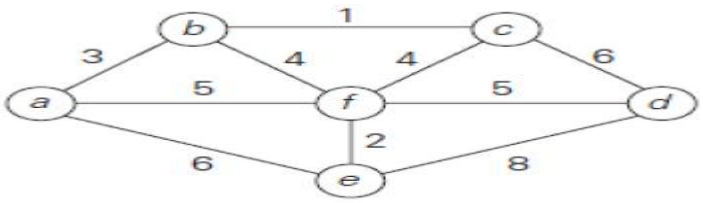
Efficiency of the Prim's algorithm

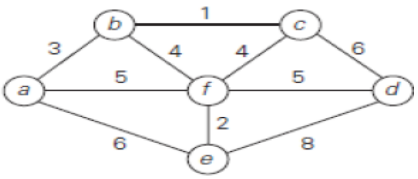
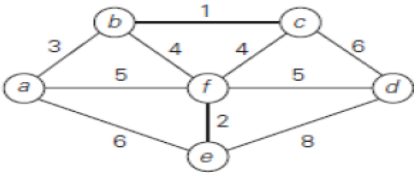
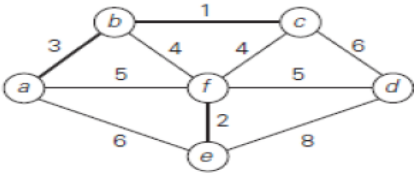
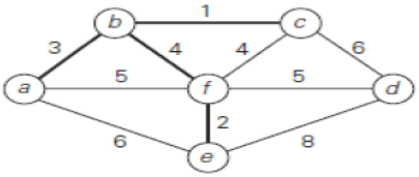
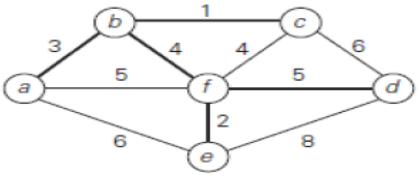
If a graph is represented by its adjacency lists and the priority queue is implemented as a min-heap, the running time of the algorithm is O[|E| log |V |] in a connected graph, where |V| − 1≤ |E|.

KRUSKAL 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.

For Example,



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	
bc 1	bc **ef** ab bf cf af df ae cd de 1 **2** 3 4 4 5 5 6 6 8	
ef 2	bc ef **ab** bf cf af df ae cd de 1 2 **3** 4 4 5 5 6 6 8	
ab 3	bc ef ab **bf** cf af df ae cd de 1 2 3 **4** 4 5 5 6 6 8	
bf 4	bc ef ab bf cf af **df** ae cd de 1 2 3 4 4 5 **5** 6 6 8	
df 5		

ALGORITHM Kruskal[G]

//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph G = V, E
//Output: ET , the set of edges composing a minimum spanning tree of G
sort E in nondecreasing order of the edge weights w[ei1] ≤ . . . ≤ w[ei|E|]
ET←Ø; ecounter ←0 //initialize the set of tree edges and its size
k←0 //initialize the number of processed edges
while ecounter < |V| − 1 do
k←k + 1
if ET∪ {eik} is acyclic
ET<-ET<- {eik}; ecounter<-ecounter + 1
return ET

Efficiency of the Kruskal's algorithm

If a graph is represented by its adjacency lists and the priority queue is implemented as a min-heap, the running time of the algorithm is O[|E| log |V |] in a connected graph, where |V| − 1≤ |E|.

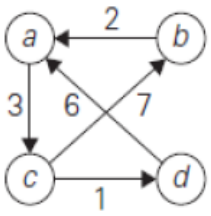
2. Write down and explain the algorithm to solve all pair's shortest paths problems? Or Solve all pairs shortest path problem for the digraph with the weight matrix given below. Or Describe all pairs shortest path problem and write procedure to compute lengths of shortest paths.[M-13][M-14] [M-15]

FLOYD'S ALGORITHM

Given a weighted connected graph (undirected or directed), the all-pairs shortest paths problem asks to find the distances—i.e., the lengths of the shortest paths— from each vertex to all other vertices.

ALGORITHM Floyd(W[1..n, 1..n])
//Implements Floyd's algorithm for the all-pairs shortest-paths problem
//Input: The weight matrix W of a graph with no negative-length cycle
//Output: The distance matrix of the shortest paths' lengths
D ←W //is not necessary if W can be overwritten
for k←1 to n do
for i ←1 to n do
for j ←1 to n do
D[i, j]←min{D[i, j], D[i, k]+ D[k, j]}
return D

Efficiency of Floyd's Algorithm: Time efficiency $\Theta(n^3)$ and Space Efficiency is $\Theta(n^2)$
For Example,



$D^{(0)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

Lengths of the shortest paths with no intermediate vertices ($D^{(0)}$ is simply the weight matrix).

$D^{(1)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	**5**	∞
c	∞	7	0	1
d	6	∞	**9**	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 1, i.e., just *a* (note two new shortest paths from *b* to *c* and from *d* to *c*).

$D^{(2)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	**9**	7	0	1
d	6	∞	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 2, i.e., **a** and *b* (note a new shortest path from *c* to *a*).

$D^{(3)} =$

	a	b	c	d
a	0	**10**	3	**4**
b	2	0	5	**6**
c	9	7	0	1
d	6	**16**	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 3, i.e., **a**, *b*, and *c* (note four new shortest paths from *a* to *b*, from a to *d*, from *b* to *d*, and from *d* to *b*).

$D^{(4)} =$

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	**7**	7	0	1
d	6	16	9	0

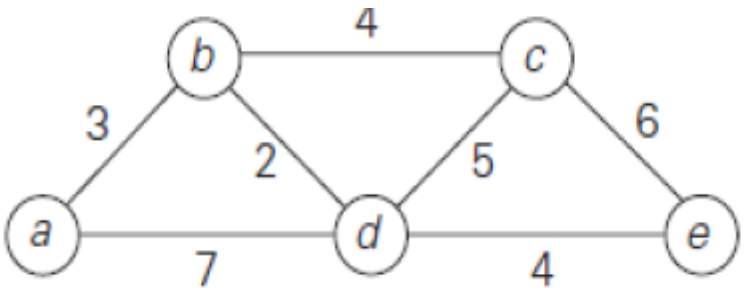
Lengths of the shortest paths with intermediate vertices numbered not higher than 4, i.e., **a**, *b*, *c*, and *d* (note a new shortest path from *c* to *a*).

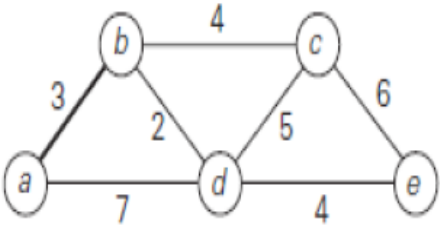
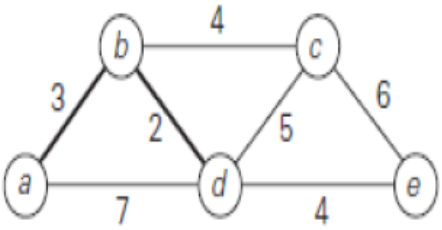
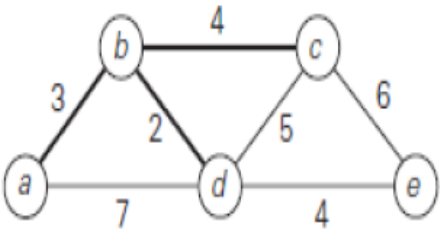
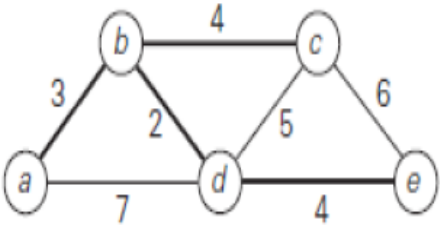
FIGURE 3.5 Application of Floyd's algorithm to the digraph shown. Updated elements are shown in bold.

DIJKSTRA'S ALGORITHM

Consider the single-source shortest-paths problem: for a given vertex called the source in a weighted connected graph, find shortest paths to all its other vertices. The single-source shortest-paths problem asks for a family of paths, each leading from the source to a different vertex in the graph, though some paths may, of course, have edges in common. There are several well-known algorithms for finding shortest paths, including Floyd's algorithm for the more general all-pairs shortest-paths problem and the best-known algorithm for the single-source shortest-paths problem, this algorithm is applicable to undirected and directed graphs with nonnegative weights only, 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 it's ith iteration commences, the algorithm has already identified the shortest paths to i − 1 other vertices nearest to the source.

