Pimpri Chinchwad Education Trust's

Pimpri Chinchwad University

School of Engineering and Technology

MCA Department

Unit 01: Introduction

Introduction: An algorithm is a well-defined sequence of steps or instructions designed to perform a specific task or solve a particular problem. Algorithms are fundamental to computer science and play a crucial role in the development of software and systems. They are used to process data, perform calculations, automate reasoning, and more.

Key Characteristics of Algorithms (Conditions)

1. **Finiteness:** An algorithm must always terminate after a finite number of steps. It cannot run indefinitely.
2. **Definiteness:** Each step of the algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified.
3. **Input:** An algorithm has zero or more inputs, which are the quantities given to it initially before the algorithm starts.
4. **Output:** An algorithm produces one or more outputs, which are the quantities produced as a result of the computation.
5. **Effectiveness:** The steps of the algorithm must be basic enough that they can, in principle, be carried out by a person using only paper and pencil. It also means that the operations should be feasible and efficient in terms of time and resources.

Importance of Algorithms

1. **Efficiency:** Well-designed algorithms can solve problems faster and use fewer resources.
2. **Automation:** Algorithms enable automation of repetitive tasks, making processes faster and less error-prone.
3. **Scalability:** Efficient algorithms are essential for handling large amounts of data or complex computations, which are common in modern applications.
4. **Foundation of Programming:** Algorithms are the building blocks of programs and software. Understanding algorithms is crucial for writing efficient and effective code.

Types of Algorithms

1. **Sorting Algorithms**: Organize data in a specific order (e.g., Quick Sort, Merge Sort, Bubble Sort).
2. **Search Algorithms**: Find specific data within a dataset (e.g., Binary Search, Linear Search).
3. **Graph Algorithms**: Solve problems related to graph theory (e.g., Dijkstra's Algorithm, A* Search).
4. **Dynamic Programming Algorithms**: Solve problems by breaking them down into simpler subproblems (e.g., Fibonacci Sequence, Knapsack Problem).
5. **Divide and Conquer Algorithms**: Divide the problem into smaller parts, solve each part, and then combine the solutions (e.g., Merge Sort, Quick Sort).
6. **Greedy Algorithms**: Make the locally optimal choice at each stage with the hope of finding the global optimum (e.g., Kruskal's Algorithm, Prim's Algorithm).

Example: Euclidean Algorithm for GCD

The Euclidean algorithm is a classic method for computing the greatest common divisor (GCD) of two integers.

Algorithm: Euclidean Algorithm

Input: Two non-negative integers a and b

Output: The greatest common divisor of a and b

Steps:

1. While $b \neq 0$
   - Set temp=b
   - Set b=a mod b
   - Set a=temp
2. Return a

Pseudocode

```
function gcd (a, b):
  while b ≠ 0:
    temp = b
    b = a mod b
    a = temp
  return a
```

Explanation

- **Initialization**: The initial values of a and b are set.

- **Main Procedure**: Repeatedly apply the modulus operation and update the values of a and b until b becomes 0.
- **Termination**: When b is 0, a contains the GCD of the original input values.

Notation of an algorithm:

An algorithm notation is a way of representing an algorithm using a formal, standardized language or set of symbols. Some common algorithm notations include:

Pseudocode: A high-level, human-readable description of an algorithm using a mix of natural language and programming-like constructs, such as control structures, variable assignments, and function calls.

Flowcharts: A visual representation of an algorithm using geometric shapes (e.g., rectangles, diamonds) connected by arrows to show the flow of execution.

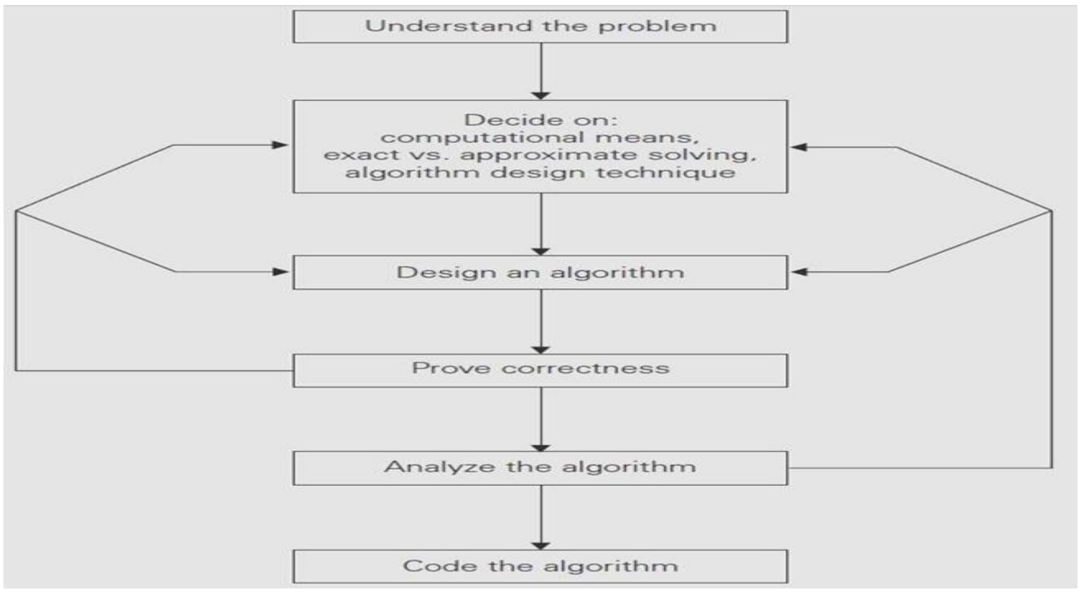
Formal mathematical notation: Algorithms can be described using mathematical symbols, logic, and set theory, allowing for more precise and rigorous representations.

Programming languages: Algorithms can be expressed directly in a programming language, such as Python, Java, or C++, providing a detailed, executable specification of the algorithm.

The choice of algorithm notation depends on the purpose and audience. Pseudocode and flowcharts are often used for high-level descriptions and communication, while formal mathematical notation and programming languages are more suitable for precise, implementation-level descriptions of algorithms.

Fundamentals of Algorithmic Problem Solving:

A sequence of steps involved in designing and analysing an algorithm is shown in the figure below.



(i) Understanding the Problem

- This is the first step in designing of algorithm.
- Read the problem's description carefully to understand the problem statement completely.
- Ask questions for clarifying the doubts about the problem.
- Identify the problem types and use existing algorithm to find solution.
- Input (instance) to the problem and range of the input get fixed.

(ii) Decision making

The Decision making is done on the following:

a) Ascertaining the Capabilities of the Computational Device

In random-access machine (RAM), instructions are executed one after another (The central assumption is that one operation at a time). Accordingly, algorithms designed to be executed on such machines are called sequential algorithms.

- In some newer computers, operations are executed concurrently, i.e., in parallel. Algorithms that take advantage of this capability are called parallel algorithms.
- Choice of computational devices like Processor and memory is mainly based on space and time efficiency

b) Choosing between Exact and Approximate Problem Solving:

- The next principal decision is to choose between solving the problem exactly or solving it approximately.
- An algorithm used to solve the problem exactly and produce correct result is called an exact algorithm.
- If the problem is so complex and not able to get exact solution, then we have to choose an algorithm called an approximation algorithm. i.e., produces an
- Approximate answer. E.g., extracting square roots, solving nonlinear equations, and evaluating definite integrals.

c) Algorithm Design Techniques

- An algorithm design technique (or "strategy" or "paradigm") is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.

- Though Algorithms and Data Structures are independent, but they are combined together to develop program. Hence the choice of proper data structure is required before designing the algorithm.
- Implementation of algorithm is possible only with the help of Algorithms and Data Structures
- Algorithmic strategy / technique / paradigm is a general approach by which many problems can be solved algorithmically. E.g., Brute Force, Divide and Conquer, Dynamic Programming, Greedy Technique and soon.

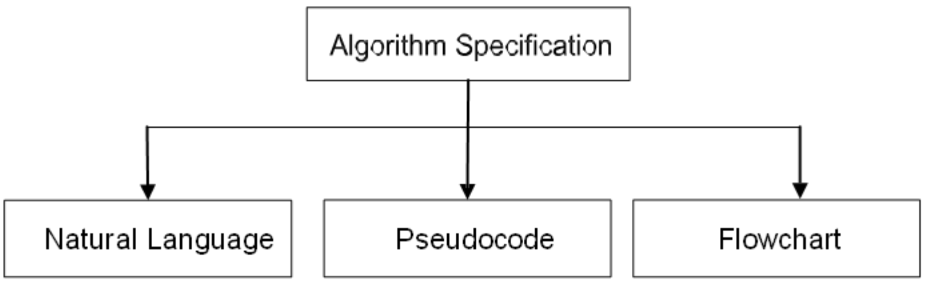
(iii) Methods of Specifying an Algorithm

There are three ways to specify an algorithm. They are:

a. Natural language

b. Pseudocode

c. Flowchart



Pseudocode and flowchart are the two options that are most widely used nowadays for specifying algorithms.

a) Natural Language

It is very simple and easy to specify an algorithm using natural language. But many times specification of algorithm by using natural language is not clear and thereby we get brief specification.

Example: An algorithm to perform addition of two numbers.

```
Step 1: Read the first number, say a.
Step 2: Read the first number, say b.
Step 3: Add the above two numbers and store the result in c.
Step 4: Display the result from c.
```

Such a specification creates difficulty while actually implementing it. Hence many programmers prefer to have specification of algorithm by means of Pseudocode.

b) Pseudocode:

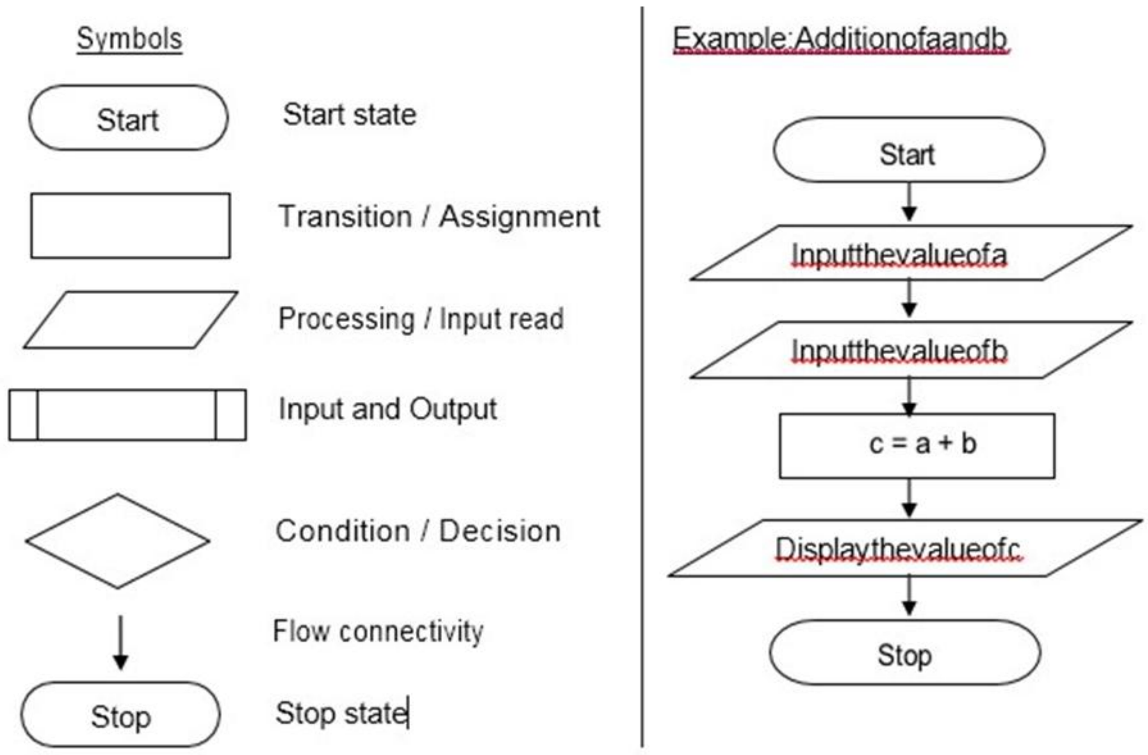
- Pseudocode is a mixture of a natural language and programming language constructs. Pseudocode is usually more precise than natural language.
- For Assignment operation left arrow "←", for comments two slashes "//",if condition, for, while loops are used.

```
ALGORITHM Sum(a,b)
        //Problem Description: This algorithm performs addition of two numbers
        //Input: Two integers a and b
        //Output: Addition of two integers
        c←a+b
        returnc
```

This specification is more useful for implementation of any language.

c) Flowchart

- In the earlier days of computing, the dominant method for specifying algorithms was a flowchart, this representation technique has proved to be inconvenient.
- Flowchart is a graphical representation of an algorithm. It is a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps



iv) Proving an Algorithm's Correctness

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

v) Analysing an Algorithm

- For an algorithm the most important is efficiency. In fact, there are two kinds of algorithm efficiency. They are:
- Time efficiency, indicating how fast the algorithm runs, and
- Space efficiency, indicating how much extra memory it uses.
- The efficiency of an algorithm is determined by measuring both time efficiency and space efficiency.
- So, factors to analyze an algorithm are:
    - Time efficiency of an algorithm
    - Space efficiency of an algorithm
    - Simplicity of an algorithm
    - Generality of an algorithm

vi) Coding an Algorithm

- The coding / implementation of an algorithm is done by a suitable programming language like C, C++, JAVA.
- The transition from an algorithm to a program can be done either incorrectly or very inefficiently. Implementing an algorithm correctly is necessary. The Algorithm power should not reduce by in efficient implementation.
- Standard tricks like computing a loop's invariant (an expression that does not change its value) outside the loop, collecting common subexpressions, replacing expensive operations by cheap ones, selection of programming language and so on should be known to the programmer.

- Typically, such improvements can speed up a program only by a constant factor, whereas a better algorithm can make a difference in running time by orders of magnitude. But once an algorithm is selected, a 10–50% speedup may be worth an effort.
- It is very essential to write an optimized code (efficient code) to reduce the burden of compiler.

Important Problem Types

Understanding different types of algorithmic problems is essential for selecting appropriate strategies and techniques to solve them. Here are some of the most important problem types in algorithm design and analysis:

1. Sorting

Sorting algorithms arrange elements in a particular order (ascending or descending).

Examples: Bubble Sort, Quick Sort, Merge Sort, Insertion Sort, Heap Sort

Applications: Data organization, search optimization, database indexing

2. Searching

Searching algorithms find the position of a specific value within a dataset.

Examples: Linear Search, Binary Search, Depth-First Search (DFS), Breadth-First Search (BFS)

Applications: Database queries, file systems, graph traversal

3. Graph Algorithms

Graph algorithms solve problems related to graph theory, such as finding the shortest path or detecting cycles.

Examples: Dijkstra's Algorithm, Bellman-Ford Algorithm, Floyd-Warshall Algorithm, Kruskal's Algorithm, Prim's Algorithm, A* Search

Applications: Network routing, social network analysis, dependency resolution

4. Dynamic Programming

Dynamic programming solves complex problems by breaking them down into simpler subproblems and storing the results of these subproblems to avoid redundant calculations.

Examples: Fibonacci Sequence, Knapsack Problem, Longest Common Subsequence (LCS), Edit Distance

Applications: Resource allocation, sequence alignment, optimization problems

5. Greedy Algorithms

Greedy algorithms make the locally optimal choice at each step with the hope of finding the global optimum.

Examples: Huffman Coding, Kruskal's Algorithm, Prim's Algorithm, Activity Selection

Applications: Compression, network design, scheduling

6. Divide and Conquer

Divide and conquer algorithms solve problems by dividing them into smaller subproblems, solving each subproblem recursively, and then combining their solutions.

Examples: Merge Sort, Quick Sort, Binary Search, Fast Fourier Transform (FFT)

Applications: Sorting, searching, signal processing

7. Backtracking

Backtracking algorithms try to build a solution incrementally and abandon a solution as soon as it determines that this solution cannot be completed.

Examples: N-Queens Problem, Sudoku Solver, Hamiltonian Path, Subset Sum Problem

Applications: Constraint satisfaction problems, combinatorial optimization, puzzles

8. Branch and Bound

Branch and bound algorithms systematically explore the decision tree by dividing the problem into subproblems (branching) and using bounds to eliminate subproblems that cannot yield better solutions than the best one found so far.

Examples: Traveling Salesman Problem (TSP), Integer Linear Programming

Applications: Combinatorial optimization, exact algorithms for NP-hard problems

9. String Matching and Manipulation

String algorithms are designed to solve problems related to string processing, such as searching, matching, and parsing.

Examples: Knuth-Morris-Pratt (KMP) Algorithm, Rabin-Karp Algorithm, Boyer-Moore Algorithm, Suffix Trees

Applications: Text search, DNA sequencing, data compression

10. Number Theory and Cryptography

Algorithms in this category are used for problems related to number theory and cryptographic applications.

Examples: Euclidean Algorithm, Sieve of Eratosthenes, RSA Algorithm, Diffie-Hellman Key Exchange

Applications: Cryptography, primality testing, public-key infrastructure

11. Geometric Algorithms

Geometric algorithms solve problems related to geometry, such as computing the convex hull or finding the closest pair of points.

Examples: Graham's Scan, Jarvis's March, Line Segment Intersection, Voronoi Diagrams

Applications: Computer graphics, geographic information systems (GIS), robotics

12. Parallel and Distributed Algorithms

These algorithms are designed to run on multiple processors or distributed systems, optimizing for parallelism and minimizing inter-process communication.

Examples: MapReduce, Parallel Sorting Algorithms, Distributed Consensus Algorithms (e.g., Paxos, Raft)

Applications: Big data processing, distributed databases, parallel computing

Fundamentals of the Analysis of Algorithm Efficiency

Analysis of the algorithm: means to investigate an algorithm's efficiency with respect to resources:

1. Space Complexity: Memory space required by an algorithm to run its completion. The space required by an algorithm is the sum of following components:

a) Fixed Part: Independent characteristics (ex. Size, number etc.) of the inputs and outputs.

This part typically include:

I) The instruction space (the space for code)

II) The space for simple variables and fixed size component variables (also called aggregates)

III) Space for constants

b) Variable Part: This part consists of

I) The space needed by component variables whose size is depends on particular instance being solved.

II) The space needed by referenced variable (to the instance that is depends on instance characteristics).

The space requirement S(p) of any algorithm 'P' is written as

S(P)=C+Sp (Instance characteristic) where 'C' is constant.

2. Time Complexity: Time complexity of a program is sum of compile time and run (execution time). The compile does not depend on instance characteristics. Hence the focus is on runtime and is denoted by tp.

The total time taken by the algorithm or program is calculated using the sum of the taken by each executable statement in an algorithm or program.

Time required by statement depends on

I) Time required for executing it once.

II) Number of times the statement is executed.

Product of equation I) and II) gives the time required by the statement.

The time complexity is measured in the following order:

1) Worst Case: The amount of time a program would take on the worst possible input configuration.

2) Average Case: The amount of time a program might be expected to take on average input data.

3) Best Case: The amount of time a program might be expected to take on best possible input data.

Experimental Studies: requires writing a program implementing the algorithm and running the program with inputs of varying size and composition. It uses a function, like the built-in clock () function, to get an accurate measure of the actual running time, then analysis is done by plotting the results.

Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used

Theoretical Analysis: It uses a high-level description of the algorithm instead of an implementation. Analysis characterizes running time as a function of the input size, n, and takes into account all possible inputs. This allows us to evaluate the speed of an algorithm independent of the hardware/software environment. Therefore, theoretical analysis can be used for analysing any algorithm.

Framework for Analysis

We use a hypothetical model with following assumptions

- Total time taken by the algorithm is given as a function on its input size
- Logical units are identified as one step
- Every step require ONE unit of time
- Total time taken = Total Num. of steps executed

Input's size: Time required by an algorithm is proportional to size of the problem instance. For e.g., more time is required to sort 20 elements than what is required to sort 10 elements.

Units for Measuring Running Time: 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 e.g., The basic operation is usually the most time-consuming operation in the algorithm’s innermost loop.

Consider the following example:

```
ALGORITHM sum_of_numbers ( A[0… n-1] )
// Functionality : Finds the Sum
// Input : Array of n numbers
// Output : Sum of ‘n’ numbers
i □ 0
sum □ 0 while i < n
sum □ sum + A[i]       n i □ i + 1
return sum
```

Total number of steps for basic operation execution, C (n) = n

NOTE: Constant of fastest growing term is insignificant: Complexity theory is an Approximation theory. We are not interested in exact time required by an algorithm to solve the problem. Rather we are interested in order of growth. i.e

- How much faster will algorithm run on computer that is twice as fast?
- How much longer does it take to solve problem of double input size?

We can crudely estimate running time by

$T(n) \approx C_{op} * C(n)$

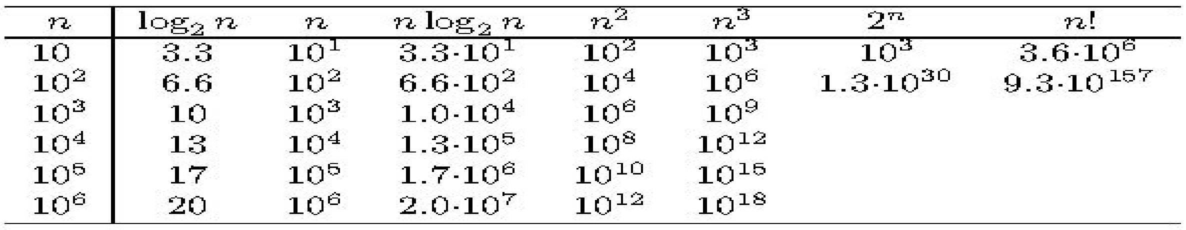
Where,

T (n): running time as a function of n.

Cop : running time of a single operation.

C (n): number of basic operations as a function of n.

Order of Growth: For order of growth, consider only the leading term of a formula and ignore the constant coefficient. The following is the table of values of several functions important for analysis of algorithms.



n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Asymptotic Notations:

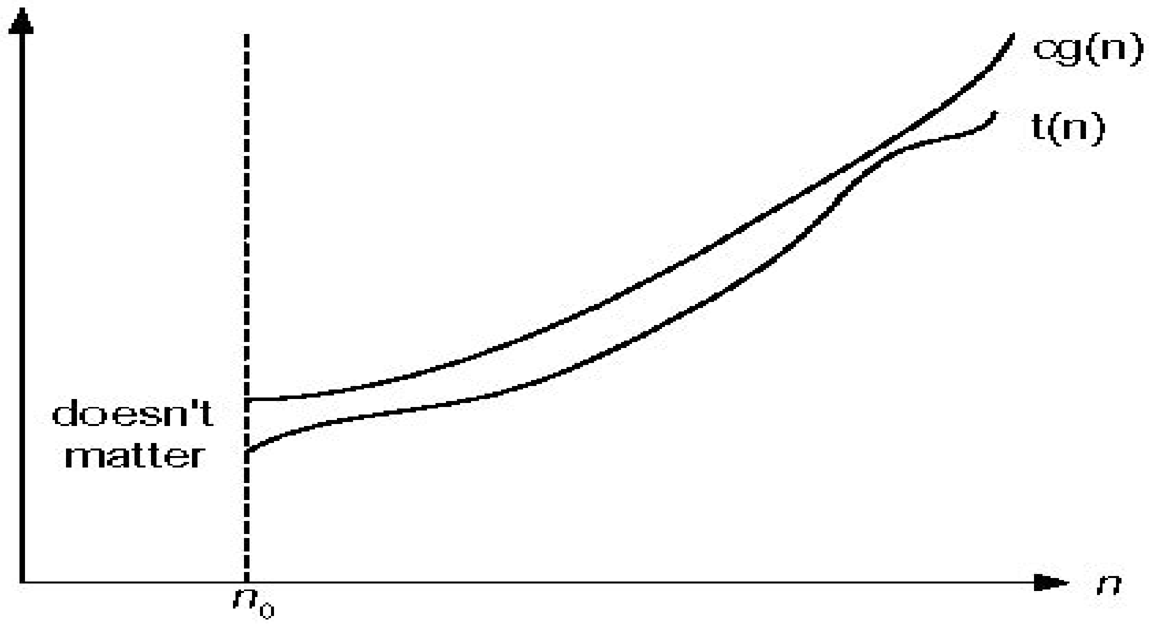
Asymptotic notation is a way of comparing functions that ignores constant factors and small input sizes. Three notations used to compare orders of growth of an algorithm's basic operation count are: O, Ω, Θ notations

Big Oh- O notation

Definition:

A function t(n) is said to be in O(g(n)), denoted t(n)∈O(g(n)), if t(n) is bounded above by some constant multiple of g(n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer n0 such that

t(n) ≤cg(n) for all n ≥ n0

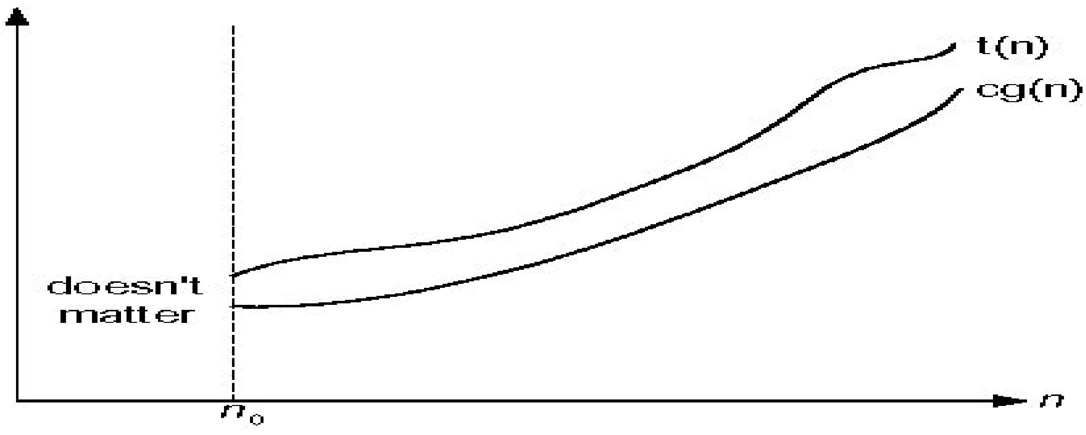


Big-oh notation: $t(n) \in O(g(n))$

Big Omega- Λ notation

Definition: A function t (n) is said to be in Λ (g(n)), denoted t(n) ∈ Λ (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 n0 such that **t(n) ≥ cg(n) for all n ≥ n0**

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



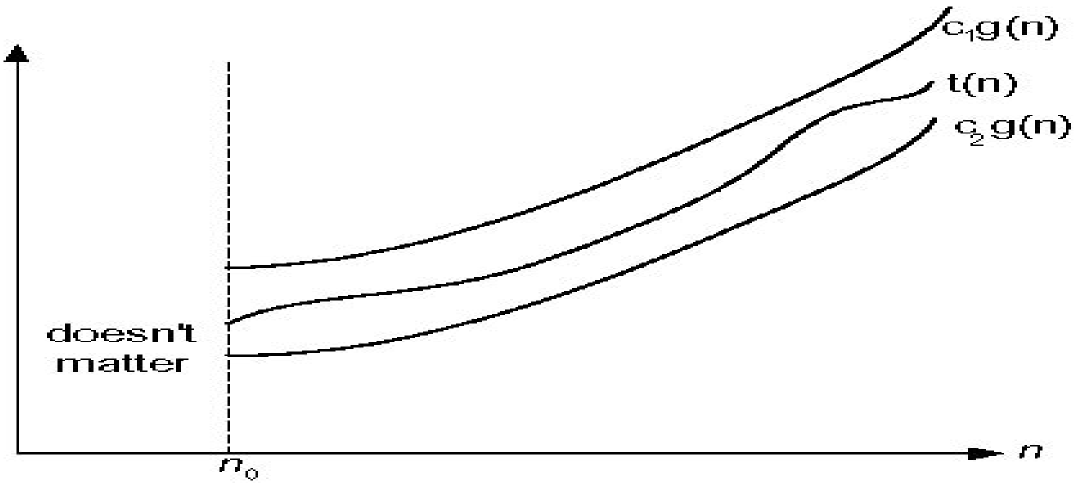
Big Theta- Θ notation

Definition:

A function t (n) is said to be in Θ (g (n)), denoted **t(n) ∈ Θ (g (n)),** if t (n) is bounded both above and below by some constant multiple of g (n) for all large n, i.e., if there exist some positive constant c1 and c2 and some nonnegative integer n0 such that

c2 g (n) ≤ t (n) ≤ c1 g (n) for all n ≥ n0

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



Basic Efficiency Classes: The time efficiencies of a large number of algorithms fall into only a few classes

1	constant
log *n*	logarithmic
n	linear
n log *n*	*n* log *n*
n^2	quadratic
n^3	cubic
2^n	exponential
n!	factorial

Mathematical analysis (Time Efficiency) of Non-recursive Algorithms

Mathematical analysis of the time efficiency of non-recursive algorithms involves determining how the running time of an algorithm grows with the size of the input. This typically involves the following steps:

1. **Identify the Input Size:** Define the input size, typically denoted as nnn. This could be the number of elements in an array, the number of vertices and edges in a graph, etc.
2. **Basic Operations:** Determine the basic operations that significantly contribute to the running time of the algorithm. These could be comparisons, swaps, arithmetic operations, etc.
3. **Counting Operations:** Count the number of times the basic operations are executed as a function of the input size nnn.
4. **Formulate the Running Time:** Express the total number of basic operations as a function of nnn, resulting in a time complexity function T(n)T(n)T(n).
5. **Big-O Notation:** Use Big-O notation to classify the time complexity, which provides an upper bound on the growth rate of T(n)T(n)T(n).

