SUBJECT CODE : CS-6402

DESIGN AND ANALYSIS OF ALGORITHM

UNIT I   INTRODUCTION

Fundamentals of algorithmic problem solving – Important problem types – Fundamentals of the analysis of algorithm efficiency – analysis frame work –Asymptotic notations – Mathematical analysis for recursive and non-recursive algorithms.

PART A

1. What is an algorithm?

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. Why is the need of studying algorithms?

From a practical standpoint, a standard set of algorithms from different areas of computing must be known, in addition to be able to design them and analyze their efficiencies. From a theoretical standpoint the study of algorithms is the cornerstone of computer science.

3. What is the formula used in Euclid's algorithm for finding the greatest common divisor of two numbers?

Euclid's algorithm is based on repeatedly applying the equality

Gcd(m,n)=gcd(n,m mod n) until m mod n is equal to 0, since gcd(m,0)=m.

4. What are the three different algorithms used to find the gcd of two numbers?

The three algorithms used to find the gcd of two numbers are

- Euclid's algorithm
- Consecutive integer checking algorithm
- Middle school procedure

5. What are the fundamental steps involved in algorithmic problem solving?

The fundamental steps are

- Understanding the problem
- Ascertain the capabilities of computational device
- Choose between exact and approximate problem solving
- Decide on appropriate data structures
- Algorithm design techniques
- Methods for specifying the algorithm
- Proving an algorithms correctness
- Analyzing an algorithm
- Coding an algorithm

6. What is an algorithm design technique?
An algorithm design technique is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.

7. What is pseudocode?
A pseudocode is a mixture of a natural language and programming language constructs to specify an algorithm. A pseudocode is more precise than a natural language and its usage often yields more concise algorithm descriptions.

8. 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

9. 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

10. What are the classical geometric problems?
The two classic geometric problems are
- *The closest pair problem:* given n points in a plane find the closest pair among them
- *The convex hull problem:* find the smallest convex polygon that would include all the points of a given set.

11. What are the steps involved in the analysis framework?
The various steps are as follows
- Measuring the input's size
- Units for measuring running time
- Orders of growth
- Worst case, best case and average case efficiencies

12. What is the basic operation of an algorithm and how is it identified?
- The most important operation of the algorithm is called the basic operation of the algorithm, the operation that contributes the most to the total running time.
- It can be identified easily because it is usually the most time consuming operation in the algorithms innermost loop.

13. What is the running time of a program implementing the algorithm?

The running time T(n) is given by the following formula

$$T(n) \approx c_{op}C(n)$$

cop is the time of execution of an algorithm's basic operation on a particular computer and
C(n) is the number of times this operation needs to be executed for the particular algorithm.

14. What are exponential growth functions?

The functions 2n and n! are exponential growth functions, because these two functions grow so fast that their values become astronomically large even for rather smaller values of n.

15. 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.

16. 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.

17. What is average case efficiency?

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.

18. What is amortized efficiency?

In some situations a single operation can be expensive, but the total time for the entire sequence of n such operations is always significantly better that the worst case efficiency of that single operation multiplied by n. this is called amortized efficiency.

19. Define O-notation?

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

$$T (n) <=cg (n) \text{ for all } n >= n_0$$

20. Define Ω-notation?

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

21. Define θ-notation?

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

$$c_2g(n) <= t(n) <= c_1g(n) \text{ for all } n >= n0$$

22. Mention the useful property, which can be applied to the asymptotic notations and its use?

If $t_1(n)$ ε $O(g_1(n))$ and $t_2(n)$ ε $O(g_2(n))$ then $t_1(n)+t_2(n)$ ε max $\{g_1(n),g_2(n)\}$ this property is also true for Ω and θ notations. This property will be useful in analyzing algorithms that comprise of two consecutive executable parts.

23. What are the basic asymptotic efficiency classes?

The various basic efficiency classes are

- Constant : 1
- Logarithmic : log n
- Linear : n
- N-log-n : nlog n
- Quadratic : n2
- Cubic : n3
- Exponential : 2n
- Factorial : n!

24. What is algorithm visualization?

Algorithm visualization is a way to study algorithms. It is defined as the use of images to convey some useful information about algorithms. That information can be a visual illustration of algorithm's operation, of its performance on different kinds of inputs, or of its execution speed versus that of other algorithms for the same problem.

25. What are the two variations of algorithm visualization?

- The two principal variations of algorithm visualization‖ ._Static algorithm visualization_: It shows the
  algorithm's progress
- through a series of still images ._Dynamic algorithm visualization_: Algorithm animation shows a
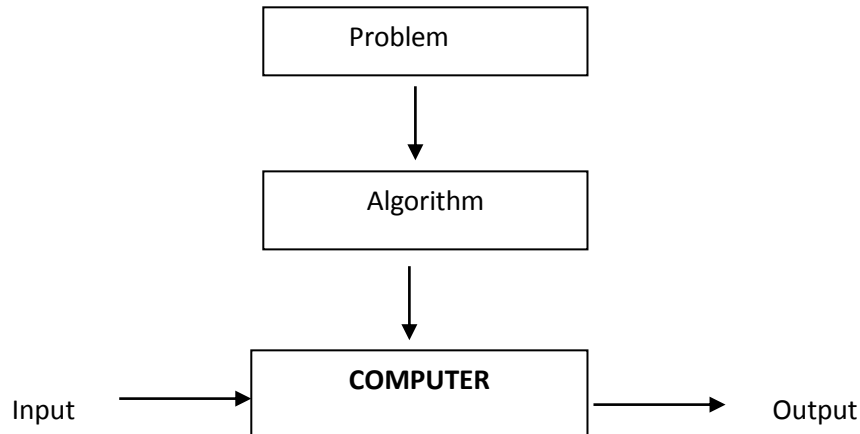- continuous movie like presentation of algorithms operations

26. What is order of growth?

Measuring the performance of an algorithm based on the input size n is called order of growth.

PART B

1. Explain about algorithm with suitable example (Notion of algorithm).

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

```
        ┌──────────────┐
        │   Problem    │
        └──────────────┘
               │
               ▼
        ┌──────────────┐
        │  Algorithm   │
        └──────────────┘
               │
               ▼
        ┌──────────────┐
 Input ─▶│  COMPUTER   │──▶ Output
        └──────────────┘
```

Algorithms – Computing the Greatest Common Divisor of Two Integers(gcd(m, n): the largest integer that divides both m and n.)

✓ **Euclid's algorithm:** gcd(m, n) = gcd(n, m mod n)

Step1: If n = 0, return the value of m as the answer and stop; otherwise, proceed to Step 2.

Step2: Divide m by n and assign the value of the remainder to r.

Step 3: Assign the value of n to m and the value of r to n. Go to Step 1.

Algorithm *Euclid(m, n)*

//Computes gcd(m, n) by Euclid's algorithm

//Input: Two nonnegative, not-both-zero integers m and n

//Output: Greatest common divisor of m and

n while n ≠ 0 do

   r ← m mod

   n m ← n

   n ← r

return m

About This algorithm

- Finiteness: how do we know that Euclid's algorithm actually comes to a stop?
- Definiteness: nonambiguity
- Effectiveness: effectively computable.

✓ **Consecutive Integer Algorithm**

Step1: Assign the value of min{m, n} to t.

Step2: Divide m by t. If the remainder of this division is 0, go to Step3;otherwise, go to Step 4.

Step3: Divide n by t. If the remainder of this division is 0, return the value of t as the answer and stop; otherwise, proceed to Step4.

Step4: Decrease the value of t by 1. Go to Step2.

About This algorithm
- Finiteness
- Definiteness
- Effectiveness

✓ **Middle-school procedure**

Step1: Find the prime factors of m.

Step2: Find the prime factors of n.

Step3: Identify all the common factors in the two prime expansions found in Step1 and Step2. (If p is a common factor occurring Pm and Pn times in m and n, respectively, it should be repeated in min{Pm, Pn} times.)

Step4: Compute the product of all the common factors and return it as the gcd of the numbers given.

2. **Write short note on Fundamentals of Algorithmic Problem Solving**

✓ Understanding the problem
- Asking questions, do a few examples by hand, think about special cases, etc.

✓ Deciding on
- Exact vs. approximate problem solving
- Appropriate data structure

✓ Design an algorithm

✓ Proving correctness

✓ Analyzing an algorithm
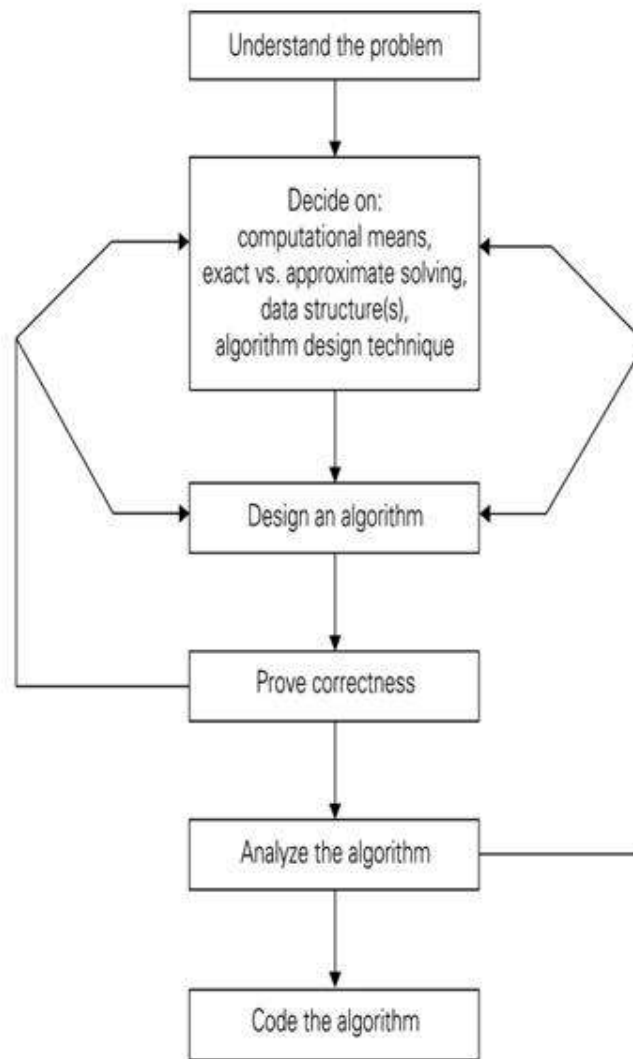
- Time efficiency : how fast the algorithm runs
- Space efficiency: how much extra memory the algorithm needs.
- ✓ Coding an algorithm

3. Discuss important problem types that you face during Algorithm Analysis.

- ✓ sorting
  - Rearrange the items of a given list in ascending order.
  - Input: A sequence of n numbers $<a_1, a_2, \ldots, a_n>$
  - Output: A reordering $<a'_1, a'_2, \ldots, a'_n>$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.
  - A specially chosen piece of information used to guide sorting. I.e., sort student records by names.

    Examples of sorting algorithms
  - Selection sort

- Bubble sort
- Insertion sort
- Merge sort
- Heap sort …

Evaluate sorting algorithm complexity: the number of key comparisons.
- Two properties
- Stability: A sorting algorithm is called stable if it preserves the relative order of any two equal elements in its input.
- In place: A sorting algorithm is in place if it does not require extra memory, except, possibly for a few memory units.

✓ searching
- Find a given value, called a search key, in a given set.
- Examples of searching algorithms
  ➢ Sequential searching
  ➢ Binary searching…

✓ string processing
- A string is a sequence of characters from an alphabet.
- Text strings: letters, numbers, and special characters.
- String matching: searching for a given word/pattern in a text.

✓ graph problems
- Informal definition
  ➢ A graph is a collection of points called vertices, some of which are connected by line segments called edges.
- Modeling real-life problems
- Modeling WWW
- communication networks
- Project scheduling …
  Examples of graph algorithms
- Graph traversal algorithms
- Shortest-path algorithms
- Topological sorting

✓ combinatorial problems
✓ geometric problems
✓ Numerical problems

4. **Discuss Fundamentals of the analysis of algorithm efficiency elaborately.**
   ✓ Algorithm's efficiency
   ✓ Three notations
   ✓ Analyze of efficiency of Mathematical Analysis of Recursive Algorithms
   ✓ Analyze of efficiency of Mathematical Analysis of non-Recursive Algorithms
   **Analysis of algorithms means to investigate an algorithm's efficiency with respect to resources: running time and memory space.**

- Time efficiency: how fast an algorithm runs.
- Space efficiency: the space an algorithm requires.
✓ Measuring an input's size
✓ Measuring running time
✓ Orders of growth (of the algorithm's efficiency function)
✓ Worst-base, best-case and average efficiency

■ **Measuring Input Sizes**
- Efficiency is defined as a function of input size.
- Input size depends on the problem.
- Example 1, what is the input size of the problem of sorting n numbers?
- Example 2, what is the input size of adding two n by n matrices?

- **Units for Measuring Running Time**
  ➢ Measure the running time using standard unit of time measurements, such as seconds, minutes?
  ➢ Depends on the speed of the computer.
  ➢ count the number of times each of an algorithm's operations is executed.
  ➢ Difficult and unnecessary
  ➢ count the number of times an algorithm's basic operation is executed.
  ➢ Basic operation: the most important operation of the algorithm, the operation contributing the most to the total running time.
  ➢ For example, the basic operation is usually the most time-consuming operation in the algorithm's innermost loop.

- **Orders of Growth**
  ➢ consider only the leading term of a formula
  ➢ Ignore the constant coefficient.

- **Worst-Case, Best-Case, and Average-Case Efficiency**
  ➢ Algorithm efficiency depends on the input size n
  ➢ For some algorithms efficiency depends on type of input.
    Example: Sequential Search
  ➢ *Problem:* Given a list of n elements and a search key K, find an element equal to K, if any.
  ➢ *Algorithm:* Scan the list and compare its successive elements with K until either a matching element is found (successful search) of the list is exhausted (unsuccessful search)

    Worst case Efficiency
  ➢ Efficiency (# of times the basic operation will be executed) for the worst case input of size n.
  ➢ The algorithm runs the longest among all possible inputs of size n.

    Best case
  ➢ Efficiency (# of times the basic operation will be executed) for the best case input of size n.

➢ The algorithm runs the fastest among all possible inputs of size n.

Average case:

➢ Efficiency (#of times the basic operation will be executed) for a typical/random input of size n.