For Example,



Tree vertices	Remaining vertices	Illustration
a(−, 0)	**b(a, 3)** c(−, ∞) d(a, 7) e(−, ∞)	
b(a, 3)	c(b, 3 + 4) **d(b, 3 + 2)** e(−, ∞)	
d(b, 5)	**c(b, 7)** e(d, 5 + 4)	
c(b, 7)	**e(d, 9)**	
e(d, 9)		

The shortest paths [identified by following nonnumeric labels backward from a destination vertex in the left column to the source] and their lengths [given by numeric labels of the tree vertices] are as follows:
from a to b : a − b of length 3
from a to d : a − b − d of length 5
from a to c : a − b − c of length 7
from a to e : a − b − d − e of length 9

ALGORITHM Dijkstra[G, s]
//Dijkstra's algorithm for single-source shortest paths
//Input: A weighted connected graph G = V, E with nonnegative weights
// and its vertex s
//Output: The length dv of a shortest path from s to v
// and its penultimate vertex pv 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
$V_T \leftarrow \emptyset$
for i ←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 − VT that is adjacent to u^* do if $d_{u^*} + w[u^*, u] < du$
$d_u \leftarrow d_{u^*} + w[u^*, u]$; $p_u \leftarrow u^*$ Decrease[Q, u, d_u]

Time efficiency analysis of Dijkstra's algorithm:
For graphs represented by their adjacency lists and the priority queue implemented as a min-heap, it is in O[|E| log |V |].

3. Write the procedure to compute Huffman code. [N-15]

Suppose we have to encode a text that comprises symbols from some n-symbol alphabet by assigning to each of the text's symbols some sequence of bits called the code word. For example, we can use a fixed-length encoding that assigns to each symbol a bit string of the same length m (m ≥ log2 n).

Step 1 Initialize n one-node trees and label them with the symbols of the alphabet given. Record the frequency of each symbol in its tree's root to indicate the tree's **weight**. (More generally, the weight of a tree will be equal to the sum of the frequencies in the tree's leaves.)

Step 2 Repeat the following operation until a single tree is obtained. Find two trees with the smallest weight. Make them the left and right sub tree of a new tree and record the sum of their weights in the root of the new tree as its weight.

A tree constructed by the above algorithm is called a **Huffman tree**. It defines—in the manner

described above—a **Huffman code**.

EXAMPLE Consider the five-symbol alphabet {A, B, C, D, _} with the following occurre frequencies in a text made up of these symbols:

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

The Huffman tree construction for this input is shown in Figure 3.18

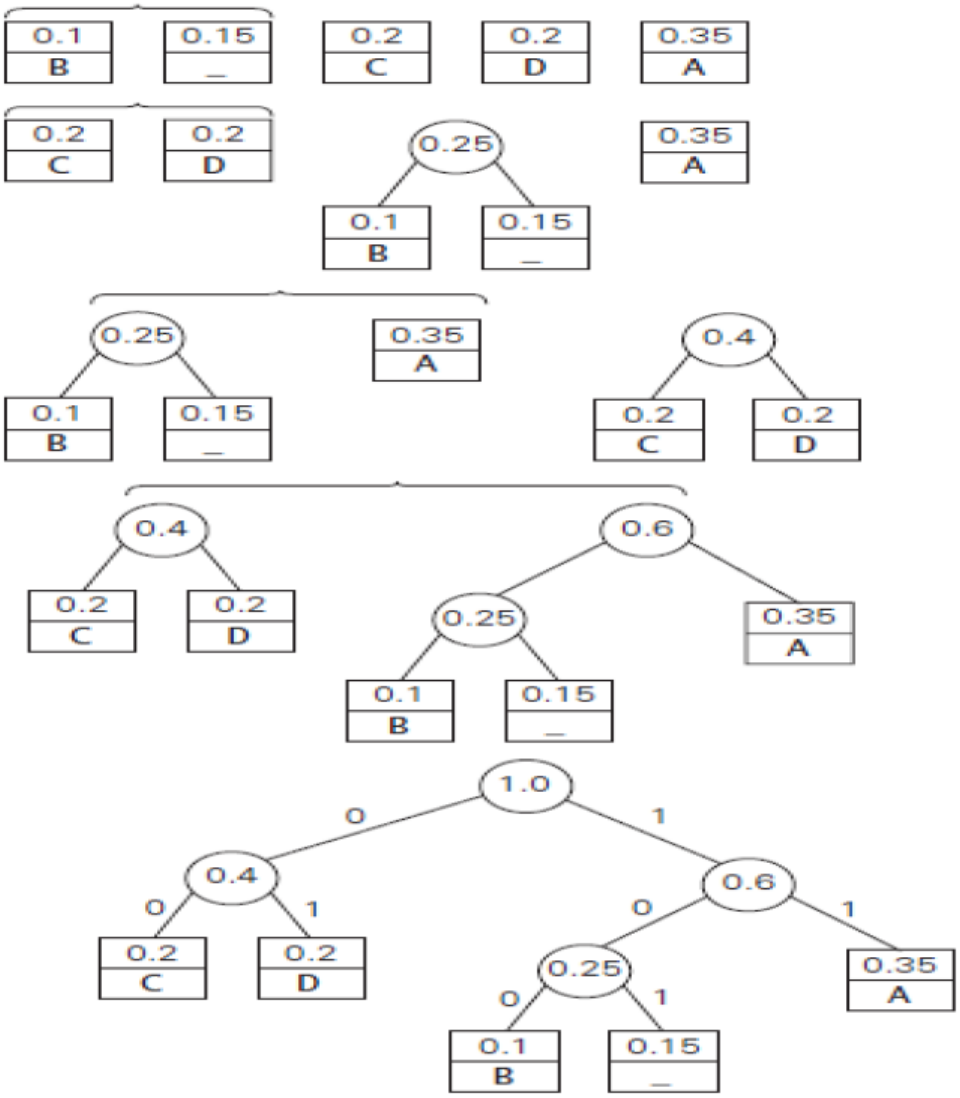


FIGURE 3.18 Example of constructing a Huffman coding tree.

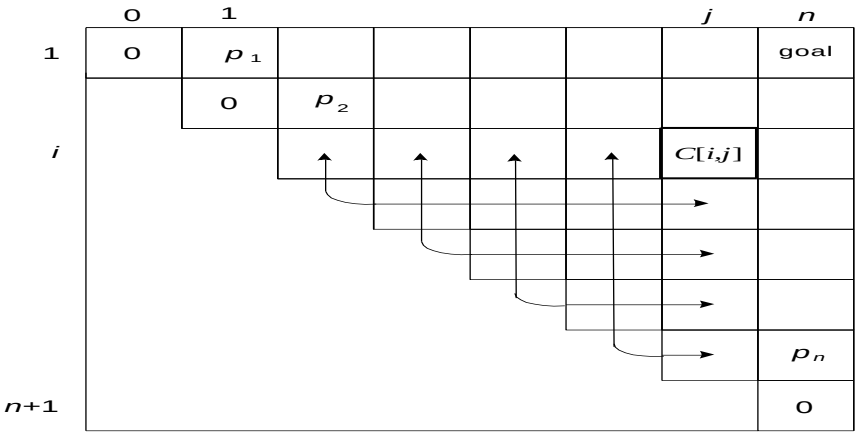
4. Write short notes on optimal binary search tree. Or Write an algorithm to construct the optimal binary search tree given the roots r(i,j), 0<=i<=j<=n. Also prove that this could be performed in time O(n)

[M-15][N-14]

Let C[i,j] be minimum average number of comparisons made in T[i,j], optimal BST for keys ai < …< aj , where 1 ≤ i ≤ j ≤ n. Consider optimal BST among all BSTs with some ak (i ≤ k ≤ j) as their root; T[i,j] is the best among them.

ALGORITHM Optimal BST(P [1..n])

//Finds an optimal binary search tree by dynamic programming
//Input: An array P[1..n] of search probabilities for a sorted list of n keys
//Output: Average number of comparisons in successful searches in the
// optimal BST and table R of subtrees' roots in the optimal BST
for i ←1 to n do
C[i, i − 1]←0
C[i, i]←P[i]
R[i, i]←i
C[n + 1, n]←0
for d ←1 to n − 1 do //diagonal count
for i ←1 to n − d do
j ←i + d
minval←∞
for k←i to j do
if C[i, k − 1]+ C[k + 1, j]< minval
minval←C[i, k − 1]+ C[k + 1, j]; kmin←k
R[i, j]←kmin
sum←P[i]; for s ←i + 1 to j do sum←sum + P[s]
C[i, j]←minval + sum
return C[1, n], R



Running time of this algorithm: Time Efficiency $\Theta(n^2)$ and Space Efficiency : $\Theta(n^3)$

For Example,

EXAMPLE: Let us illustrate the algorithm by applying it to the four-key set we used at the beginning of this section:

key	A	B	C	D
probability	0.1	0.2	0.4	0.3

The initial tables are:

main table

	0	1	2	3	4
1	0	0.1			
2		0	0.2		
3			0	0.4	
4				0	0.3
5					0

root table

	0	1	2	3	4
1		1			
2			2		
3				3	
4					4
5					

Let us compute $C(1, 2)$:

$$C(1, 2) = \min \left\{ \begin{array}{ll} k = 1: & C(1, 0) + C(2, 2) + \sum_{s=1}^{2} p_s = 0 + 0.2 + 0.3 = 0.5 \\ k = 2: & C(1, 1) + C(3, 2) + \sum_{s=1}^{2} p_s = 0.1 + 0 + 0.3 = 0.4 \end{array} \right\}$$
$$= 0.4.$$

Thus, out of two possible binary trees containing the first two keys, A and B, the root of the optimal tree has index 2 (i.e., it contains B), and the average number of comparisons in a successful search in this tree is 0.4.