Let's go through an example with a simple non-recursive algorithm: Bubble Sort.

Bubble Sort Analysis

Bubble Sort Algorithm:

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
```

```
        arr[j], arr[j+1] = arr[j+1], arr[j]
```

1. **Input Size:** The input size nnn is the number of elements in the array arr.
2. **Basic Operations:** The basic operation here is the comparison arr[j] > arr[j+1].
3. **Counting Operations:**
   - The outer loop runs nnn times.
   - The inner loop runs n−i−1times for each iteration of the outer loop.

4. **Formulate the Running Time:** T(n)=n2−n/2
5. **Big-O Notation:** T(n)=O(n2)

Thus, the time complexity of Bubble Sort is O(n2) indicating that its running time grows quadratically with the size of the input.

General Steps for Analysing Non-recursive Algorithms

1. **Identify Loops:** Determine the number of times loops execute.
2. **Nested Loops:** Analyze the running time of nested loops by multiplying the complexities of the loops.
3. **Conditional Statements:** Include the time taken by conditional statements.
4. **Aggregate Operations:** Sum up the time taken by different parts of the algorithm.
5. **Worst-Case, Average-Case, Best-Case:** Analyze the algorithm in different scenarios, though often the worst-case analysis is sufficient.

Example: Linear Search

Linear Search Algorithm:

```
def linear_search (arr, target):
    for i in range (len(arr)):
        if arr[i] == target:
            return i
    return -1
```

1. **Input Size:** The input size n is the number of elements in the array **'arr'.**
2. **Basic Operations:** The basic operation here is the comparison **arr [i] == target.**
3. **Counting Operations:** In the worst case, the loop runs n times.
4. **Formulate the Running Time:** T(n)=n
5. **Big-O Notation:** T(n)=O(n)

Thus, the time complexity of Linear Search is O(n)O(n)O(n), indicating that its running time grows linearly with the size of the input.

Mathematical analysis (Time Efficiency) of recursive Algorithms

General plan for analyzing efficiency of recursive algorithms:

1. Decide on parameter n indicating input size
2. Identify algorithm's basic operation
3. 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.
4. Set up recurrence relation, with an appropriate initial condition, for the number of times the algorithm's basic operation is executed.
5. Solve the recurrence.

Example: Factorial function

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

Analysis:

1. Input size: given number = n
2. Basic operation: multiplication
3. NO best, worst, average cases.
4. Let M (n) denotes number of multiplications.

$$M(n) = M(n-1) + 1 \qquad \text{for } n > 0$$
$$M(0) = 0 \qquad \textit{initial condition}$$

Where: M (n – 1) : to compute Factorial (n – 1)
1 :to multiply Factorial (n – 1) by n

5. Solve the recurrence: Solving using *"Backward substitution method":*

$$\begin{aligned} M(n) &= M(n-1) + 1 \\ &= [\, M(n-2) + 1] + 1 \\ &= M(n-2) + 2 \\ &= [\, M(n-3) + 1] + 3 \\ &= M(n-3) + 3 \\ &\ldots \end{aligned}$$

In the ith recursion, we haveWhen i = n, we have

Since M (0) = 0

$= M(n - i) + i$

$= M(n - n) + n = M(0) + n$

$= n$

$$\mathbf{M}(n) \in \Theta(\mathbf{n})$$

Properties of Asymptotic Notation

Asymptotic notation has several important properties that make it useful in the analysis of algorithms.

- **Reflexivity:** The first property is reflexivity. This property states that a function is always O(f(n)), Ω(f(n)), and Θ(f(n)) of itself. In other words, every function is a subset of itself.
- **Transitivity:** The second property is transitivity. This property states that if f(n) is O(g(n)) and g(n) is O(h(n)), then f(n) is O(h(n)). In other words, if a function is bounded above by another function, and that function is bounded above by a third function, then the first function is also bounded above by the third function.
- **Symmetry:** The third property is symmetry. This property states that if f(n) is Θ(g(n)), then g(n) is Θ(f(n)). In other words, if a function is both O(g(n)) and Ω(g(n)), then it is also Θ(g(n)).
- **Transpose Symmetry:** The fourth property is transpose symmetry. This property states that if f(n) is O(g(n)), then g(n) is Ω(f(n)), and if f(n) is Ω(g(n)), then g(n) is O(f(n)). In other words, if a function is bounded above by another function, then that function is bounded below by the first function.
- **Additivity:** The fifth property is additivity. This property states that if f(n) is O(h(n)) and g(n) is O(h(n)), then f(n) + g(n) is also O(h(n)). In other words, if two functions are bounded above by the same function, then their sum is also bounded above by that function.

The Analysis Framework

1. Measuring an Input's Size: Let's start with the obvious observation that almost all algorithms run longer on larger inputs. For example, it takes longer to sort larger arrays, multiply larger matrices, and so on. Therefore, it is logical to investigate an algorithm's efficiency as a function of some parameter n indicating the algorithm's input size.

2. Units for Measuring Running Time: The next issue concerns units for measuring an algorithm's running time. Of course, we can simply use some standard unit of time measurement—a second, or millisecond, and so on—to measure the running time of a program implement-ing the algorithm. There are obvious drawbacks to such an approach, however: dependence on the speed of a particular computer, dependence on the quality of a program implementing the algorithm and of the compiler used in generating the machine code, and the difficulty of clocking the actual running time of the pro-gram. Since we are after a measure of an algorithm's efficiency, we would like to have a metric that does not depend on these extraneous factors.

3. Orders of Growth: Why this emphasis on the count's order of growth for large input sizes? A differ-ence in running times on small inputs is not what really distinguishes efficient algorithms from inefficient ones. When we must compute, for example, the greatest common divisor of two small numbers, it is not immediately clear how much more efficient Euclid's algorithm is compared to the other two algorithms discussed previously or even why we should care which of them is faster and by how much.

4. Worst-Case, Best-Case, and Average-Case Efficiencies: In the beginning of this section, we established that it is reasonable to measure an algorithm's efficiency as a function of a parameter indicating the size of the algorithm's input. But there are many algorithms for which running time depends not only on an input size but also on the specifics of a particular input. Consider, as an example, sequential search.

5. Recapitulation of the Analysis Framework: Both time and space efficiencies are measured as functions of the algorithm's input size.

Time efficiency is measured by counting the number of times the algorithm's basic operation is executed. Space efficiency is measured by counting the number of extra memory units consumed by the algorithm.

The efficiencies of some algorithms may differ significantly for inputs of the same size. For such algorithms, we need to distinguish between the worst-case, average-case, and best-case efficiencies.

EXAMPLE 1: Consider the problem of finding the value of **the largest element in a list of n numbers**. Assume that the list is implemented as an array for simplicity.

```
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
```

Algorithm analysis

- The measure of an input's size here is the number of elements in the array, i.e., n.
- There are two operations in the for loop's body:
    - The comparison A[i]> maxval and
    - The assignment maxval←A[i].

The comparison operation is considered as the algorithm's basic operation, because the comparison is executed on each repetition of the loop and not the assignment.

- The number of comparisons will be the same for all arrays of size n; therefore, there is no need to distinguish among the worst, average, and best cases here.
- Let C(n) denotes the number of times this comparison is executed. The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable i within the bounds 1 and n − 1, inclusive. Therefore, the sum for C(n) is calculated as follows:

PECT's PCU

School of Engineering and Technology

Department MCA/BCA/BSc (CS)

Unit 02. Brute Force and Divide and Conquer

Introduction:

What is the Brute Force Algorithm?

A brute force algorithm is a simple, comprehensive search strategy that systematically explores every option until a problem's answer is discovered. It's a generic approach to problem-solving that's employed when the issue is small enough to make an in-depth investigation possible. However, because of their high temporal (time based) complexity, brute force techniques are inefficient for large-scale issues.

Key takeaways:

- **Methodical Listing:** Brute force algorithms investigate every potential solution to an issue, usually in an organized and detailed way. This involves attempting each option in a specified order.
- **Relevance:** When the issue space is small and easily explorable in a fair length of time, brute force is the most appropriate method. The temporal complexity of the algorithm becomes unfeasible for larger issue situations.
- **Not using optimization or heuristics:** Brute force algorithms don't use optimization or heuristic approaches. They depend on testing every potential outcome without ruling out any using clever pruning or heuristics.

Features of the brute force algorithm

- It is an intuitive, direct, and straightforward technique of problem-solving in which all the possible ways or all the possible solutions to a given problem are enumerated.
- Many problems are solved in day-to-day life using the brute force strategy, for example, exploring all the paths to a nearby market to find the minimum shortest path.
- Arranging the books in a rack using all the possibilities to optimize the rack spaces, etc.
- Daily life activities use a brute force nature, even though optimal algorithms are also possible.

Examples:

1. Computing an: a * a * a * … * a (n times)
2. Computing n!: The n! can be computed as n*(n-1)* … *3*2*1
3. Multiplication of two matrices: C=AB

4. Searching a key from list of elements (Sequential search)

Pros And Cons of Brute Force Algorithm:

Pros:

- The brute force approach is a guaranteed way to find the correct solution by listing all the possible candidate solutions for the problem.
- It is a generic method and not limited to any specific domain of problems.
- The brute force method is ideal for solving small and simpler problems.
- It is known for its simplicity and can serve as a comparison benchmark.
- The brute-force approach yields reasonable algorithms of at least some practical value with no limitation on instance size for sorting, searching, and string matching.

Cons:

- The brute force approach is inefficient. For real-time problems, algorithm analysis often goes above the O(N!) order of growth.
- This method relies more on compromising the power of a computer system for solving a problem than on a good algorithm design.
- Brute force algorithms are slow.
- Brute force algorithms are not constructive or creative compared to algorithms that are constructed using some other design paradigms.

Computing and String-Matching Algorithm-The Brut Force Method

The brute force approach is the simplest string-matching algorithm. It involves comparing the pattern with every substring of the text until a match is found. This algorithm has a time complexity of O (mn), where 'm' is the length of the pattern and 'n' is the length of the text. Although straightforward, it becomes inefficient for large texts or patterns.

How the Brute-Force String-Matching Algorithm Works

1. **Start at the beginning of the text:** The algorithm begins by aligning the pattern with the first character of the text.
2. **Compare each character:** Compare each character of the pattern to the corresponding character in the text.
    - If all characters match, the pattern has been found at that position.
    - If a mismatch occurs, the algorithm moves the pattern one position to the right in the text.

3. **Repeat:** This process is repeated until the pattern is found or all possible positions in the text have been checked.

Pseudocode for Brute-Force String Matching in C programming

```
int brute_force_string_match (char text [], char pattern [])
{
        int n = strlen(text);     // Length of the text
        int m = strlen(pattern);  // Length of the pattern
         // Loop through each position in the text where the pattern could fit
         for (int i = 0; i <= n - m; i++)
                {
                        int j = 0;
                        // Check if the pattern matches the text starting at position i
                        while (j < m && text[i + j] == pattern[j])
                        {
                                j++;
                        }
                        // If the pattern matches entirely, return the starting index
                        if (j == m)
                        {
                                return i;
                        }
                }
                // If no match is found, return -1
                return -1;
}
```

Example

Suppose you want to find the pattern "ABCD" in the text "ABC ABCDAB ABCDABCDABDE".

1. Start by aligning "ABCD" with the first four characters of the text.
2. Compare "ABCD" with "ABCA". There is a mismatch on the fourth character.
3. Move the pattern one position to the right and repeat the comparison.
4. Continue this process until you find a match or exhaust all possibilities.

Use Cases

- The brute-force approach is best suited for small-scale problems or scenarios where the text and pattern sizes are small.
- It serves as a foundational concept for more advanced string-matching algorithms like the Knuth-Morris-Pratt (KMP) or Boyer-Moore algorithms, which are optimized for efficiency.

Selection Sort Algorithm

Selection Sort is a simple comparison-based sorting algorithm. It works by repeatedly selecting the smallest (or largest, depending on the sorting order) element from the unsorted portion of the list and swapping it with the first unsorted element. This process is repeated until the entire list is sorted.

How Selection Sort Works

1. **Start with the first element:**
   - Assume the first element is the minimum.
2. **Find the minimum:**
   - Traverse the rest of the array to find the actual minimum element.
3. **Swap:**
   - Swap the found minimum element with the first element.
4. **Repeat:**
   - Move the boundary between the sorted and unsorted parts of the array one position to the right and repeat the process for the remaining unsorted elements.

Pseudocode for Selection Sort in C

```
void selectionSort(int array[], int n)
{
   int i, j, min_index, temp;
   // Loop through each element in the array (except the last one)
   for (i = 0; i < n-1; i++)
{
      // Assume the current element is the minimum
      min_index = i;
      // Find the index of the minimum element in the remaining unsorted part
      for (j = i + 1; j < n; j++)
 {
         if (array[j] < array[min_index])
{
```

```
                min_index = j;
            }
        }
        // Swap the found minimum element with the first element
        if (min_index != i)
{
            temp = array[i];
            array[i] = array[min_index];
            array[min_index] = temp;
        }
    }
}
```

Explanation:

1. **Outer Loop (for (i = 0; i < n-1; i++)):**
   - This loop iterates through each element in the array, assuming that the current element (array[i]) is the minimum.
2. **Inner Loop (for (j = i + 1; j < n; j++)):**
   - This loop checks the remaining elements to find the actual minimum element in the unsorted portion of the array. If a smaller element is found, min_index is updated to the index of this smaller element.
3. **Swap (if (min_index != i)):**
   - After the inner loop completes, the smallest element is swapped with the element at the current position (i), ensuring that the smallest element of the unsorted portion is placed at the beginning of the unsorted portion.
4. **End of Sort:**
   - The process continues until the entire array is sorted.

Key Points:

- **In-Place:** The algorithm sorts the array without needing additional space.
- **Time Complexity:** The time complexity of this algorithm is O(n2) O(n^2) O(n2), making it inefficient for large datasets.
- **Unstable:** It is not a stable sort, meaning equal elements might not retain their original relative order.

Closest-Pair and Convex-Hull Problems

We consider a straight forward approach (Brute Force) to two well-known problems dealing with a finite set of points in the plane. These problems are very useful in important applied areas like computational geometry and operations research.

Closest-Pair

The Closest Pair Problem is a fundamental problem in computational geometry. The goal is to find the two points in a set of points that are closest to each other. This problem can be solved in different ways, depending on the desired trade-off between simplicity and efficiency.

Problem Statement

Given a set of n points in a 2D plane, the task is to find the pair of points with the smallest Euclidean distance between them.

Naive Approach (Brute Force)

The brute-force approach for solving the Closest Pair Problem involves checking the distance between every pair of points in each set and finding the pair with the smallest distance. This approach has a time complexity of O(n2) where n is the number of points.

Brute-Force Closest Pair Algorithm - Pseudocode in C

```
#include <stdio.h>
#include <math.h>
#include <float.h>
// Define a structure for a Point
struct Point
{
   int x, y;
};
// Function to calculate the distance between two points
float distance(struct Point p1, struct Point p2)
{
   return sqrt((p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y));
}
// Function to find the closest pair of points using brute-force
float bruteForceClosestPair(struct Point points[], int n) {
   float min_dist = FLT_MAX;  // Initialize minimum distance to a large value
   // Loop through each pair of points and compute their distance
   for (int i = 0; i < n; ++i)
{
        for (int j = i + 1; j < n; ++j)
        {
            float dist = distance(points[i], points[j]);
            if (dist < min_dist) {
                min_dist = dist;  // Update min_dist if a smaller distance is found
            }
        }
    }
        return min_dist;  // Return the smallest distance found
}
int main()
{
    struct Point points[] =
    { {2, 3}, {12, 30}, {40, 50}, {5, 1}, {12, 10}, {3, 4} };
    int n = sizeof(points) / sizeof (points[0]);
   // Call the brute-force function and print the result
    Printf ("The smallest distance is %f\n", bruteForceClosestPair(points, n));
    return 0;
}
```

Explanation:

1. **Point Structure:**
    - A 'Point' structure is defined with 'x' and 'y' coordinates to represent points in a 2D space.
2. **Distance Function:**
    - The distance function calculates the Euclidean distance between two points p1 and p2 using the formula:
    - distance= $\sqrt{(p1.x - p2.x)2 + (p1.y - p2.y)2}$
3. **Brute-Force Closest Pair Function:**
    - The 'bruteForceClosestPair' function iterates over all possible pairs of points in the array.
    - It computes the distance for each pair and keeps track of the minimum distance encountered.

4. **Main Function:**
    - The main function demonstrates how to use the 'bruteForceClosestPair' function. It defines an array of points, calculates the smallest distance using the brute-force method, and then prints the result.

Time Complexity

- **Time Complexity:** O(n2), where n is the number of points. This is because the algorithm checks every possible pair of points, and the number of pairs is n(n−1)/2, which is O(n2).
- **Space Complexity:** O (1), as the algorithm uses only a constant amount of extra space.

Closest-Pair Problem

Euclidean distance $d(P_i, P_j) = \sqrt{[(x_i-x_j)^2 + (y_i-y_j)^2]}$

Find the minimal distance between a pair in a set of points