➢ NOT the average of worst and best case
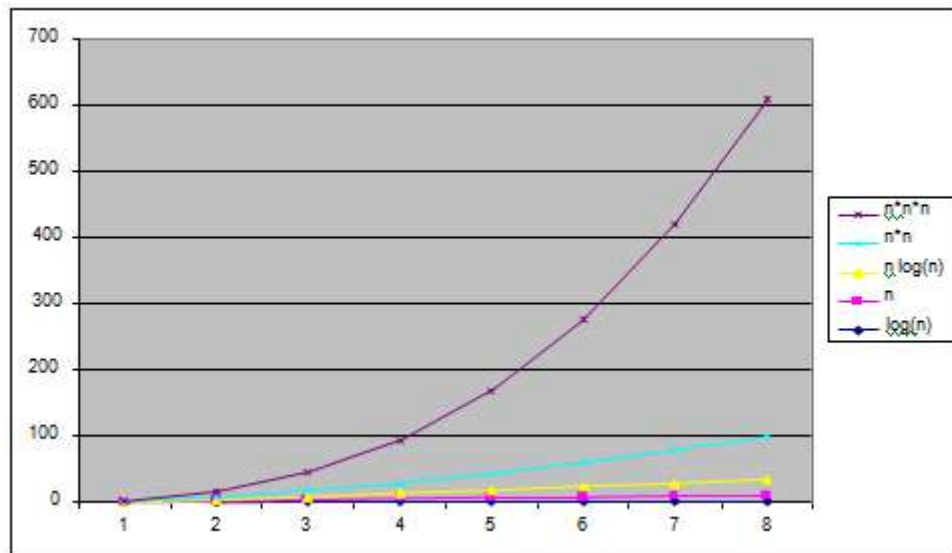
5. **Explain Asymptotic Notations**

Three notations used to compare orders of growth of an algorithm's basic operation count

a. O($g(n)$): class of functions $f(n)$ that grow *no faster* than $g(n)$

A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \in O(g(n))$, if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large *n*, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that $t(n) \leq cg(n)$ for all $n \geq n_0$
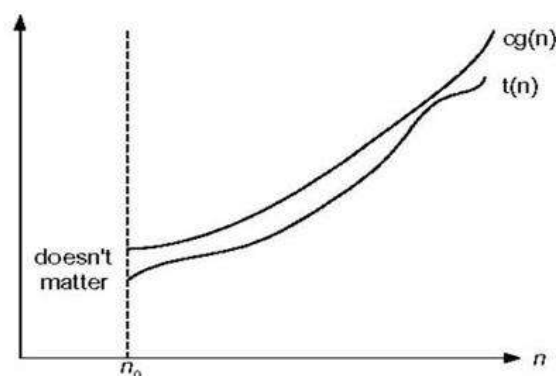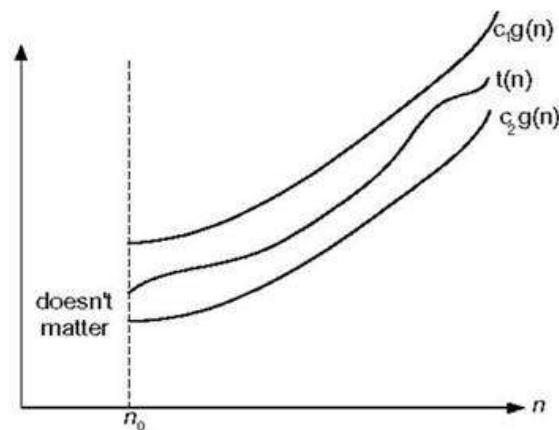
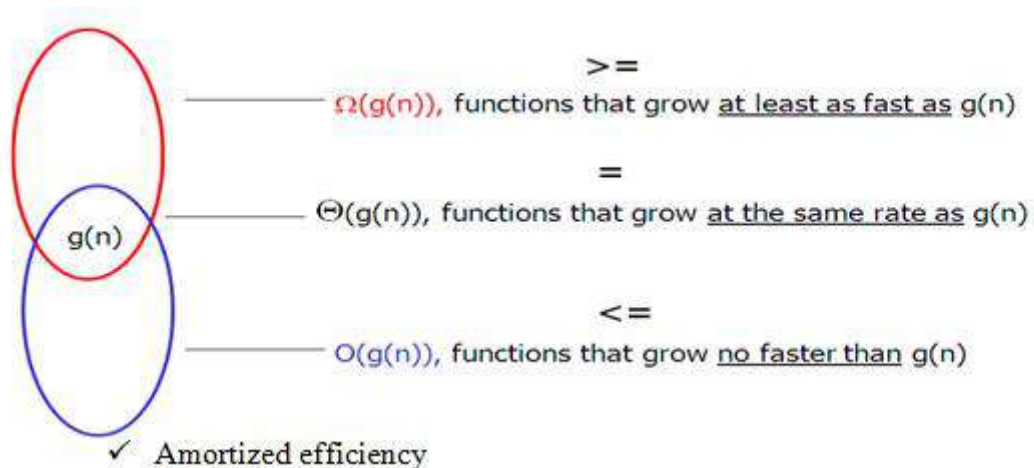a. $\Omega(g(n))$: class of functions $f(n)$ that grow _at least as fast_ as $g(n)$

A function $t(n)$ is said to be in $\Omega(g(n))$, denoted $t(n) \in \Omega(g(n))$, if $t(n)$ is bounded below by some constant multiple of $g(n)$ for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that $t(n) \geq cg(n)$ for all $n \geq n_0$
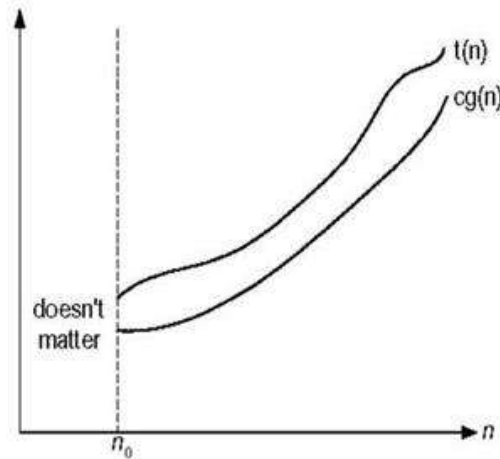
b. $\Theta(g(n))$: class of functions $f(n)$ that grow at same rate as $g(n)$

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

$\geq$

$\Omega(g(n))$, functions that grow <u>at least as fast as</u> g(n)

$=$

$\Theta(g(n))$, functions that grow <u>at the same rate as</u> g(n)

$\leq$

$O(g(n))$, functions that grow <u>no faster than</u> g(n)

✓ Amortized efficiency

fast			High time efficiency
	1	constant	
	log n	logarithmic	
	n	linear	
	n log n	n log n	
	n^2	quadratic	
	n^3	cubic	
	2^n	exponential	
slow	$n!$	factorial	low time efficiency

6. List out the Steps in Mathematical Analysis of non recursive Algorithms.

   ✓ Steps in mathematical analysis of nonrecursive algorithms:

- Decide on parameter n indicating input size
- Identify algorithm's basic operation
- Check whether the number of times the basic operation is executed depends only on the input size n. If it also depends on the type of input, investigate worst, average, and best case efficiency separately.
- Set up summation for C(n) reflecting the number of times the algorithm's basic operation is executed.

   ✓ Example: Finding the largest element in a given array

      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]

        for i ← 1 to n-1 do

            if A[i] > maxval

                maxval ← A[i]

        return maxval

7. List out the Steps in Mathematical Analysis of Recursive Algorithms.

    8. Decide on parameter n indicating *input size*

    9. Identify algorithm's *basic operation*

    10. Determine *worst*, *average*, and *best* case for input of size n

✓ Set up a recurrence relation and initial condition(s) for $C(n)$-the number of times the basic operation will be executed for an input of size n (alternatively count recursive calls).

✓ Solve the recurrence or estimate the order of magnitude of the

    solution F(n) = 1   if n = 0

            n * (n-1) * (n-2)… 3 * 2 * 1   if n > 0

✓ Recursive definition

        F(n) = 1   if n = 0

        n * F(n-1)   if n > 0

  Algorithm *F(n)*

        if *n*=0

                    return 1   //base case

        else

             return F (*n* -1) * *n*   //general case

    Example Recursive evaluation of *n* ! (2)

✓ Two Recurrences

The one for the factorial function value: F(n)

        F(n) = F(n − 1) * n for every n > 0

        F(0) = 1

The one for number of multiplications to compute n!, M(n)

    M(n) = M(n − 1) + 1 for every n > 0

    M(0) = 0

    M(n) ∈ Θ (n)

8. Explain in detail about Sequential search

    Sequential Search searches for the key value in the given set of items sequentially and returns the position of the key value else returns -1.

```
ALGORITHM   SequentialSearch(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: The index of the first element of A that matches K
    //          or −1 if there 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
```

Analysis:

For sequential search, best-case inputs are lists of size n with their first elements equal to a search key; accordingly,

$$C_{bw}(n) = 1.$$

Average Case Analysis:

The standard assumptions are that
(a) the probability of a successful search is equal top ($0 <=p<-=1$) and
(b) the probability of the first match occurring in the ith position of the list is the same for every i.

Under these assumptions- the average number of key comparisons $C_{avg}(n)$ *is found* as follows.

In the case of a successful search, the probability of the first match occurring in the i th position of the list is p / n for every i, and the number of comparisons made by the algorithm in such a situation is obviously i. In the case of an unsuccessful search, the number of comparisons is n with the probability of such a search being $(1- p)$. Therefore,

$$C_{avg}(n) = [1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}] + n \cdot (1 - p)$$

$$= \frac{p}{n}[1 + 2 + \cdots + i + \cdots + n] + n(1 - p)$$

$$= \frac{p}{n} \frac{n(n+1)}{2} + n(1 - p) = \frac{p(n+1)}{2} + n(1 - p).$$

For example, if $p = 1$ (i.e., the search must be successful), the average number of key comparisons made by sequential search is $(n + 1) /2$; i.e., the algorithm will inspect, on average, about half of the list's elements. If $p = 0$ (i.e., the search must be unsuccessful), the average number of key comparisons will be n because the algorithm will inspect all n elements on all such inputs.

9. **Explain in detail about Tower of Hanoi.**

In this puzzle, there are n disks of different sizes and three pegs. 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, if necessary. On1y one disk can be moved at a time, and it is forbidden to place a larger disk on top of a smaller one.
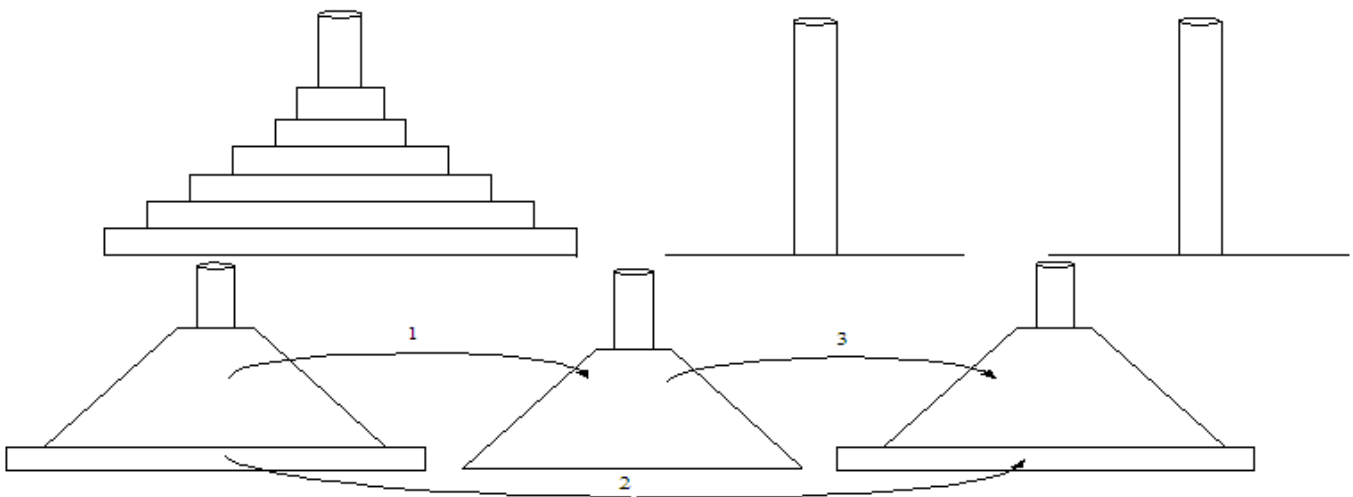
The general plan to the Tower of Hanoi problem.

The number of disks *n* is the obvious choice for the input's size indicator, and so is moving one disk as the algorithm's basic operation. Clearly, the number of moves *M(n)* depends on *n* only, and we get the following recurrence equation for it:

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

With the obvious initial condition $M(1) = 1$, the recurrence relation for the number of moves M(n) is:

$$M(n) = 2M(n-1) + 1 \text{ for } n> 1, M(1) = 1.$$

The total number of calls made by the Tower of Hanoi algorithm: *n-1*

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

$$= 2n-1$$

UNIT II

DIVIDE AND CONQUER METHOD AND GREEDY METHOD

Divide and conquer methodology – Merge sort – Quick sort – Binary search – Binary tree traversal – Multiplication of large integers – Strassen's matrix multiplication – Greedy method – Prim's algorithm – Kruskal's algorithm – Dijkstra's algorithm.

PART A

1. What is brute force algorithm?
A straightforward approach, usually based directly on the problem's statement and definitions of the concepts involved.

2. List the strength and weakness of brute force algorithm.

    Strengths
       a. wide applicability,
       b. simplicity
       c. yields reasonable algorithms for some important problems (e.g., matrix multiplication, sorting, searching, string matching)

    Weaknesses
       a. rarely yields efficient algorithms
       b. some brute-force algorithms are unacceptably slow not as constructive as some other design techniques

3. 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.

4. Give the general plan of exhaustive search.
  Method:
- generate a list of all potential solutions to the problem in a systematic manner
- evaluate potential solutions one by one, disqualifying infeasible ones and, for an optimization problem, keeping track of the best one found so far
- when search ends, announce the solution(s) found

5. Give the general plan for divide-and-conquer algorithms.
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-comquer strategy suggests splitting the inputs in

to'k' distinct susbsets, 1<k <n, yielding _k' subproblems. The subproblems must be solved, and then a method must be found to combine subsolutions into a solution of the whole. If the subproblems are still relatively large, then the divide-and conquer strategy can possibly be reapplied.

6. List the advantages of Divide and Conquer Algorithm

Solving difficult problems, Algorithm efficiency, Parallelism, Memory access, Round off control.

7. Define of feasibility

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.

8. Define Hamiltonian circuit.

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

9. State the Master theorem and its use.

If $f(n) \in \theta(n^d)$ where $d \geq 0$ in recurrence equation $T(n) = aT(n/b)+f(n)$, then

$$T(n) \in \begin{cases} \theta(n^d) & \text{if } a<b^d \\ \theta(n^d\log n) & \text{if } a=b^d \\ \theta(n^{\log_b a}) & \text{if } a>b^d \end{cases}$$

The efficiency analysis of many divide-and-conquer algorithms is greatly simplified by the use of Master theorem.

10.   What is the general divide-and-conquer recurrence relation?

An instance of size _n' can be divided into several instances of size n/b, with _a' of them needing to be solved. Assuming that size _n' is a power of _b', to simplify the analysis, the following recurrence for the running time is obtained:

$$T(n) = aT(n/b)+f(n)$$

Where f(n) is a function that accounts for the time spent on dividing the problem into smaller ones and on combining their solutions.

11.   Define mergesort.

Mergesort 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.

12.   List the Steps in Merge Sort

   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

13. List out Disadvantages of Divide and Conquer Algorithm
- Conceptual difficulty
- Recursion overhead
- Repeated subproblems

14. 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.

15. List out the Advantages in Quick Sort
- It is in-place since it uses only a small auxiliary stack.
- It requires only n log(n) time to sort n items.
- It has an extremely short inner loop
- This algorithm has been subjected to a thorough mathematical analysis, a very precise statement can be made about performance issues.

16. List out the Disadvantages in Quick Sort
- It is recursive. Especially if recursion is not available, the implementation is extremely complicated.
- It requires quadratic (i.e., n2) time in the worst-case.
- It is fragile i.e., a simple mistake in the implementation can go unnoticed and cause it to perform badly.

17. What is the difference between quicksort and mergesort?

Both quicksort and mergesort use the divide-and-conquer technique in which the given array is partitioned into subarrays and solved. The difference lies in the technique that the arrays are partitioned. For mergesort the arrays are partitioned according to their position and in quicksort they are partitioned according to the element values.