We arrive at the following final tables:

main table

	0	1	2	3	4
1	0	0.1	0.4	1.1	1.7
2		0	0.2	0.8	1.4
3			0	0.4	1.0
4				0	0.3
5					0

root table

	0	1	2	3	4
1		1	2	3	3
2			2	3	3
3				3	3
4					4
5					

Thus, the average number of key comparisons in the optimal tree is equal to 1.7. Since R(1, 4) = 3, the root of the optimal tree contains the third key, i.e., C. Its left subtree is made up of keys A and B, and its right subtree contains just key D. To find the specific structure of these subtrees, we find first their roots by consulting the root table again as follows. Since R(1, 2) = 2, the root of the optimal tree containing A and B is B, with A being its left child (and the root of the one node tree: R(1, 1) = 1). Since R(4, 4) = 4, the root of this one-node optimal tree is its only key D. Figure 3.10 presents the optimal tree in its entirety.

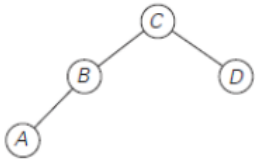


FIGURE 3.10 Optimal binary search tree for the above example.

UNIT IV ITERATIVE IMPROVEMENT

2 marks

1. Define flow and flow conservation requirement. [N – 15]

A *flow* is an assignment of real numbers x*ij* to edges [*i,j*] of a given network that satisfy the following:

flow-conservation requirements: The total amount of material entering an intermediate vertex must be equal to the total amount of the material leaving the vertex capacity constraints

0 ≤ *xij* ≤ *uij* for every edge [*i,j*] □ *E*

2. What is cut and min cut? [M - 15]

Let X be a set of vertices in a network that includes its source but does not include its sink, and let X, the complement of X, be the rest of the vertices including the sink. The *cut* induced by this partition of the vertices is the set of all the edges with a tail in X and a head in X.

Capacity of a cut is defined as the sum of capacities of the edges that compose the cut.

Minimum cut is a cut of the smallest capacity in a given network

3. State max – flow – min – cut theorem.

The value of maximum flow in a network is equal to the capacity of its minimum cut.

4. Define Bipartite Graphs. [M – 15]

Bipartite graph: a graph whose vertices can be partitioned into two disjoint sets V and U, not necessarily of the same size, so that every edge connects a vertex in V to a vertex in U. A graph is bipartite if and only if it does not have a cycle of an odd length.

7. What is augmentation and augmentation path?

- The length of an augmenting path is always odd
- Adding to M the odd numbered path edges and deleting from it the even numbered path edges increases the matching size by 1 [*augmentation*]

8. Write the detailed description about simplex method.

Step 0 [Initialization]
Step 1 [Optimality test]
Step 2 [Find entering variable].
Step 3 [Find departing variable]
Step 4 [Form the next tableau]

9. Define - Maximum flow problem

Problem of maximizing the flow of a material through a transportation network [e.g., pipeline system, communications or transportation networks]

10. State Extreme point theorem. [M – 16]

Extreme point theorem states that if S is convex and compact in a locally convex space, then S is the closed convex hull of its extreme points: Convex set has its extreme points at the boundary. Extreme points should be the end points of the line connecting any two points of convex set.

16 marks

1. Summarize the Simplex method. Or Write the procedure to initialize simplex which determine if a linear program is feasible or not . Or Use simplex to solve the farmers problem given below.[M-16][N-15][M-15]

Simplex method procedure:

Step 0 [Initialization] Present a given LP problem in standard form and set up initial tableau.

Step 1 [Optimality test] If all entries in the objective row are nonnegative then stop: the tableau represents an optimal solution.

Step 2 [Find entering variable] Select the most negative entry in the objective row. Mark its column to indicate the entering variable and the pivot column.

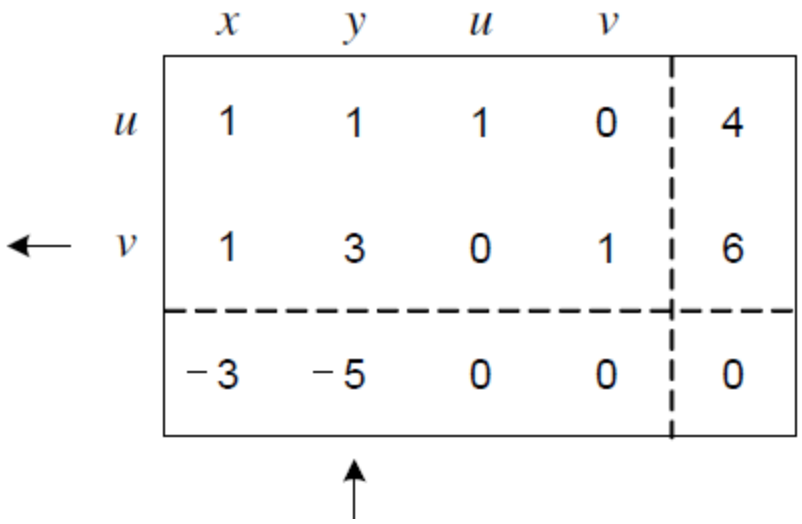
Step 3 [Find departing [leaving] variable] For each positive entry in the pivot column, calculate the θ-ratio by dividing that row's entry in the rightmost column [solution] by its entry in the pivot column. [If there are no positive entries in the pivot column then stop: the problem is unbounded.] Find the row with the smallest θ-ratio, mark this row to indicate the departing variable and the pivot row.

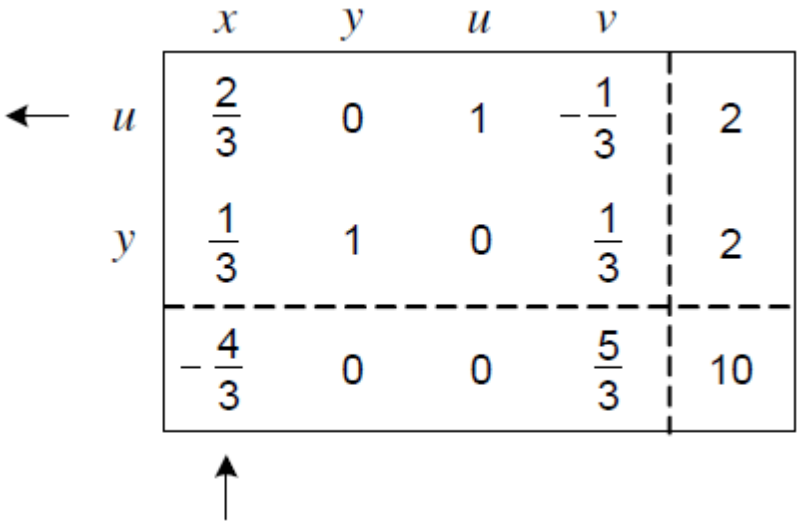
Step 4 [Form the next tableau] Divide all the entries in the pivot row by its entry in the pivot column. Subtract from each of the other rows, including the objective row, the new pivot row multiplied by the entry in the pivot column of the row in question. Replace the label of the pivot row by the variable's name of the pivot column and go back to Step 1.

Standard form of LP problem

- Must be a maximization problem
- All constraints [except the nonnegativity constraints] must be in the form of linear equations
- All the variables must be required to be nonnegative
- Thus, the general linear programming problem in standard form with *m* constraints and *n* unknowns [$n \geq m$] is
  Maximize $c1\ x1 + ... + cn\ xn$
  Subject to $ai\ 1x1 + ... + ain\ xn = bi\ ,\ i = 1,...,m,\ x1 \geq 0, ...\ ,\ xn \geq 0$
  Example
  maximize $3x + 5y$ maximize $3x + 5y + 0u + 0v$
  subject to $x + y \leq 4$ subject to $x + y + u = 4$
  $x + 3y \leq 6$ to $x + 3y + v = 6$
  $x \geq 0,\ y \geq 0$ $x \geq 0,\ y \geq 0,\ u \geq 0,\ v \geq 0$
  Variables *u* and *v*, transforming inequality constraints into equality constrains, are called *slack Variables*

	x	*y*	*u*	*v*	
u	1	1	1	0	4
v	1	3	0	1	6
	−3	−5	0	0	0





	x	y	u	v	
x	1	0	3/2	−1/3	3
y	0	1	−1/2	1/2	1
	0	0	2	1	14

basic feasible sol. $(3, 1, 0, 0)$ $z = 14$

Example 1:

Use Simplex method to solve the formers problem given below.

A farmer has a 320 acre farm on which she plants two crops: corn and soybeans. For each acre of corn planted, her expenses are $50 and for each acre of soybeans planted, her expenses are $100. Each acre of corn requires 100 bushels of storage and yields a profit of $60; each acre of soybeans requires 40 bushels of storage and yields a profit of $90. If the total amount of storage space available is 19,200 bushels and the farmer has only $20,000 on hand, how many acres of each crop should she plant in order to maximize her profit? What will her profit be if she follows this strategy?