```
Algorithm BruteForceClosestPoints(P)
// P is list of points
dmin ← ∞
for i ← 1 to n-1 do
for j ← i+1 to n do
d ← sqrt((x_i-x_j)^2 + (y_i-y_j)^2)
if d < dmin then
dmin ← d; index1 ← i; index2 ← j
return index1, index2
```

Note the algorithm does not have to calculate the square root

Then the basic operation is squaring

$C(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 2 = 2\sum_{j=i+1}^{n} (n-i) = 2n(n-1)/2 \in \Theta(n^2)$

Convex-Hull Problem

The Convex Hull Algorithm is used to find the convex hull of a set of points in computational geometry. The convex hull is the smallest convex set that encloses all the points, forming a convex polygon. This algorithm is important in various applications such as image processing, route planning, and object modelling.

Convex Set:

A set of points (finite or infinite) in the plane is called ***convex*** if for any two points *p* and *q* in the set, the entire line segment with the endpoints at *p* and *q* belongs to the set.

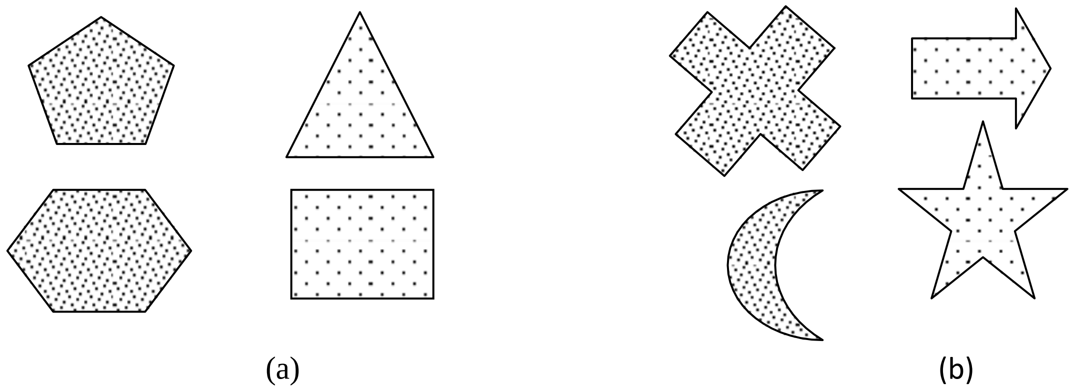


FIGURE 2.1 (a) Convex sets. (b) Sets that are not convex.

- All the sets depicted in Figure 2.1 (a) are convex, and so are a straight line, a triangle, a rectangle, and, more generally, any convex polygon, a circle, and the entire plane.
- On the other hand, the sets depicted in Figure 2.1 (b), any finite set of two or more distinct points, the boundary of any convex polygon, and a circumference are examples of sets that are not convex.
- Take a rubber band and stretch it to include all the nails, then let it snap into place. The convex hull is the area bounded by the snapped rubber band as shown in Figure 2.2

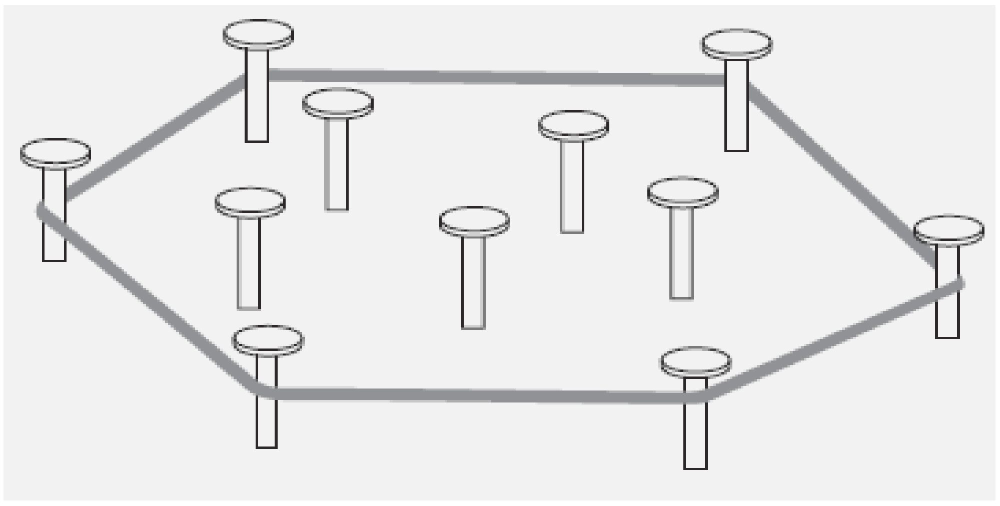


FIGURE 2.2 Rubber-band interpretation of the convex hull.

Convex hull

The convex hull of a set S of points is the smallest convex set containing S. (The smallest convex hull of S must be a subset of any convex set containing S.)

If S is convex, its convex hull is obviously S itself. If S is a set of two points, its convex hull is the line segment connecting these points. If S is a set of three points not on the same line, its convex hull is the triangle with the vertices at the three points given; if the three points do lie on the same line, the convex hull is the line segment with its endpoints at the two points that are farthest apart. For an example of the convex hull for a larger set, see Figure 2.3.

THEOREM

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

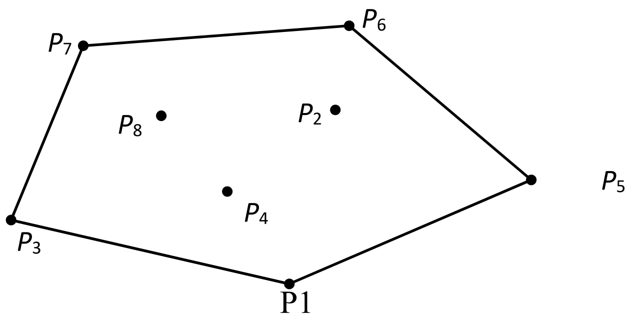


FIGURE 2.3 The convex hull for this set of eight points is the convex polygon with vertices at p1, p5, p6, p7, and p3.

The convex-hull problem is the problem of constructing the convex hull for a given set S of n points. To solve it, we need to find the points that will serve as the vertices of the polygon in question. Mathematicians call the vertices of such a polygon “extreme points.” By definition, an extreme point of a convex set is a point of this set that is not a middle point of any line segment with endpoints in the set. For example, the extreme points of a triangle are its three vertices, the extreme points of a circle are all the points of its circumference, and the extreme points of the convex hull of the set of eight points in Figure 2.3 are p1, p5, p6, p7, and p3.

Application

Extreme points have several special properties other points of a convex set do not have. One of them is exploited by the simplex method, This algorithm solves linear programming Problems.

We are interested in extreme points because their identification solves the convex-hull problem. Actually, to solve this problem completely, we need to know a bit more than just which of n points of a given set are extreme points of the set’s convex hull. we need to know which pairs of points need to be connected to form the boundary of the convex hull. Note that this issue can also be addressed by listing the extreme points in a clockwise or a counterclockwise order.

We can solve the convex-hull problem by brute-force manner. The convex hull problem is one with no obvious algorithmic solution. there is a simple but inefficient algorithm that is based on the following observation about line segments making up the boundary of a convex hull: a line segment connecting two points pi and pj of a set of n points is a part of the convex hull's boundary if and only if all the other points of the set lie on the same side of the straight line through these two points. Repeating this test for every pair of points yields a list of line segments that make up the convex hull's boundary.

Facts

A few elementary facts from analytical geometry are needed to implement the above algorithm.

- First, the straight line through two points (x1, y1), (x2, y2) in the coordinate plane can be defined by the equation ax + by = c, where a = y2 − y1, b = x1 − x2, c = x1y2 − y1x2.
- Second, such a line divides the plane into two half-planes: for all the points in one of them, ax + by > c, while for all the points in the other, ax + by < c. (For the points on the line itself, of course, ax + by = c.) Thus, to check whether certain points lie on the same side of the line, we can simply check whether the expression ax + by − c has the same sign for each of these points.

Time efficiency of this algorithm.

Time efficiency of this algorithm is in O(n3): for each of n (n − 1)/2 pairs of distinct points, we may need to find the sign of ax + by – c for each of the other n − 2 points.

Bubble Sort

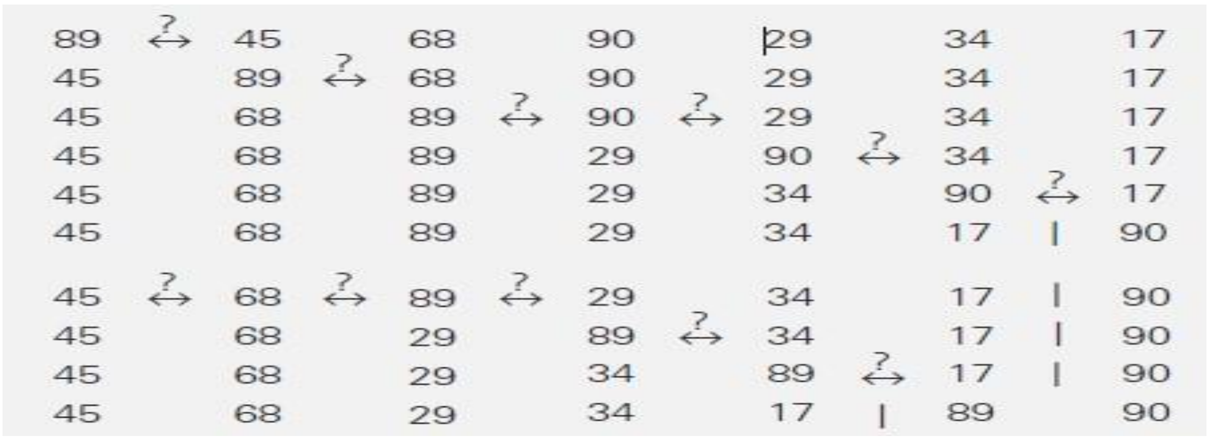
The bubble sorting algorithm is to compare adjacent elements of the list and exchange them if they are out of order. By doing it repeatedly, we end up "bubbling up" the largest element to the last position on the list. The next pass bubbles up the second largest element, and so on, until aftern − 1 passes the list is sorted. Pass i (0 ≤ i ≤ n − 2) of bubble sort can be represented by the

$$\overset{?}{}$$

following: $A_0, \dots, A_j \leftrightarrow A_{j+1}, \dots, A_{n-i-1} \mid A_{n-i} \leq \dots \leq A_{n-1}$

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

The action of the algorithm on the list 89, 45, 68, 90, 29, 34, 17 is illustrated as an example.



89	$\overset{?}{\leftrightarrow}$	45		68		90		29		34		17
45		89	$\overset{?}{\leftrightarrow}$	68		90		29		34		17
45		68		89	$\overset{?}{\leftrightarrow}$	90	$\overset{?}{\leftrightarrow}$	29		34		17
45		68		89		29		90	$\overset{?}{\leftrightarrow}$	34		17
45		68		89		29		34		90	$\overset{?}{\leftrightarrow}$	17
45		68		89		29		34		17	|	90
45	$\overset{?}{\leftrightarrow}$	68	$\overset{?}{\leftrightarrow}$	89	$\overset{?}{\leftrightarrow}$	29		34		17	|	90
45		68		29		89	$\overset{?}{\leftrightarrow}$	34		17	|	90
45		68		29		34		89	$\overset{?}{\leftrightarrow}$	17	|	90
45		68		29		34		17	|	89		90

The number of key comparisons for the bubble-sort version given above is the same for all arrays of size n; it is obtained by a sum that is almost identical to the sum for selection sort: The number of key swaps, however, depends on the input. In the worst case of decreasing arrays, it is the same as the number of key comparisons.

Exhaustive Search

For discrete problems in which no efficient solution method is known, it might be necessary to test each possibility sequentially in order to determine if it is the solution. Such ***exhaustive*** examination of all possibilities is known as ***exhaustive search, complete search or* direct *search.***

Exhaustive search is simply a brute force approach to combinatorial problems (Minimization or maximization of optimization problems and constraint satisfaction problems).

Reason to choose brute-force / ***exhaustive search*** approach as an important algorithm design strategy

1. First, unlike some of the other strategies, brute force is applicable to a very wide variety of problems. In fact, it seems to be the only **general approach** for which it is

more difficult to point out problems it *cannot* tackle.

2. Second, for some important problems, e.g., sorting, searching, matrix multiplication, string matching the brute-force approach yields reasonable algorithms of at least some practical value **with no limitation on instance size**.
3. Third, the expense of designing a more efficient algorithm may be unjustifiable if only a few instances of a problem need to be solved and a brute-force algorithm can solve those instances with **acceptable speed**.
4. Fourth, even if too **inefficient** in general, a brute-force algorithm can still be **useful for solving small-size instances** of a problem.

Exhaustive Search is applied to the important problems like

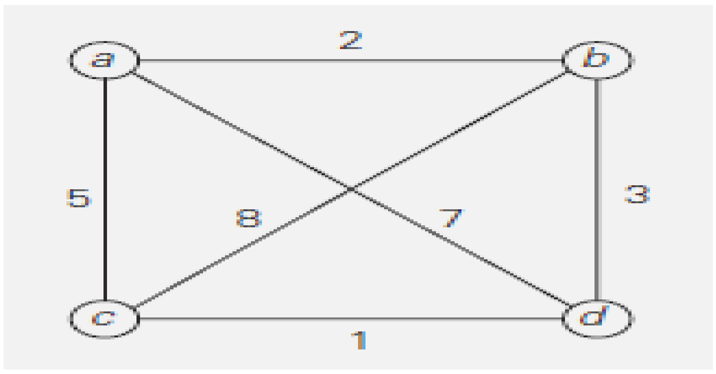
- Traveling Salesman Problem
- Knapsack Problem
- Assignment Problem.

Traveling Salesman Problem

The traveling salesman problem (TSP) is one of the combinatorial problems. The problem asks to find the shortest tour through a given set of n cities that visits each city exactly once before returning to the city where it started.

The problem can be conveniently modelled by a weighted graph, with the graph's vertices representing the cities and the edge weights specifying the distances. Then the problem can be stated as the problem of finding the shortest Hamiltonian circuit of the graph. (A Hamiltonian circuit is defined as a cycle that passes through all the vertices of the graph exactly once).

A Hamiltonian circuit can also be defined as a sequence of n + 1 adjacent vertices vi0, vi1, . . . , vin−1, vi0, where the first vertex of the sequence is the same as the last one and all the other n − 1 vertices are distinct. All circuits start and end at one particular vertex. Figure 2.4 presents a small instance of the problem and its solution by this method.



Tour	**Length**
a ---> b ---> c ---> d ---> a	I = 2 + 8 + 1 + 7 = 18
a ---> b ---> d ---> c ---> a	**I = 2 + 3 + 1 + 5 = 11 optimal**
a ---> c ---> b ---> d ---> a	I = 5 + 8 + 3 + 7 = 23
a ---> c ---> d ---> b ---> a	**I = 5 + 1 + 3 + 2 = 11 optimal**
a ---> d ---> b ---> c ---> a	I = 7 + 3 + 8 + 5 = 23
a ---> d ---> c ---> b ---> a	I = 7 + 1 + 8 + 2 = 18

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

Time efficiency

- We can get all the tours by generating all the permutations of n − 1 intermediate cities from a particular city i.e. **(n - 1)!**
- Consider two intermediate vertices, say, *b* and *c*, and then only permutations in which *b* precedes *c*. (This trick implicitly defines a tour's direction.)
- An inspection of Figure 2.4 reveals three pairs of tours that differ only by their direction. Hence, we could cut the number of vertex permutations by **half** because cycle total lengths in both directions are same. 1
- The total number of permutations needed is still 1/2 $(n-1)!$, which makes the exhaustive-search approach impractical for large n. It is useful for very small values of *n*.

Example:

Cities: A, B, C, D

Distances between cities (in km):

- A to B: 10
- A to C: 15
- A to D: 20
- B to C: 35
- B to D: 25
- C to D: 30

Step 1: List All Possible Routes

Since there are 4 cities, the number of possible routes is given by the factorial of (n-1) cities, as the last city returns to the starting point:

Number of routes = (4-1)! = 3! = 6 routes.

Here are the possible routes:

1. A → B → C → D → A
2. A → B → D → C → A
3. A → C → B → D → A
4. A → C → D → B → A
5. A → D → B → C → A
6. A → D → C → B → A

Step 2: Calculate the Distance for Each Route

Let's calculate the total distance for each route:

1. **A → B → C → D → A**:
   - A → B = 10
   - B → C = 35
   - C → D = 30
   - D → A = 20
   - Total = 10 + 35 + 30 + 20 = 95 km
2. **A → B → D → C → A**:
   - A → B = 10
   - B → D = 25
   - D → C = 30
   - C → A = 15
   - Total = 10 + 25 + 30 + 15 = 80 km
3. **A → C → B → D → A**:
   - A → C = 15
   - C → B = 35 (same as B → C)
   - B → D = 25
   - D → A = 20
   - Total = 15 + 35 + 25 + 20 = 95 km
4. **A → C → D → B → A**:
   - A → C = 15
   - C → D = 30
   - D → B = 25 (same as B → D)
   - B → A = 10
   - Total = 15 + 30 + 25 + 10 = 80 km
5. **A → D → B → C → A**:
   - A → D = 20

    - D → B = 25
    - B → C = 35
    - C → A = 15
    - Total = 20 + 25 + 35 + 15 = 95 km

6. **A → D → C → B → A**:
    - A → D = 20
    - D → C = 30
    - C → B = 35
    - B → A = 10
    - Total = 20 + 30 + 35 + 10 = 95 km

Step 3: Identify the Shortest Route

From the calculated distances, the shortest routes are:

- **A → B → D → C → A** with a total distance of 80 km
- **A → C → D → B → A** with a total distance of 80 km

So, the shortest possible route(s) are **A → B → D → C → A** and **A → C → D → B → A**, both with a total distance of 80 km.

Knapsack Problem

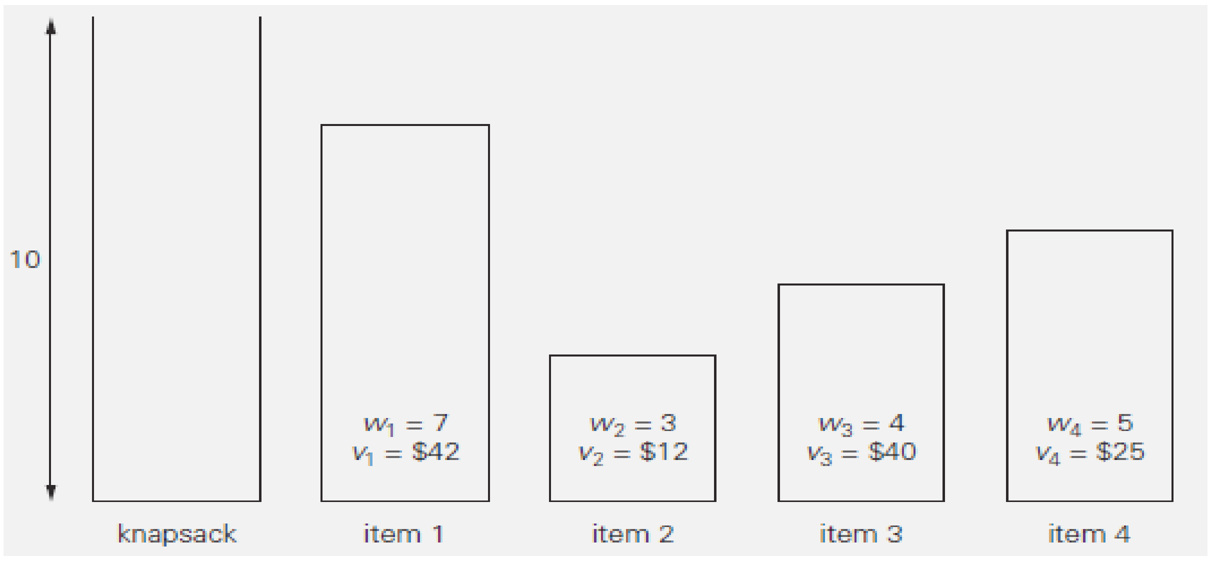
Given n items of known weights w1, w2, . . . , wn and values v1, v2, . . . , vn and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack.

Real time examples:

• A Thief who wants to steal the most valuable loot that fits into his knapsack,

• A transport plane that must deliver the most valuable set of items to a remote location without exceeding the plane's capacity.

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



Instance of the knapsack problem.

Subset	Total weight	Total value
Φ	0	$0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1, 2}	10	$54
{1, 3}	11	not feasible
{1, 4}	12	not feasible
{2, 3}	7	$52
{2, 4}	8	$37
{3, 4}	**9**	**$65 (Maximum-Optimum)**
{1, 2, 3}	14	not feasible
{1, 2, 4}	15	not feasible
{1, 3, 4}	16	not feasible
{ 2, 3, 4}	12	not feasible
{1, 2, 3, 4}	19	not feasible

knapsack problem's solution by exhaustive search. The information about the optimal selection is in bold.

Time efficiency: As given in the example, the solution to the instance of Figure is given in Figure 2.6. Since the number of subsets of an n-element set is 2n, the exhaustive search leads to a Ω(2n) algorithm, no matter how efficiently individual subsets are generated.

Assignment Problem

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

Assignment problem solved by exhaustive search is illustrated with an example as shown in figure 2.8. A small instance of this problem follows, with the table entries representing the assignment costs C [i, j].

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

FIGURE 2.7 Instance of an Assignment problem.

An instance of the assignment problem is completely specified by its cost matrix *C.*

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

The problem is to select one element in each row of the matrix so that all selected elements are in different columns and the total sum of the selected elements is the smallest possible. We can describe feasible solutions to the assignment problem as n-tuples <j1, . . . , jn > in which the ith component, i = 1, . . . , n, indicates the column of the element selected in the ith row (i.e., the job number assigned to the ith person). For example, for the cost matrix above, <2, 3, 4, 1> indicates the assignment of Person 1 to Job 2, Person 2 to Job 3, Person 3 to Job 4, and Person 4 to Job 1. Similarly, we can have 4! = 4 · 3 · 2 · 1 = 24, i. e. 24 permutations. The requirements of the assignment problem imply that there is a one-to-one correspondence between feasible assignments and permutations of the first n integers. Therefore, the exhaustive-search approach to the assignment problem would require generating all the permutations of integers 1, 2, . . . , n, computing the total cost of each assignment by summing up the corresponding elements of the cost matrix, and finally selecting the one with the smallest sum. A few first iterations of applying this algorithm to the instance given above are given below.

<1, 2, 3, 4>	cost = 9 + 4 + 1 + 4 = **18**
<1, 2, 4, 3>	cost = 9 + 4 + 8 + 9 = 30
<1, 3, 2, 4>	cost = 9 + 3 + 8 + 4 = 24
<1, 3, 4, 2>	cost = 9 + 3 + 8 + 6 = 26
<1, 4, 2, 3>	cost = 9 + 7 + 8 + 9 = 33
<1, 4, 3, 2>	cost = 9 + 7 + 1 + 6 = 23

<2, 1, 3, 4>	**cost = 2 + 6 + 1 + 4 = 13 (Min)**
<2, 1, 4, 3>	cost = 2 + 6 + 8 + 9 = 25
<2, 3, 1, 4>	cost = 2 + 3 + 5 + 4 = 14
<2, 3, 4, 1>	cost = 2 + 3 + 8 + 7 = 20
<2, 4, 1, 3>	cost = 2 + 7 + 5 + 9 = 23
<2, 4, 3, 1>	cost = 2 + 7 + 1 + 7 = 17, etc

FIGURE 2.8 First few iterations of solving a small instance of the assignment problem by exhaustive search.

Since the number of permutations to be considered for the general case of the assignment problem is n!, exhaustive search is impractical for all but very small instances of the problem. Fortunately, there is a much more efficient algorithm for this problem called the Hungarian method.

Example Problem:

Let's consider an example with 3 agents and 3 tasks.

Agents: A1, A2, A3

Tasks: T1, T2, T3

Cost Matrix:

	T1	**T2**	**T3**
A1	**10**	**5**	**9**
A2	**4**	**7**	**8**
A3	**15**	**3**	**6**

The goal is to assign each agent to exactly one task such that the total cost is minimized.

Step 1: List All Possible Assignments

For 3 agents and 3 tasks, the number of possible assignments is given by the factorial of the number of agents (or tasks):

Number of assignments = 3! = 6 assignments.

Here are the possible assignments:

1. A1 → T1, A2 → T2, A3 → T3
2. A1 → T1, A2 → T3, A3 → T2
3. A1 → T2, A2 → T1, A3 → T3
4. A1 → T2, A2 → T3, A3 → T1
5. A1 → T3, A2 → T1, A3 → T2
6. A1 → T3, A2 → T2, A3 → T1

Step 2: Calculate the Cost for Each Assignment

Let's calculate the total cost for each possible assignment:

1. A1 → T1, A2 → T2, A3 → T3:

- A1 → T1 = 10
- A2 → T2 = 7
- A3 → T3 = 6
- Total Cost = 10 + 7 + 6 = 23

2. A1 → T1, A2 → T3, A3 → T2:

- A1 → T1 = 10
- A2 → T3 = 8
- A3 → T2 = 3
- Total Cost = 10 + 8 + 3 = 21

3. A1 → T2, A2 → T1, A3 → T3:

- A1 → T2 = 5
- A2 → T1 = 4
- A3 → T3 = 6
- Total Cost = 5 + 4 + 6 = 15

4. A1 → T2, A2 → T3, A3 → T1:

- A1 → T2 = 5
- A2 → T3 = 8
- A3 → T1 = 15
- Total Cost = 5 + 8 + 15 = 28

5. A1 → T3, A2 → T1, A3 → T2:

- A1 → T3 = 9
- A2 → T1 = 4
- A3 → T2 = 3
- Total Cost = 9 + 4 + 3 = 16

6. A1 → T3, A2 → T2, A3 → T1:

- A1 → T3 = 9
- A2 → T2 = 7

- A3 → T1 = 15
- Total Cost = 9 + 7 + 15 = 31

Step 3: Identify the Minimum Cost Assignment

From the calculated costs, the minimum cost is:

- A1 → T2, A2 → T1, A3 → T3 with a total cost of 15.

Solution:

The optimal assignment that minimizes the total cost is:

- Assign A1 to T2
- Assign A2 to T1
- Assign A3 to T3

This results in a minimum cost of 15.

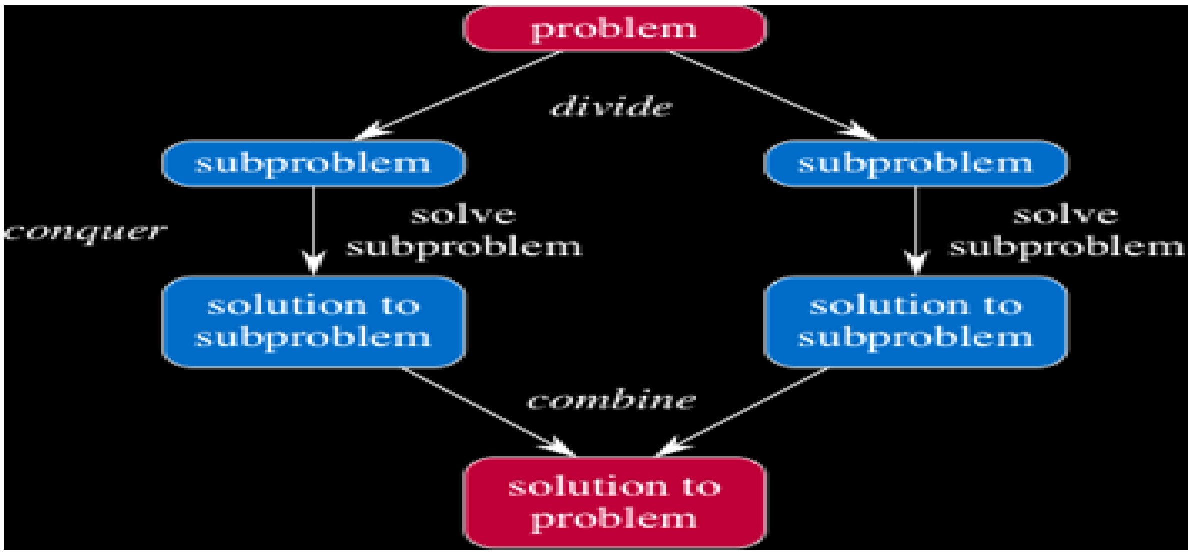
Divide and Conquer Methodology

A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same (or related) type (divide), until these become simple enough to be solved directly (conquer).

Divide-and-conquer algorithms work according to the following general plan:

1. A problem is divided into several subproblems of the same type, ideally of about equal size.
2. The subproblems are solved (typically recursively, though sometimes a different algorithm is employed, especially when subproblems become small enough).
3. If necessary, the solutions to the subproblems are combined to get a solution to the original problem.

The divide-and-conquer technique as shown in figure which depicts the case of dividing a problem into two smaller subproblems, then the subproblems solved separately. Finally, solution to the original problem is done by combining the solutions of subproblems



Divide and conquer methodology can be easily applied on the following problem.

1. Merge sort
2. Quick sort
3. Binary search

Merge Sort

The Merge Sort algorithm is a classic example of the divide-and-conquer methodology. It's an efficient, stable, and comparison-based sorting algorithm. Here's how it works:

Merge Sort Algorithm Steps:

1. **Divide**:
   - If the array has more than one element, split the array into two halves.
   - Recursively apply merge sort to each half.
2. **Conquer**:
   - Sort the two halves recursively by applying the merge sort algorithm to them.
3. **Combine**:
   - Merge the two sorted halves back together into a single sorted array.

Merge Sort Algorithm

```
function MergeSort(array)
   if length(array) > 1
      mid = length(array) / 2
      // Divide the array into two halves
      leftHalf = array[0 : mid]
      rightHalf = array[mid : length(array)]
      // Recursively sort each half
```

```
MergeSort(leftHalf)
MergeSort(rightHalf)
// Merge the sorted halves
i = j = k = 0
while i < length(leftHalf) and j < length(rightHalf)
   if leftHalf[i] < rightHalf[j]
      array[k] = leftHalf[i]
      i = i + 1
   else
      array[k] = rightHalf[j]
      j = j + 1
   k = k + 1
// Copy any remaining elements of leftHalf, if any
while i < length(leftHalf)
   array[k] = leftHalf[i]
   i = i + 1
   k = k + 1
// Copy any remaining elements of rightHalf, if any
while j < length(rightHalf)
   array[k] = rightHalf[j]
   j = j + 1
   k = k + 1
```

Explanation:

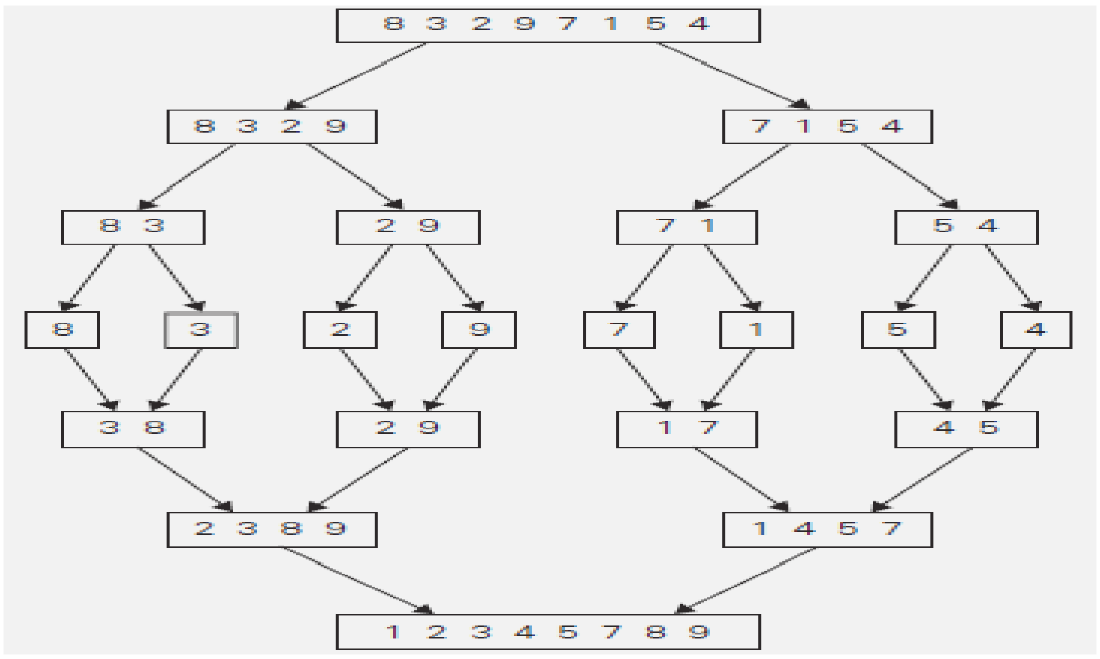
1. **Base Case**:
   - If the length of the array is 1 or less, it is already sorted, so the function returns immediately.
2. **Divide**:
   - The array is divided into two halves, leftHalf and rightHalf.
3. **Conquer**:
   - The MergeSort function is called recursively on each half to sort them.
4. **Combine**:
   - After both halves are sorted, they are merged back together in sorted order.
   - Two pointers (i for leftHalf and j for rightHalf) are used to traverse the halves.
   - The elements are compared, and the smaller element is placed into the original

array.

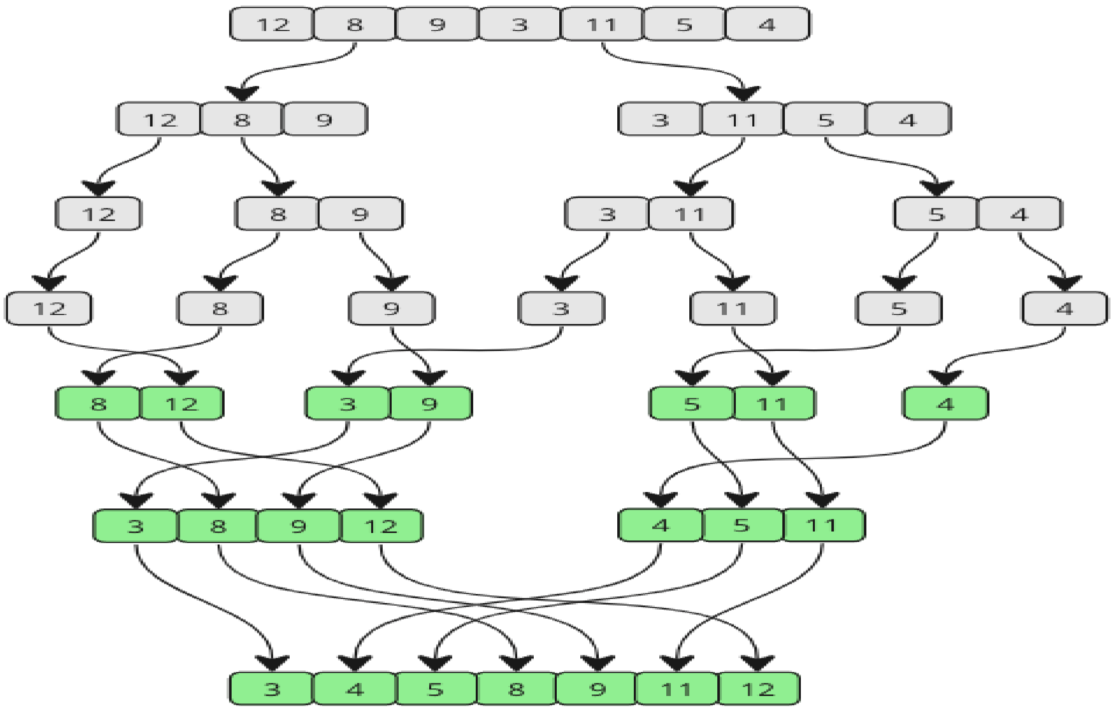
- If there are remaining elements in either half, they are copied into the original array.

This process continues recursively until the entire array is sorted.

The operation of the algorithm on the list 8, 3, 2, 9, 7, 1, 5, 4 is illustrated



Example 2



Quick Sort

Quicksort is an efficient, recursive, and in-place sorting algorithm that uses a divide-and-conquer approach. It is one of the most commonly used sorting algorithms because of its average-case time complexity of O(nlogn).

Quicksort Algorithm Overview:

1. **Choose a Pivot:** Select an element from the array as the pivot. The pivot can be any element, but common choices are the first element, the last element, or the middle element.
2. **Partition:** Rearrange the array so that all elements less than the pivot are on its left, and all elements greater than the pivot are on its right. The pivot element is now in its correct position.
3. **Recursively Apply Quicksort:** Recursively apply the same process to the left and right subarrays (elements before and after the pivot).

Quicksort Pseudocode:

```
function QuickSort(array, low, high)
   if low < high then
      // Partition the array and get the pivot index
      pivotIndex = Partition(array, low, high)
      // Recursively apply quicksort to the left subarray
      QuickSort(array, low, pivotIndex - 1)
      // Recursively apply quicksort to the right subarray
      QuickSort(array, pivotIndex + 1, high)
function Partition(array, low, high)
   pivot = array[high] // Choosing the last element as the pivot
   i = low - 1 // Pointer for the smaller element
   for j = low to high - 1 do
      if array[j] <= pivot then
         i = i + 1
         swap(array[i], array[j])
   swap(array[i + 1], array[high])
   return i + 1
```

Example: Sorting the Array [10, 7, 8, 9, 1, 5] using Quicksort

Step 1: Initial Call to QuickSort

- **Array**: [10, 7, 8, 9, 1, 5]

- **Low**: 0
- **High**: 5

Step 2: Partitioning

- **Pivot**: Choose the last element as the pivot, so pivot = 5.
- **Initial Array**: [10, 7, 8, 9, 1, 5]
    - Start with i = -1 and j = 0.
    - Compare array[j] with the pivot (5):
        - array[0] = 10 (greater than 5) → No change.
        - array[1] = 7 (greater than 5) → No change.
        - array[2] = 8 (greater than 5) → No change.
        - array[3] = 9 (greater than 5) → No change.
        - array[4] = 1 (less than or equal to 5) → Increment i to 0 and swap array[i] with array[4].
        - The array becomes [1, 7, 8, 9, 10, 5].
- Swap the pivot (5) with array[i + 1] (element at index 1): The array becomes [1, 5, 8, 9, 10, 7].
- **Pivot index** after partitioning: 1

Step 3: Recursively Apply Quicksort to Subarrays

1. **Left Subarray** [1] (from index 0 to 0):
    - Only one element, so it is already sorted.
2. **Right Subarray** [8, 9, 10, 7] (from index 2 to 5):
    - **Pivot**: Choose 7 (last element).
    - **Partition**:
        - Compare and rearrange to get [5, 1, 7, 9, 10, 8].
        - Pivot index after partitioning: 2.
    - **Recursively sort:**
        - **Left Subarray** [8] (from index 2 to 1): No elements, so skip.
        - **Right Subarray** [9, 10, 8] (from index 3 to 5):
            - **Pivot**: Choose 8 (last element).
            - **Partition**:
                - Compare and rearrange to get [5, 1, 7, 8, 10, 9].
                - Pivot index after partitioning: 3.
            - Recursively sort:
                - **Left Subarray** [9] (from index 3 to 2): No elements, so

skip.

- **Right Subarray** [10, 9] (from index 4 to 5):
    - **Pivot**: Choose 9 (last element).
    - **Partition**:
        - Compare and rearrange to get [5, 1, 7, 8, 9, 10].
        - Pivot index after partitioning: 4.
        - Recursively sort:
            - **Left Subarray** [9] (from index 4 to 3): No elements, so skip.
            - **Right Subarray** [10] (from index 5 to 5): Only one element, so it is already sorted.

Final Sorted Array:

After all recursive steps, the array becomes [1, 5, 7, 8, 9, 10], which is the sorted version of the original array [10, 7, 8, 9, 1, 5].

Key Points of Quicksort:

- **Time Complexity**: Average and Best Case: O(nlogn), Worst Case: O(n2) (e.g., when the pivot is always the smallest or largest element).
- **Space Complexity**: O(logn) due to recursive stack space.
- **In-Place Sorting**: Does not require additional memory.
- **Not Stable**: It may not preserve the relative order of equal elements.

Binary Search

Binary Search is an efficient algorithm for finding an item from a sorted list of elements. It works by repeatedly dividing the search interval in half, a classic example of the divide-and-conquer approach.

How Binary Search Works:

1. **Divide**: Begin with an interval covering the whole list. Compare the target value to the middle element of the array.
2. **Conquer**:
    - If the target value is equal to the middle element, the search is complete.
    - If the target value is less than the middle element, narrow the interval to the lower half.
    - If the target value is greater than the middle element, narrow the interval to the

upper half.

3. **Combine**: This step involves the recursion or iteration, where the search continues on the subarray until the size of the subarray is reduced to zero.

Binary Search Pseudocode:

Here's the pseudocode for binary search using the divide-and-conquer approach:

```
function BinarySearch(array, target, low, high)
  if low > high then
    return -1 // Target is not found
  mid = low + (high - low) / 2 // Calculate the mid index
  if array[mid] == target then
    return mid // Target found at index mid
  else if array[mid] > target then
    return BinarySearch(array, target, low, mid - 1) // Search in the left half
  else
    return BinarySearch(array, target, mid + 1, high) // Search in the right half
```

Example: Searching for 7 in the Array [1, 3, 5, 7, 9, 11, 13]

Initial Call to BinarySearch:

- **Array**: [1, 3, 5, 7, 9, 11, 13]
- **Target**: 7
- **Low**: 0 (first index)
- **High**: 6 (last index)

Step 1: Find the Midpoint

- **Mid**: Calculate the midpoint: mid = 0 + (6 - 0) / 2 = 3.
- Compare array[mid] with the target:
  - array[3] = 7, which is equal to the target 7.
- **Target found** at index 3.

Hear Sort

Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure. It is an in-place algorithm with a time complexity of O(nlogn) and is particularly well-suited for sorting large datasets.

How Heap Sort Works:

1. **Build a Max Heap**: Convert the array into a max heap, where the largest element is at the root of the heap.
2. **Extract Elements**: Repeatedly remove the root (the maximum element) and move it

to the end of the array, reducing the heap size by one. Then, heapify the root to restore the max heap property.

3. **Repeat**: Continue extracting the maximum element from the heap until the entire array is sorted.

Heap Sort Pseudocode:

Here's the pseudocode for Heap Sort:

```
function HeapSort(array)
   n = length(array)
   // Build a max heap
   for i = n/2 - 1 down to 0 do
      Heapify(array, n, i)
   // Extract elements from the heap
   for i = n - 1 down to 1 do
      swap(array[0], array[i]) // Move current root to the end
      Heapify(array, i, 0) // Heapify the reduced heap
function Heapify(array, n, i)
   largest = i // Initialize largest as root
   left = 2 * i + 1 // Left child
   right = 2 * i + 2 // Right child
// If left child is larger than root
   if left < n and array[left] > array[largest] then
      largest = left
   // If right child is larger than largest so far
   if right < n and array[right] > array[largest] then
      largest = right
   // If largest is not root
   if largest != i then
      swap(array[i], array[largest]) // Swap root with largest
      Heapify(array, n, largest) // Recursively heapify the affected subtree
```

Example: Sorting the Array [4, 10, 3, 5, 1] Using Heap Sort

Step 1: Build a Max Heap

- **Array**: [4, 10, 3, 5, 1]
- Start heapifying from the non-leaf nodes (starting from index n/2 - 1).

1. **Heapify Subtree with Root at Index 1**:

- Subtree: [10, 5, 1] (10 is the root, 5 is left child, 1 is right child).
- **No change needed**: Already a max heap.

2. **Heapify Subtree with Root at Index 0**:
    - Subtree: [4, 10, 3, 5, 1] (4 is root, 10 is left child, 3 is right child).
    - 10 > 4, so swap 10 with 4.
    - Resulting Subtree: [10, 4, 3, 5, 1]
    - Heapify the affected subtree with root at index 1.
    - Subtree: [4, 5, 1]
    - 5 > 4, so swap 5 with 4.
    - **Max Heap**: [10, 5, 3, 4, 1]

Step 2: Extract Elements from the Heap

1. **Swap Root with Last Element**:
    - Swap 10 with 1.
    - Array: [1, 5, 3, 4, 10]
    - **Heapify** the root to maintain the heap property.
    - Subtree: [1, 5, 3, 4]
    - 5 > 1, so swap 5 with 1.
    - Subtree: [5, 4, 3, 1]
    - **Array after heapify**: [5, 4, 3, 1, 10]
2. **Swap Root with Last Element of Reduced Heap**:
    - Swap 5 with 1.
    - Array: [1, 4, 3, 5, 10]
    - **Heapify** the root to maintain the heap property.
    - Subtree: [1, 4, 3]
    - 4 > 1, so swap 4 with 1.
    - **Array after heapify**: [4, 1, 3, 5, 10]
3. **Swap Root with Last Element of Reduced Heap**:
    - Swap 4 with 3.
    - Array: [3, 1, 4, 5, 10]
    - **Heapify** the root to maintain the heap property.
    - Subtree: [3, 1]
    - **No change needed**: Already a max heap.
    - **Array after heapify**: [3, 1, 4, 5, 10]
4. **Final Swap**:

    - Swap 3 with 1.
    - Array: [1, 3, 4, 5, 10]
    - No heapify needed since only one element is left.

Final Sorted Array:

After all steps, the array is [1, 3, 4, 5, 10].

Summary of Heap Sort:

- **Time Complexity**: O(nlogn) for all cases.
- **Space Complexity**: O(1) as it sorts in place.
- **Not Stable**: Heap Sort does not maintain the relative order of equal elements.
- **In-Place**: No extra space is required, making it memory efficient.

Heap Sort is particularly useful when space is limited, and you need a reliable O(nlogn) sort, though it might not be the fastest in practice compared to Quicksort for many real-world applications.

PECT's PCU

School of Engineering and Technology

Department MCA/BCA/BSc (CS)

Unit 03. Dynamic Programming and Greedy Technique

Introduction:

Dynamic programming is defined as a computer programming technique where an algorithmic problem is first broken down into sub-problems, the results are saved, and then the sub-problems are optimized to find the overall solution — which usually has to do with finding the maximum and minimum range of the algorithmic query.

What Is Dynamic Programming?

Dynamic programming is a computer programming technique where an algorithmic problem is first broken down into sub-problems, the results are saved, and then the sub-problems are optimized to find the overall solution — which usually has to do with finding the maximum and minimum range of the algorithmic query.

Richard Bellman was the one who came up with the idea for dynamic programming in the 1950s. It is a method of mathematical optimization as well as a methodology for computer programming. It applies to issues one can break down into either overlapping subproblems or optimum substructures.

How Does Dynamic Programming Work?

1. Top-down approach

In computer science, problems are resolved by recursively formulating solutions, employing the answers to the problems' subproblems. If the answers to the subproblems overlap, they may be memorized or kept in a table for later use. The top-down approach follows the strategy of memorization. The memorization process is equivalent to adding the recursion and caching steps. The difference between recursion and caching is that recursion requires calling the function directly, whereas caching requires preserving the intermediate results.

The top-down strategy has many benefits, including the following:

- The top-down approach is easy to understand and implement. In this approach, problems are broken down into smaller parts, which help users identify what needs to be done. With each step, more significant, more complex problems become smaller, less complicated, and, therefore, easier to solve. Some parts may even be reusable for the same problem.

- It allows for subproblems to be solved upon request. The top-down approach will enable problems to be broken down into smaller parts and their solutions stored for reuse. Users can then query solutions for each part.
- It is also easier to debug. Segmenting problems into small parts allows users to follow the solution quickly and determine where an error might have occurred.

2. Bottom-up approach

In the bottom-up method, once a solution to a problem is written in terms of its subproblems in a way that loops back on itself, users can rewrite the problem by solving the smaller subproblems first and then using their solutions to solve the larger subproblems.

The advantages of the bottom-up approach include the following:

- It makes decisions about small reusable subproblems and then decides how they will be put together to create a large problem.
- It removes recursion, thus promoting the efficient use of memory space. Additionally, this also leads to a reduction in timing complexity.

The principle of optimality

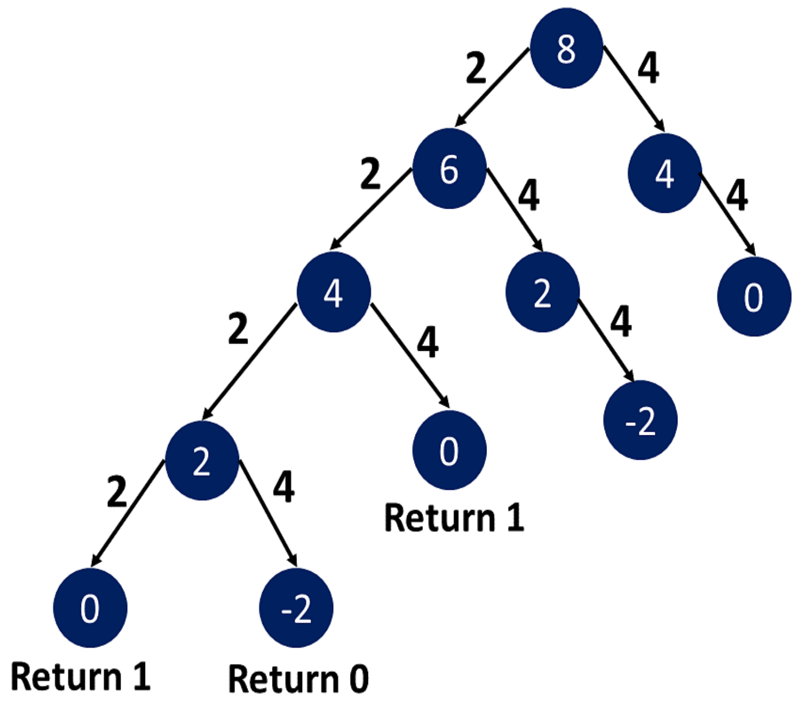
The principle of optimality is a fundamental concept in dynamic programming that states that the optimal solution to a dynamic optimization problem can be found by combining the optimal solutions to its sub-problems. It was developed by Richard Bellman in 1957.

Explanation: The principle states that an optimal policy, or sequence of decisions, has the property that whatever the initial state and decisions are, the remaining decisions must constitute an optimal policy regarding the state resulting from the first decision.

Applications: The principle of optimality is used in applications of dynamic programming such as the matrix chain problem, all-pairs shortest path problem, and optimal binary search trees.

Coin Changing Problem

You are given an array of coins with varying denominations and an integer sum representing the total amount of money; you must return the fewest coins required to make up that sum; if that sum cannot be constructed, return -1.



Given an integer array of coins [ ] of size N representing different types of denominations and an integer sum, the task is to count all combinations of coins to make a given value sum.

Examples:

Input: sum = 4, coins [] = {1,2,3}

Output: 4

Explanation: there are four solutions: {1, 1, 1, 1}, {1, 1, 2}, {2, 2} and {1, 3}

Input: sum = 10, coins [] = {2, 5, 3, 6}

Output: 5

Explanation: There are five solutions:

{2,2,2,2,2}, {2,2,3,3}, {2,2,6}, {2,3,5} and {5,5}

Input: sum = 10, coins[] = {10}

Output: 1

Explanation: The only is to pick 1 coin of value 10.

Input: sum = 5, coins[] = {4}

Output: 0

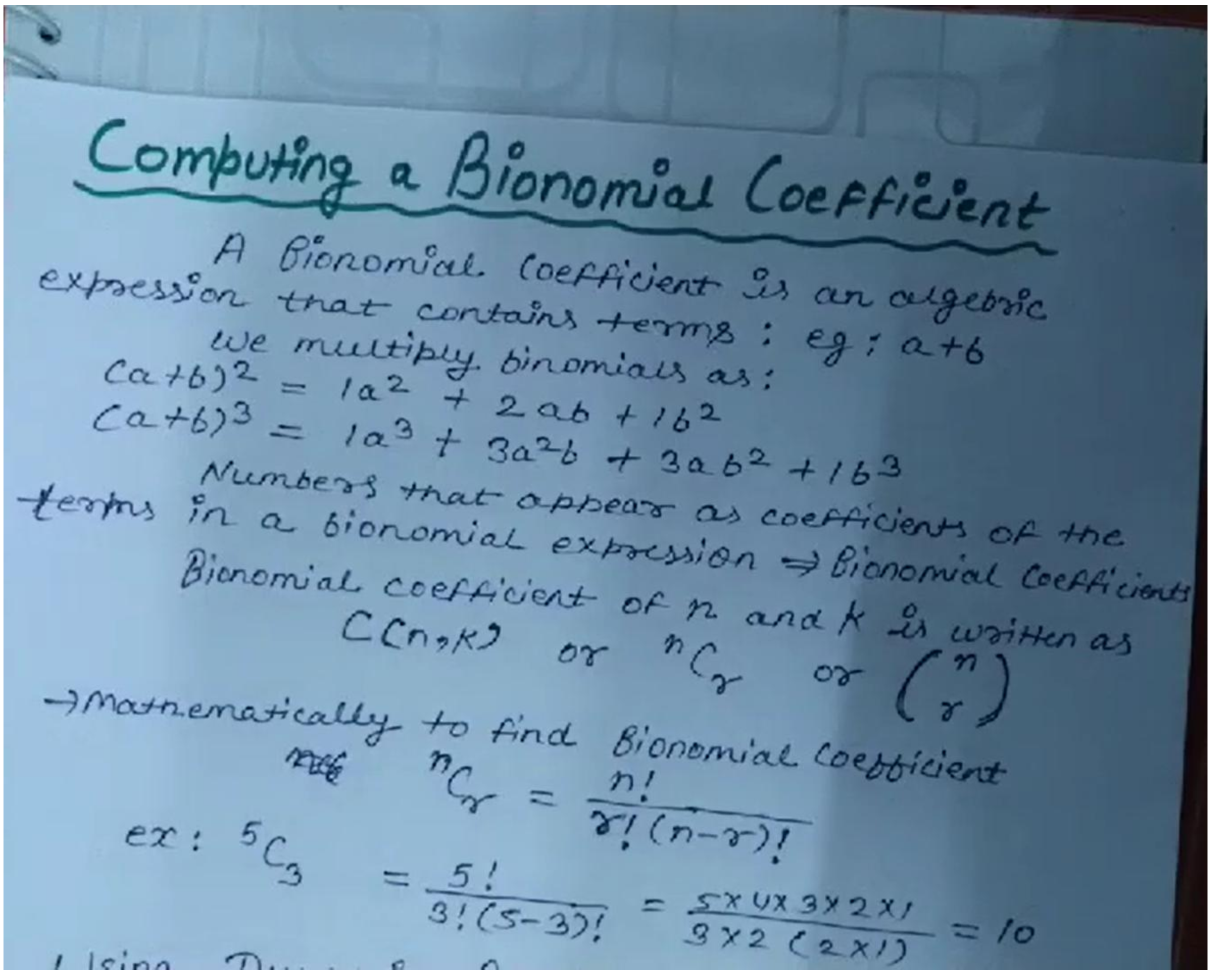
Explanation: We cannot make sum 5 with the given coins

Recursive solution code for the coin change problem