18. **What is binary search?**

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].

$$K$$
$$\uparrow$$
$$A[0]\ldots\ldots A[m-1]\ A[m]\ A[m+1]\ldots\ldots A[n-1]$$

search here if K<A[m]   search here if K>A[m]
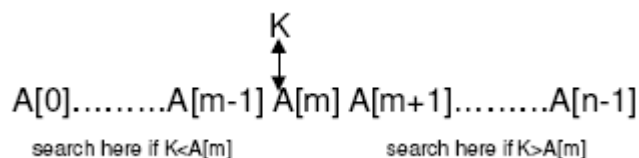
19. List out the 4 steps in Strassen's Method?

1. Divide the input matrices A and B into n/2 * n/2 submatrices, as in equation (1).
2. Using $\Theta(n2)$ scalar additions and subtractions, compute 14 n/2 * n/2 matrices A1, B1, A2, B2, …, A7, B7.
3. Recursively compute the seven matrix products Pi =AiBi for i =1, 2, 7.

4. Compute the desired submatrices r, s, t, u of the result matrix C by adding and/or subtracting various combinations of the Pi matrices, using only $\Theta(n2)$ scalar additions and subtractions.

PART B

1. **Explain Divide And Conquer Method**
✓ The most well known algorithm design strategy is Divide and Conquer Method. It
  • Divide the problem into two or more smaller subproblems.
  • Conquer the subproblems by solving them recursively.
  • Combine the solutions to the subproblems into the solutions for the original problem.
✓ Divide and Conquer Examples
  • Sorting: mergesort and quicksort
  • Tree traversals

• Binary search
• Matrix multiplication-Strassen's algorithm
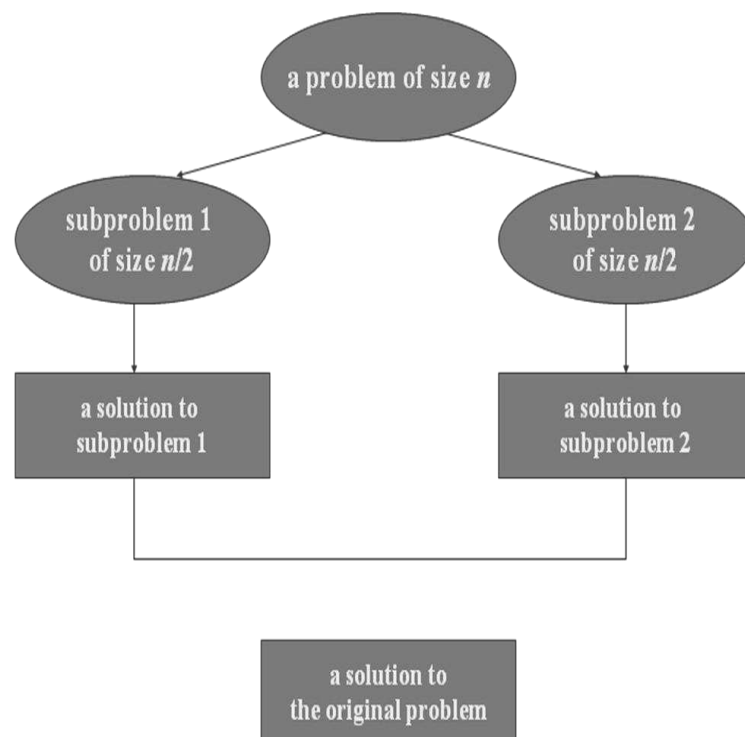
2. Explain Merge Sort with suitable example.

✓ Merge sort definition**.**
   Mergesort 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.

✓ Steps in Merge Sort
   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

 Algorithm for merge sort.

✓ ALGORITHM *Mergesort*(A[0..n-1])
   //Sorts an 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)

 Algorithm to merge two sorted arrays into one.

✓ 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 & 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]

3. Discuss Quick Sort

✓ Quick Sort definition

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.

✓ Quick Sort and divide and conquer

- Divide: Partition array A[l..r] into 2 subarrays, A[l..s-1] and A[s+1..r] such that each element of the first array is ≤A[s] and each element of the second array is ≥ A[s]. (Computing the index of s is part of partition.)
- Implication: A[s] will be in its final position in the sorted array.
- Conquer: Sort the two subarrays A[l..s-1] and A[s+1..r] by recursive calls to quicksort
- Combine: No work is needed, because A[s] is already in its correct place after the partition is done, and the two subarrays have been sorted.

✓ Steps in Quicksort

- Select a pivot w.r.t. whose value we are going to divide the sublist. (e.g., p = A[l])
- Rearrange the list so that it starts with the pivot followed by a ≤ sublist (a sublist whose elements are all smaller than or equal to the pivot) and a ≥ sublist (a sublist whose elements are all greater than or equal to the pivot) Exchange the pivot with the last element in the first sublist(i.e., ≤ sublist) – the pivot is now in its final position

Sort the two sublists recursively using quicksort

$$A[i] \leq p \qquad A[i] \geq p$$
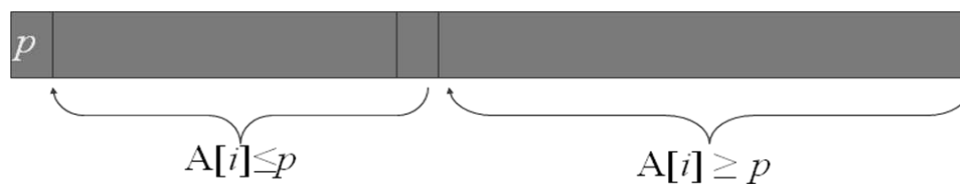
✓ The Quicksort Algorithm

ALGORITHM Quicksort(A[l..r])
//Sorts a subarray by quicksort
//Input: A subarray A[l..r] of A[0..n-1],defined by its left and right indices l and r
//Output: The 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 ← A[l]
i ← l; j ← r + 1;
Repeat
   repeat i ← i + 1 until A[i]>=p //left-right scan
   repeat j ← j – 1 until A[j] <= p//right-left scan
   if (i < j)   //need to continue with the scan
       swap(A[i], a[j])
until i >= j   //no need to scan
swap(A[l], A[j])
   return j

✓ Advantages in Quick Sort
• It is in-place since it uses only a small auxiliary stack.
• It requires only n log(n) time to sort n items.
• It has an extremely short inner loop
• This algorithm has been subjected to a thorough mathematical analysis, a very precise statement can be made about performance issues.

✓ Disadvantages in Quick Sort
• It is recursive. Especially if recursion is not available, the implementation is extremely complicated.
• It requires quadratic (i.e., n2) time in the worst-case.
• It is fragile i.e., a simple mistake in the implementation can go unnoticed and cause it to perform badly.

✓ Efficiency of Quicksort
Based on whether the partitioning is balanced.
Best case: split in the middle — $\Theta(n \log n)$
$C(n) = 2C(n/2) + \Theta(n)$ //2 subproblems of size n/2 each
Worst case: sorted array! — $\Theta(n^2)$
$C(n) = C(n-1) + n+1$ //2 subproblems of size 0 and n-1 respectively
Average case: random arrays — $\Theta(n \log n)$

4. Explain Binary Search.

✓ **Binary Search –Iterative Algorithm**

ALGORITHM
BinarySearch(A[0..n-1], K)
//Implements nonrecursive binary search
//Input: An array A[0..n-1] sorted in ascending order 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                    //l and r crosses over→ can't find K.
    m ←⌊(l + r) / 2⌋
    if K = A[m] return m          //the key is found
    else
        if K < A[m] r← m − 1//the key is on the left half of the array
        else    l ← m + 1      // the key is on the right half of the array
return -1
```

✓ **Binary Search – a Recursive Algorithm**

ALGORITHM BinarySearchRecur(A[0..n-1], l, r, K)

```
 if l > r              //base case 1: l and r cross over→ can't find K
      return −1
  else
      m ← ⌊(l + r) / 2⌋
      if K = A[m]            //base case 2: K is
              found return m
      else                   //general case: divide the problem.
              if K < A[m]    //the key is on the left half of the
                     array return BinarySearchRecur(A[0..n-1],
                     l, m-1, K)
              else           //the key is on the left half of the
                     array return BinarySearchRecur(A[0..n-1],
                     m+1, r, K)
```

✓ **Binary Search – Efficiency**
  • Recurrence
    relation $C(n) =$
    $C(\lfloor n / 2 \rfloor) + 2$
  • Efficiency
    $C(n) \in \Theta(\log n)$

5. **Explain Strassen's Algorithm**
    ✓ **Multiplication of Large Integers**
        o Multiplication of two n-digit integers.
        o Multiplication of a m-digit integer and a n-digit integer (where n > m) can be
          modeled as the multiplication of 2 n-digit integers (by padding n − m 0s before the
          first digit of the m-digit integer)

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

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

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

o **Brute-force algorithm**

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

$$= \begin{Bmatrix} a_{00}*b_{00}+a_{01}*b_{10} & a_{00}*b_{01}+a_{01}*b_{11} \\ a_{10}*b_{00}+a_{11}*b_{10} & a_{10}*b_{01}+a_{11}*b_{11} \end{Bmatrix}$$

- o 8 multiplications
- o 4 additions
- o Efficiencyclass:Θ (n3)
- ✓ Strassen'sAlgorithm (two 2x2 matrices)

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

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

- Efficiency of Strassen's Algorithm
  - o If n is not a power of 2, matrices can be padded with rows and columns with zeros
  - o Number of multiplications
  - o Number of additions
  - o Other algorithms have improved this result, but are even more complex

6. **Explain in detail about Travelling Salesman Problem using exhaustive search.**

   Given *n* cities with known distances between each pair, find the shortest tour that passes through all the cities exactly once before returning to the starting city

   Alternatively: Find shortest *Hamiltonian circuit* in a weighted connected graph.

   Example :
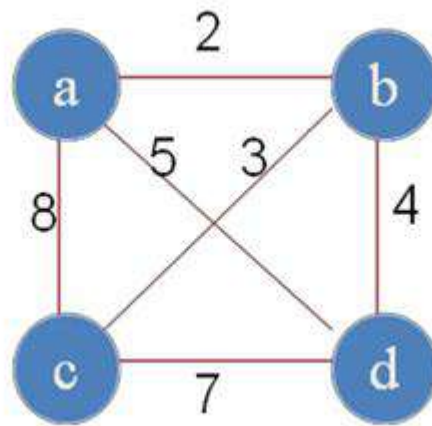
Tour	Cost
a→b→c→d→a	2+3+7+5 = 17
a→b→d→c→a	2+4+7+8 = 21
a→c→b→d→a	8+3+4+5 = 20
a→c→d→b→a	8+7+4+2 = 21
a→d→b→c→a	5+4+3+8 = 20
a→d→c→b→a	5+7+3+2 = 17

Efficiency: $\Theta((n-1)!)$

7. Explain in detail about knapsack problem.

Given *n* items:

weights: $w_1\ w_2\ …\ w_n$

values: $\quad v_1\quad v_2\ …$

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

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

Efficiency: $\Theta(2^n)$

8. Explain in detail about closest pair problem.

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   BruteForceClosestPoints(P)
    //Input: A list P of n (n ≥ 2) points P₁ = (x₁, y₁),..., Pₙ = (xₙ, yₙ)
    //Output: Indices index1 and index2 of the closest pair of points
    dmin ← ∞
    for i ← 1 to n − 1 do
        for j ← i + 1 to n do
            d ← sqrt((xᵢ − xⱼ)² + (yᵢ − yⱼ)²) //sqrt is the square root function
            if d < dmin
                dmin ← d; index1 ← i; index2 ← j
    return index1, index2
```

Efficiency: $\Theta(n^2)$ **multiplications (or sqrt)**

UNIT – III

DYNAMIC PROGRAMMING AND GREEDY TECHNIQUE

Computing a Binomial Coefficient – Warshall"s and Floyd" algorithm – Optimal Binary Search Trees – Knapsack Problem and Memory functions. Greedy Technique– Prim"s algorithm- Kruskal's Algorithm- Dijkstra's Algorithm- Huffman Trees.

PART A

1. Define dynamic programming.

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

Dynamic programming is a technique for solving problems with overlapping subproblems. These sub problems arise from a recurrence relating a solution to a given problem with solutions to its smaller sub problems only once and recording the results in a table from which the solution to the original problem is obtained. It was invented by a prominent U.S Mathematician, Richard Bellman in the 1950s.\

2. What are the features of dynamic programming?
- Optimal solutions to sub problems are retained so as to avoid recomputing their values.
- Decision sequences containing subsequences that are sub optimal are not considered.
- It definitely gives the optimal solution always.

3. What are the drawbacks of dynamic programming?
- Time and space requirements are high, since storage is needed for all level.
- Optimality should be checked at all levels.

4. Write the general procedure of dynamic programming.

The development of dynamic programming algorithm can be broken into a sequence of 4 steps.
- Characterize the structure of an optimal solution.
- Recursively define the value of the optimal solution.
- Compute the value of an optimal solution in the bottom-up fashion.
- Construct an optimal solution from the computed information.

5. Define principle of optimality.

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.

6. Write the difference between the Greedy method and Dynamic programming.
- Greedy method
1. Only one sequence of decision is generated.
2. It does not guarantee to give an optimal solution always.

- Dynamic programming
1. Many number of decisions are generated.
2. It definitely gives an optimal solution always.

7. 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.*

8. Write any two characteristics of Greedy Algorithm?

- To solve a problem in an optimal way construct the solution from given set of candidates.As the algorithm proceeds, two other sets get accumulated among this one set contains the candidates that have been already considered and chosen while the other set contains the candidates that have been considered but rejected.

9. What is the Greedy choice property?

- The first component is greedy choice property (i.e.) a globally optimal solution can arrive at by making a locally optimal choice.
- The choice made by greedy algorithm depends on choices made so far but it cannot depend on any future choices or on solution to the sub problem.
- It progresses in top down fashion.

10. What is greedy method?

Greedy method is the most important design technique, which makes a choice that looks best at that moment. A given _n' inputs are required us to obtain a subset that satisfies some constraints that is the feasible solution. A greedy method suggests that one can device an algorithm that works in stages considering one input at a time.

11. What are the steps required to develop a greedy algorithm?

- Determine the optimal substructure of the problem.
- Develop a recursive solution.
- Prove that at any stage of recursion one of the optimal choices is greedy choice. Thus it is always safe to make greedy choice.
- Show that all but one of the sub problems induced by having made the greedy choice are empty.
- Develop a recursive algorithm and convert into iterative algorithm.

12. 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

13. What are the labels in Prim's algorithm used for?

Prim's algorithm makes it necessary to provide each vertex not in the current tree with the information about the shortest edge connecting the vertex to a tree vertex. The information is provided by attaching two labels to a vertex.

- The name of the nearest tree vertex.
- The length of the corresponding edge

14. How are the vertices not in the tree split into?

The vertices that are not in the tree are split into two sets

- Fringe : It contains the vertices that are not in the tree but are adjacent to atleast one tree vertex.
- Unseen : All other vertices of the graph are called unseen because they are yet to be affected by the algorithm.

15. What are the operations to be done after identifying a vertex u* to be added to the tree?

After identifying a vertex u* to be added to the tree, the following two operations need to be performed

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

16. What is the use of Dijksra's algorithm?

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.