Solution

Linear Programming Problem Formulation

	Corn	Soybean	**Total**
Expenses	$50	$100	$20,000
Storage(bushels)	100	40	19,200
Profit	60	90	Maximize profit

A farmer has a 320 acre farm is unwanted data but c+s<=320.

c = corn planted acres and s = soybean planted acres

$50c + 100s \leq 20{,}000$

$100c + 40s \leq 19{,}200$

Maximize: $60c + 90s = P$

Canonical form of LPP

Maximize: $60c + 90s$

Subject to $50c + 100s = 20000$

$100c + 40s = 19200$

$c \geq 0, s \geq 0$

Solving by algebra (Intersection of lines)

Maximize: $60c + 90s$

Subject to $50c + 100s = 20000$ (1)

$100c + 40s = 19200$ (2)

(1)/50 => $c + 2s = 400$

(2)/20 => $5c + 2s = 960$

(2) – (1) => $4c = 560$

$c = 140$

Substitute c = 140 in (1) then s = 130

Profit: $p = 60c + 90s = 60(140) + 90(130) = \$20{,}100$

She should plant 140 acres corn and 130 acres of soybean for $20,100.

2. State and Prove Maximum Flow Min cut Theorem. Or Explain Max-Flow Problem Or How do you compute maximum flow for the following graph using ford-Fulkerson method?[M-15][N-15][M-16]

Maximum Flow Problem

Problem of maximizing the flow of a material through a transportation network [e.g., pipeline system, communications or transportation networks]

Formally represented by a connected weighted digraph with *n* vertices numbered from 1 to *n* with the following properties:

• Contains exactly one vertex with no entering edges, called the ***source*** [numbered 1]

• Contains exactly one vertex with no leaving edges, called the ***sink*** [numbered *n*]

• Has positive integer weight *uij* on each directed edge [*i.j*], called the ***edge capacity*****,** indicating the upper bound on the amount of the material that can be sent from *i* to *j* through this edge.

A digraph satisfying these properties is called a **flow network** or simply a network

Flow value and Maximum Flow Problem

Since no material can be lost or added to by going through intermediate vertices of the network, the total amount of the material leaving the source must end up at the sink:

$\Sigma\, x1j = \Sigma\, xjn$

j: [1,j] ϵ E j: [j,n] ϵ E

The *value* of the flow is defined as the total outflow from the source [= the total inflow into the sink].

The *maximum flow problem* is to find a flow of the largest value [maximum flow] for a given network.

Max-Flow Min-Cut Theorem

1. The value of maximum flow in a network is equal to the capacity of its minimum cut
2. The shortest augmenting path algorithm yields both a maximum flow and a minimum cut:
   - Maximum flow is the final flow produced by the algorithm
   - Minimum cut is formed by all the edges from the labeled vertices to unlabeled vertices on the last iteration of the algorithm.
   - All the edges from the labeled to unlabeled vertices are full, i.e., their flow amounts are equal to the edge capacities, while all the edges from the unlabeled to labeled vertices, if any, have zero flow amounts on them.

Flow augmenting path algorithm or Ford Fulkerson method to solve max-flow problem

Thus, to find a flow-augmenting path for a flow x, we need to consider paths from source to sink in the underlying undirected graph in which any two consecutive vertices i, j are either

i. connected by a directed edge from i to j with some positive unused capacity rij = uij − xij (so that we can increase the flow through that edge by up to rij units), or

ii. connected by a directed edge from j to i with some positive flow xji (so that we can decrease the flow through that edge by up to xji units)

For example,

Edges of the first kind are called forward edges because their tail is listed before their head in th vertex list 1→. . . i →j . . .→n defining the path; edges of the second kind are called backward edges because their tail is listed after their head in the path list 1→. . . i ←j . . .→n. To illustrate, for the path 1→4→3←2→5→6 of the last example, (1, 4), (4, 3), (2, 5), and (5, 6) are the forward edges, and (3, 2) is the backward edge.

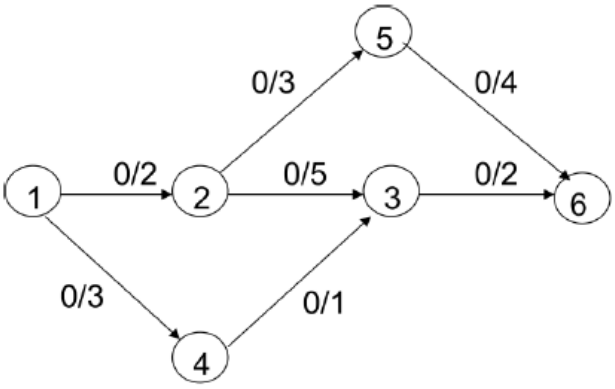
Shortest-augmenting-path algorithm

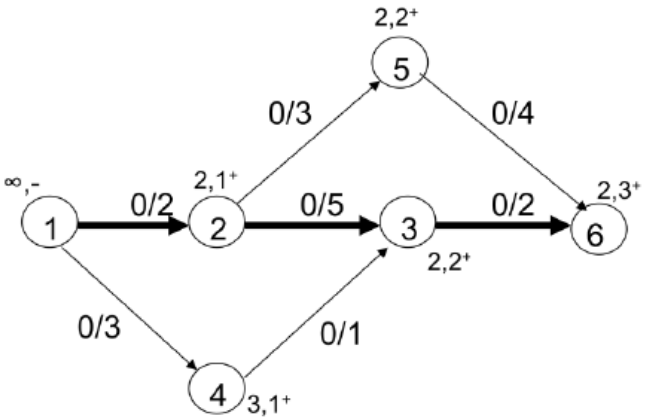
Input: A network with single source 1, single sink n, and
positive integer capacities u_{ij} on its edges (i, j)

Output: A maximum flow x

assign $x_{ij} = 0$ to every edge (i, j) in the network
label the source with $\infty, -$ and add the source to the empty queue Q
while not *Empty*(Q) **do**
    $i \leftarrow$ *Front*(Q); *Dequeue*(Q)
    for every edge from i to j **do** //forward edges
        if j is unlabeled
            $r_{ij} \leftarrow u_{ij} - x_{ij}$
            if $r_{ij} > 0$
                $l_j \leftarrow \min\{l_i, r_{ij}\}$; label j with l_j, i^+
                Enqueue(Q, j)
    for every edge from j to i **do** //backward edges
        if j is unlabeled
            if $x_{ji} > 0$
                $l_j \leftarrow \min\{l_i, x_{ji}\}$; label j with l_j, i^-
                Enqueue(Q, j)
    if the sink has been labeled
        //augment along the augmenting path found
        $j \leftarrow n$ //start at the sink and move backwards using second labels
        while $j \neq 1$ //the source hasn't been reached
            if the second label of vertex j is i^+
                $x_{ij} \leftarrow x_{ij} + l_n$
            else //the second label of vertex j is i^-
                $x_{ji} \leftarrow x_{ji} - l_n$
            $j \leftarrow i$
        erase all vertex labels except the ones of the source
        reinitialize Q with the source
return x //the current flow is maximum