```
#include <stdio.h>
```

```
int coins[] = {1,2,3};
int numberofCoins = 3, sum = 4;
 int solution(int sol, int i){
    if(numberofCoins == 0 || sol > sum || i>=numberofCoins)
        return 0;
    else if(sol == sum)
        return 1;
      return solution(sol+coins[i],i) + solution(sol,i+1) ;
}
int main()
{
    printf("Total solutions: %d",solution(0,0));
    return 0;
}
```

Computing a Binomial Coefficient



Computing a Bionomial Coefficient

A Bionomial coefficient is an algebric expression that contains terms : eg: a+b

We multiply binomials as:

$(a+b)^2 = 1a^2 + 2ab + 1b^2$

$(a+b)^3 = 1a^3 + 3a^2b + 3ab^2 + 1b^3$

Numbers that appear as coefficients of the terms in a bionomial expression ⇒ Bionomial Coefficients

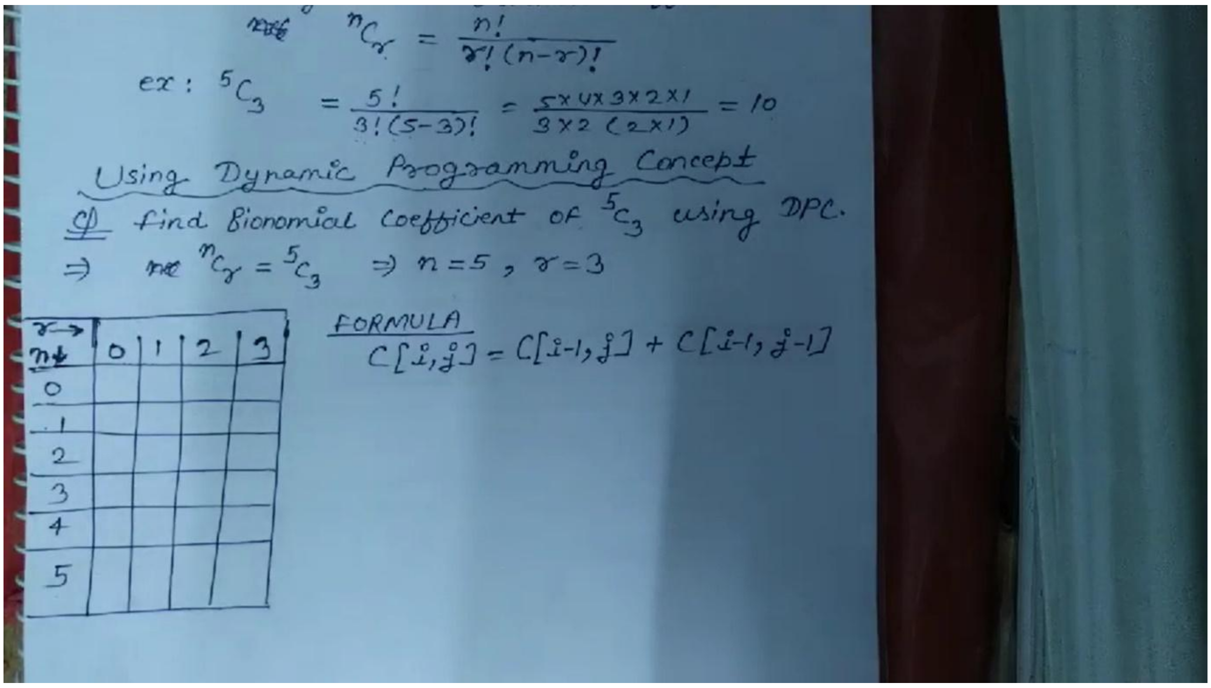
Bionomial coefficient of n and k is written as

$C(n,k)$ or ^nC_r or $\binom{n}{r}$

→ Mathematically to find Bionomial coefficient

$$^nC_r = \frac{n!}{r!(n-r)!}$$

ex: $^5C_3 = \frac{5!}{3!(5-3)!} = \frac{5\times4\times3\times2\times1}{3\times2(2\times1)} = 10$



[illegible] $^{n}C_{r} = \frac{n!}{r!(n-r)!}$

ex : $^{5}C_{3} = \frac{5!}{3!(5-3)!} = \frac{5\times4\times3\times2\times1}{3\times2\ (2\times1)} = 10$

Using Dynamic Programming Concept

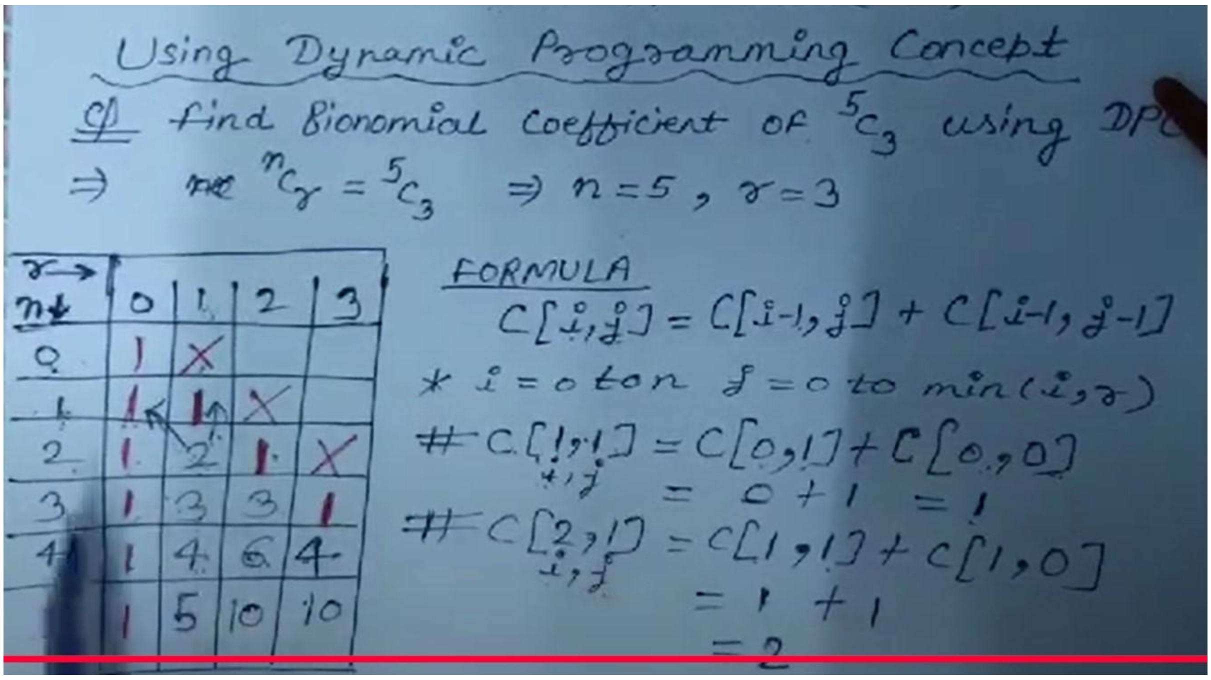
Q. find Bionomial Coefficient of $^{5}C_{3}$ using DPC.

$\Rightarrow$ ~~nc~~ $^{n}C_{r} = {}^{5}C_{3} \Rightarrow n=5,\ r=3$

r→ / n↓	0	1	2	3
0				
1				
2				
3				
4				
5				

FORMULA

$C[i,j] = C[i-1,j] + C[i-1,j-1]$



Using Dynamic Programming Concept

Q. find Bionomial Coefficient of $^{5}C_{3}$ using DPC

$\Rightarrow$ ~~nc~~ $^{n}C_{r} = {}^{5}C_{3} \Rightarrow n=5,\ r=3$

r→ / n↓	0	1	2	3
0	1	X		
1	1	1	X	
2	1	2	1	X
3	1	3	3	1
4	1	4	6	4
[illegible]	1	5	10	10

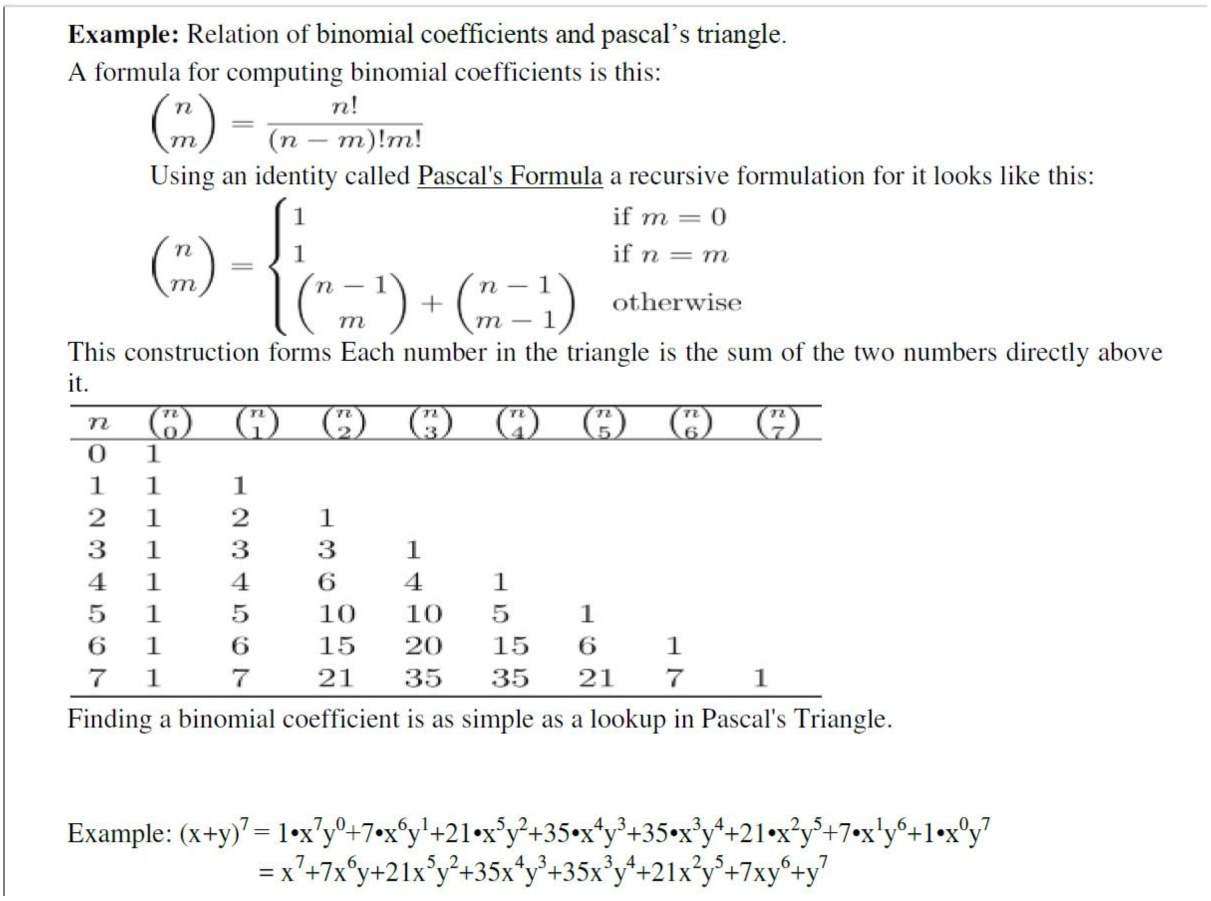
FORMULA

$C[i,j] = C[i-1,j] + C[i-1,j-1]$

* $i = 0$ to n $\quad j = 0$ to $\min(i, r)$

$\underset{i,j}{C[1,1]} = C[0,1] + C[0,0]$
$= 0 + 1 = 1$

$\underset{i,j}{C[2,1]} = C[1,1] + C[1,0]$
$= 1 + 1$
$= 2$



Example: Relation of binomial coefficients and pascal's triangle.

A formula for computing binomial coefficients is this:

$$\binom{n}{m} = \frac{n!}{(n-m)!m!}$$

Using an identity called Pascal's Formula a recursive formulation for it looks like this:

$$\binom{n}{m} = \begin{cases} 1 & \text{if } m = 0 \\ 1 & \text{if } n = m \\ \binom{n-1}{m} + \binom{n-1}{m-1} & \text{otherwise} \end{cases}$$

This construction forms Each number in the triangle is the sum of the two numbers directly above it.

n	$\binom{n}{0}$	$\binom{n}{1}$	$\binom{n}{2}$	$\binom{n}{3}$	$\binom{n}{4}$	$\binom{n}{5}$	$\binom{n}{6}$	$\binom{n}{7}$
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4	6	4	1			
5	1	5	10	10	5	1		
6	1	6	15	20	15	6	1	
7	1	7	21	35	35	21	7	1

Finding a binomial coefficient is as simple as a lookup in Pascal's Triangle.

Example: $(x+y)^7 = 1\bullet x^7y^0+7\bullet x^6y^1+21\bullet x^5y^2+35\bullet x^4y^3+35\bullet x^3y^4+21\bullet x^2y^5+7\bullet x^1y^6+1\bullet x^0y^7$
$= x^7+7x^6y+21x^5y^2+35x^4y^3+35x^3y^4+21x^2y^5+7xy^6+y^7$

Floyd's Algorithm

It is used to find the shortest paths between all pairs of nodes in a weighted graph. This algorithm is highly efficient and can handle graphs with both positive and negative edge weights, making it a versatile tool for solving a wide range of network and connectivity problems.

This algorithm works for both the directed and undirected weighted graphs. But it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative). It follows Dynamic Programming approach to check every possible path going via every possible node in order to calculate shortest distance between every pair of nodes.

Floyd Algorithm:

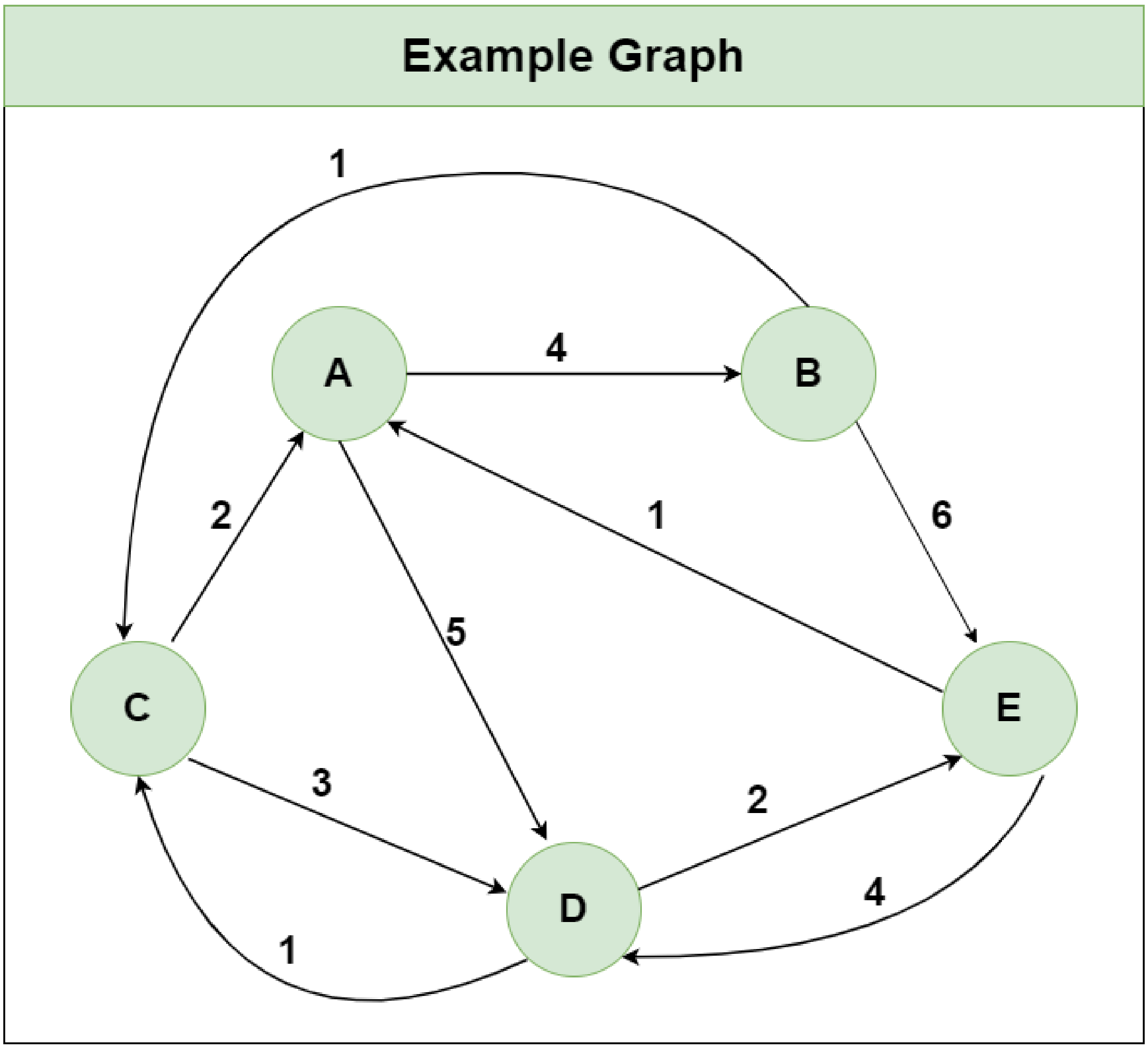
- Initialize the solution matrix same as the input graph matrix as a first step.
- Then update the solution matrix by considering all vertices as an intermediate vertex.
- The idea is to pick all vertices one by one and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path.
- When we pick vertex number **k** as an intermediate vertex, we already have considered vertices **{0, 1, 2, .. k-1}** as intermediate vertices.

- For every pair **(i, j)** of the source and destination vertices respectively, there are two possible cases.
  - **k** is not an intermediate vertex in shortest path from **i** to **j**. We keep the value of **dist[i][j]** as it is.
  - **k** is an intermediate vertex in shortest path from **i** to **j**. We update the value of **dist[i][j]** as **dist[i][k] + dist[k][j],** if **dist[i][j] > dist[i][k] + dist[k][j]**

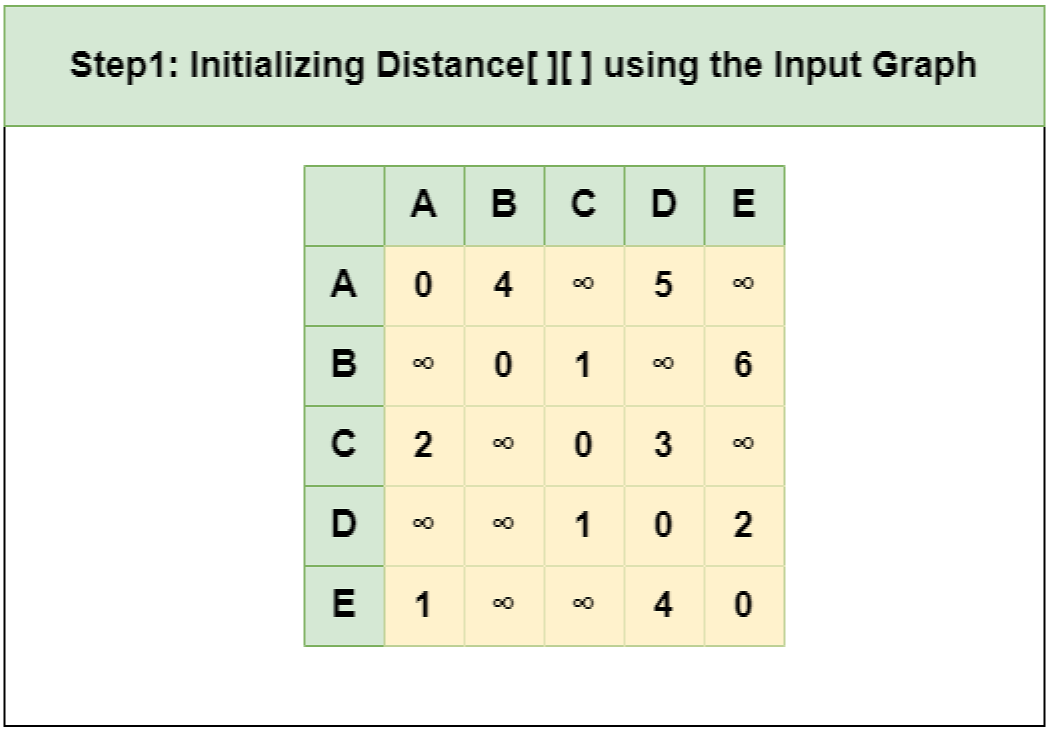
Pseudo-Code of Floyd Warshall Algorithm:

```
For k = 0 to n – 1
For i = 0 to n – 1
For j = 0 to n – 1
Distance[i, j] = min(Distance[i, j], Distance[i, k] + Distance[k, j])
```

where i = source Node, j = Destination Node, k = Intermediate Node



Step 1: Initialize the Distance[][] matrix using the input graph such that Distance[i][j]= weight of edge from i to j, also Distance[i][j] = Infinity if there is no edge from i to j.



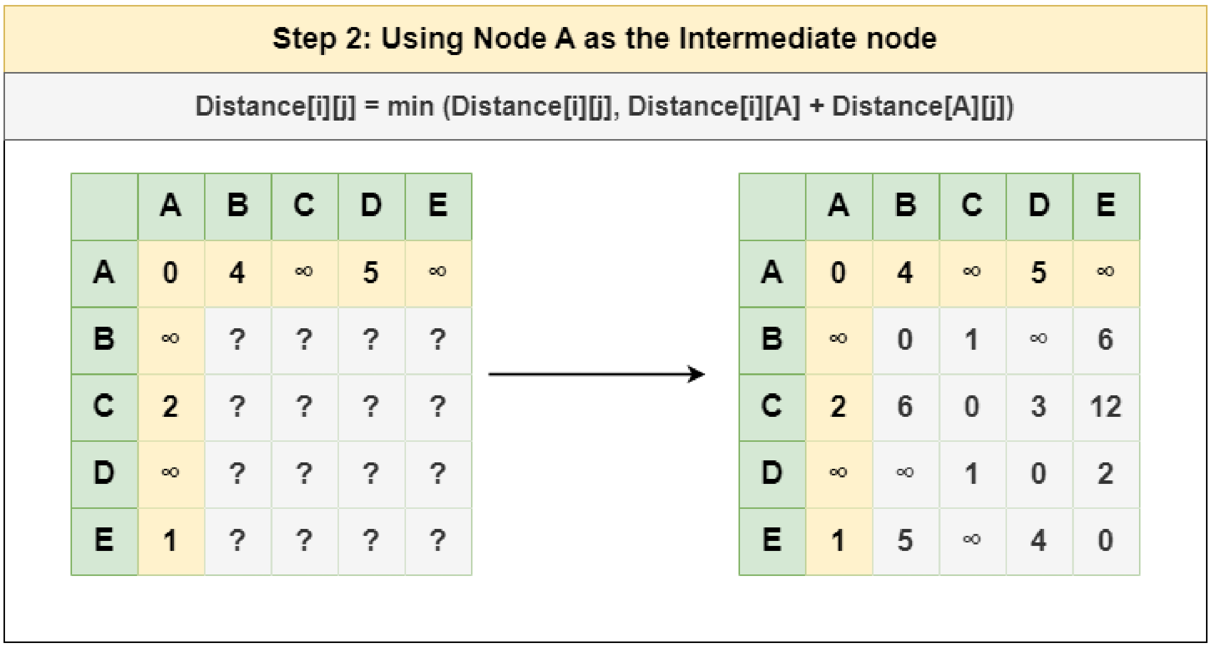
Step1: Initializing Distance[ ][ ] using the Input Graph

	A	B	C	D	E
A	0	4	∞	5	∞
B	∞	0	1	∞	6
C	2	∞	0	3	∞
D	∞	∞	1	0	2
E	1	∞	∞	4	0

Step 2: Treat node A as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:

= Distance[i][j] = minimum (Distance[i][j], (Distance from i to A) + (Distance from A to j))

= Distance[i][j] = minimum (Distance[i][j], Distance[i][A] + Distance[A][j])



Step 2: Using Node A as the Intermediate node

Distance[i][j] = min (Distance[i][j], Distance[i][A] + Distance[A][j])

	A	B	C	D	E
A	0	4	∞	5	∞
B	∞	?	?	?	?
C	2	?	?	?	?
D	∞	?	?	?	?
E	1	?	?	?	?

$\longrightarrow$

	A	B	C	D	E
A	0	4	∞	5	∞
B	∞	0	1	∞	6
C	2	6	0	3	12
D	∞	∞	1	0	2
E	1	5	∞	4	0

Step 3: Treat node B as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:

= Distance[i][j] = minimum (Distance[i][j], (Distance from i to B) + (Distance from B to j))

= Distance[i][j] = minimum (Distance[i][j], Distance[i][B] + Distance[B][j])

Step 3: Using Node B as the Intermediate node

Distance[i][j] = min (Distance[i][j], Distance[i][B] + Distance[B][j])

	A	B	C	D	E
A	?	4	?	?	?
B	∞	0	1	∞	6
C	?	6	?	?	?
D	?	∞	?	?	?
E	?	5	?	?	?

	A	B	C	D	E
A	0	4	5	5	10
B	∞	0	1	∞	6
C	2	6	0	3	12
D	∞	∞	1	0	2
E	1	5	6	4	0

Step 4: Treat node C as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:

= Distance[i][j] = minimum (Distance[i][j], (Distance from i to C) + (Distance from C to j))

= Distance[i][j] = minimum (Distance[i][j], Distance[i][C] + Distance[C][j])

Step 4: Using Node C as the Intermediate node

Distance[i][j] = min (Distance[i][j], Distance[i][C] + Distance[C][j])

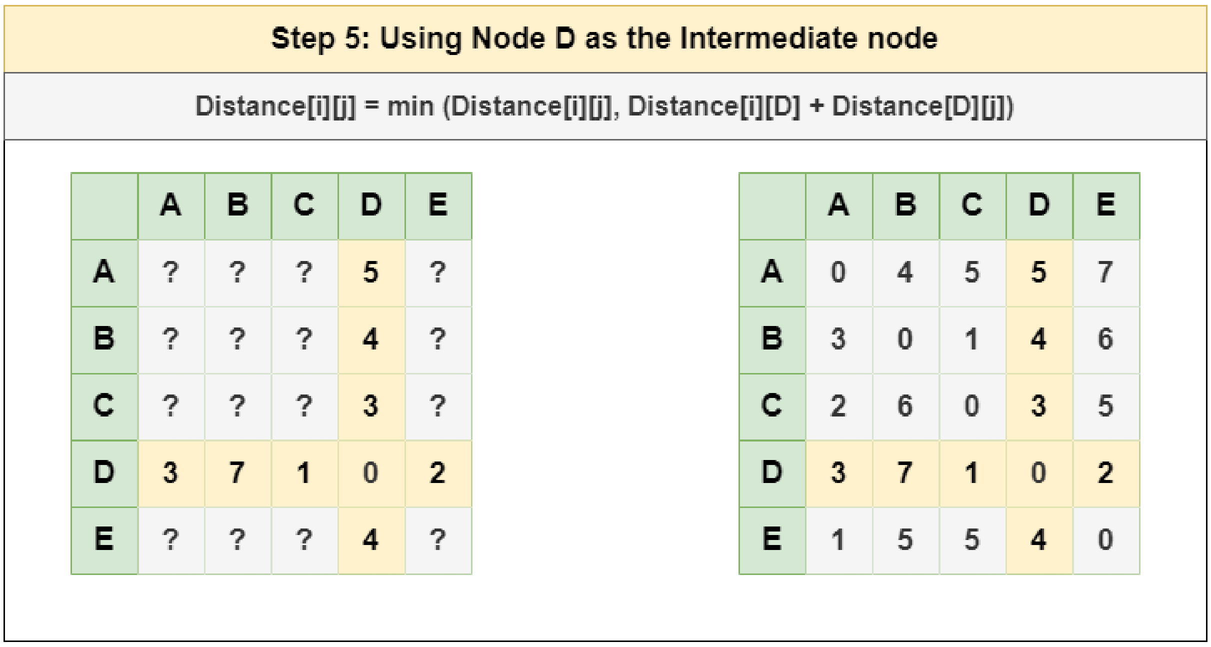
	A	B	C	D	E
A	?	?	5	?	?
B	?	?	1	?	?
C	2	6	0	3	12
D	?	?	1	?	?
E	?	?	6	?	?

	A	B	C	D	E
A	0	4	5	5	10
B	3	0	1	4	6
C	2	6	0	3	12
D	3	7	1	0	2
E	1	5	6	4	0

Step 5: Treat node D as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:

= Distance[i][j] = minimum (Distance[i][j], (Distance from i to D) + (Distance from D to j))

= Distance[i][j] = minimum (Distance[i][j], Distance[i][D] + Distance[D][j])



Step 5: Using Node D as the Intermediate node

Distance[i][j] = min (Distance[i][j], Distance[i][D] + Distance[D][j])

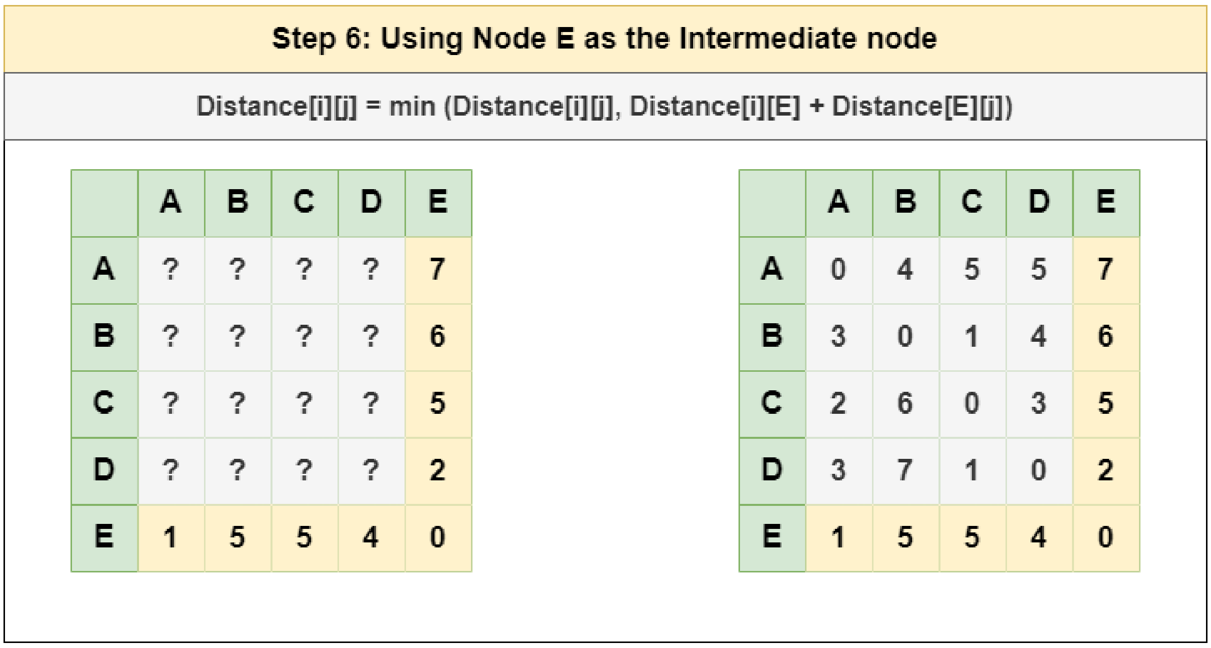
	A	B	C	D	E
A	?	?	?	5	?
B	?	?	?	4	?
C	?	?	?	3	?
D	3	7	1	0	2
E	?	?	?	4	?

	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

Step 6: Treat node E as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:

= Distance[i][j] = minimum (Distance[i][j], (Distance from i to E) + (Distance from E to j))

= Distance[i][j] = minimum (Distance[i][j], Distance[i][E] + Distance[E][j])



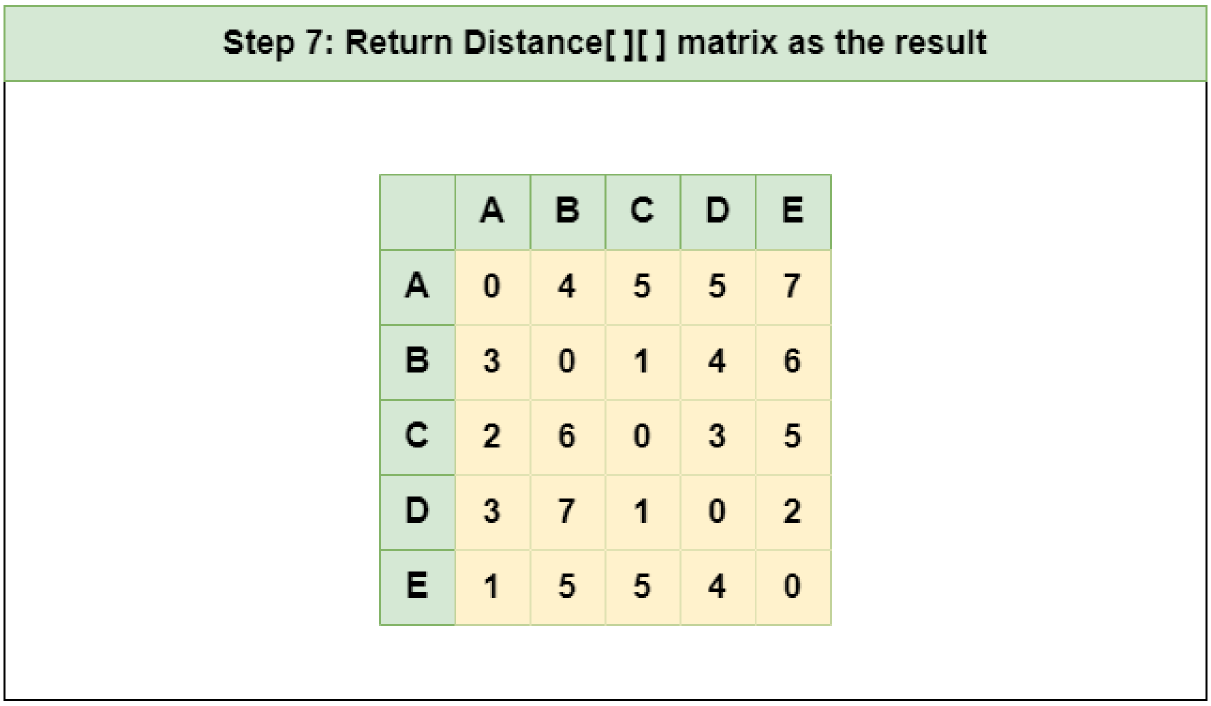
Step 6: Using Node E as the Intermediate node

Distance[i][j] = min (Distance[i][j], Distance[i][E] + Distance[E][j])

	A	B	C	D	E
A	?	?	?	?	7
B	?	?	?	?	6
C	?	?	?	?	5
D	?	?	?	?	2
E	1	5	5	4	0

	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

Step 7: Since all the nodes have been treated as an intermediate node, we can now return the updated Distance[][] matrix as our answer matrix.



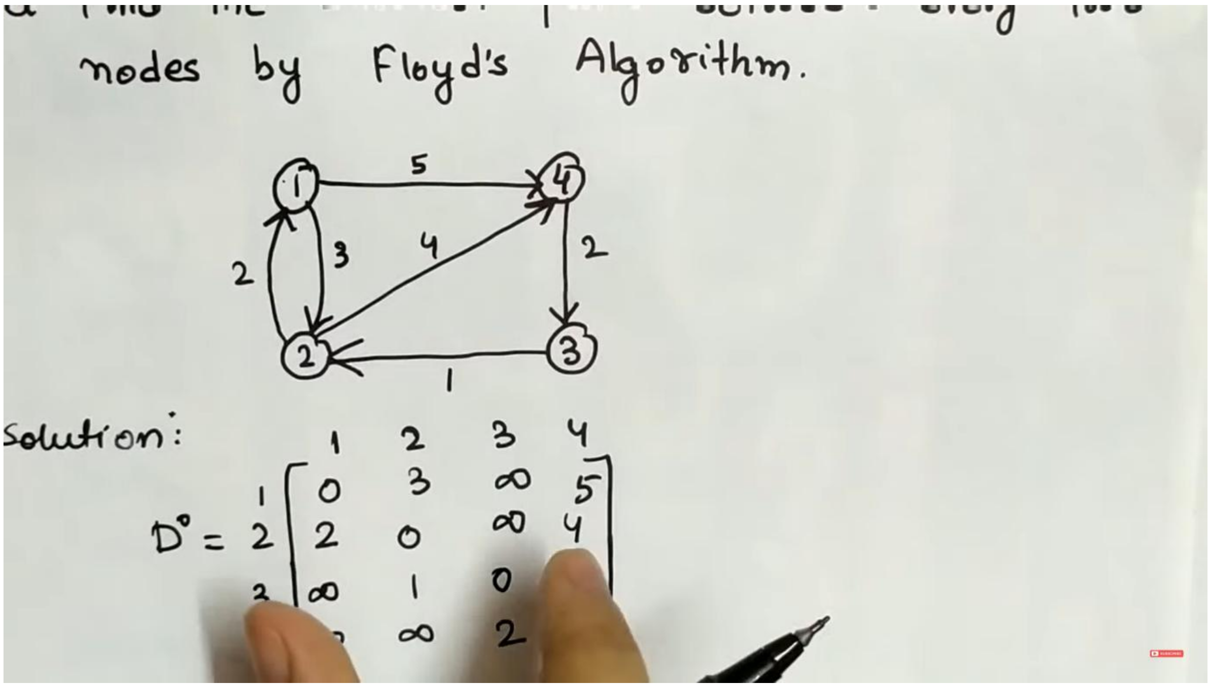
Step 7: Return Distance[ ][ ] matrix as the result

	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

Complexity Analysis of Floyd's Algorithm:

Time Complexity: O(V3), where V is the number of vertices in the graph and we run three nested loops each of size V

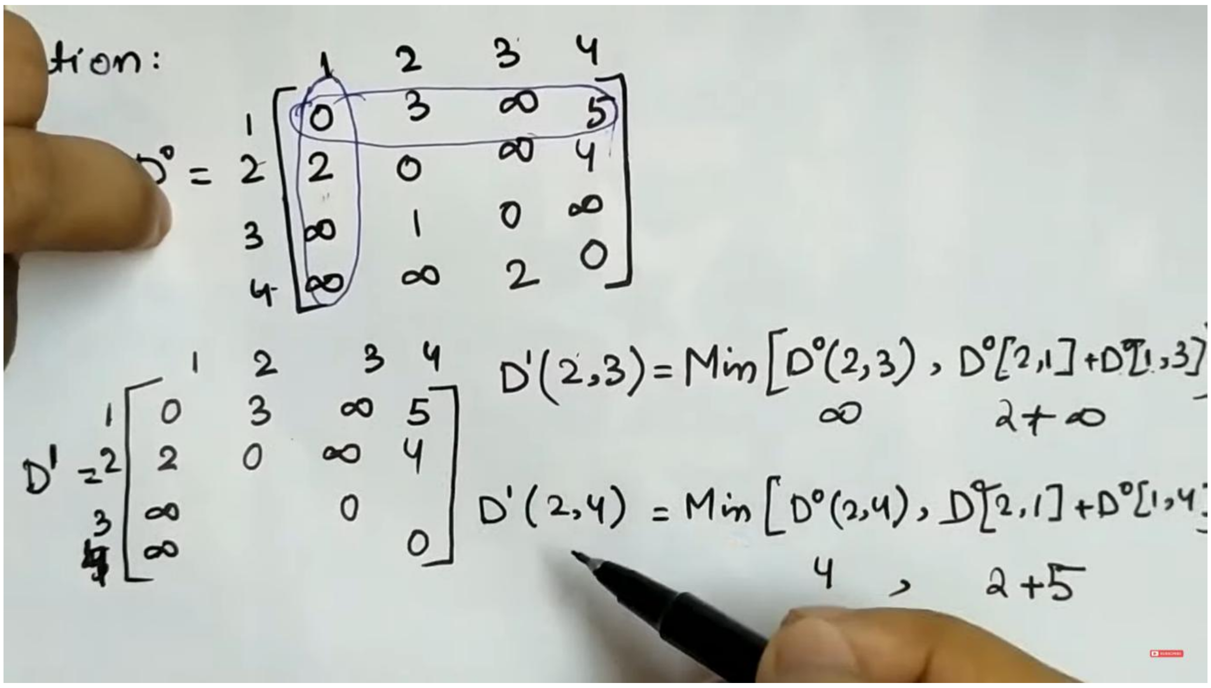
Auxiliary Space: O(V2), to create a 2-D matrix in order to store the shortest distance for each pair of nodes.



nodes by Floyd's Algorithm.

Solution:

$$D^0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & \infty & 5 \\ 2 & 2 & 0 & \infty & 4 \\ 3 & \infty & 1 & 0 & [illegible] \\ [illegible] & [illegible] & \infty & 2 & [illegible] \end{array}$$



tion:

$$D^0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & \infty & 5 \\ 2 & 2 & 0 & \infty & 4 \\ 3 & \infty & 1 & 0 & \infty \\ 4 & \infty & \infty & 2 & 0 \end{array}$$

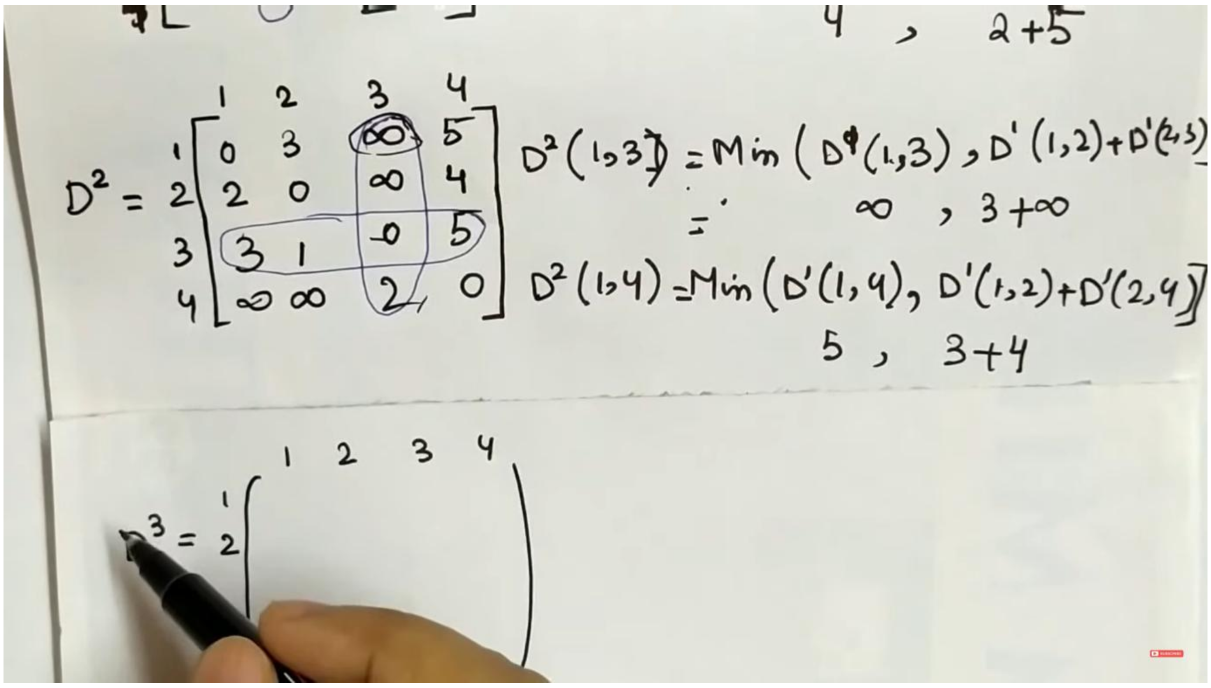
$$D^1 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & \infty & 5 \\ 2 & 2 & 0 & \infty & 4 \\ 3 & \infty & & 0 & \\ 4 & \infty & & & 0 \end{array}$$

$D^1(2,3) = \text{Min}[D^0(2,3), D^0[2,1] + D^0[1,3]]$

$\infty \qquad 2 + \infty$

$D^1(2,4) = \text{Min}[D^0(2,4), D^0[2,1] + D^0[1,4]]$

$4 \quad , \quad 2 + 5$



$4 \quad , \quad 2 + 5$

$$D^2 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & \infty & 5 \\ 2 & 2 & 0 & \infty & 4 \\ 3 & 3 & 1 & 0 & 5 \\ 4 & \infty & \infty & 2 & 0 \end{array}$$

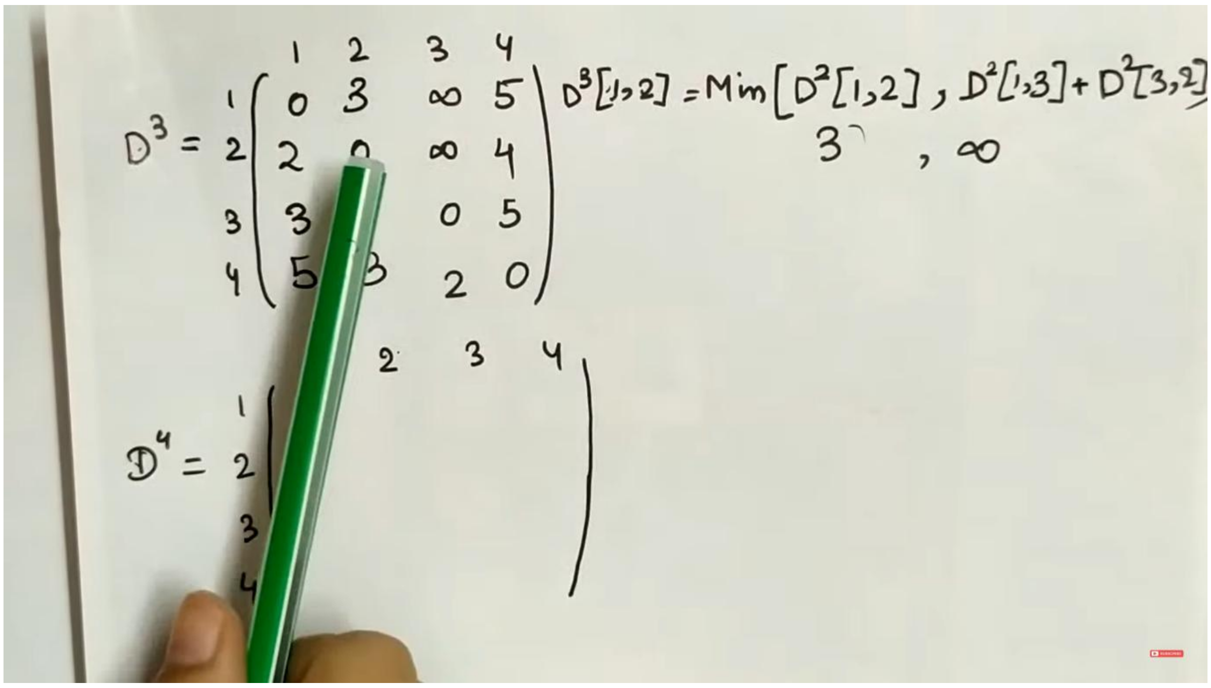
$D^2(1,3) = \text{Min}(D^1(1,3), D^1(1,2) + D^1(2,3))$

$= \quad \infty \quad , \quad 3 + \infty$

$D^2(1,4) = \text{Min}(D^1(1,4), D^1(1,2) + D^1(2,4))$

$5 \quad , \quad 3 + 4$

$$D^3 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & & & & \\ 2 & & & & \end{array}$$

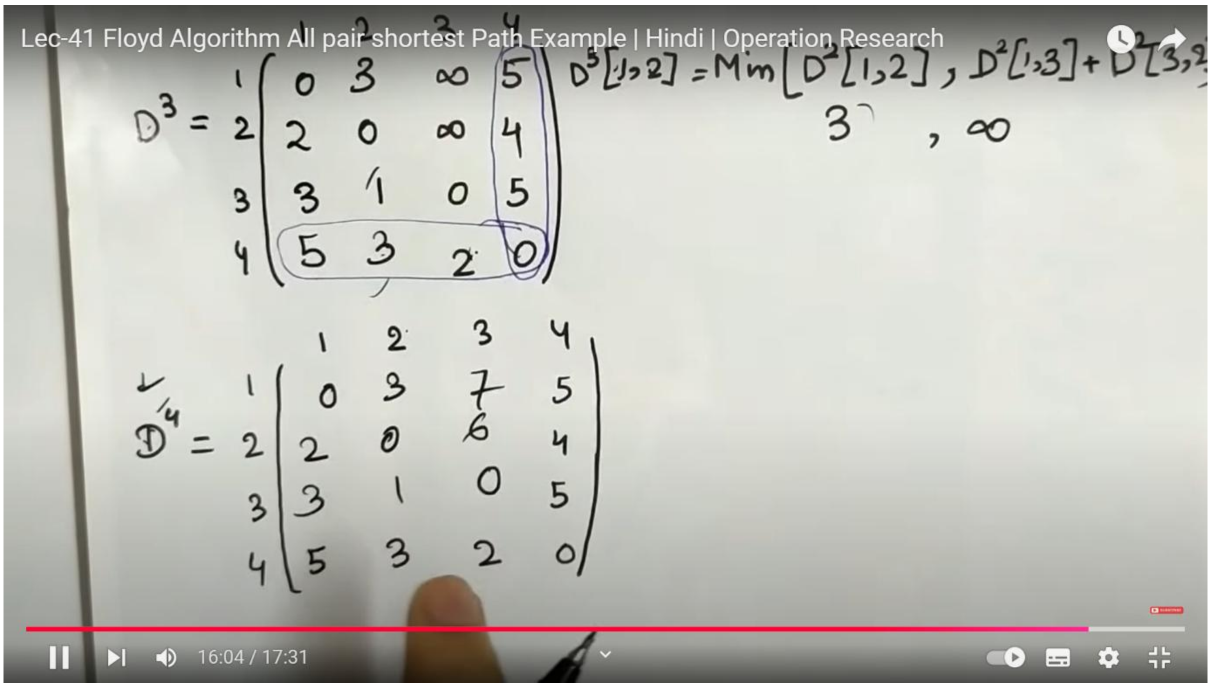


$$D^3 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & \infty & 5 \\ 2 & 2 & 0 & \infty & 4 \\ 3 & 3 & [illegible] & 0 & 5 \\ 4 & 5 & 3 & 2 & 0 \end{array}$$

$$D^3[1,2] = \mathrm{Min}\left[D^2[1,2],\ D^2[1,3] + D^2[3,2]\right]$$

$$3,\ \infty$$

$$D^4 = \begin{array}{c|cccc} & & 2 & 3 & 4 \\ \hline 1 & & & & \\ 2 & & & & \\ 3 & & & & \\ 4 & & & & \end{array}$$



Lec-41 Floyd Algorithm All pair shortest Path Example | Hindi | Operation Research

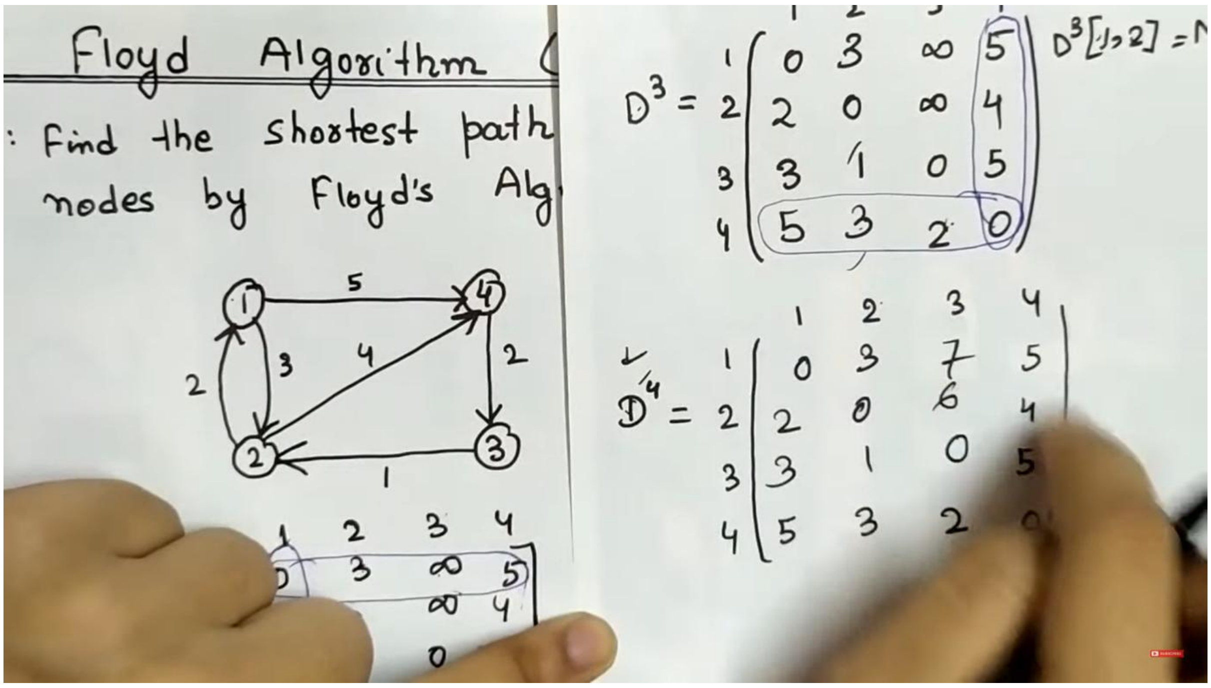
$$D^3 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & \infty & 5 \\ 2 & 2 & 0 & \infty & 4 \\ 3 & 3 & 1 & 0 & 5 \\ 4 & 5 & 3 & 2 & 0 \end{array}$$

$$D^3[1,2] = \mathrm{Min}\left[D^2[1,2],\ D^2[1,3] + D^2[3,2]\right]$$

$$3,\ \infty$$

✓

$$D^4 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 7 & 5 \\ 2 & 2 & 0 & 6 & 4 \\ 3 & 3 & 1 & 0 & 5 \\ 4 & 5 & 3 & 2 & 0 \end{array}$$



Floyd Algorithm (

: Find the shortest path

nodes by Floyd's Alg

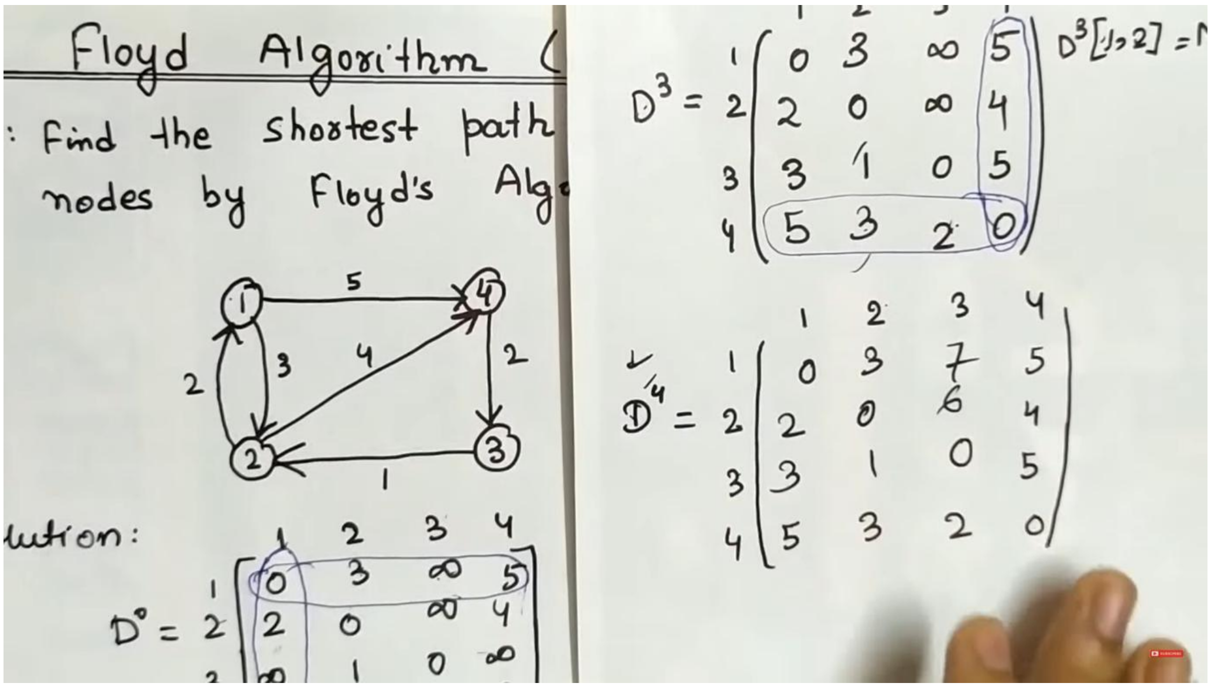
$$\begin{tikzcd}
1 \arrow[r, "5"] \arrow[d, bend left, "3"] & 4 \arrow[d, "2"] \\
2 \arrow[u, bend left, "2"] \arrow[ur, "4"] & 3 \arrow[l, "1"]
\end{tikzcd}$$

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline & 0 & 3 & \infty & 5 \\ & & & \infty & 4 \\ & & & 0 & \end{array}$$

$$D^3 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & \infty & 5 \\ 2 & 2 & 0 & \infty & 4 \\ 3 & 3 & 1 & 0 & 5 \\ 4 & 5 & 3 & 2 & 0 \end{array}$$

$D^3[1,2] =$

$$\checkmark\ D^4 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 7 & 5 \\ 2 & 2 & 0 & 6 & 4 \\ 3 & 3 & 1 & 0 & 5 \\ 4 & 5 & 3 & 2 & 0 \end{array}$$



Floyd Algorithm (

: Find the shortest path

nodes by Floyd's Alg

$$\begin{tikzcd}
1 \arrow[r, "5"] \arrow[d, bend left, "3"] & 4 \arrow[d, "2"] \\
2 \arrow[u, bend left, "2"] \arrow[ur, "4"] & 3 \arrow[l, "1"]
\end{tikzcd}$$

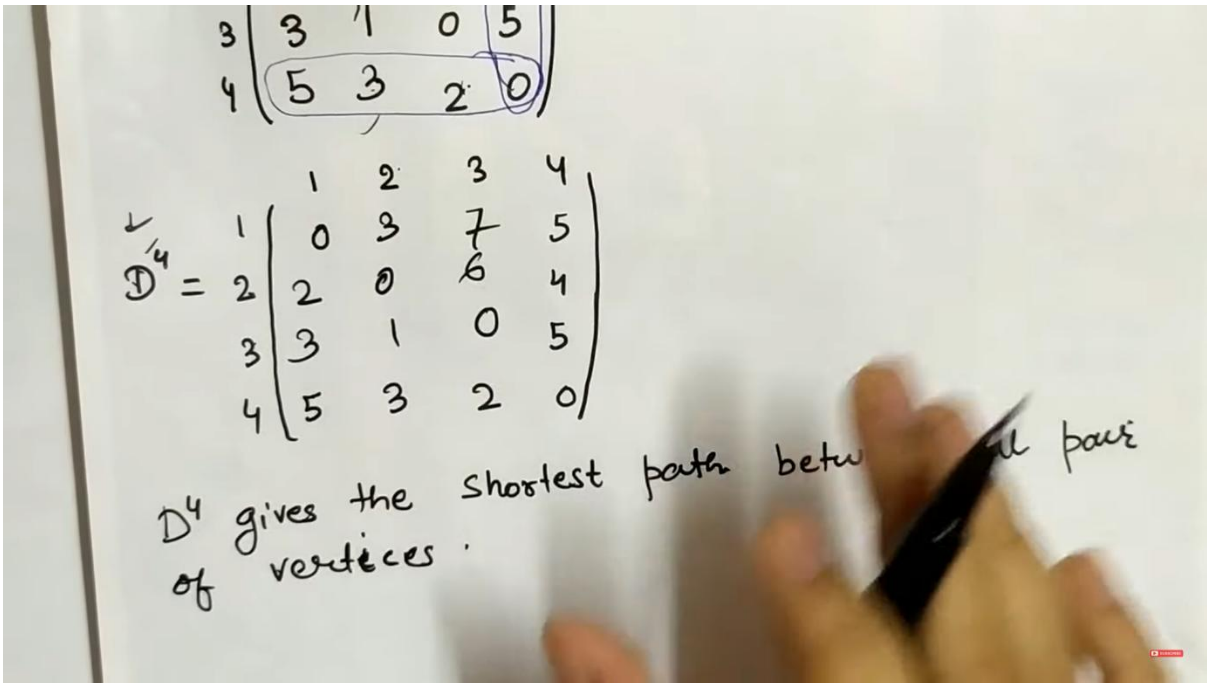
lution:

$$D^0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & \infty & 5 \\ 2 & 2 & 0 & \infty & 4 \\ & \infty & 1 & 0 & \infty \end{array}$$

$$D^3 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & \infty & 5 \\ 2 & 2 & 0 & \infty & 4 \\ 3 & 3 & 1 & 0 & 5 \\ 4 & 5 & 3 & 2 & 0 \end{array}$$

$D^3[1,2] =$

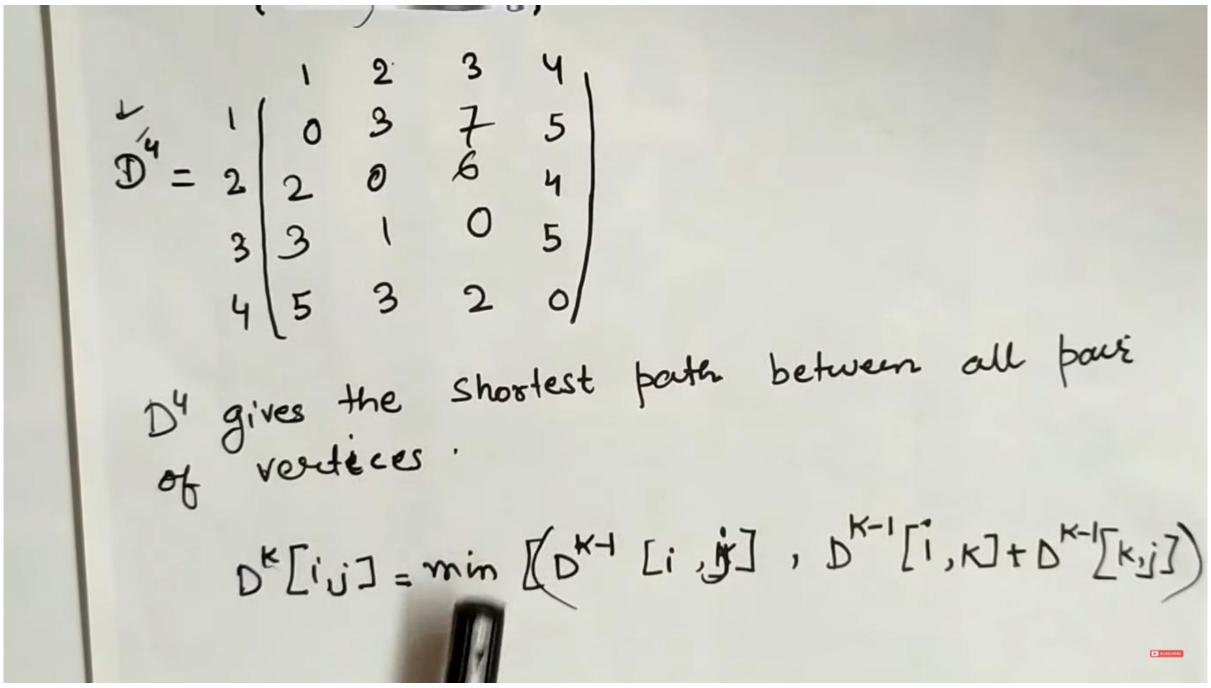
$$\checkmark\ D^4 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 7 & 5 \\ 2 & 2 & 0 & 6 & 4 \\ 3 & 3 & 1 & 0 & 5 \\ 4 & 5 & 3 & 2 & 0 \end{array}$$



$$\begin{array}{c|cccc} 3 & 3 & 1 & 0 & 5 \\ 4 & 5 & 3 & 2 & 0 \end{array}$$

$$D^4 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 7 & 5 \\ 2 & 2 & 0 & 6 & 4 \\ 3 & 3 & 1 & 0 & 5 \\ 4 & 5 & 3 & 2 & 0 \end{array}$$

D^4 gives the shortest path betw[illegible]ll pair of vertices.



$$D^4 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 7 & 5 \\ 2 & 2 & 0 & 6 & 4 \\ 3 & 3 & 1 & 0 & 5 \\ 4 & 5 & 3 & 2 & 0 \end{array}$$

D^4 gives the shortest path between all pair of vertices.

$D^K[i,j] = \min[(D^{K-1}[i,j], D^{K-1}[i,K] + D^{K-1}[K,j])$

- Floyd–Warshall's Algorithm is used to find the shortest paths between all pairs of vertices in a graph, where each edge in the graph has a weight which is positive or negative.
- The biggest advantage of using this algorithm is that all the shortest distances between any 2 vertices could be calculated in O(V3), where V is the number of vertices in a graph.
- Floyd-Warshall algorithm is also called as Floyd's algorithm, Roy-Floyd algorithm, Roy-Warshall algorithm, or WFI algorithm.

- This algorithm follows the dynamic programming approach to find the shortest paths.

Algorithm

For a graph with N vertices:

Step 1: Initialize the shortest paths between any 2 vertices with Infinity.

Step 2: Find all pair shortest paths that use 0 intermediate vertices, then find the shortest paths that use 1 intermediate vertex and so on.. until using all N vertices as intermediate nodes.

Step 3: Minimize the shortest paths between any 2 pairs in the previous operation.

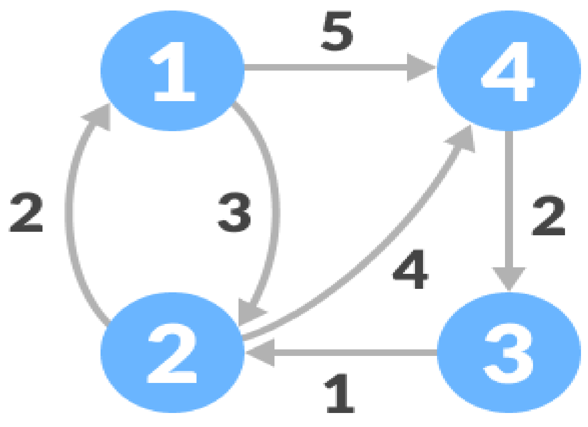
Step 4: For any 2 vertices (i,j) , one should actually minimize the distances between this pair using the first K nodes, so the shortest path will be: min(dist[i][k]+dist[k][j],dist[i][j]).

dist[i][k] represents the shortest path that only uses the first K vertices, dist[k][j] represents the shortest path between the pair k,j. As the shortest path will be a concatenation of the shortest path from i to k, then from k to j.