17. Define Spanning tree.

Spanning tree of a connected graph *G*: a connected acyclic subgraph of *G* that includes all of *G*'s
Vertices

18. What is minimum spanning tree.

Minimum spanning tree of a weighted, connected graph *G*: a spanning tree of *G* of the minimum total weight

PART B

1. **Write Short note on Dijkstra's Algorithm Shortest Paths – Dijkstra's Algorithm Shortest Path Problems**

o All pair shortest paths (Floy's algorithm)
o Single Source Shortest Paths Problem (Dijkstra's algorithm): Given a weighted graph G, find the

shortest paths from a source vertex s to each of the other vertices.
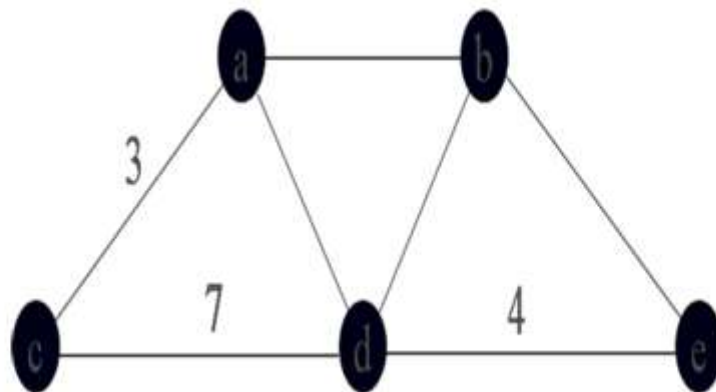
o **Prim's and Dijkstra's Algorithms**
   o Generate different kinds of spanning trees
           o Prim's: a minimum spanning tree.
                   ▪ Dijkstra's : a spanning tree rooted at a given source s, such that the

                     distance from s to every other vertex is the shortest.
           o Different greedy strategies
                   ▪ Prims': Always choose the closest (to the tree) vertex in the priority

                     queue Q to add to the expanding tree V_T.
                   ▪ Dijkstra's : Always choose the closest (to the source) vertex in the

                     priority queue Q to add to the expanding tree V_T.
           o Different labels for each vertex
                   ▪ Prims': parent vertex and the distance from the tree to the vertex.
                   ▪ Dijkstra's : parent vertex and the distance from the source to the vertex.

2. **Explain Kruskal's Algorithm**
   a. Greedy Algorithm for MST: Kruskal
                   o Edges are initially sorted by increasing weight

       i. Start with an empty forest
      ii. ―grow‖ MST one edge at a time
              1. intermediate stages usually have forest of trees (not connected)
     iii. at each stage add minimum weight edge among those not yet used that does not

create a cycle
1. at each stage the edge may:
2. expand an existing tree
3. combine two existing trees into a single tree
4. create a new tree

iv. need efficient way of detecting/avoiding cycles
✓ algorithm stops when all vertices are included

ALGORITHM Kruscal(G)
//Input: A weighted connected graph $G = <V, E>$
//Output: E_T, the set of edges composing a minimum spanning tree of G.
Sort E in nondecreasing order of the edge weights
$w(e_{i1}) <= \ldots <= w(e_{i|E|})$
$E_T \leftarrow \varnothing;\ ecounter \leftarrow 0$ //initialize the set of tree edges and its size
$k \leftarrow 0$
while *encounter < |V| - 1* do
    $k \leftarrow k + 1$
    if E_T U {e_{ik}} is acyclic
        $E_T \leftarrow E_T$ U {e_{ik}} ; *ecounter $\leftarrow$ ecounter + 1*
return E_T

3. **Discuss Prim's Algorithm**
✓ Minimum Spanning Tree (MST)
   o Spanning tree of a connected graph G: a connected acyclic subgraph (tree) of G that includes all of G's vertices.
   o Minimum Spanning Tree of a weighted, connected graph G: a spanning tree of G of minimum total weight.
   o Example:
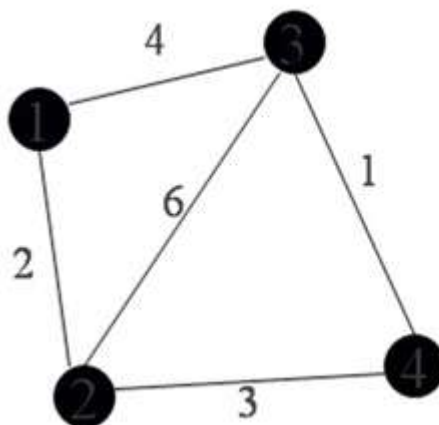
✓ Prim's MST algorithm
✓ Start with a tree , T_0 ,consisting of one vertex
✓ ―Grow‖ tree one vertex/edge at a time
   o Construct a series of expanding subtrees T_1, T_2, … T_{n-1} .At each stage construct T_{i+1} from T_i by

   ▪ adding the minimum weight edge connecting a vertex in tree (T_i) to one not yet in tree
      • choose from ―fringe‖ edges

✓ (this is the ―greedy‖ step!) Or (another way to understand it)
✓ expanding each tree (Ti) in a greedy manner by attaching to it the nearest vertex not in that tree. (a vertex not in the tree connected to a vertex in the tree by an edge of the smallest weight)
✓ Algorithm stops when all vertices are included
✓ **Algorithm**:
  ALGORITHM
  Prim(G)
  //Prim‘s algorithm for constructing a minimum spanning tree
  //Input A weighted connected graph G= V, E
  //Output E_T, the set of edges composing a minimum spanning tree of
  G V_T {v0}
  E_T← F
  for i← 1 to |V|-1 do
  Find the minimum-weight edge e*=(v*,u*) among all the edges (v,u) such that v is in V_T and u is in V-V_T
  V_T←V_T U
  {u*} E_T E_T U
  {e*}
  return E_T

4. **Write short note on Greedy Method**
✓ A greedy algorithm makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.
  o The choice made at each step must be:
  o Feasible
    ▪ Satisfy the problem‘s constraints
  o locally optimal
    ▪ Be the best local choice among all feasible choices
  o Irrevocable
    ▪ Once made, the choice can‘t be changed on subsequent steps.
✓ Applications of the Greedy Strategy
  o Optimal solutions:
    ▪ change making
    ▪ Minimum Spanning Tree (MST)
    ▪ Single-source shortest paths
    ▪ Huffman codes
  o Approximations:
    ▪ Traveling Salesman Problem (TSP)
    ▪ Knapsack problem
    ▪ other optimization problems

5. **What does dynamic programming has in common with divide-and-Conquer?**
  a. **Dynamic Programming**
  • Dynamic Programming is a general algorithm design technique. ―Programming‖ here means ―planning‖.
  • Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems
    Main idea:

✓ solve several smaller (overlapping) subproblems
✓ record solutions in a table so that each subproblem is only solved once
✓ final state of the table will be (or contain) solution

Dynamic programming vs. divide-and-conquer

✓ partition a problem into overlapping subproblems and independent ones
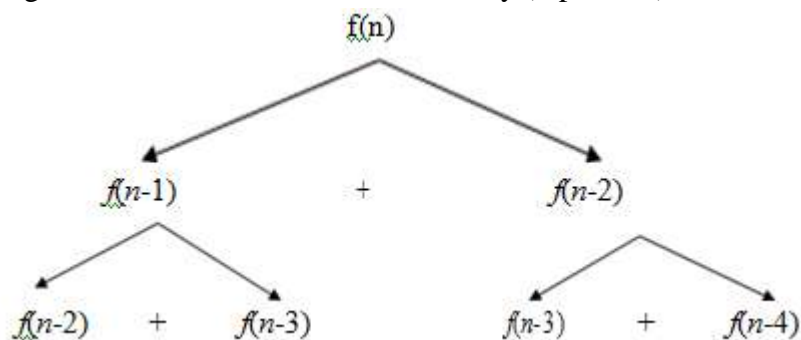✓ store and not store solutions to subproblems
✓

Example: Fibonacci numbers
Recall definition of Fibonacci
numbers: $f(0) = 0$
$f(1) = 1$
$f(n) = f(n-1) + f(n-2)$

- Computing the nth Fibonacci number recursively (top-down):

Computing the nth fibonacci number using bottom-up iteration:

$$f(0) = 0$$
$$f(1) = 1$$
$$f(2) = 0+1 = 1$$
$$f(3) = 1+1 = 2$$
$$f(4) = 1+2 = 3$$
$$f(5) = 2+3 = 5$$
$$f(n-2) =$$

$$f(n-1) =$$
$$f(n) = f(n-1) + f(n-2)$$

ALGORITHM Fib(n)
F[0] ← 0, F[1] ← 1
for i←2 to n do
        F[i] ← F[i-1] + F[i-2]
return F[n]

6. **Disuss Warshall's Algorithm with suitable diagrams?**

- Main idea: Use a bottom-up method to construct the transitive closure of a given digraph with *n* vertices through a series of *nxn* boolean matrices:

$R^{(0)}, ..., R^{(k-1)}, R^{(k)}, ..., R^{(n)}$

$\phantom{R^{(k)} : r_{ij}}^{(k)} = 1$ in R, iff

$R^{(k)} : r_{ij}$ $^{(k)}$

there is an edge from i to j; or
there is a path from i to j going through vertex 1; or
there is a path from i to j going through vertex 1 and/or 2; or
...
there is a path from i to j going through 1, 2, … and/or k
there is a path from i to j going through 1, 2, … and/or k
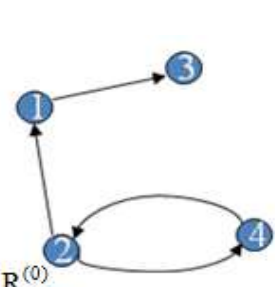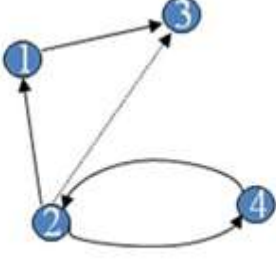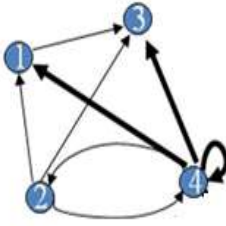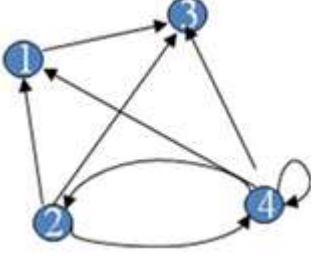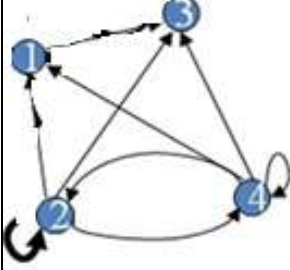
$R^{(0)}$

0 0 1 0

1 0 0 1

0 0 0 0

0 1 0 0

Does not allow an intermediate node

$R^{(1)}$

0 0 1 0

1 0 **1** 1

0 0 0 0

0 1 0 0

Allow 1 to be an intermediate node

$R^{(2)}$

0 0 1 0

1 0 1 1

0 0 0 0

1 1 **1 1**

Allow 1,2 to be an intermediate node

$R^{(3)}$

0 0 1 0

1 0 1 1

0 0 0 0

1 1 1 1

Allow 1,2,3 to be an intermediate node

$R^{(4)}$

0 0 1 0

1 **1** 1 1

0 0 0 0

1 1 1 1

Allow 1,2,3,4 to be an intermediate node

In the kth stage: to determine $R^{(k)}$ is to determine if a path exists between two vertices i, j using just vertices among $1,\dots,k$

$$r_{ij}^{(k)} = 1: \begin{cases} r_{ij}^{(k-1)} = 1 & \text{(path using just } 1,\dots,k\text{-1)} \\ \quad \text{or} \\ (r_{ik}^{(k-1)} = 1 \text{ and } r_{kj}^{(k-1)}) = 1 & \text{(path from } i \text{ to } k \\ & \text{and from } k \text{ to } i \\ & \text{using just } 1,\dots,k\text{-1)} \end{cases}$$

7. **Explain how to Floyd's Algorithm works.**
   - All pairs shortest paths problem: In a weighted graph, find shortest paths between every pair of vertices.
   - Applicable to: undirected and directed weighted graphs; no negative weight.
   - Same idea as the Warshall's algorithm : construct solution through series of matrices D(0) , D(1), …, D(n)

$$\begin{matrix} 0 & \infty & 4 & \infty \\ 1 & 0 & 6 & 3 \\ \infty & \infty & 0 & \infty \\ \infty & 5 & 1 & 0 \end{matrix} \qquad \begin{matrix} 0 & \infty & 4 & \infty \\ 1 & 0 & 5 & 3 \\ \infty & \infty & 0 & \infty \\ 6 & 5 & 1 & 0 \end{matrix}$$

Weight matrix $\longrightarrow$ distancematrix
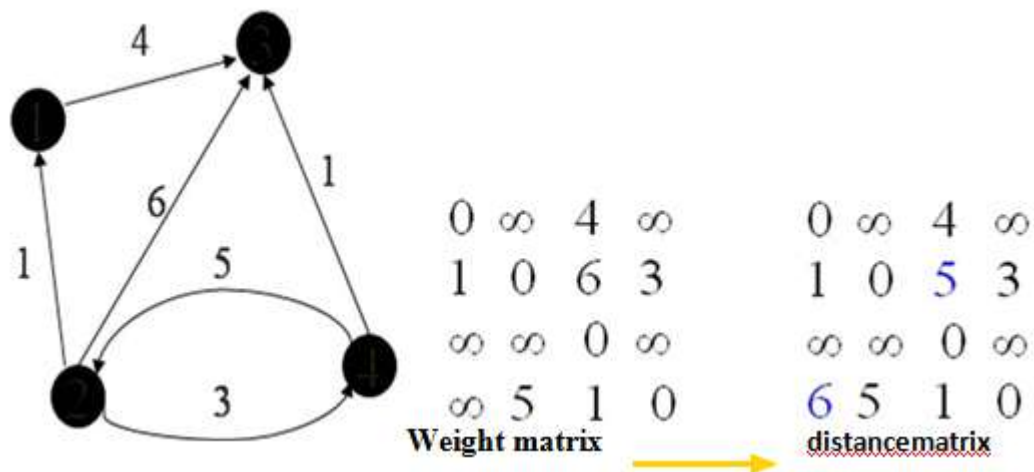
   - D(k) : allow 1, 2, …, k to be intermediate vertices.
   - In the kth stage, determine whether the introduction of k as a new eligible intermediate vertex will bring about a shorter path from i to j.
   - dij(k) = min{dij(k-1) , dik(k-1) + dkj(k-1)}   for k ≥ 1, dij(0) = wij
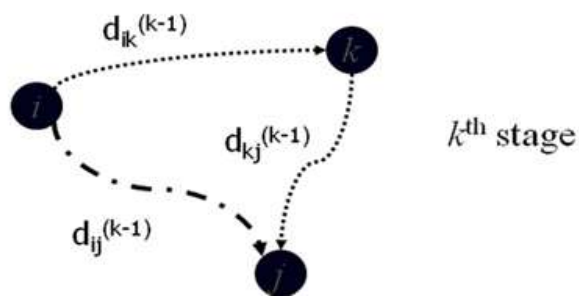
8. **Explain Knapsack problem**

✓ **The problem**

- Find the most valuable subset of the given n items that fit into a knapsack of capacity W.