Example: Shortest-Augmenting-Path Algorithm

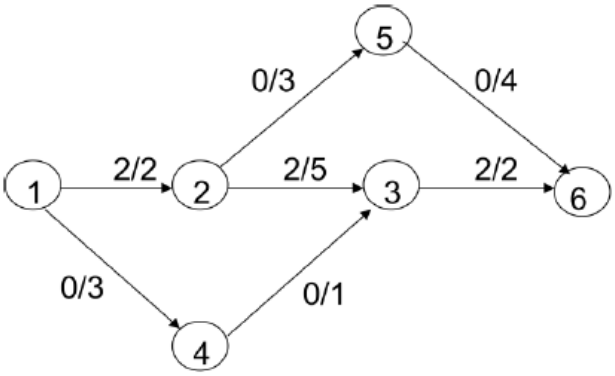


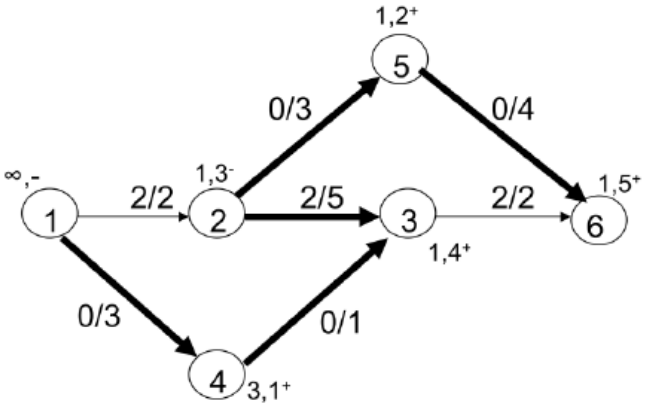


Queue: 1 2 4 3 5 6

↑ ↑ ↑ ↑

Augment the flow by 2 (the sink's first label) along the path 1→2→3→6

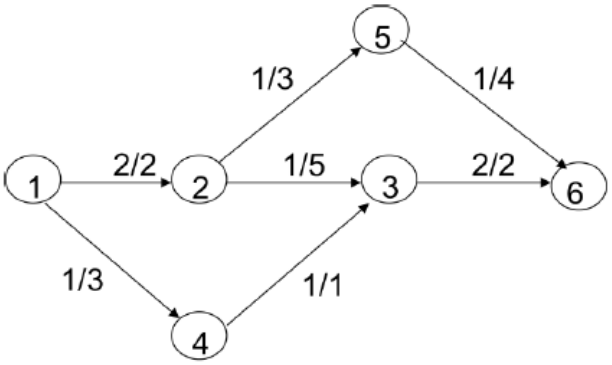


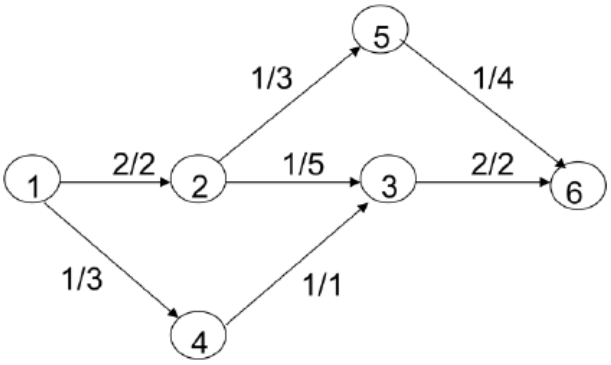


Queue: 1 4 3 2 5 6

↑ ↑ ↑ ↑ ↑

Augment the flow by 1 (the sink's first label) along the path 1→4→3←2→5→6





Queue: 1 4

↑ ↑

No augmenting path (the sink is unlabeled) the current flow is maximum

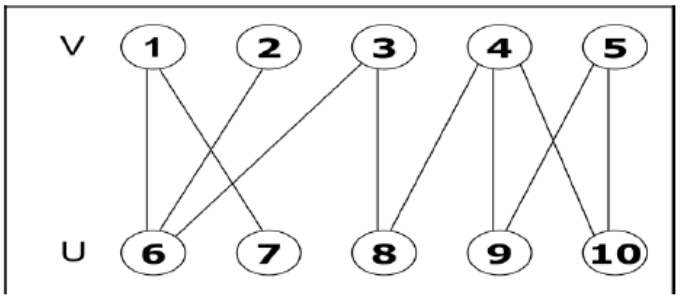
3. Write down the optimality condition and algorithmic implementation for finding M-augumenting paths in bipartite graphs? Or Illustrate the workings of the maximum matching algorithm on the following weighted trees. [N-15][M-15]

A matching in a graph is a subset of its edges with the property that no two edges share a vertex. A maximum matching—more precisely, a maximum cardinality matching—is a matching with the largest number of edges.

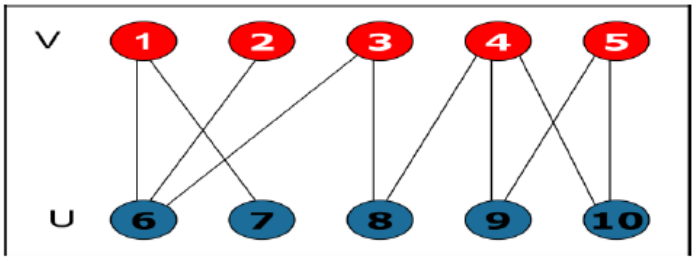
Case 1 (the queue's front vertex w is in V) If u is a free vertex adjacent to w, it is used as the other endpoint of an augmenting path; so the labeling stops and augmentation of the matching commences. The augmenting path in question is obtained by moving backward along the vertex labels (see below) to alternately add and delete its edges to and from the current matching. If u is not free and connected to w by an edge not in M, label u with w unless it has been already labeled.

Case 2 (the front vertex w is in U) In this case, w must be matched and we label its mate in V with w.

A graph is bipartite if and only if it does not have a cycle of an odd length.



A bipartite graph is *2-colorable*: the vertices can be colored in two colors so that every edge has its vertices colored differently



ALGORITHM Maximum Bipartite Matching(G)

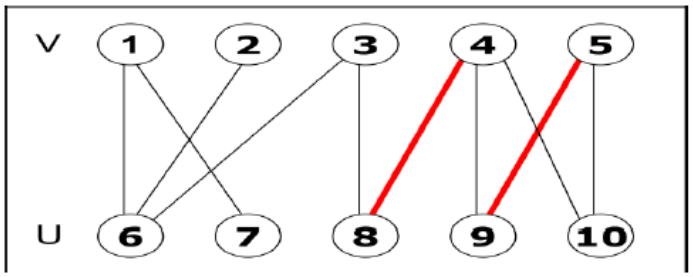
//Finds a maximum matching in a bipartite graph by a BFS-like traversal
//Input: A bipartite graph G = V, U, E
//Output: A maximum-cardinality matching M in the input graph
initialize set M of edges with some valid matching (e.g., the empty set)
initialize queue Q with all the free vertices in V (in any order)
while not Empty(Q) do
w←Front(Q); Dequeue(Q)
if w ∈ V
for every vertex u adjacent to w do
if u is free
//augment
M ←M ∪ (w, u)
v←w
while v is labeled do
u←vertex indicated by v's label; M ←M − (v, u)
v←vertex indicated by u's label; M ←M ∪ (v, u)
remove all vertex labels
reinitialize Q with all free vertices in V
break //exit the for loop
else //u is matched
if (w, u) ∈M and u is unlabeled
label u with w
Enqueue(Q, u)
else //w ∈ U (and matched)
label the mate v of w with w

Enqueue(Q, v)

return M //current matching is maximum

For Example,

A *matching* in a graph is a subset of its edges with the property that no two edges share a vertex

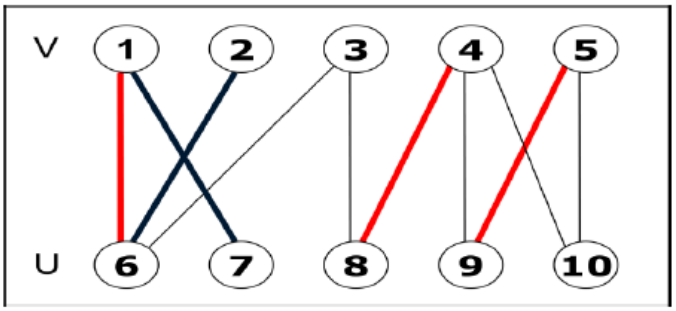


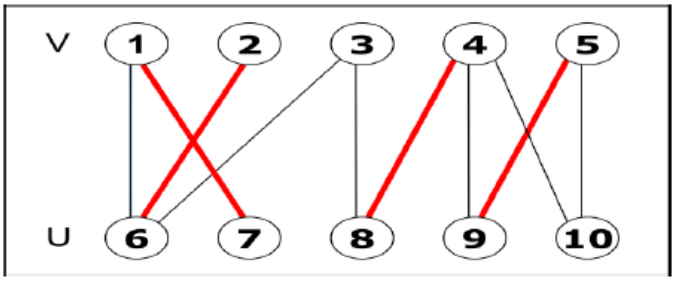
a matching in this graph M = {(4,8), (5,9)}

Augmenting Paths and Augmentation

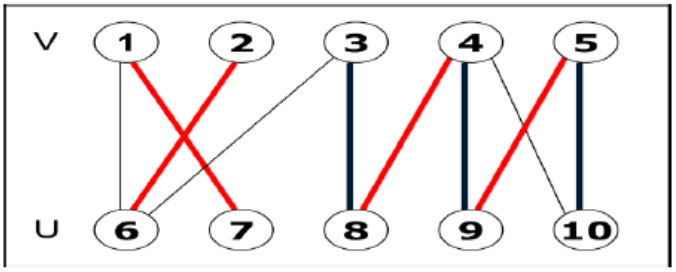
An *augmenting path* for a matching M is a path from a free vertex in V to a free vertex in U whose edges alternate between edges not in M and edges in M

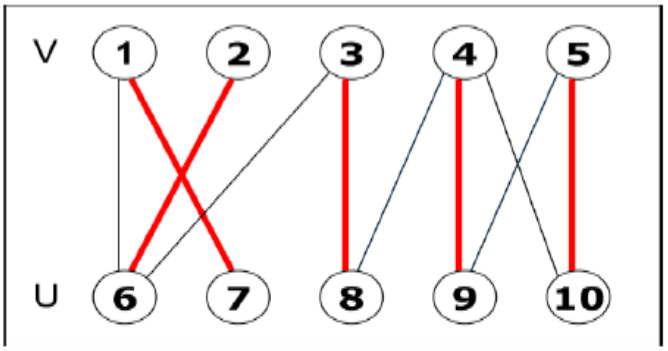
- The length of an augmenting path is always odd
- Adding to M the odd numbered path edges and deleting from it the even numbered path edges increases the matching size by 1 (*augmentation*)
- One-edge path between two free vertices is special case of augmenting path





Augmentation along path 2,6,1,7





Augmentation along 3, 8, 4, 9, 5, 10

Matching on the right is maximum (*perfect* matching).