```
for(int k = 1; k <= n; k++){
   for(int i = 1; i <= n; i++){
      for(int j = 1; j <= n; j++){
         dist[i][j] = min( dist[i][j], dist[i][k] + dist[k][j] );
      }
   }
}
```

Example

Let the given graph be:



Follow the steps below to find the shortest path between all the pairs of vertices.

1. Create a matrix A0 of dimension n*n where n is the number of vertices. The row and the column are indexed as i and j respectively. i and j are the vertices of the graph.

Each cell A[i][j] is filled with the distance from the ith vertex to the jth vertex. If there is no path from ith vertex to jth vertex, the cell is left as infinity.

$$A^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & \infty & 4 \\ \infty & 1 & 0 & \infty \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

2. Now, create a matrix A1 using matrix A0. The elements in the first column and the first row are left as they are. The remaining cells are filled in the following way.

Let k be the freezing vertex in the shortest path from source to destination. In this step, k is the first vertex.A[i][j] is filled with (A[i][k] + A[k][j]) if (A[i][j] > A[i][k] + A[k][j]).

That is, if the direct distance from the source to the destination is greater than the path through the vertex k, then the cell is filled with A[i][k] + A[k][j].

In this step, k is vertex 1.

We calculate the distance from the source vertex to destination vertex through this vertex k.

$$A^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & & \\ \infty & & 0 & \\ \infty & & & 0 \end{bmatrix} \end{array} \longrightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & 9 & 4 \\ \infty & 1 & 0 & 8 \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

For example: For A1[2, 4], the direct distance from vertex 2 to 4 is 4 and the sum of the distance from vertex 2 to 4 through vertex (i.e. from vertex 2 to 1 and from vertex 1 to 4) is 7. Since 4 < 7, A0[2, 4] is filled with 4.

3. In a similar way, A2 is created using A3. The elements in the second column and the second row are left as they are.

In this step, k is the second vertex (i.e. vertex 2). The remaining steps are the same as in **step 2**.

$$A^2 = \begin{array}{c|cccc|} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & & \\ 2 & 2 & 0 & 9 & 4 \\ 3 & & 1 & 0 & \\ 4 & & \infty & & 0 \\ \hline \end{array} \longrightarrow \begin{array}{c|cccc|} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 9 & 5 \\ 2 & 2 & 0 & 9 & 4 \\ 3 & 3 & 1 & 0 & 5 \\ 4 & \infty & \infty & 2 & 0 \\ \hline \end{array}$$

4. Similarly, A3 and A4 is also created.

$$A^3 = \begin{array}{c|cccc|} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & & \infty & \\ 2 & & 0 & 9 & \\ 3 & \infty & 1 & 0 & 8 \\ 4 & & & 2 & 0 \\ \hline \end{array} \longrightarrow \begin{array}{c|cccc|} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 9 & 5 \\ 2 & 2 & 0 & 9 & 4 \\ 3 & 3 & 1 & 0 & 5 \\ 4 & 5 & 3 & 2 & 0 \\ \hline \end{array}$$

5. A4 gives the shortest path between each pair of vertices.

Confused to select the shortest path, then use the below code to find it. But enter the correct distances.

$$A^4 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & & & 5 \\ 2 & & 0 & & 4 \\ 3 & & & 0 & 5 \\ 4 & 5 & 3 & 2 & 0 \end{array} \longrightarrow \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 7 & 5 \\ 2 & 2 & 0 & 6 & 4 \\ 3 & 3 & 1 & 0 & 5 \\ 4 & 5 & 3 & 2 & 0 \end{array}$$

5. A4 gives the shortest path between each pair of vertices.

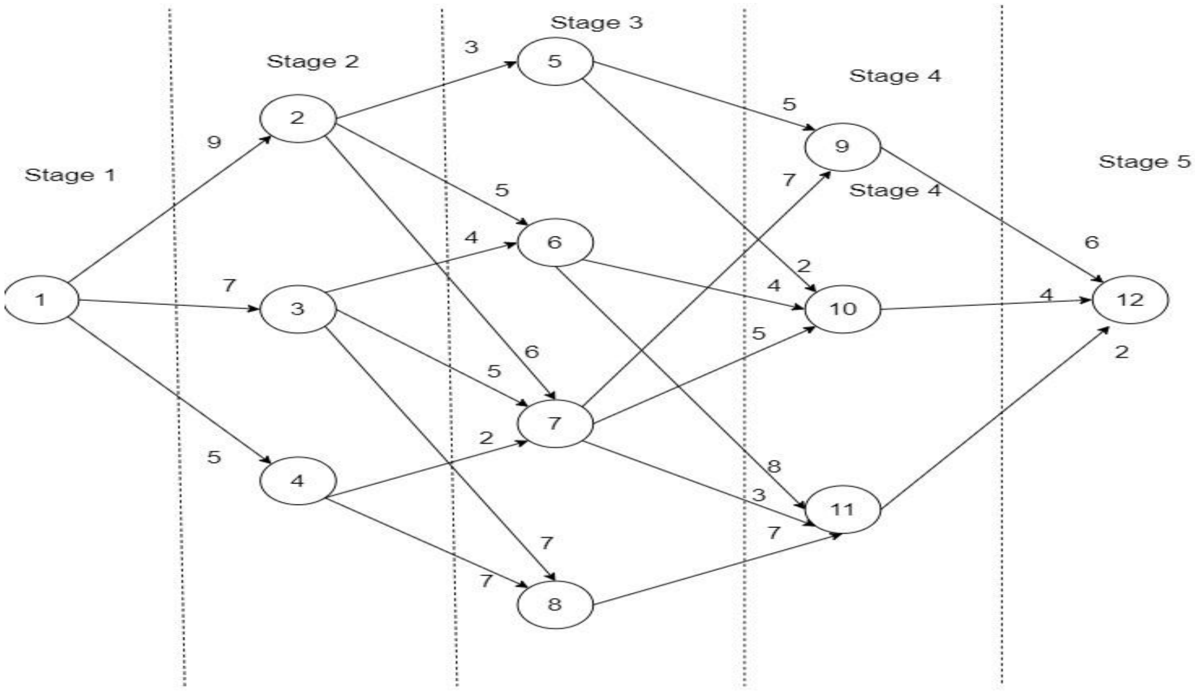
Confused to select the shortest path, then use the below code to find it. But enter the correct distances.

Multistage graph

A multistage graph is a directed and weighted graph, in which all vertices are divided into stages, such that all the edges are only directed from the vertex of the current stage to the vertex of the next stage (i.e there is no edge between the vertex of the same stage and to the previous stage vertex). Both the first and last stage contains only one vertex called source and destination/sink respectively.

Mathematically, a multistage graph can be defined as:

Multistage graph G = (V, E, W) is a weighted directed graph in which vertices are partitioned into k ≥ 2 disjoint subsets V = {V1, V2, …, Vk} such that if edge (u, v) is present in E then u ∈ Vi and v ∈ Vi+1, 1 ≤ i ≤ k. The sets V1 and Vk are such that |V1 | = |Vk|=1



In the multistage graph problem, we are required to find the shortest path between the source and the sink/destination. This problem can be easily solved by dynamic programming. Before getting started, let's understand what is Dynamic Programming (DP).

The multistage graph problem can be solved in two ways using dynamic programming:

1. Forward approach
2. Backward approach

Forward approach

In the forward approach, we assume that there are k stages in the graph. We start from the last stage and find out the cost of each and every node to the first stage. We then find out the minimum cost path from the source to destination (i.e from stage 1 to stage k).

Procedure:

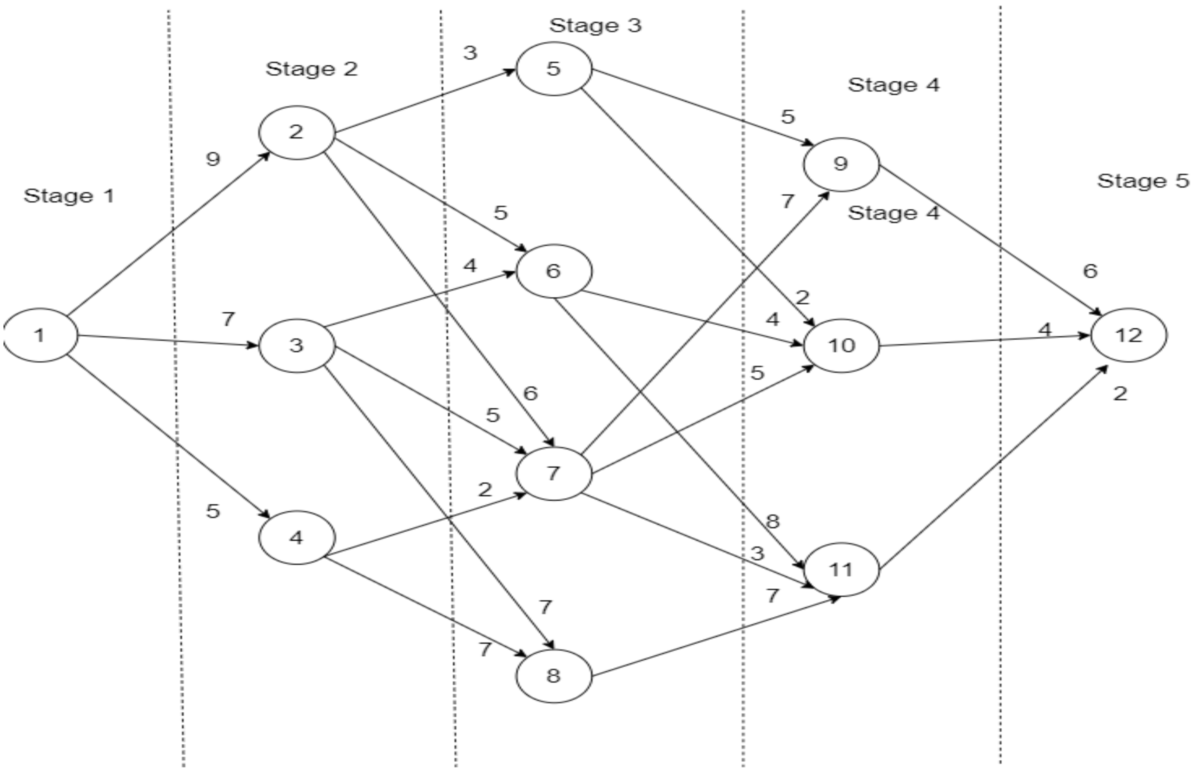
1. Maintain a cost array cost[n] which stores the distance from any vertex to the destination.
2. If a vertex has more than one path, then the path with the minimum distance is chosen and the intermediate vertex is stored in the distance array D[n]. This will give us a minimum cost path from each and every vertex.
3. Finally, the cost from 1st vertex cost(1,k) gives the minimum cost of the shortest path from source to destination.
4. For finding the path, start from vertex-1 then the distance array D(1) will give the minimum cost neighbour vertex which in turn gives the next nearest vertex and proceed in this way till we reach the destination. For a 'k' stage graph, there will be 'k' vertex in the path.
5. For forward approach,

$\text{Cost}(i,j) = \min \{C(j,l) + \text{Cost}(i+1,l)\}\ l \in V_i + 1\ \&\ (j,l) \in E$

Algorithm:

```
MULTI_STAGE(G, k, n, p)
// Description: Solve multi-stage problem using dynamic programming
// Input:
k: Number of stages in graph G = (V, E)
c[i, j]:Cost of edge (i, j)
// Output: p[1:k]:Minimum cost path
cost[n] ← 0
for j ← n — 1 to 1 do
//Let r be a vertex such that (j, r) E and c[j, r] + cost[r] is minimum
```