✓ **Consider the following sub problem P(i, j)**

- Find the most valuable subset of the first i items that fit into a knapsack of capacity j, where $1 \leq i \leq n$, and $1 \leq j \leq W$

- Let V[i, j] be the value of an optimal solution to the above subproblem P(i, j). Goal: V[n, W]

9. **Explain Memory Function algorithm for the Knapsack problem**

✓ The Knapsack Problem and Memory Functions

- The Recurrence

  a. Two possibilities for the most valuable subset for the subproblem P(i, j)

     i. It does not include the ith item: *V[i, j] = V[i-1, j]*

     ii. It includes the ith item: *V[i, j] = v$_i$+ V[i-1, j − w$_i$]*

$$V[i, j] = \ max\{V[i-1, j], v_i + V[i-1, j − w_i] \}, if\ j − w_i \geq 0$$

$$V[i-1, j] \qquad\qquad if\ j − w_i < 0$$

$$V[0, j] = 0\ for\ j \geq 0\ and\ V[i, 0] = 0\ for\ i \geq 0$$

✓ Memory functions:

- Memory functions: a combination of the top-down and bottom-up method. The idea is to solve the subproblems that are necessary and do it only once.

- Top-down: solve common subproblems more than once.

- Bottom-up: Solve subproblems whose solution are not necessary for the solving the original problem.

✓ ALGORITHM MFKnapsack(i, j)

    if V[i, j] < 0 //if subproblem P(i, j) hasn't been solved yet.

        if j < Weights[i]

value ← MFKnapsack(i – 1, j)

else

value ←max(MFKnapsack(i – 1, j),

values[I] + MFKnapsck(i – 1, j – Weights[i]))

V[i, j] ← value

return V[i, j]

10. **Explain in detail about Huffman tree.**

Any binary tree with edges labeled with 0's and 1's yields a prefix-free code of characters assigned to its leaves. Optimal binary tree minimizing the average length of a codeword can be constructed as follows:

Huffman's algorithm

- Initialize *n* one-node trees with alphabet characters and the tree weights with their frequencies.
- Repeat the following step *n*-1 times: join two binary trees with smallest weights into one (as left and right subtrees) and make its weight equal the sum of the weights of the two trees.
- Mark edges leading to left and right subtrees with 0's and 1's, respectively.

Example:

character	A	B	C	D	_
codeword	0.35	0.1	0.2	0.2	0.15
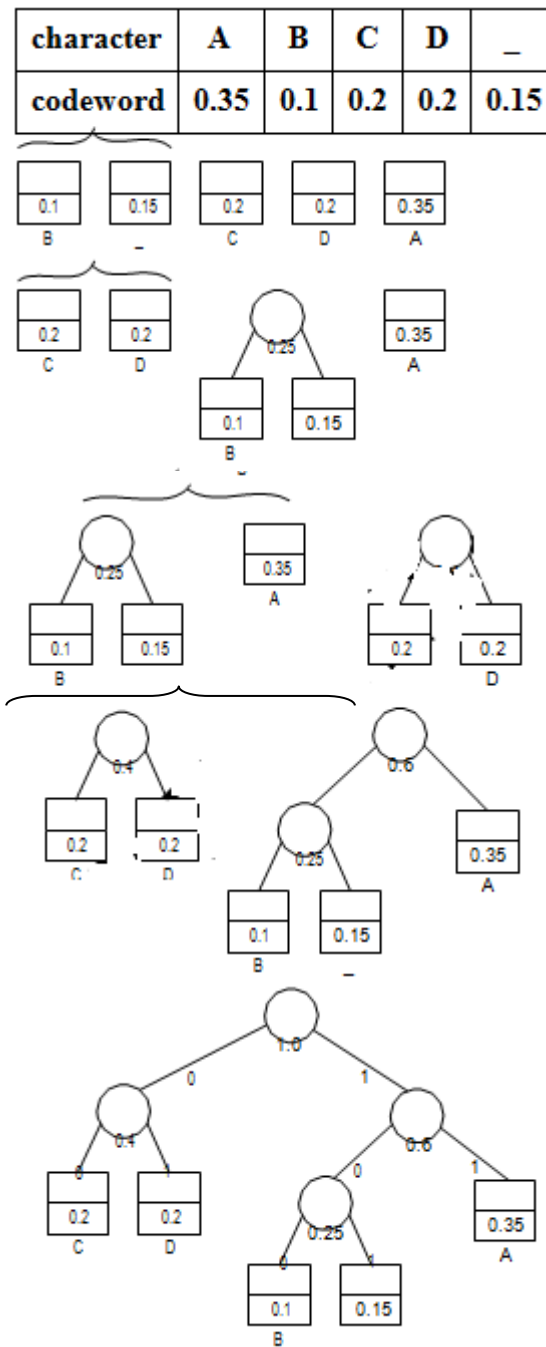
UNIT IV
ITERATIVE IMPROVEMENT

The Simplex MeFthod-The Maximum-Flow Problem – Maximum Matching in Bipartite Graphs- The Stable marriage Problem.

PART A

1. Define linear programming.
Every LP problem can be represented in such form

maximize $\quad 3x + 5y$ $\qquad\qquad\qquad$ maximize $3x + 5y + 0u + 0v$

subject to $\quad x + \; y \le 4$ $\qquad\qquad\qquad$ subject to $\quad x + \; y + u \qquad = 4$

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad x + 3y \le 6$

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad x + 3y \qquad + \; v \; = 6$

$x \ge 0, \;\; y \ge 0$ $\qquad\qquad\qquad\qquad\qquad\qquad x \ge 0, \; y \ge 0, \; u \ge 0, \; v \ge 0$

Variables u and v, transforming inequality constraints into equality constrains, are called *slack variables*

2. What is basic solution?
A *basic solution* to a system of m linear equations in n unknowns ($n \ge m$) is obtained by setting $n - m$ variables to 0 and solving the resulting system to get the values of the other m variables. The variables set to 0 are called *nonbasic*; the variables obtained by solving the system are called *basic*.
A basic solution is called *feasible* if all its (basic) variables are nonnegative.

3. Define Bipartite graph.
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

4. What is cut and min cut?

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.

- We'll denote a cut and its capacity by C(X,X) and c(X,X)
- Note that if all the edges of a cut were deleted from the network, there would be no directed path from source to sink
- *Minimum cut* is a cut of the smallest capacity in a given network

5. **State max – flow – min – cut theorem.**
   The value of maximum flow in a network is equal to the capacity of its minimum cut.

6. **Define Bipartite Graphs.**
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?**
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.

PART B

1. **Explain in detail about simplex method.**
Every LP problem can be represented in such form

maximize $\quad 3x + 5y$ $\qquad\qquad\qquad$ maximize $3x + 5y + 0u + 0v$

subject to $\quad x + y \le 4$ $\qquad\qquad\qquad$ subject to $\quad x + y + u \quad\quad = 4$

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad x + 3y \le 6$

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad x + 3y \quad\quad + v = 6$

$x \ge 0, \quad y \ge 0$ $\qquad\qquad\qquad\qquad\qquad\qquad x \ge 0, \ y \ge 0, \ u \ge 0, \ v \ge 0$

Variables *u* and *v*, transforming inequality constraints into equality constrains, are called *slack variables*
A *basic solution* to a system of *m* linear equations in *n* unknowns ($n \ge m$) is obtained by setting $n - m$ variables to 0 and solving the resulting system to get the values of the other *m* variables. The variables set to 0 are called *nonbasic*; the variables obtained by solving the system are called *basic*.
A basic solution is called *feasible* if all its (basic) variables are nonnegative.

Example $x + y + u \quad\quad = 4$

$\qquad x + 3y \quad\quad + v = 6$

$\qquad$ **(0, 0, 4, 6) is basic feasible solution**

$\qquad\qquad\qquad$ (*x*, *y* are nonbasic; *u*, *v* are basic)

There is a 1-1 correspondence between extreme points of LP's feasible region and its basic feasible solutions.

maximize $\quad z = 3x + 5y + 0u + 0v$

subject to $\qquad x + y + u \qquad =$

4

$\qquad\qquad\qquad x + 3y \qquad + v = 6$

$x \geq 0, \; y \geq 0, \; u \geq 0, \; v \geq 0$

basic feasible solution

(0, 0, 4, 6)

Simplex method:

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 — 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 variable] For each positive entry in the pivot column, calculate the θ-ratio by dividing that row's entry in the rightmost column by its entry in the pivot column. (If there are no positive entries in the pivot column — 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.

maximize $z = 3x + 5y + 0u + 0v$

subject to $x + \; y + \; u \qquad = 4$

$\qquad\qquad\qquad x + 3y \qquad + \; v \; = 6$

$x \geq 0, \; y \geq 0, \; u \geq 0, \; v \geq 0$

	x	y	u	v		
u	1	1	1	0	:	a
← v	1	3	0	1	:	6
	−3	−5		0	:	0

	x	y	u	v		
← u	$\frac{2}{3}$	0	1	$-\frac{1}{3}$	:	2
y	$\frac{1}{3}$	1	0	$\frac{1}{3}$	:	2
	$-\frac{4}{3}$	0	0	$\frac{5}{3}$	:	10

	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.
(0, 0, 4, 6)
z = 0

basic feasible sol.
(0, 2, 2, 0)
z = 10

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

2.   Explain in detail about maximum flow problem.   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 u_{ij} 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

Definition of flow:

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*

    $0 \le x_{ij} \le u_{ij}$ **for every edge** $(i,j) \in E$

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:

$$\sum_{j:\ (1,j)\ \epsilon\ E} x_{1j} \quad = \quad \sum_{j:\ (j,n)\ \epsilon\ E} x_{jn}$$

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

Maximum-Flow Problem as LP problem

Maximize $\quad v = \sum_{j:\ (1,j)\ \in\ E} x_{1j}$

subject to

$$\sum_{j:\ (j,i)\ \in\ E} x_{ji} \ - \ \sum_{j:\ (i,j)\ \in\ E} x_{ij} \ = \ 0 \quad \text{for } i = 2, 3, \ldots, n\text{-}1$$

$$0 \le x_{ij} \le u_{ij} \quad \text{for every edge } (i,j) \in E$$

Augmenting Path (Ford-Fulkerson) Method

- Start with the zero flow ($x_{ij} = 0$ for every edge)
- On each iteration, try to find a *flow-augmenting path* from source to sink, which a path along which some additional flow can be sent
- If a flow-augmenting path is found, adjust the flow along the edges of this path to get a flow of increased value and try again
- If no flow-augmenting path is found, the current flow is maximum
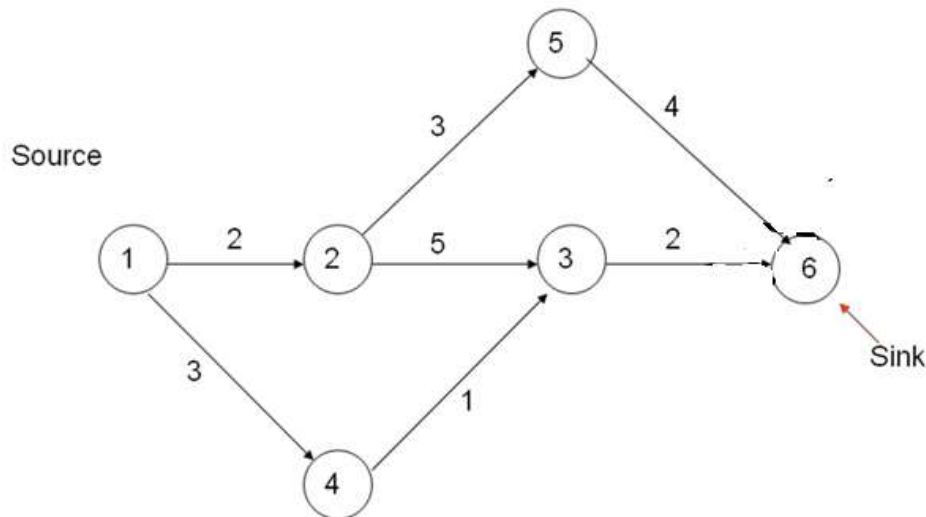
Augmenting path: $1\rightarrow2 \rightarrow3 \rightarrow6$

Augmenting path: $1 \rightarrow4 \rightarrow3\leftarrow2 \rightarrow5 \rightarrow6$

max flow value = 3

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 u_{ij} 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

Definition of flow:

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*

   $0 \leq x_{ij} \leq u_{ij}$ **for every edge** $(i,j) \in E$

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:

$$\sum_{j:\ (1,j)\ \in\ E} x_{1j} \quad = \quad \sum_{j:\ (j,n)\ \in\ E} x_{jn}$$

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

Maximum-Flow Problem as LP problem
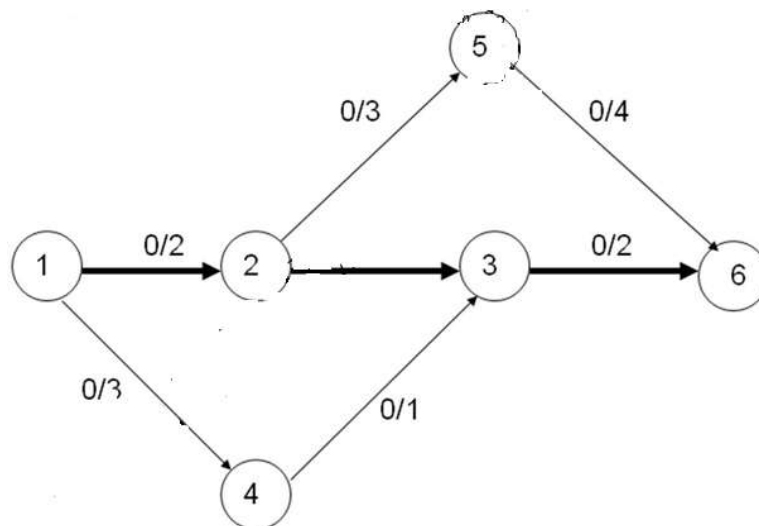
Maximize $\quad v = \sum_{j:\ (1,j)\ \in\ E} x_{1j}$

subject to

$$\sum_{j:\,(j,i)\,\in\,E} x_{ji} \;-\; \sum_{j:\,(i,j)\,\in\,E} x_{ij} \;=\; 0 \quad \text{for } i = 2, 3, \dots, n\text{-}1$$

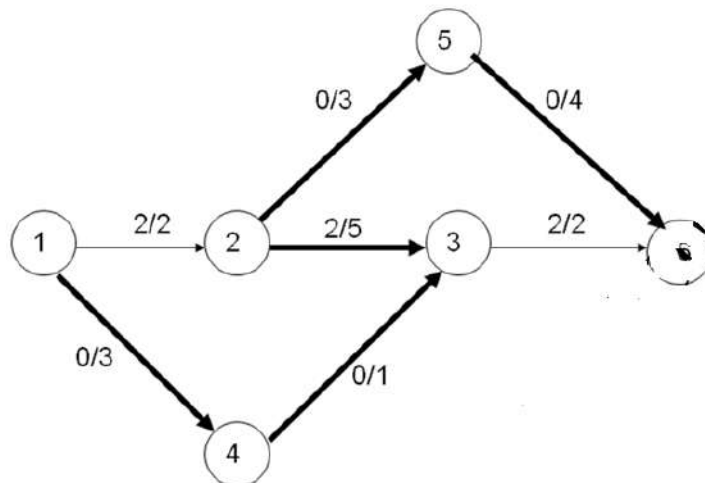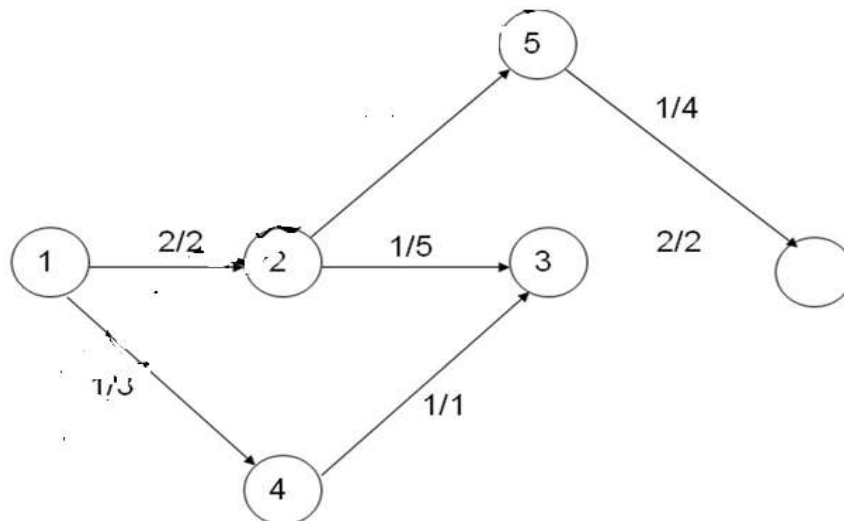$$0 \le x_{ij} \le u_{ij} \quad \text{for every edge } (i,j) \in E$$