4.Briefly describe on the stable marriage problem. [M-15]

Stable marriage algorithm

A marriage matching M is a set of n (m, w) pairs whose members are selected from disjoint n-element sets Y and X in a one-one fashion, i.e., each man m from Y is paired with exactly one woman w from X and vice versa

Consider an interesting version of bipartite matching called the stable marriage problem.

Consider a set Y = {m1, m2, . . . , mn } of n men and a set X = {w1, w2, . . . , wn } of n women. Each man has a preference list ordering the women as potential marriage partners with no ties allowed. Similarly, each woman has a preference list of the men, also with no ties.

Input: A set of n men and a set of n women along with rankings of the women by each man and rankings of the men by each woman with no ties allowed in the rankings

Output: A stable marriage matching

Step 0 Start with all the men and women being free.

Step 1 While there are free men, arbitrarily select one of them and do the following:

Proposal The selected free man m proposes to w, the next woman on his preference list (who is the highest-ranked woman who has not rejected him before).

Response If w is free, she accepts the proposal to be matched with m. If she is not free, she compares m with her current mate. If she prefersm to him, she accepts m's proposal, making her former mate free; otherwise, she simply rejects m's proposal, leaving m free.

Step 2 Return the set of n matched pairs.

For Example,

Instance of the Stable Marriage Problem

An instance of the stable marriage problem can be specified either by two sets of preference lists or by a ranking matrix, as in the example below.

men's preferences

	1st	2nd	3rd
Bob:	Lea	Ann	Sue
Jim:	Lea	Sue	Ann
Tom:	Sue	Lea	Ann

women's preferences

	1st	2nd	3rd
Ann:	Jim	Tom	Bob
Lea:	Tom	Bob	Jim
Sue:	Jim	Tom	Bob

ranking matrix

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

Example

Free men: Bob, Jim, Tom

	Ann	Lea	Sue
Bob	2,3	**1,2**	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

Bob proposed to Lea, Lea accepted Bob

Free men: Jim, Tom

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

Jim proposed to Lea, Lea rejected

Free men: Jim, Tom

	Ann	Lea	Sue
Bob	2,3	**1,2**	3,3
Jim	3,1	1,3	**2,1**
Tom	3,2	2,1	1,2

Jim proposed to Sue, Sue accepted

Free men: Tom

	Ann	Lea	Sue
Bob	2,3	**1,2**	3,3
Jim	3,1	1,3	**2,1**
Tom	3,2	2,1	1,2

Tom proposed to Sue, Sue rejected

Free men: Tom

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	**2,1**
Tom	3,2	**2,1**	1,2

Tom proposed to Lea, Lea replaced Bob with Tom

Free men: Bob

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

Bob proposed to Ann, Ann accepted

UNIT V COPING WITH THE LIMITATIONS OF ALGORITHM POWER

2 marks

1. What is meant by n-queen Problem?

The problem is to place n queens on an n-by-n chessboard so that no two queens attack each other by being in the same row or in the same column or in the same diagonal.

2. Define - Backtracking

Backtracking is used to solve problems with tree structures. Even problems seemingly remote to trees such as a walking a maze are actually trees when the decision \'back-left-straight-right\' is considered a node in a tree. The principle idea is to construct solutions one component at a time and evaluate such partially constructed candidates

3. Define -Subset-Sum Problem [M – 13]

This problem find a subset of a given set S={s1,s2,……,sn} of n positive integers whose sum is equal to a given positive integer d.

4. Define -Traveling Salesman Problem [M-15]

Given a complete undirected graph G=[V, E] that has nonnegative integer cost c[u, v] associated with each edge [u, v] in E, the problem is to find a hamiltonian cycle [tour] of G with minimum cost.

5. Define - Knapsack Problem [N-14] [M – 13]

Given n items of known weight wi and values vi=1,2,..,n and a knapsack of capacity w, find the most valuable subset of the items that fit in the knapsack.

6. Define - Branch and Bound

A counter-part of the backtracking search algorithm which, in the absence of a cost criteria, the algorithm traverses a spanning tree of the solution space using the breadth-first approach. That is, a queue is used, and the nodes are processed in first-in-first-out order.

7. What is a state space tree? [M-15] [N – 15]

The processing of backtracking is implemented by constructing a tree of choices being made. This is called the state-space tree. Its root represents a initial state before the search for a solution begins. The nodes of the first level in the tree represent the choices made for the first component of the solution, the nodes in the second level represent the choices for the second component and so on.

10. Define - Hamiltonian circuit. [M – 15]

Hamiltonian is defined as a cycle that passes through all the vertices of the graph exactly once. It is named after the Irish mathematician Sir William Rowan Hamilton [1805-1865].It is a sequence of n+1 adjacent vertices vi0, vi1,……, vin-1, vi0 where the first vertex of the sequence is same as the last one while all the other n-1 vertices are distinct.

11. What is a feasible solution and what is an optimal solution?

In optimization problems, a feasible solution is a point in the problem‘s search space that satisfies all the

problem's constraints, while an optimal solution is a feasible solution with the best value of the objective function.

12. Define - Integer Linear Programming [N – 15]

It is a Programming to find the minimum value of a linear function of several integer-valued variables subject to a finite set of constraints in the form of linear equalities and/or in equalities.

13. Define –NP[nondeterministic Polynomial] [M-14]

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

14. Define NP-Complete [M -15]

An NP-Complete problem is a problem in NP that is as difficult as any other problem in this class because any other problem in NP can be reduced to it in Polynomial time.

15. What are the strengths of backtracking and branch-and-bound? [M-15]

The strengths are as follows

- It is typically applied to difficult combinatorial problems for which no efficient algorithm for finding exact solution possibly exist
- It holds hope for solving some instances of nontrivial sizes in an acceptable amount of time

Even if it does not eliminate any elements of a problem's state space and ends up generating all its elements, it provides a specific technique for doing so, which can be of some value

16 marks

1. State the subset-sum problem and Complete state-space tree of the backtracking algorithm applied to the instance A={3, 5, 6, 7} and d=15 of the subset-sum problem.[M-16]

The ***subset-sum problem*** finds a subset of a given set $A = \{a1, \ldots, an\}$ of n positive integers whose sum is equal to a given positive integer d. For example, for $A = \{1, 2, 5, 6, 8\}$ and $d = 9$, there are two solutions: {1, 2, 6} and {1, 8}. Of course, some instances of this problem may have no solutions.

It is convenient to sort the set's elements in increasing order. So, we will assume that $a1 < a2 < \ldots < an$.

For Example,

A = {3, 5, 6, 7} and d = 15 of the subset-sum problem. The number inside a node is the sum of the elements already included in the subsets represented by the node. The inequality below a leaf indicates the reason for its termination.

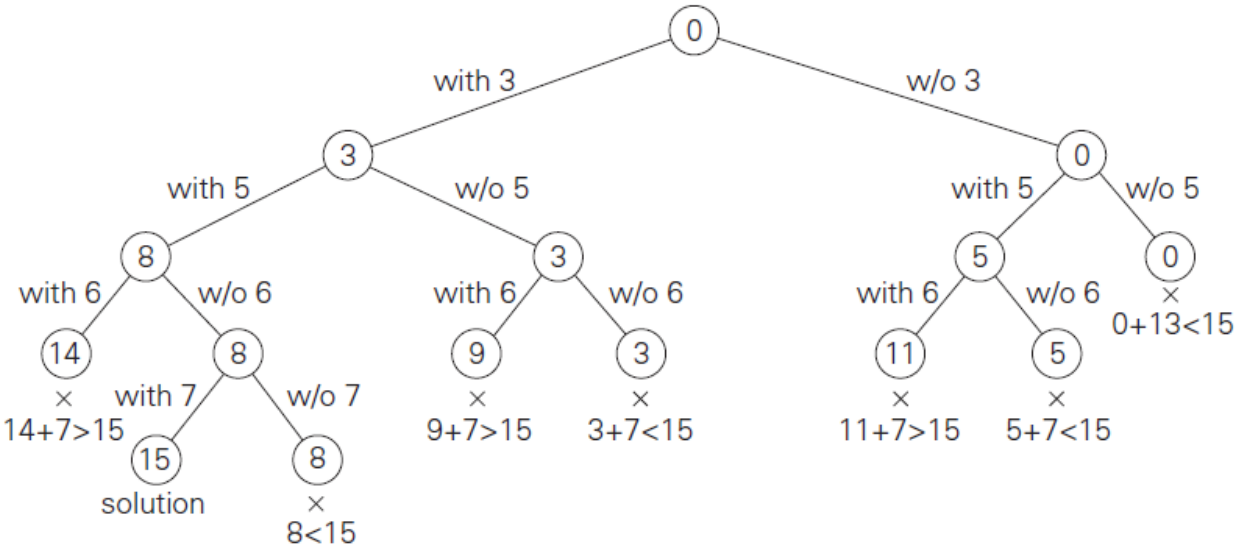


FIGURE Complete state-space tree of the backtracking algorithm applied to the instance

- The state-space tree can be constructed as a binary tree like that in the instance *A* = {3, 5, 6, 7} and *d* = 15.
- The root of the tree represents the starting point, with no decisions about the given elements made as yet.
- Its left and right children represent, respectively, inclusion and exclusion of *a*1 in a set being sought. Similarly, going to the left from a node of the first level corresponds to inclusion of *a*2 while going to the right corresponds to its exclusion, and so on.
- Thus, a path from the root to a node on the *i*th level of the tree indicates which of the first *I* numbers have been included in the subsets represented by that node. We record the value of *s*, the sum of these numbers, in the node.
- If *s* is equal to *d*, we have a solution to the problem. We can either report this result and stop or, if all the solutions need to be found, continue by backtracking to the node's parent.
- If *s* is not equal to *d,* we can terminate the node as non-promising if either of the following two inequalities holds:

  $s + a_{i+1} > d$ [the sum *s* is too large],

  $s + a\ nj$

  $=i+1\ j < d$ [the sum *s* is too small].

2. What is an approximation algorithm? Give example. Or Suggest an approximation algorithm for TSP. Assume that the cost functions satisfies the triangle inequality. Or Write a greedy algorithm for solving the knapsack optimization problem. Or Implement an algorithm for Knapsack problem using NP-Hard approach.[N-13][M-15][N-15]

A polynomial-time approximation algorithm is said to be a approximation algorithm, where c ≥ 1, if the accuracy ratio of the approximation it produces does not exceed c for any instance of the problem in question: r(sa) ≤ c.

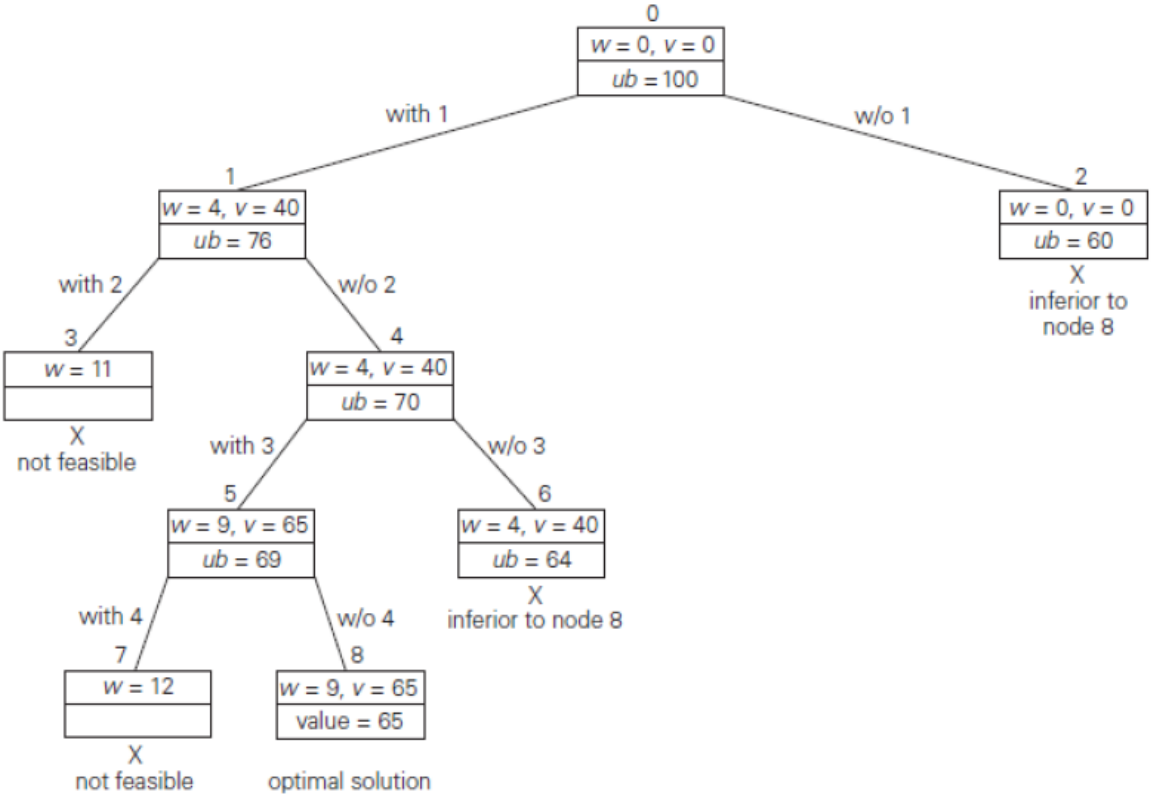
KNAPSACK PROBLEM

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

Item	Weight	value	value / weight	capacity
1	4	$40	10	$W = 10$
2	7	$42	6	
3	5	$25	5	
4	3	$12	4	
	w=19	v=119	$v_{i+1}/w_{i+1}=25$	

A simple way to compute the upper bound ub is to add to v, the total value of the items already selected, the product of the remaining capacity of the knapsack $W - w$ and the best per unit payoff among the remaining items, which is v_{i+1}/w_{i+1}:

$$ub = v + (W - w)(v_{i+1}/w_{i+1}).$$
$$= 0+(10-0)\ (10)$$
$$= 100$$



Greedy algorithm for the discrete knapsack problem

Step 1 Compute the value-to-weight ratios ri= vi/wi, i = 1, . . . , n, for the items given.

Step 2 Sort the items in non-increasing order of the ratios computed in Step 1.

Step 3 Repeat the following operation until no item is left in the sorted list: if the current item on the list fits into the knapsack, place it in the knapsack and proceed to the next item; otherwise, just proceed to the next item.

TRAVELING SALESMAN PROBLEM

Greedy Algorithms for the TSP

The simplest approximation algorithms for the traveling salesman problem are based on the greedy technique..

Nearest-neighbor algorithm

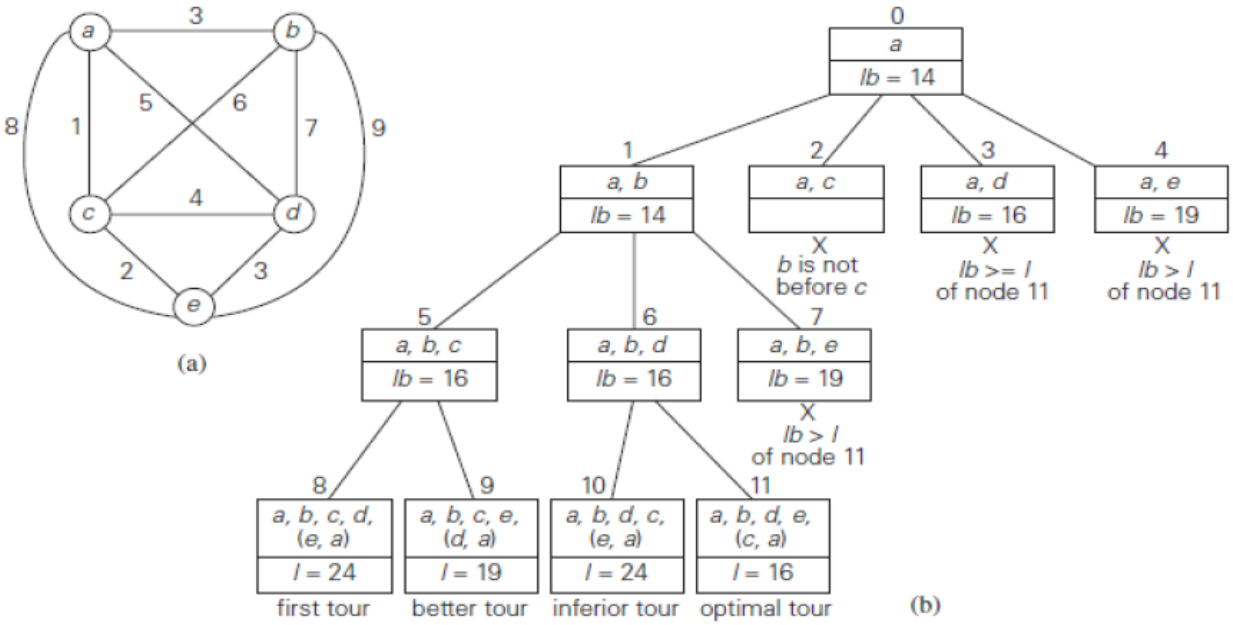
The following well-known greedy algorithm is based on the ***nearest-neighbor*** heuristic: always go next to the nearest unvisited city.

Step 1 Choose an arbitrary city as the start.

Step 2 Repeat the following operation until all the cities have been visited:go to the unvisited city nearest the one visited last (ties can be broken arbitrarily).

Step 3 Return to the starting city.

For Example,



3. Using an example prove that, satisfiability of Boolean formula in 3-conjunctive normal form is NP-Complete. Or Briefly explain NP-Hard and NP-Completeness with example. Or Write short notes on deterministic and non-deterministic algorithms. [M-14] [M-15] [N-15]

Class P: An algorithm solves a problem in polynomial time if its worst-case time efficiency belongs to *O(p(n))* where *p(n)* is a polynomial of the problem's input size *n.* (Note that since we are using big-oh notation here, problems solvable in, say, logarithmic time are solvable in polynomial time as well.) Problems that can be solved in polynomial time are called ***tractable***, and problems that cannot be solved in polynomial time are called ***intractable***.