Augmenting Path (Ford-Fulkerson) Method

- Start with the zero flow ($x_{ij} = 0$ for every edge)
- On each iteration, try to find a *flow-augmenting path* from source to sink, which a path along which some additional flow can be sent
- If a flow-augmenting path is found, adjust the flow along the edges of this path to get a flow of increased value and try again
- If no flow-augmenting path is found, the current flow is maximum

Augmenting path: 1→2 →3 →6

Augmenting path: $1 \rightarrow 4 \rightarrow 3 \leftarrow 2 \rightarrow 5 \rightarrow 6$

max flow value = 3

Finding a flow-augmenting path

To find a flow-augmenting path for a flow x, consider paths from source to sink in the underlying <u>undirected</u> graph in which any two consecutive vertices i,j are either:

- connected by a directed edge (i to j) with some positive unused capacity $r_{ij} = u_{ij} - x_{ij}$
 - known as *forward edge* ($\rightarrow$)

OR

- connected by a directed edge (j to i) with positive flow x_{ji}
  - known as *backward edge* ($\leftarrow$)

If a flow-augmenting path is found, the current flow can be increased by r units by increasing x_{ij} by r on each forward edge and decreasing x_{ji} by r on each backward edge, where

r = min $\{r_{ij}$ on all forward edges, x_{ji} on all backward edges$\}$

- Assuming the edge capacities are integers, r is a positive integer

- On each iteration, the flow value increases by at least 1
- Maximum value is bounded by the sum of the capacities of the edges leaving the source; hence the augmenting-path method has to stop after a finite number of iterations
- The final flow is always maximum, its value doesn't depend on a sequence of augmenting paths used
- The augmenting-path method doesn't prescribe a specific way for generating flow-augmenting paths
- Selecting a bad sequence of augmenting paths could impact the method's efficiency

Example:
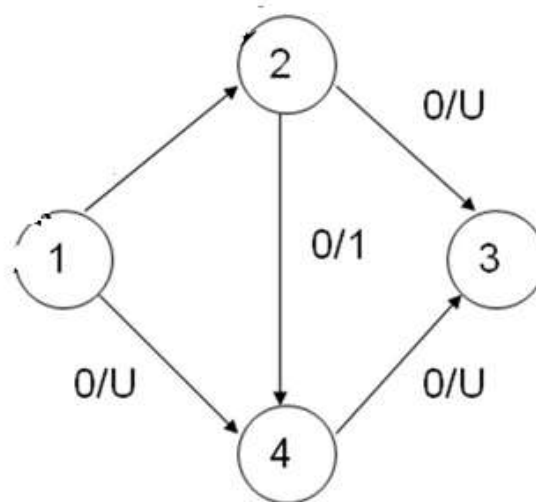
U = large positive integer
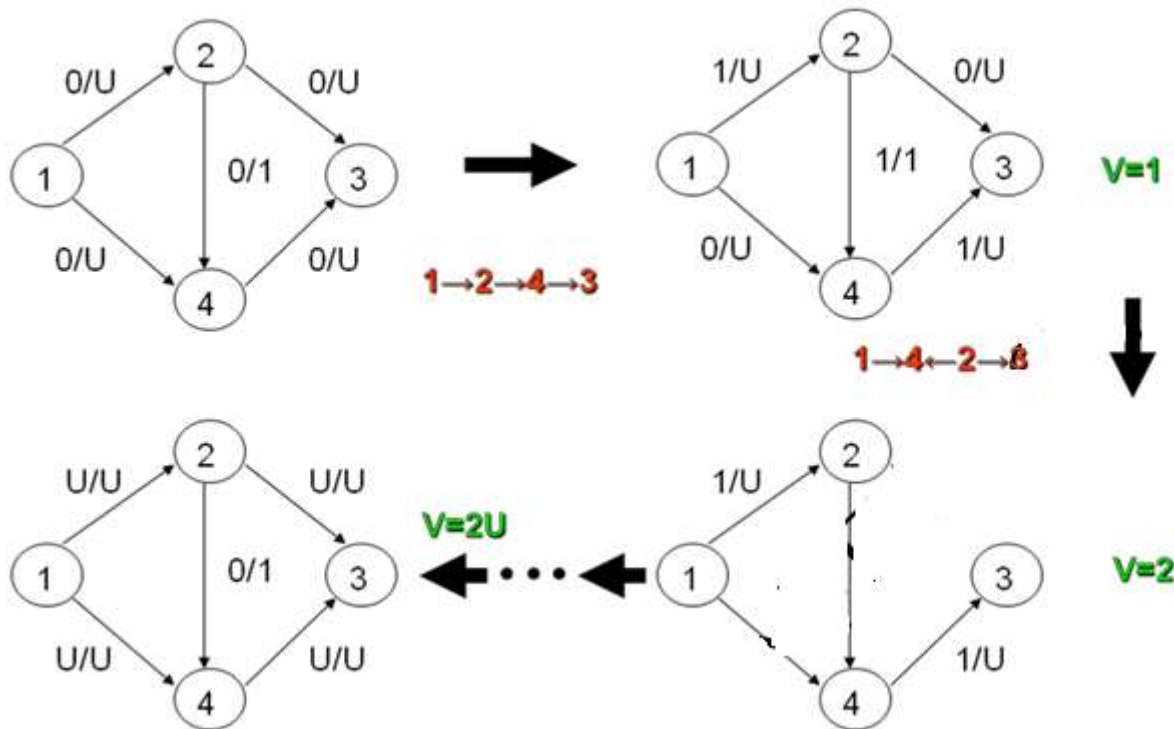
Requires 2U iterations to reach the maximum flow of value 2U

Shortest-Augmenting-Path Algorithm

Generate augmenting path with the least number of edges by BFS as follows.

Starting at the source, perform BFS traversal by marking new (unlabeled) vertices with two labels:

- first label – indicates the amount of additional flow that can be brought from the source to the vertex being labeled

second label – indicates the vertex from which the vertex being labeled was reached, with —+‖ or —–‖ added to the second label to indicate whether the vertex was reached via a forward or backward edge

- The source is always labeled with ∞,-
- All other vertices are labeled as follows:
    - If unlabeled vertex j is connected to the front vertex i of the traversal queue by a directed edge from i to j with positive unused capacity $r_{ij} = u_{ij} - x_{ij}$ (forward edge), vertex j is labeled with l_j, i^+, where $l_j = \min\{l_i, r_{ij}\}$
    - If unlabeled vertex j is connected to the front vertex i of the traversal queue by a directed edge from j to i with positive flow x_{ji} (backward edge), vertex j is labeled l_j, i^-, where $l_j = \min\{l_i, x_{ji}\}$
- If the sink ends up being labeled, the current flow can be augmented by the amount indicated by the sink's first label
- The augmentation of the current flow is performed along the augmenting path traced by following

the vertex second labels from sink to source; the current flow quantities are increased on the forward edges and decreased on the backward edges of this path
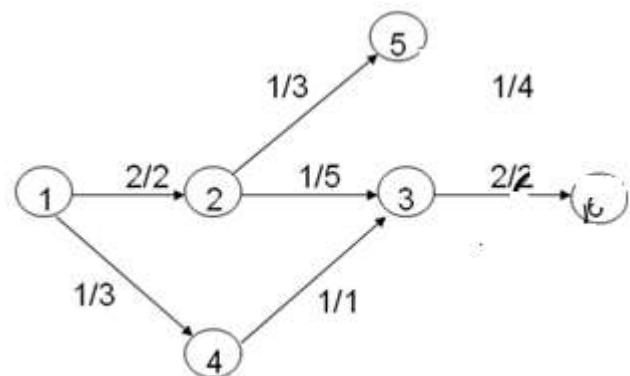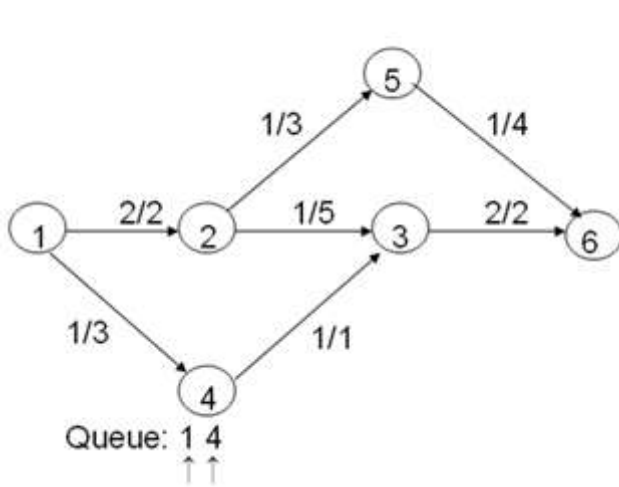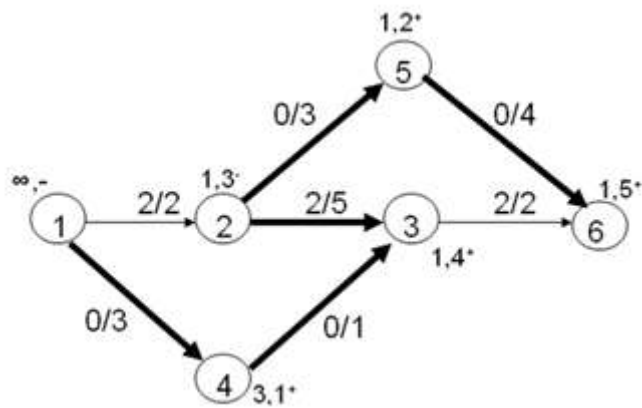
- If the sink remains unlabeled after the traversal queue becomes empty, the algorithm returns the current flow as maximum and stops
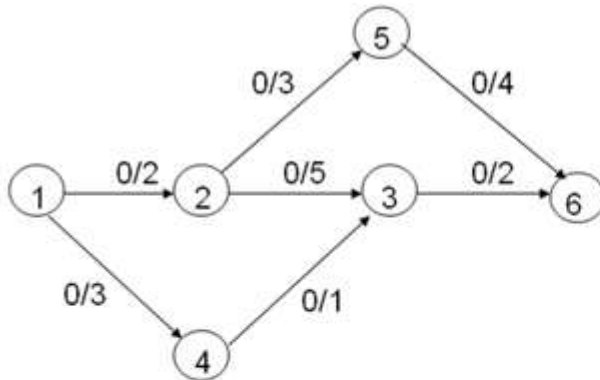

Queue: 1 2 4 3 5 6
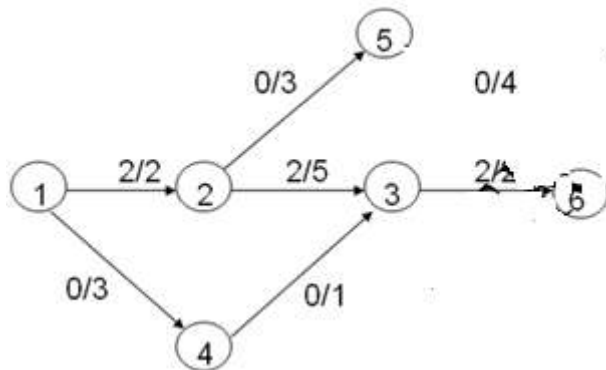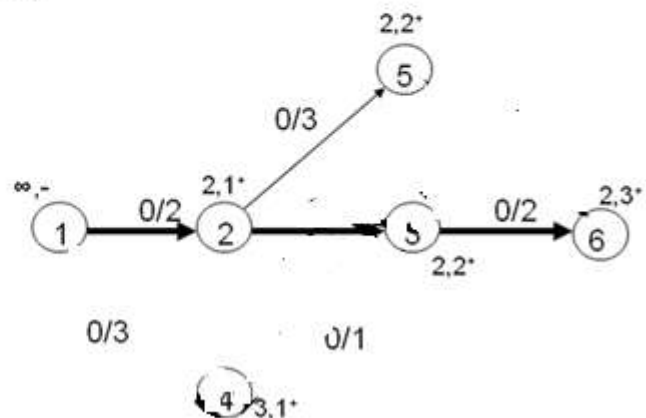
Queue: 1 2 4 3 5 6

Definition of a Cut:

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.

- We'll denote a cut and its capacity by C(X,X) and c(X,X)
- Note that if all the edges of a cut were deleted from the network, there would be no directed path from source to sink
- *Minimum cut* is a cut of the smallest capacity in a given network

Examples of network cuts
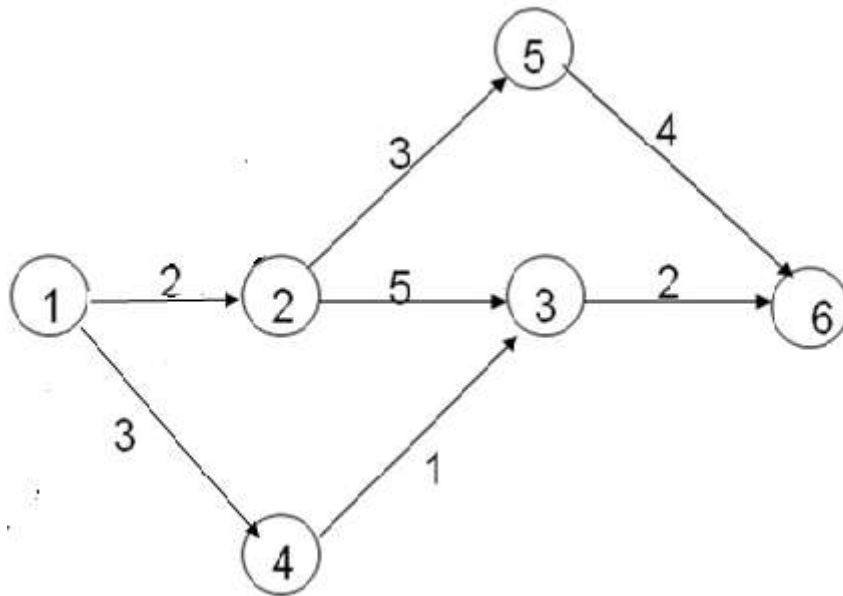
If X = {1} and X = {2,3,4,5,6}, C(X,X) = {(1,2), (1,4)}, c = 5
 If X ={1,2,3,4,5} and X = {6}, C(X,X) = {(3,6), (5,6)}, c = 6
If X = {1,2,4} and X ={3,5,6}, C(X,X) = {(2,3), (2,5), (4,3)}, c = 9

Max-Flow Min-Cut Theorem
- The value of maximum flow in a network is equal to the capacity of its minimum cut
- 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

\

```
Shortest-augmenting-path algorithm
Input: A network with single source 1, single sink n, and
        positive integer capacities uij on its edges (i, j)
Output: A maximum flow x
assign xij = 0 to every edge (i, j) in the network
label the source with ∞, - and add the source to the empty queue Q
while not Empty(Q) do
    i ← Front(Q);   Dequeue(Q)
    for every edge from i to j do //forward edges
        if j is unlabeled
            rij ← uij − xij
            if rij > 0
                lj ← min{li, rij};  label j with lj, i+
                Enqueue(Q, j)
    for every edge from j to i do //backward edges
        if j is unlabeled
            if xji > 0
            lj ← min{li, xji};  label j with lj, i−
            Enqueue(Q, j)
if the sink has been labeled
    //augment along the augmenting path found
    j ← n //start at the sink and move backwards using second labels
    while j ≠ 1 //the source hasn't been reached
        if the second label of vertex j is i+
            xij ← xij + ln
        else //the second label of vertex j is i−
            xji ← xji − ln
        j ← i
    erase all vertex labels except the ones of the source
    reinitialize Q with the source
return x //the current flow is maximum
```

Time Efficiency

- The number of augmenting paths needed by the shortest-augmenting-path algorithm never exceeds $nm/2$, where n and m are the number of vertices and edges, respectively
- Since the time required to find shortest augmenting path by breadth-first search is in $O(n+m)=O(m)$ for networks represented by their adjacency lists, the time efficiency of the shortest-augmenting-path algorithm is in $O(nm^2)$ for this representation
- More efficient algorithms have been found that can run in close to $O(nm)$ time, but these algorithms don't fall into the iterative-improvement paradigm

3.   Explain in detail about maximum bipartite matching Bipartite Graphs

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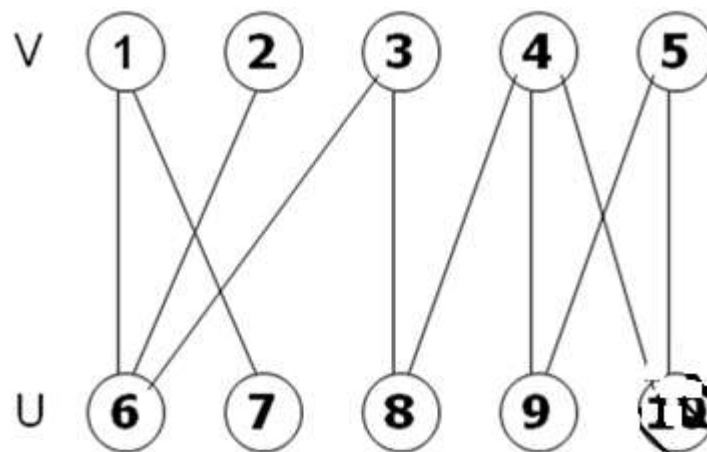
A bipartite graph is *2-colorable*: the vertices can be colored in two colors so that every edge has its vertices colored differently

Matching in a Graph

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

$$M = \{(4,8), (5,9)\}$$

A *maximum* (or *maximum cardinality*) *matching* is a matching with the largest number of edges

- always exists
- not always unique

Free Vertices and Maximum Matching

- A matching in this graph (M)

For a given matching M, a vertex is called *free* (or *unmatched*) if it is not an endpoint of any edge in M; otherwise, a vertex is said to be *matched*

- If every vertex is matched, then M is a maximum matching
- If there are unmatched or free vertices, then M may be able to be improved
- We can immediately increase a matching by adding an edge connecting two free vertices (e.g., (1,6) above)

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

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

Theorem   A matching M is maximum if and only if there exists no augmenting path with respect to M

Augmenting Path Method (template)

- Start with some initial matching
    - e.g., the empty set
- Find an augmenting path and augment the current matching along that path
    - e.g., using breadth-first search like method
- When no augmenting path can be found, terminate and return the last matching, which is maximum

BFS-based Augmenting Path Algorithm

- Initialize queue Q with all free vertices in one of the sets (say V)

- While Q is not empty, delete front vertex w and label every unlabeled vertex u adjacent to w as follows:

    - Case 1 (w is in V): If u is free, augment the matching along the path ending at u by moving backwards until a free vertex in V is reached. After that, erase all labels and reinitialize Q with all the vertices in V that are still free If u is matched (not with w), label u with w and enqueue u

- Case 2 (w is in U) Label its matching mate v with w and enqueue v

- After Q becomes empty, return the last matching, which is maximum

Each vertex is labeled with the vertex it was reached from. Queue deletions are indicated by arrows. The free vertex found in U is shaded and labeled for clarity; the new matching obtained by the augmentation is shown on the next slide.

a. This matching is maximum since there are no remaining free vertices in V (the queue is empty)

b Note that this matching differs from the maximum matching found earlier.

Maximum-matching algorithm for bipartite graphs

Input: A bipartite graph $G = \langle V, U, E \rangle$

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**

    $w \leftarrow Front(Q);\ Dequeue(Q)$

    if $w \in V$

        for every vertex u adjacent to w **do**

            if u is free

                //augment

                $M \leftarrow M \cup (w, u)$

                $v \leftarrow w$

                while v is labeled **do**

                    $u \leftarrow$ vertex indicated by v's label; $\quad M \leftarrow M - (v, u)$

                    $v \leftarrow$ vertex indicated by u's label; $\quad M \leftarrow M \cup (v, u)$

                remove all vertex labels

                reinitialize Q with all free vertices in V

                break //exit the for loop

            else //u is matched

                if $(w, u) \notin M$ **and** u is unlabeled

                    label u with w

    else //$w \in U$ (and matched)

        label the mate v of w with "w]

        $Enqueue(Q, v)$

return M //current matching is maximum

4. Explain detail stable marriage problem

Stable Marriage Problem:

There is a set $Y = \{m_1, \ldots, m_n\}$ of n men and a set $X = \{w_1, \ldots, w_n\}$ of n women. Each man has a ranking list of the women, and each woman has a ranking list of the men (with no ties in these lists).

A *marriage matching* M is a set of n pairs (m_i, w_j).

A pair (m, w) is said to be a *blocking pair* for matching M if man m and woman w are not matched in M but prefer each other to their mates in M.

A marriage matching M is called *stable* if there is no blocking pair for it; otherwise, it's called *unstable*.

The *stable marriage problem* is to find a stable marriage matching for men's and women's given preferences.

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.

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

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 prefers m 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

men's preferences   women's preferences

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

{(Bob, Ann) (Jim, Lea) (Tom, Sue)} is unstable
{(Bob, Ann) (Jim, Sue) (Tom, Lea)} is stable
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
Free men: Jim Tom

Lea

	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
rejected

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

Free men:
Jim, Tom

Jim proposed to Sue
Sue accepted

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

Free men:
Tom

Tom proposed to Sue
Sue rejected

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

Free men:
Tom

Tom proposed to Lea
Lea replaced Bob
with 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

Free men:
Bob

Bob proposed to Ann
Ann accepted

- The algorithm terminates after no more than n 2 iterations with a stable marriage output
- The stable matching produced by the algorithm is always man-optimal: each man gets the highest rank woman on his list under any stable marriage. One can obtain the womanoptimal matching by making women propose to men
- A man (woman) optimal matching is unique for a given set of participant preferences
- The stable marriage problem has practical applications such as matching medical-school graduates with hospitals for residency training

UNIT V

COPING WITH THE LIMITATIONS OF ALGORITHM POWER

Limitations of Algorithm Power-Lower-Bound Arguments-Decision Trees-P, NP and NP-Complete Problems-- Coping with the Limitations - Backtracking – n-Queens problem – Hamiltonian Circuit Problem – Subset Sum Problem-Branch and Bound – Assignment problem – Knapsack Problem – Traveling Salesman ProblemApproximation Algorithms for NP – Hard Problems – Traveling Salesman problem – Knapsack problem.

PART A

1. Define backtracking?

Depth first node generation with bounding function is called backtracking. The backtracking algorithm has its virtue the ability to yield the answer with far fewer than m trials.

2. What is Hamiltonian cycle in an undirected graph?

A Hamiltonian cycle is round trip along n edges of G that visits every vertex once and returns to its starting position.

3. What is Feasible solution?

It is obtained from given n inputs Subsets that satisfies some constraints are called feasible solution. It is obtained based on some constraints

4. What is optimal solution?

It is obtained from feasible solution.
Feasible solution that maximizes or minimizes a given objective function. It is obtained based on objective function.

5. **List the application of backtracking** technique? 8-
Qeens problem

6. Given an application for knapsack problem?

The knapsack problem is problem combinatorial optimization. It derives its name from the maximum problem of choosing possible essential that can fit too bag to be carried on trip. A similar problem very often appears business, combinatory, complexity theory, cryptography and applied mathematics.

7. Define subset sum problem?

Subset sum problem is problem, which is used to find a subset of a given set S={S1,S2,S3,…….Sn} of positive integers whose sum is equal to given positive integer d.

8. What is heuristic?

A heuristic is common sense rule drawn from experience rather than from mathematically proved assertion.
For example, going to the nearest un visited city in the travelling salesman problem is good example for heuristic.

9. State the concept of branch and bound method?

The branch and bound method refers to all state space search methods in which all children of the E-Node are generated before any other live node can become the E-node.

10. Give the upper bound and lower bound of matrix multiplication algorithm?

Upper bound: The given algorithm does n*n*n multiplication hence at most n*n*n multiplication are necessary.

Lower bound: It has been proved in the literature that at least n*n multiplication are necessary.

11. What is state space tree?

Backtracking and branch bound are based on the construction of a state space tree, whose nodes reflect specific choices made for a solution's component .Its root represents an initial state before the search for a solution begins.

The nodes of the first level the tree represent the made for the first component of solution, the nodes of the second evel represent the
Choices for the second components & so on

12. What is promising and non promising node?

A node state space tree is said to be promising, if it corresponds to a partially constructed solution that may still lead to complete solution. Otherwise, node is called non- promising.

13. What are the additional items are required for branch and bound compare to backtracking technique?

Compared to backtracking, branch and bound requires 2 additional items.
1) A way to proved , for every node of node of state space tree, a bound on the best value of the objective function on any solution that can be obtain d by adding further components to the partial solutionrepresented by the node.
2) The value of the best solution seen so far.

14. Differentiate backtracking and branch bound techniques.
- Backtracking is applicable only to non optimization problems.
- Backcktracking generates state space tree depth first manner.
- Branch and bound is applicable only to optimization problem.
- Branch and bound generated a node of state space tree using best first rule

15. What is called all pair shortest path problem?

Given a weighted connected graph, the all pairs shortest paths problem is to find the distances from each vertex to all other vertices

16. How will you construct an optimal binary search tree?

A binary search tree is one of the most important data structures in computer sciences. its principal applications are to implement a dictionary, a set of elements with the operations of searching, insertion and deletion. If probabilities of searching for elements of a set are known as optimal binary search tree, for which the average number of comparisons in a sear h is the smallest possible.

17. What is the runtime of shortest path algorithm?

The runtime of shortest path algorithm is ((n+|E|) log n)

18. What is mathematical modelling?

Mathematical modelling is method of ex pressing problem in terms of purely Mathematical objects such as variables, functions and equations.

19. What is pre structure?

Pre structuring is type of technique that exploits space for time tradeoffs simply uses extra space of facilities faster and or more flexible access to data.

20. Define Heap sort?
Heap sort is   sorting algorithm. Heap sort is 2 stage algorithm. The two stages are Stage
 1- Heap Construction- Construct heap for a given array of elements
Stage 2- Maximum Deletion- Apply the root deletion operation n-1 times to the remaining heap.

PART B

1. **Explain the backtracking algorithm for the n-queens problem.**
    The problem is to place eight queens on a 8 x 8 chessboard so that no two queen ―attack‖ that is, so that no two of them are on the same row, column or on the diagonal.
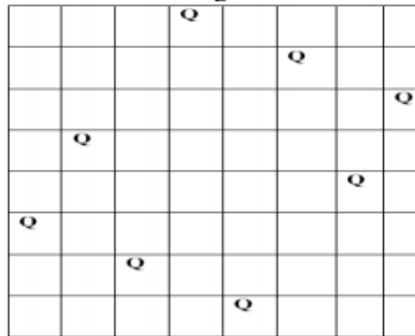
```
Algorithm place(k,I)
{

    for j := 1 to k-1 do
    if(x[j]=I) or(abs(x[j]-I)=abs(j-k))) then return false;
    return true;
    }
    Algorithm Nqueens(k,n)
    {
    for I:= 1 to n do
    {
    if( place(k,I) then
    {
    x[k]:= I;
    if(k=n) then write(x[1:n]);
    else
    Nqueens(k+1,n) } } }
```

2. **Explain Backtracking technique Backtracking technique is a refinement of this approach.**
    Backtracking is a surprisingly simple approach and can be used even for solving the hardest Sudoku puzzle. Problems that need to find an element in a domain that grows exponentially with the size of the input, like the Hamiltonian circuit and the Knapsack problem, are not solvable in polynomial time. Such problems can be solved by the exhaustive search technique, which requires identifying the correct solution from many candidate solutions.

Pseudocode of Backtracking Algorithm

```
Backtrack (B[1..i])
// Input: B[1..i] indicates the first i promising parts of a solution
//Output: All the tuples which are the solution of the problem
If B[1..i] is a solution write B[1..i]
else
for each element e ∈ S i +1 consistent with B[1..i] and the constraints do
        B[1..i] ← e
        Backtrack (B[1..i+1])
```

Steps to achieve Goal:

- Backtracking possible by constructing the state-space tree, this is a tree of choices.
- The root of the state-space tree indicates the initial state, before the search for the solution begins.
- The nodes of each level of this tree signify the candidate solutions for the corresponding component.
- A node of this tree is considered to be promising if it represents a partially constructed solution that can lead to a complete solution, else they are considered to be non-promising.
- The leaves of the tree signify either the non-promising dead-ends or the complete solutions.
- Depth-First-search method usually for constructing these state-space-trees.
- If a node is promising, then a child-node is generated by adding the first legitimate choice of the next component and the processing continues for the child node.
- If a node is non-promising, then the algorithm backtracks to the parent node and considers the next promising solution for that component.
- If there are no more choices for that component, the algorithm backtracks one more level higher. The algorithm stops when it finds a complete solution.