- Class P is a class of decision problems that can be solved in polynomial time by (deterministic) algorithms. This class of problems is called polynomial. (Class P)
- Examples: Searching, Element uniqueness, primality test, graph acyclicity

Class NP: A nondeterministic algorithm is a two-stage procedure that takes as its input an instance I of a decision problem and does the following. Nondeterministic ("guessing") stage: An arbitrary string S is generated that can be thought of as a candidate solution to the given instance I (but may be complete gibberish as well).

- Class NP is the class of decision problems that can be solved by nondeterministic polynomial algorithms. This class of problems is called nondeterministic polynomial.
- Examples: TSP, AP, Graph coloring problem, partition problem, Hamiltonian circuit problem

Class- NP Complete: A decision problem D1 is said to be polynomially reducible to a decision problem D2, if there exists a function t that transforms instances of D1 to instances of D2 such that:

1. t maps all yes instances of D1 to yes instances of D2 and all no instances of D1 to no instances of D2
2. t is computable by a polynomial time algorithm

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

1. it belongs to class NP
2. every problem in NP is polynomially reducible to D

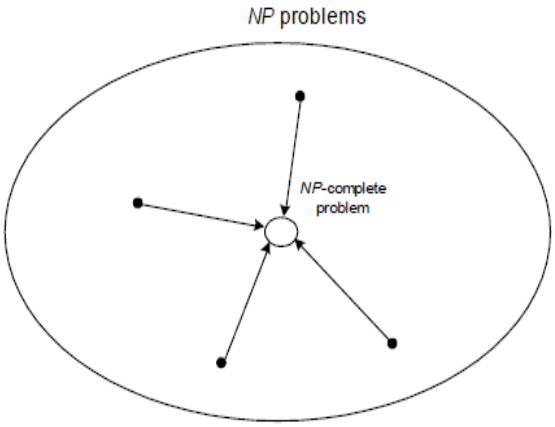


FIGURE 5.6 Polynomial-time reductions of *NP* problems to an *NP*-complete problem

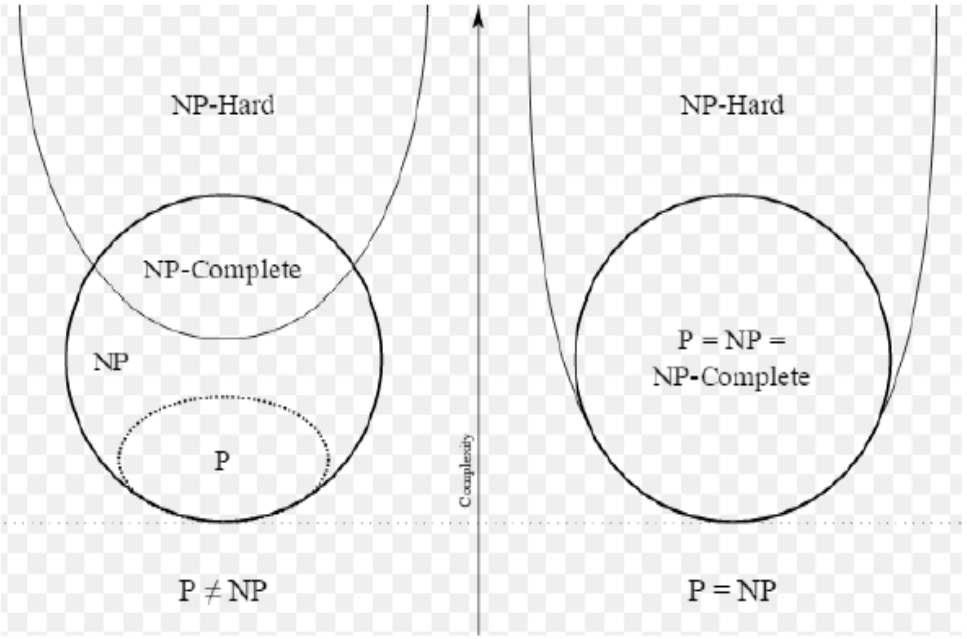


FIGURE 5.8 Relation among P, NP, NP-hard and NP Complete problems

4. Solve n-Queens problem. Or Explain 8-Queens problem with an algorithm. Explain why backtracking is the default procedure for solving problems. Or Explain 8 Queens problem with example.[M-14][N-13] [N-14]

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.

For *n* = 1, the problem has **a trivial solution**.

Q

For n = 2, it is easy to see that there is **no solution** to place 2 queens in 2 × 2 chessboard.

Q	

For *n* = 3, it is easy to see that there is **no solution** to place 3 queens in 3 × 3 chessboard.

	1	2	3	
1	Q			← queen 1
2			Q	← queen 2
3				

Or

	1	2	3	
1	Q			← queen 1
2				
3		Q		← queen 2

Or

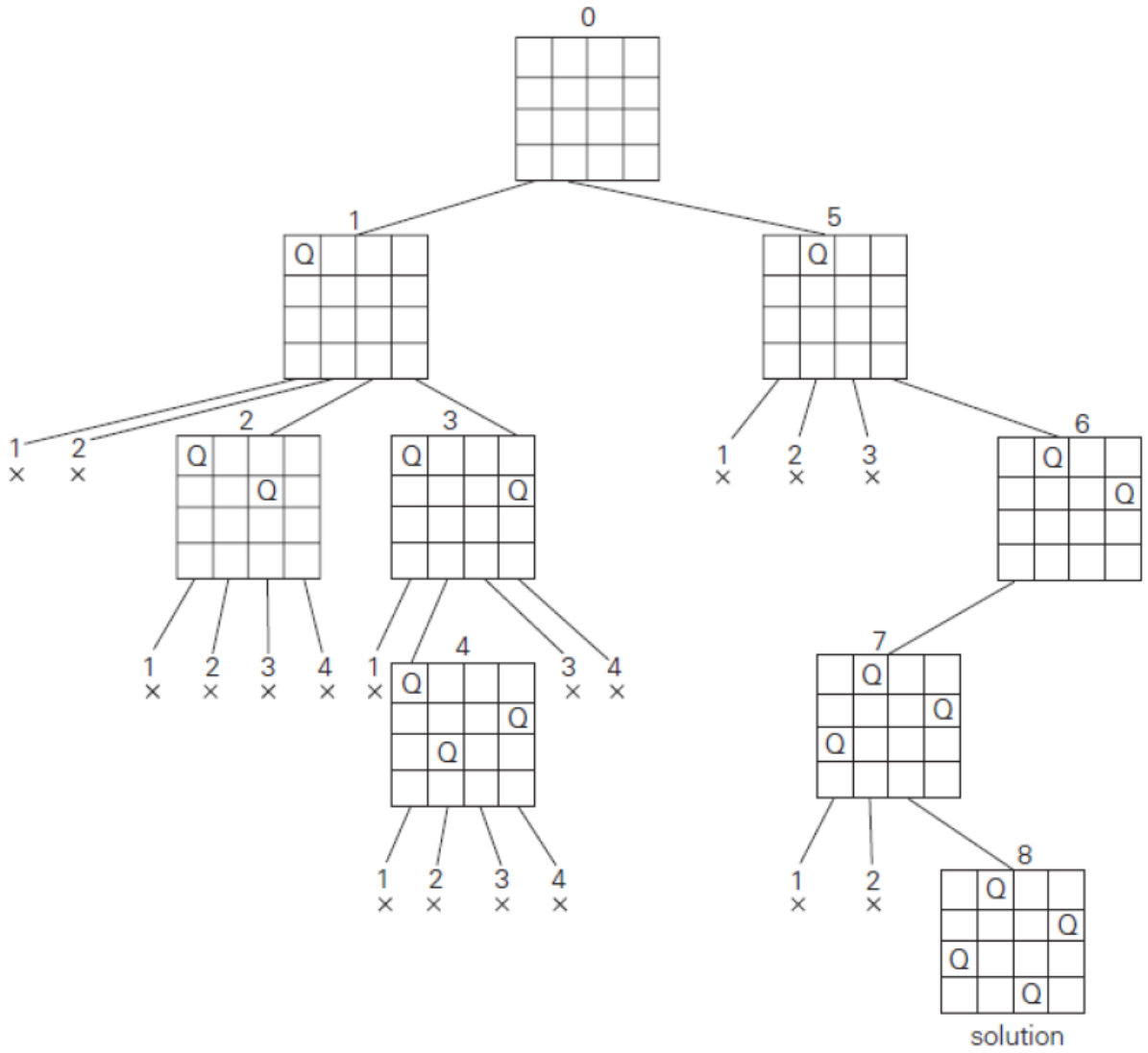
	1	2	3	
1		Q		← queen 1
2				
3	Q			← queen 2

For *n* = 4, There is **solution** to place 4 queens in 4 × 4 chessboard. the four-queens problem solved by the backtracking technique.

	1	2	3	4
1		Q		
2				Q
3	Q			
4			Q	

For *n* = 8, There is **solution** to place 8 queens in 8 × 8 chessboard.

	1	2	3	4	5	6	7	8
1				Q				
2						Q		
3								Q
4			Q					
5	Q							
6							Q	
7					Q			
8		Q						



State space tree for 4-queen problem. Similar to other queen problem also.