Example:

- This technique is illustrated by the following figure.
- Here the algorithm goes from the start node to node 1 and then to node 2.
- When no solution is found it backtracks to node1 and goes to the next possible solution node 3. But node 3 is also a dead-end.
- Hence the algorithm backtracks once again to node 1 and then to the start node. From here it goes to node 4 and repeats the procedure till node 6 is identified as the solution.
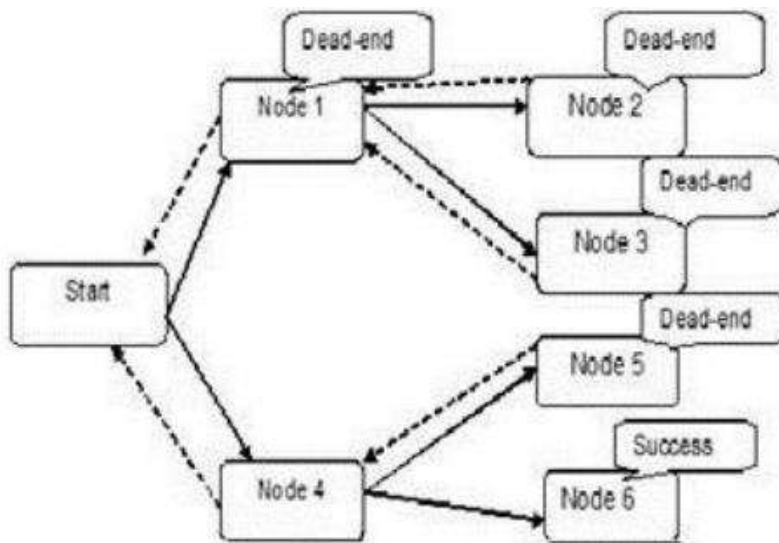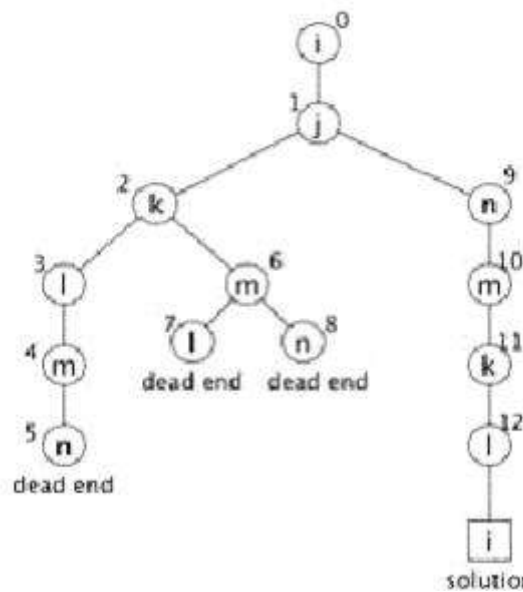
3.Give solution to Hamiltonian circuit using Backtracking technique

- The graph of the Hamiltonian circuit is shown below.
- In the below Hamiltonian circuit, circuit starts at vertex i, which is the root of the state-space tree.
- From i, we can move to any of its adjoining vertices which are j, k, and l. We first select j, and then move to k, then l, then to m and thereon to n. But this proves to be a dead-end.
- So, we backtrack from n to m, then to l, then to k which is the next alternative solution. But moving from k to m also leads to a dead-end.
- So, we backtrack from m to k, then to j. From there we move to the vertices n, m, k, l and correctly return to i. Thus the circuit traversed is i-> j -> n-> m-> k-> l-> i.
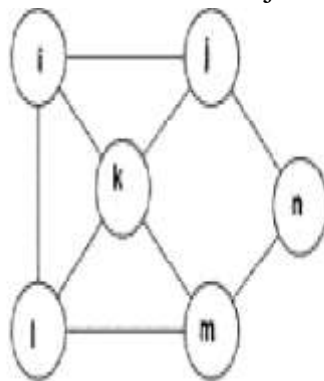
State-Space Tree for Finding a Hamiltonian Circuit

4.Give solution to Subset sum problem using Backtracking technique.

- In the Subset-Sum problem, we have to find a subset of a given set $S = \{s1, s2, ..., sn\}$ of n positive integers whose sum is equal to a positive integer t.
- Let us assume that the set S is arranged in ascending order. For example, if $S = \{2, 3, 5, 8\}$ and if t = 10, then the possible solutions are $\{2, 3, 5\}$ and $\{2, 8\}$.
- The root of the tree is the starting point and its left and right children represent the inclusion and exclusion of 2.
- Similarly, the left node of the first level represents the inclusion of 3 and the right node the exclusion of 3.
- Thus the path from the root to the node at the ith level shows the first i numbers that have been included in the subsets that the node represents.
- Thus, each node from level 1 records the sum of the numbers Ssum along the path upto that particular node.
- If Ssum equals t, then that node is the solution. If more solutions have to be found, then we can backtrack to that node's parent and repeat the process. The process is terminated for any non-promising node that meets any of the following two conditions:

$$S_{sum} + S_{i+1} > t \text{ (the sum is too large)}$$

$$S_{sum} + \sum_{j=i+1}^{n} s_j < t \text{ (the sum is too small)}$$
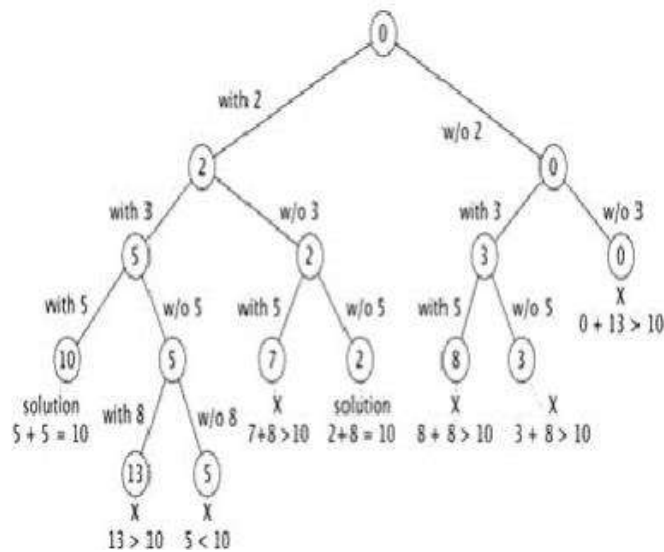
5. Explain P, NP and NP complete problems.

Definition: I

An algorithm solves a problem in polynomial time if its worst case time efficiency belongs to O (P (n)). P (n) is a polynomial of the problem's input size n.

Tractable:

Problems that can be solved in polynomial time are called tractable.

Intractable:

Problems that cannot be solved in polynomial time are called intractable. We cannot solve arbitrary instances in reasonable amount of time.

Huge difference between running time. Sum and composition of 2 polynomial results in polynomial .Development of extensive theory called computational complexity

Definition: II

P and NP Problems:

Class P

It is a class of decision problems that can be solved in polynomial time by deterministic algorithm, where as deterministic algorithm is every operation uniquely defined.

Decision problems:

A problem with yes/no answers called decision problem

Restriction of P to decision problems. Sensible to execute problems not solvable in polynomial time because of their large output. Eg: Subset of the given set

Many important problems that are decision problem can be reduced to a series of decision problems that are easier to study.

Undecidable problem:

Some decision problems cannot be solved at all by any algorithm.

Halting Problem:

Given a computer program and an input to it, determine whether the program halt at input or continue work indefinitely on it.

Proof:

Consider A is an Algorithm that solve halting problem.

$A[P,I]= \begin{cases} 1, \text{if P halts on input I} \\ 0, \text{P does not halt on input I} \end{cases}$

If two answers are true

There are some problems that can be solved in polynomial time(i.e) no polynomial time algorithm.
Some are decision problems that can also be solved.
Eg:

- Hamiltonian circuit problem
- Travelling salesman problem
- Knapsack problem
- Partition problem
- Bin Packing
- Graph coloring

Definition: III (A non-deterministic algorithm)
Here two stage procedure:
1. Non-deterministic *(Guessing stage)*
2. Deterministic *(Verification stage)*

A Non-deterministic polynomial means, time efficiency of its verification stage is polynomial.
Definition: IV
Class NP is the class of decision problem that can be solved by non-deterministic polynomial
algorithm. This class of problems is called non-deterministic polynomial.

P ⊆ NP

P=NP imply many hundreds of difficult combinational decision problem can be solved by polynomial time.
NP-Complete:
Many well-known decision problems known to be NP-Complete where P=NP is more doubt.
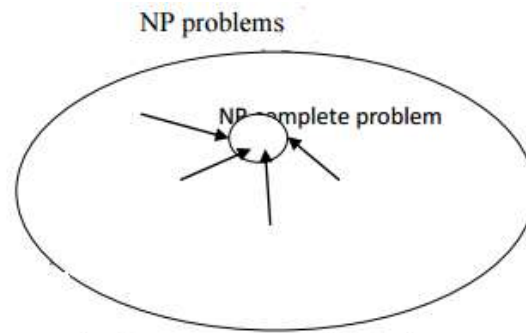Any problem can be reduced to polynomial time.
Definition: V
A decision problem D1 is ploynomially reducible to a decision problem D2. If there exists a function t,

t transforms instances D1 to D2

t map yes instances of D1 to yes instance of D2

t map no instances of D1 to no instances of D2.

D1 reduced D2

D1 → D2

(Yes) (Yes)

D1 → D2

(No) (No)

NP problems

NP-complete problem
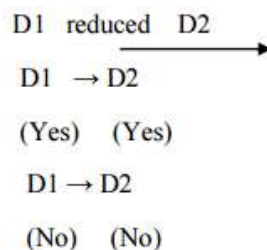
Definition: VI

A decision problem D is said to be NP Complete if it belongs to class NP.Every problem in NP is ploynomially reducible to p.

E.g.: CNF statisfiability problem

It deals with Boolean expressions. Boolean expression represented in conjunctive normal form.

Consider a expression needs 3 variables x1, x2 and x3 where ~x1, ~x2, ~x3 are negation.

(x1V~x2V~x3) & (~x1Vx2) & (~x1V~x2V~x3)
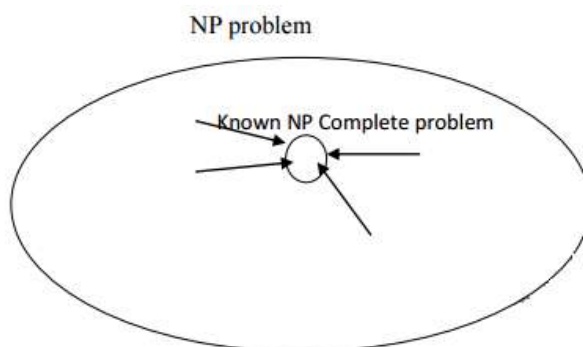
P≠NP,

Decision problem in NP-Complete:

A decision problem in NP- complete can be done in 2 ways.

1. A string is generated in polynomial time for Non-Polynomial problem. That string says whether it is possible or not to solve the problem.
2. . Every problem in NP is reducible to a problem in polynomial time

NP problem

Known NP Complete problem

6. Explain Chromatic Number decision problem:

Graph coloring:

A coloring of a graph G=(V,E) is a function f:->{1,2…..k} defined V I ∈ v.If (u.v) ∈ E,then f(u)≠f(v).

The chromatic number decision problem is to determine whether G has a coloring for a given K.

E.g.:

Red — Black — Red

Red — Black
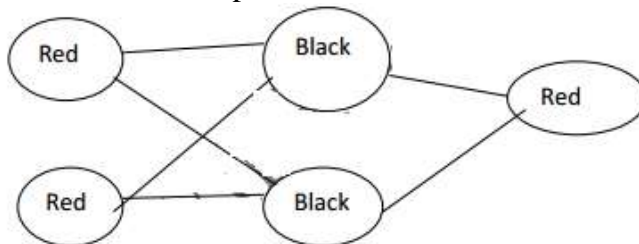
Let G=(V,E) where v={v1,v2,….vn} and the colors be positive integers, the sequential coloring strategy(sc) always colors the next vertex say vi with minimum acceptable color.

Algorithm:

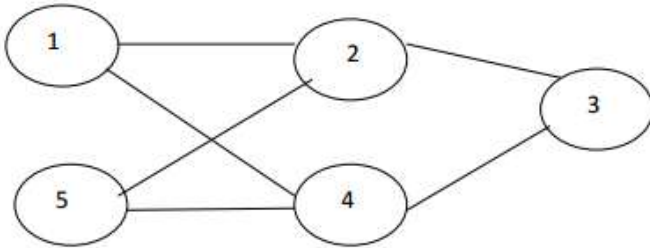Input: G= (V, E) an undirected graph where v= {v1, v2…vn}

Output: A coloring of G
Sequential coloring (V, E)
 int c,i
for(i=1; i<=n;i++)
 for(c=1; c<=n;c++)
    if no vertex adjacent to ui has color c
    color ui with c
 break I   //exit for(c)
//continue for(c)
//continue for(i)
E.g:



for(i=1;i<=5;i++)
 for(c=1;c<=5;c++)
i=1, c=1   No vertex adjacent to v1 has color i
                Color v1 with color 1
                Break
i=2, c=1   v1 is adjacent to v2 having the same color 1
                Continue for c
   C=2   No vertex adjacent to v2 having the color 2
                Color v2 with color 2
                Break
i=3, c=1   No vertex adjacent to v3 having the color 1
                color v3 with color 1
                break
i=4, c=1   v3 is adjacent to v4 having color 1
                continue for c
   c=2   No vertex adjacent to v3 having the color 2.
                Color v4 with color 2
                Break
i=5,c=1   No vertex adjacent to v5 having color 1
                color u5 with color 1
                break
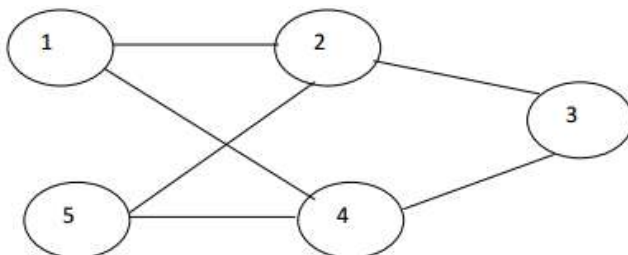
8. **Explain in detail about approximation algorithm for the Knapsack problem**

Knapsack problem:

Given n items of known weights w1, w2….wn and values v1, v2...n and a knapsack of weight capacity w,find the most valuable subset of times that fits in to the knapsack.
This problem can be solved by

- Exhaustive search
- Dynamic programming
- Branch and Bound

Greedy algorithms for the knapsack problem

- If items selected in decreasing order(weight)
  - ➢ Heavier Items may not have high value
  - ➢ No guarantee for knapsack
- Use Greedy strategy
      Vi/wi, i=1, 2...n

Algorithm based on greedy Heuristic. It is for discrete knapsack problem.
      Step1: compute $r_i = v_i / w_i$ i=1…n
      Step2: Sort items in decreasing order with ratio
      Step3: Proceed till knapsack filled

Item	weight	Value
1	7	$42
2	3	$12
3	4	$40
4	5	$25

      Capacity=10
    Compute value to weight ratio
  Sort items in decreasing order

Item	weight	value	Value/weight
1	4	$40	10
2	7	$42	6
3	5	$25	5
4	3	$12	4

1&4 items are added. Does this always true? NO

Greedy algorithm for the continuous knapsack problem:
      1. compute vi/wi, i=1,…n
      2. sort items in decreasing order
      3. Take large fraction till sack fills/no items left.

Approximation Schemes:

Approximation Sa (K)

$F (Sa (k))/f(S^*) <= 1+1/k$ for any instances of size n

Where k is integer parameter. Range $0 <= k < n$

Example: K=2

Items	weight	value	Value/weight
1	4	40	10
2	7	42	6
3	5	25	5
4	1	4	4

W=10

Solution = {1, 3, 4}

- It is theoretical than practical
- Approximating the optimal solution with any predefined accuracy level.
- Time efficiency of this algorithm is polynomial in a.
- Total no of subsets the algorithm generates before adding extra element is O (n)   time to determine the subsets possible. Thus algorithm efficiency is O.

Subset	Added item	Value
Ø	1,3,4	$69
{1}	3,4	$69
{2}	4	$46
{3}	1,4	$69
{4}	1,3	69
{1,2}	Not feasible	
{1,3}	4	69
{1,4}	3	69
{2,3}	Not feasible	1
{2,4}		46
{3,4}	1	69