```
cost[j] ← c[j, r] + cost[r]
π[j] ← r
end
//Find minimum cost path
p[1] ← 1
p[k] ← n
for j ← 2 to k — 1 do
p[j] ← π[p[j — 1]]
end
```



In the above graph, cost of an edge is represented as c(i, j). We need to find the minimum cost path from vertex 1 to vertex 12. Using the below formula we can find the shortest cost path from source to destination: cost(i,j)=minc(j,l)+cost(i+1,l)cost(i,j)=minc(j,l)+cost(i+1,l)

Step 1

Step 1 uses the forwarded approach (cost(5,12) = 0).

Here, 5 represents the stage number and 12 represents a node in that stage. Since there are no outgoing edges from vertex 12, the cost is 0.

Step 2

cost(4,9)=c(9,12)=6

cost(4,10)=c(10,12)=4 cost(4,11)=c(11,12)= 2

Step 3

cost(3,5)=min{c(5,9)+cost(4,9),c(5,10)+cost(4,10)}

min{5+6,2+4}

min{11,6}=6

cost(3,5)=6

cost(3,6)=min{c(6,10)+cost(4,10),c(6,11)+cost(4,11)}

min{4+4,8+2}

min{8,10}=8

cost(3,6)=8

cost(3,7)=min{c(7,9)+cost(4,9),c(7,10)+cost(4,10),c(7,11)+cost(4,11)} min{7+6,5+4,3+2}

min{13,9,5}=5

cost(3,7)=5

cost(3,8)=c(8,11)+cost(4,11)=7+2=9 cost(3,8)=9

Step 4

cost(2,2)=min{c(2,5)+cost(3,5),c(2,6)+cost(3,6),c(2,7)+cost(3,7)} min{3+6,5+8,6+5}

min{9,13,11}=9

cost(2,2)=9

cost(2,3)=min{c(3,6)+cost(3,6),c(3,7)+cost(3,7),c(3,8)+cost(3,8)} min{4+8,5+5,7+9}

min{12,10,16}=10

cost(2,3)=10

cost(2,4)=min{c(4,7)+cost(3,7),c(4,8)+cost(3,8)}

min{2+5,7+9}

min{7,16}=7

cost(2,4)=7

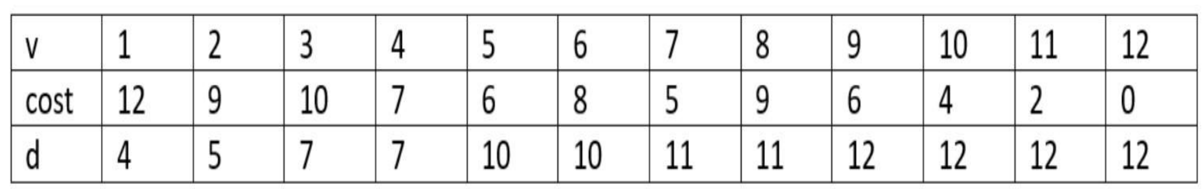
Step-5

cost(1,1)=min{c(1,2)+cost(2,2),c(1,3)+cost(2,3),c(1,4)+cost(2,4)} min{9+9,7+10,5+7}

min {18,17,12}=12

cost(1,1)=12

From the above calculations we get:



v	1	2	3	4	5	6	7	8	9	10	11	12
cost	12	9	10	7	6	8	5	9	6	4	2	0
d	4	5	7	7	10	10	11	11	12	12	12	12

Here, d represents which vertex has given minimum cost to above vertex

Therefore:

d[1]=4

d[4]=7

d[7]=11

d[11]=12

shortest path 1--4--7--11--12 i.e 12

Page No. ______
Date ______

Activity selection Problem

The activity selection problem is combinatorial optimization problem concerning the selection of non-conflecting activities to perform within a given time frame, given set of activities each marked by a start time (si) and finish time (fi)

1. Sort the activities as per finishing time in asending order.
2. select the first activity
3. Selection of new activity if its starting time is greater than or equal to the previously selected activity's finish time
4. Repeat step 3. till all activities are checked.

Activity	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
start	1	0	1	4	2	5	3	4
Finish	3	4	2	6	9	8	5	5

1. Sort the activity according the finishing time assending

Activity	a_3	$\overset{*}{a_1}$	a_2	a_7	a_8	a_4	a_6	a_5
start	1	1	0	3	4	4	5	2
Finish	2	3	4	5	5	6	8	9

2. Select the first activity

Activity	a_3
Start	1
Finish	2

Page No
Date

3. Selection of new activity if its starting time is greater than or equal to the previously selected activity's finish time.

Activity	a3	a7	a6
start	1	3	5
Finish	2	5	8

$s_i \geq f_k$

Previous activity a3
starting time one

① next activity a_1 which startime time is 1 which is not greater than the finishing time of a3 which is 2

a_1 $1 >= 2$ ✗ condition not satisfied

② a_2 $0 >= 2$ ✗ ~~condition not satisfied~~

③ a_7 ~~7~~

$3 >= 2$ ✓ condition satisfied
so a_7 is included in the table.

④ a_8

$4 \geq 5$ ✗ condition not satisfied

⑤ a_4

$4 7 = 5$ ✗ not satisfied

⑥ a_6

$5 \geq 5$ ✓ condition satisfied so included

⑦ a_5

$2 \geq 8$ ✗ condition not satisfied

and we finish the table so the final selected Activity table is

Activity	a3	a7	a6
start time	1	3	5
finish time	2	5	8

(a)

Activity i	a1	a2	a3	a4	a5	a6	a7	a8	a9	a10
Start time Si	1	2	3	4	7	8	9	9	11	12
Finishing time fi	3	5	4	7	10	9	11	13	12	14

- Sort the given activities according to assending order of finish time

Activity i	a1	a3	a2	a4	a6	a5	a7	a9	a8	a10
Start time Si	1	3	2	4	8	7	9	11	9	12
finishing time fi	3	4	5	7	9	10	11	12	13	14

- i - Activity
  Si - Start time
  fi - finish time

$$S_i \geq F_i$$

Activity	a1	a3	a4	a6	a7	a9	a10
Start	1	3	4	8	9	11	12
finish	3	4	7	9	11	12	14

1) a2 ⇒
   2 ≥ 4 condition not satisfied ✗

2) a4 ⇒ 4 ≥ 4 condition satisfied ✓

3) a6 ⇒ 8 ≥ 7 condition satisfied ✓

4) a5 ⇒ 7 ≥ 9 condition not satisfied ✗

5) $a_7 \Rightarrow 9 \geq 9$ condition satisfied ✓

6) $a_9 \Rightarrow 11 \geq 11$ condition satisfied ✓

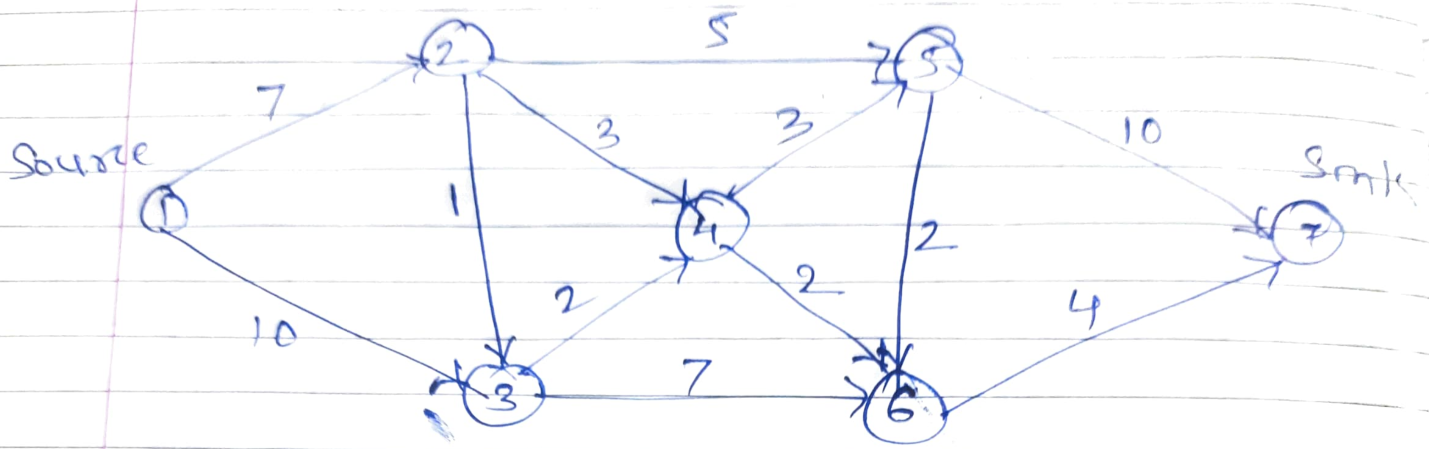
7) $a_8 \Rightarrow 9 \geq 12$ condition not satisfied ✗

8) $a_{10} \Rightarrow 12 \geq 12$ condition satisfied ✓

Total 7 activities are selected in given exam

Maximum Flow Problem

Ex. Find the Maximum Flow through the given network using ford fulkerson algorithm.



1. The Ford Fulkerson Algorithm is used for solving maximum flow problem

2. Basic Terms
   - Source
   - Sink
   - Capacity and bottle neck capacity
   - Flow
   - Argumenting path
   - Residual ~~path~~ capacity

- Source: The source vertex has all outward edges no inward edge

- Sink: Sink will have all inward edges no outward edges.

- capacity: The value present on edge from a vertex to next. You can say a weight of the edge.

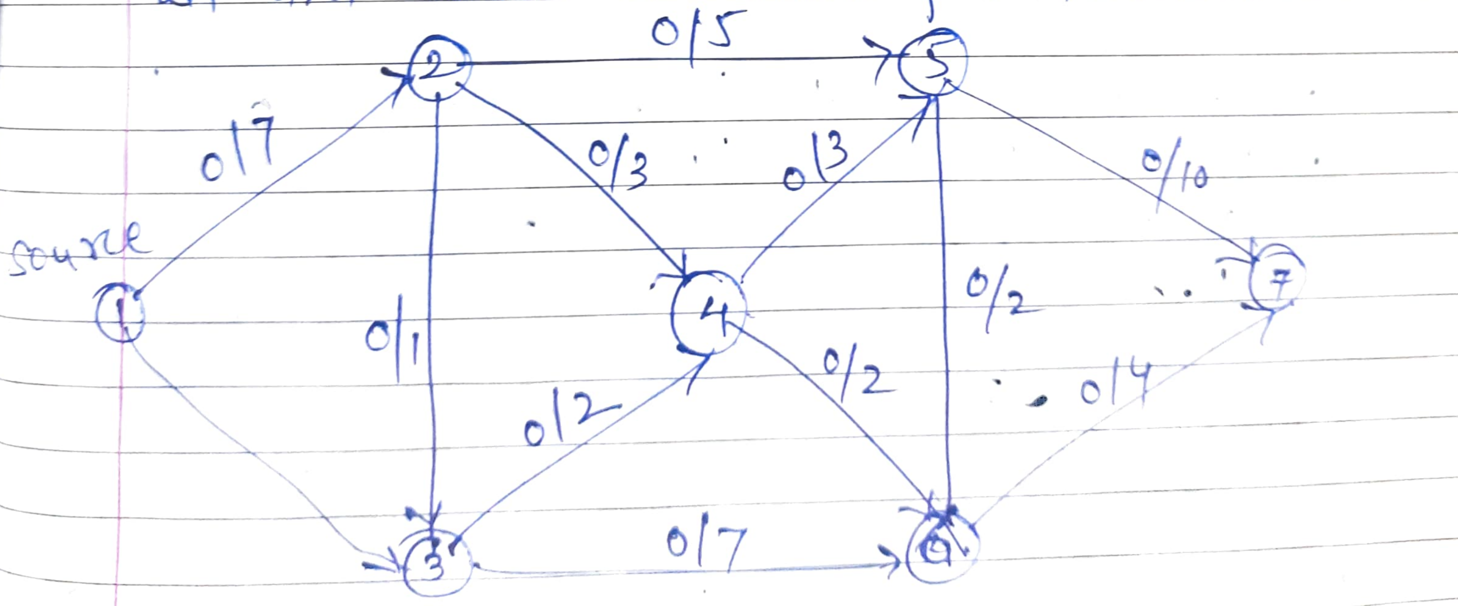
- Bottleneck capacity : Bottoleneck capacity of the path is minimum capacity of any edge on the path.

- Flow : The amount of something that passes through network edges.

- The Argumenting Path : The choosen path from source to sink is called as argumenting path.

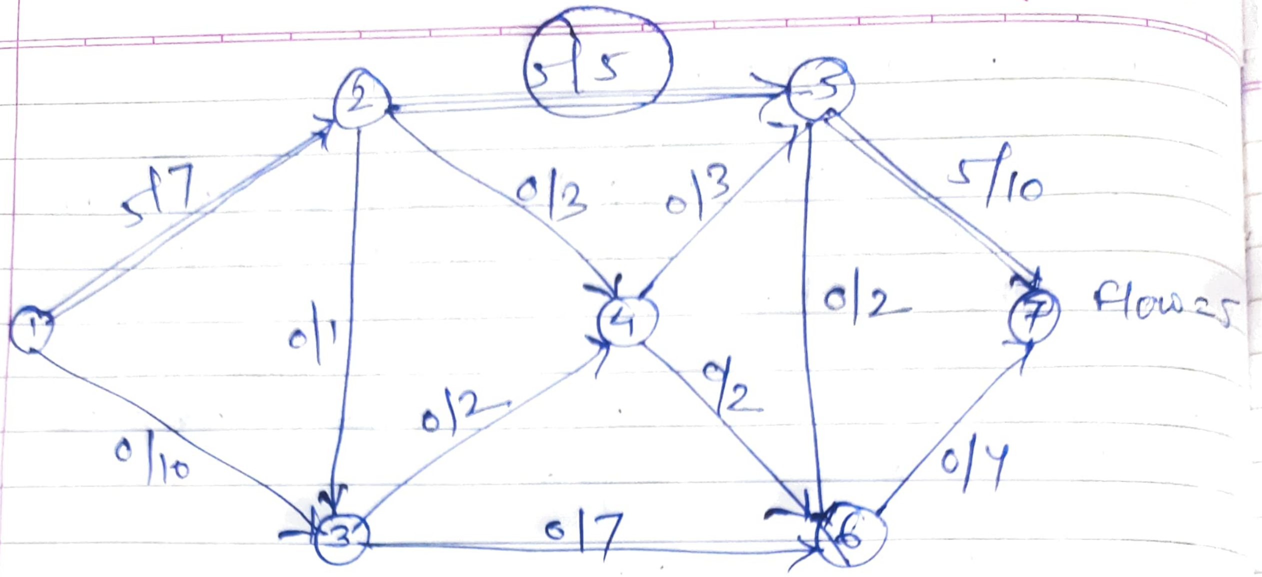
- Residual Capacity : Every edge of residual graph has a value called residual capacity, which is equal to original capacity minus current flow. In simple capacity - flow.

Solution :- Initial we will do the flow zero for all and choose a random path towards sink.



Arugmenting Path
1-2-5-7

Bottleneck capacity
5 weight
(minimum amongest the choosen path)

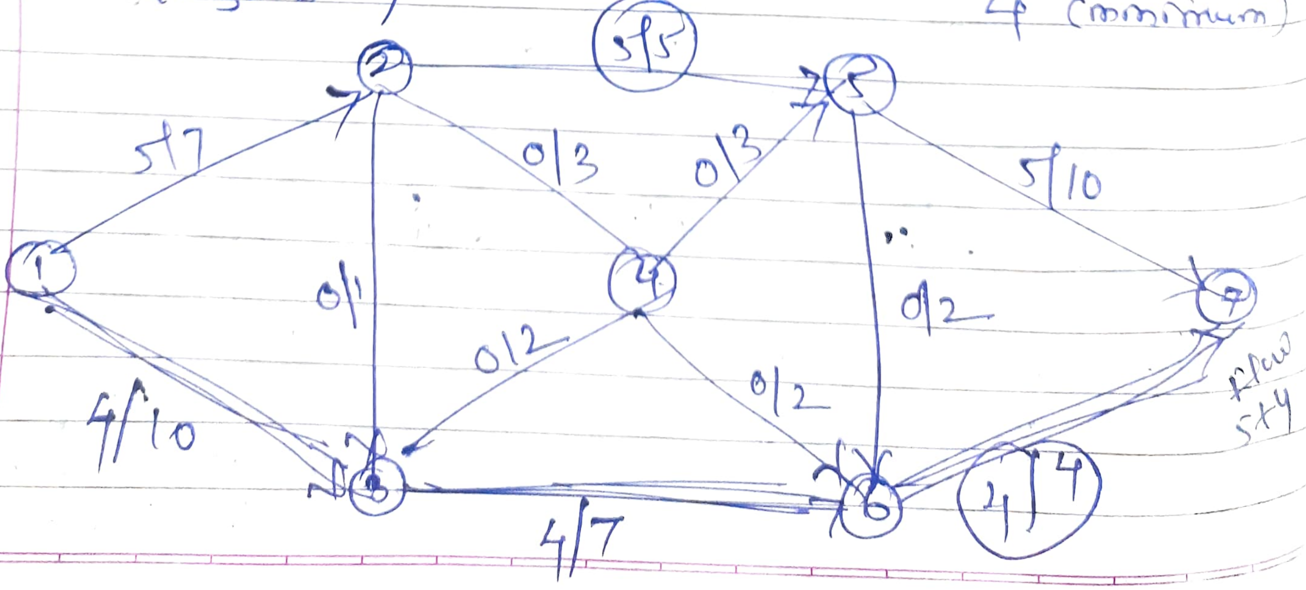


- After find out the path and flow we will find out residual capacity for the choosen path.
  Residual capacity = ~~flow~~ Capacity - flow

- So vertex (2) and (5) the residual capacity is 0, which means the path from 2 to 5 will not be considered for next consideration.

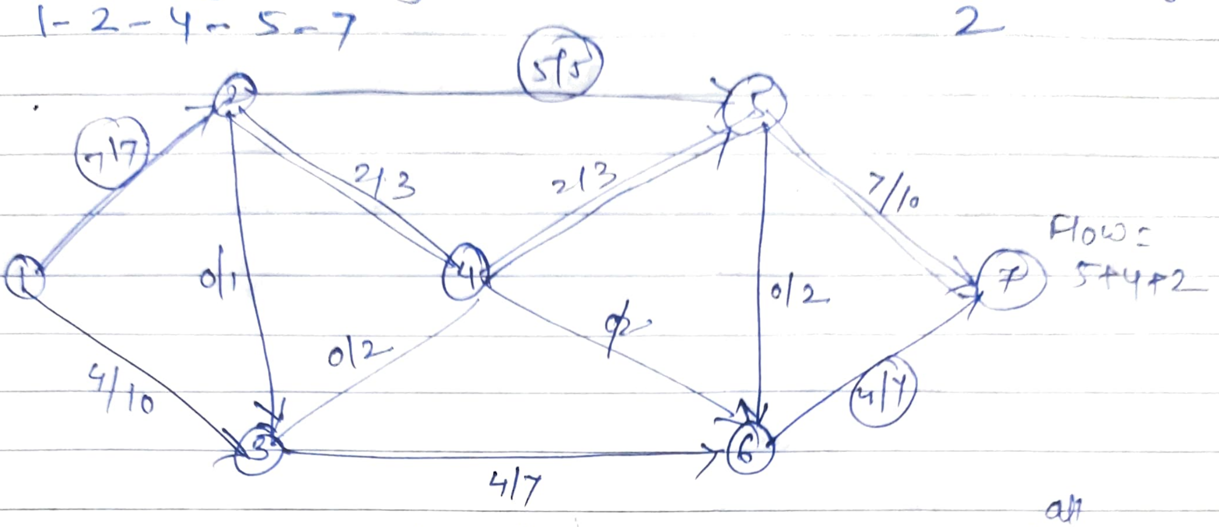
- Next we will find new path and Bottleneck capacity.

Augumenting Path	Bottleneck capacity
1-3-6-7	4 (minimum)



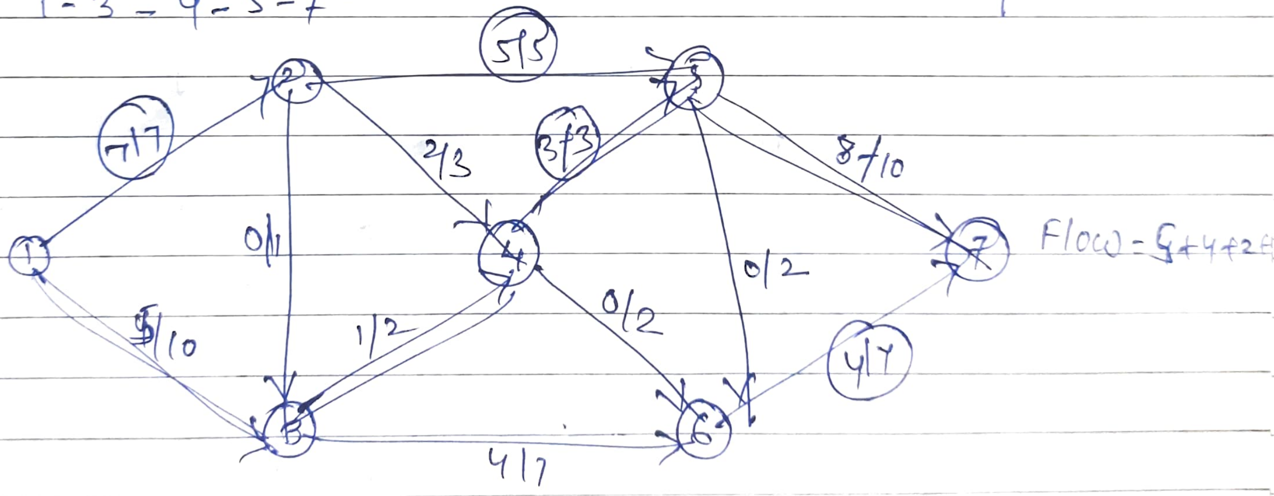
- choose the next path

Argumenting Path	Bottoleneck Capacity
1-2-4-5-7	2



- we will continue to choose the path until all the possibilities would be possible to reach sink.
- choose the next path

Augumenting Path	Bottoleneck Capacity
1-3-4-5-7	1

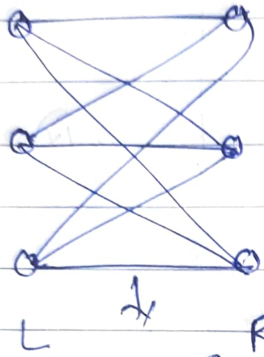


- Maximum flow through the given network using ford fulkerson method is 5+4+2+1 = 12.

Maximum Matching in Bipartite Graphs

- A Bipartite graph is graph whose vertices can b divided into two parts L and R such that every edge connects a vertix in L to vertix in R
$G = (L \cup R, E)$ $L \cup R = V$

eg. ①

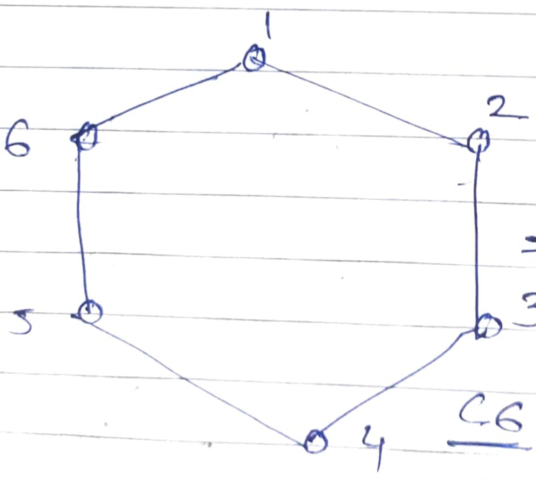


K3,3

complete Bipartite graph

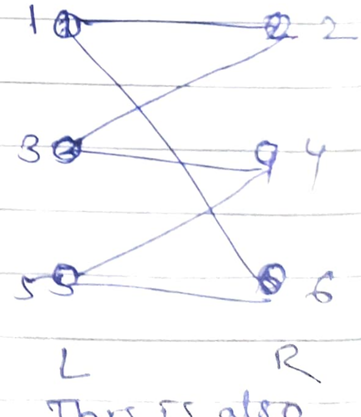
All the nodes of L are connected with all other nodes of R so it is complete bipartite graph

②



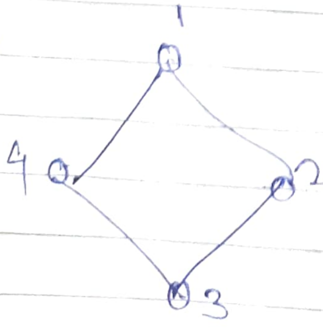
C6

$\Rightarrow$

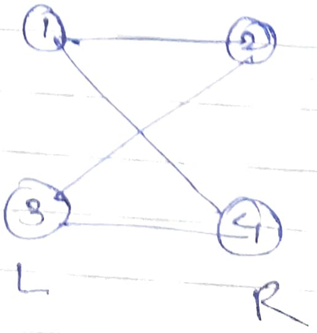


This is also Bipartite graph but not complete

③



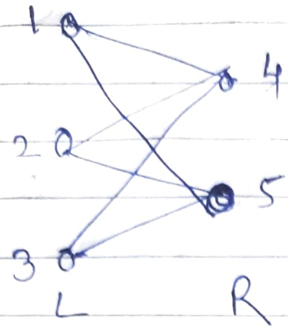
$\Rightarrow$



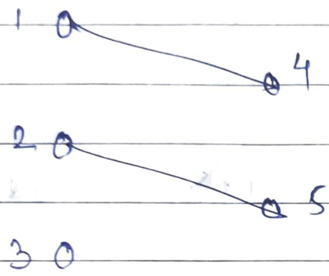
$\Rightarrow$ complete Bipartite graph
K2,2

- Any odd no vertices we can not convert it into a bipartite graph.

* How to find maximum Matching in bipartite graph.



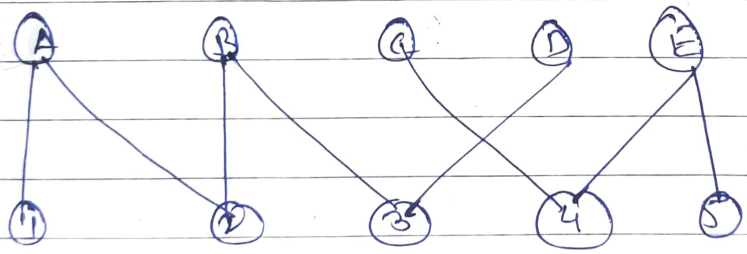
graph is
An Bipartite if and only if it does not have a cycle of an odd length.
- The degree of a bipartite graph is always less than or equal to 1.



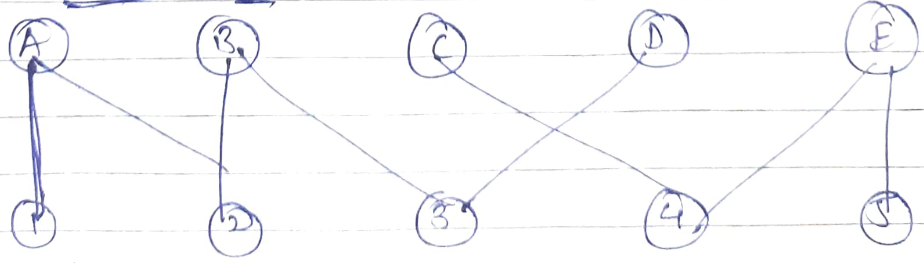
Here 1,4 & 2,5 are said to be Matched vertix &
3 ⇒ unmatched vertix.
$|M| = 2$

Maximum matching in bipartite graph

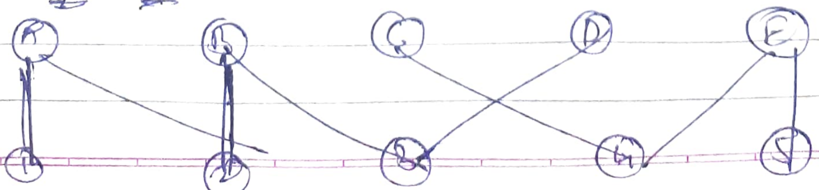
② Maximum Matching in bipartite graph



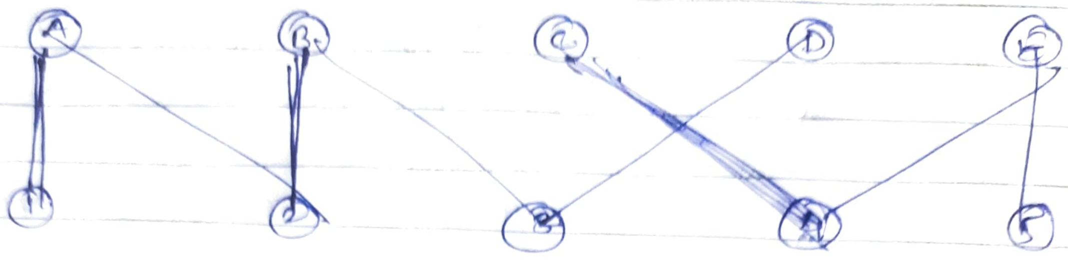
step 1 queue A B C D E



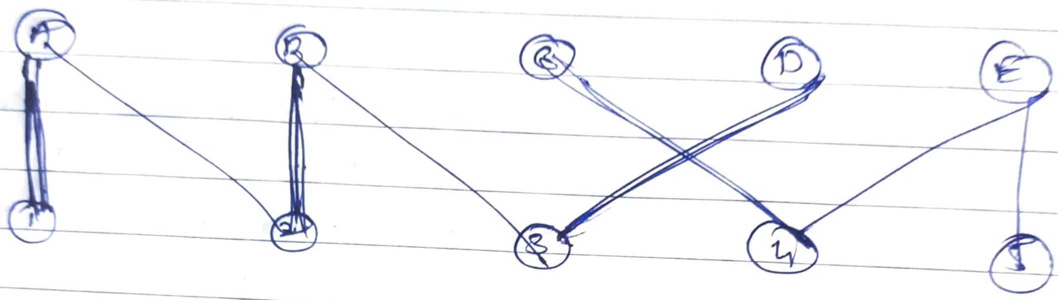
step 2 ~~A~~ ~~B~~ C D E



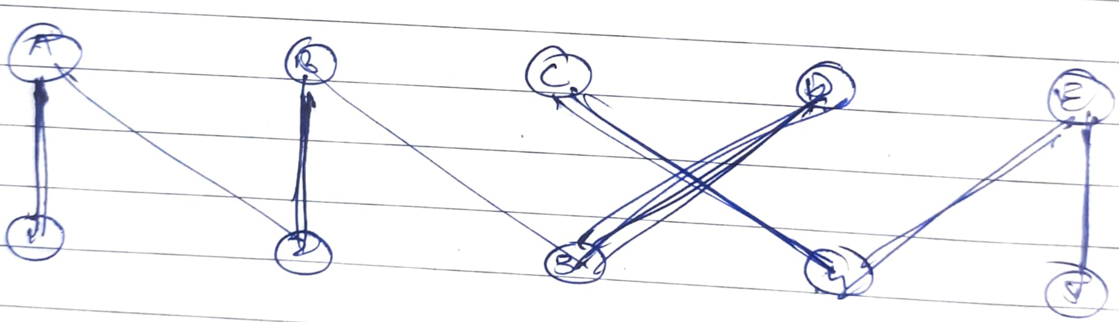
step 3 ~~A~~ ~~B~~ ~~C~~ D E



step 4 ~~A~~ ~~B~~ ~~C~~ ~~D~~ E



steps A B C D E



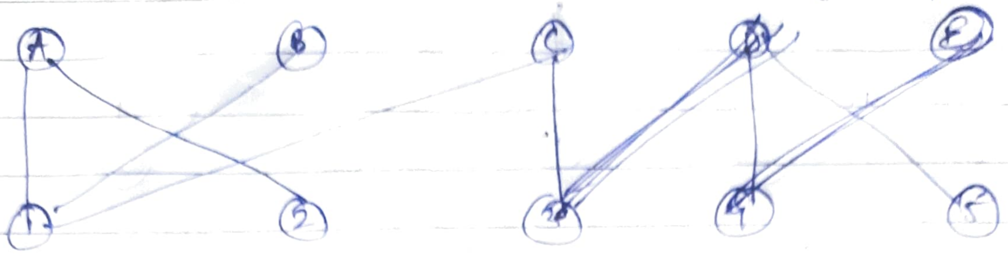
Pairs

A	1
B	2
C	4
D	3
E	5

This is about maximum matching in bipartite graph

Page No. ______
Date ______

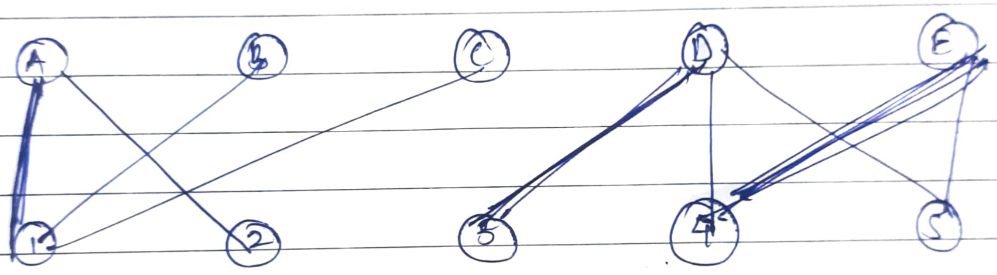
(3) Let M be the Matching in bipartite graph G
Ma {(D,3), (E,4)}



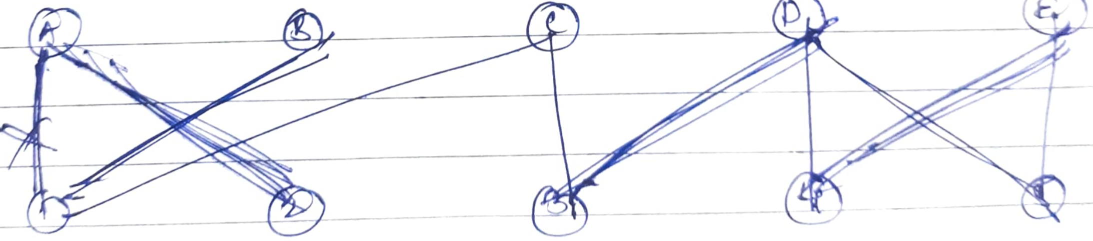
Sol^n find the free vertices
L { A, B, C, D, E }

R { 1, 2, 3, 4, 5 }

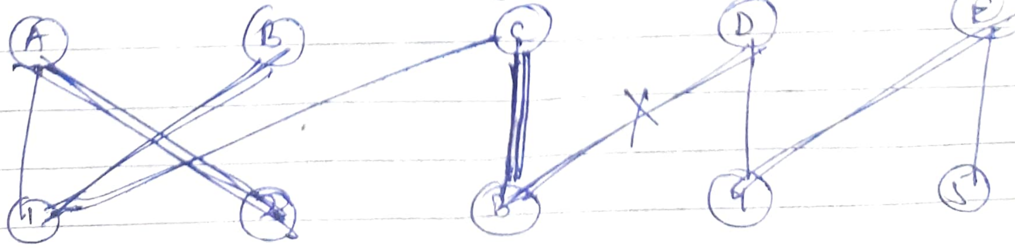
Lets consider ~~free vertices~~ vertex A : Mb {(A,1), (D,3) (E,4)}



consider vertex B : Mc {(A,2), (B,1), (D,3), (E,4)}

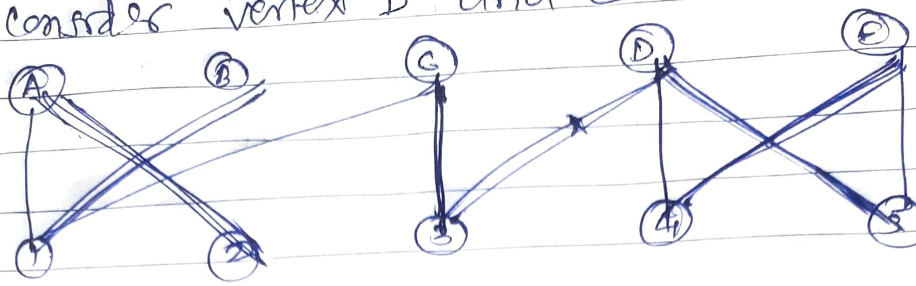


consider vertex C : Md {(A,2) (B,1) (C,3) (E,4)}



Page No.
Date

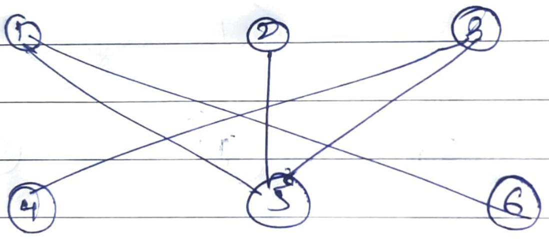
Consider vertex D and E



M = {(A,2), (B,1) (C,3) (D,5) (E,4)}

Thus the maximum matching for the bipartite graph is M {(A,2), (B,1), (C,3) (D,5) (E,4)}

* Find the maximum matching for the following bipartite graph.



Unit No. 4 Iterative Method

1 Simplex Method

Solve the following LPP using simplex Method

Maximize $Z = 12x_1 + 16x_2$

Subject to $10x_1 + 20x_2 \leq 120$

$8x_1 + 8x_2 \leq 80$

x_1 and $x_2 \geq 0$

Solⁿ Add the slack variable to balance the eqⁿ

Max $Z = 12x_1 + 16x_2 + 0S_1 + 0S_2$

subject to $10x_1 + 20x_2 + S_1 = 120$ — ①

$8x_1 + 8x_2 + S_2 = 80$ — ②

$x_1, x_2, S_1, S_2 \geq 0$

Initial Simplex table

CBi (coefficient of basic variable)	Cj	12	16	0	0		
	Basic variable	x_1	x_2	S_1	S_2	Solution	Ra
0	S_1 key Row	10	20 key element	1	0	120	120 = 6
	key column						
0	S_2	8	8	0	1	80	80/8 =
Initial simplex table	Z_j	0	0	0	0	0	

$$Z_j = \sum_{i=1}^{2} (CB_i)(a_{ij})$$

1) $0 \times 10 + 0 \times 8$
$0 + 0 = 0$.

2) $0 \times 20 + 0 \times 8$
$0 + 0 = 0$

3) $0 \times 1 + 0 \times 0$
$0 + 0 = 0$

4) $0 \times 0 + 0 \times 1 = 0$
$0 + 0$

5) $0 \times 120 + 0 \times 80$
$0 + 0 = 0$

* $C_j - Z_j$

$12 - 0 = 12$, $16 - 0 = 0$, $0 - 0 = 0$, $0 - 0 = 0$

$C_j - Z_j = 12 \quad 16 \quad 0 \quad 0$.

Look at optimality condition :

For Max :
all $C_j - Z_j \leq 0$

For Min :
all $C_j - Z_j \geq 0$

• As this problem is Maximize so all $C_j - Z_j \leq 0$ or negative values.

- To reach the optimality condition select the maximum value from $c_j - z_j$.
  Here which is 16.
- Then you need to find key coloum and ratio which is between solution column and key column.
  so, Here $120/20 = 6$ & $80/08 = 10$
- In order find key row select the least value in the ratio column.
  in this case first row is key row and the intersection element is called key element.
- x_2 is entering variable and
  s_1 is leaving variable

Iteration - I

CBi	c_j / B.V.	12	16	0	0	Solution	Ratio
		x_1	x_2	s_1	s_2		
16	x_2	$1/2$	1	$1/20$	0	6	$6/\cancel{}\frac{1}{2}=1$
0	s_2	$\boxed{4}$	0	$-2/5$	1	32	$32/4=$
	z_j	8	16	$4/5$	0		
	$c_j - z_j$	$\boxed{4}$	0	$-4/5$	0		

- In Previous table we found that s_1 is leaving variable and x_2 is entering variab

Page No. ______
Date ______

- In order to find new value just divide the key element with all other old values in the previous. table.

1) $10/20 = 1/2$ 2) $\frac{20}{20} = 1$, 3) $1/20 = 1/20$ 4) $0/20 = 0$

5) $120/20 = 6$

- For second row write the value of s_2 & find out the new value for Iteration I row - II
- To find the the second row value

formula: New value = old vale $-$ $\frac{\text{corrousponding key coloumn value} \times \text{corrousponding key row value}}{\text{Key element}}$

1) $8 - \frac{8 \times 10}{20} = 8 - \frac{80}{20} = 4$

2) $8 - \frac{8 \times 20}{20} = 8 - \frac{160}{20} = 8 - 8 = 0$

3) $\frac{0 - 8 \times 1}{20} = 0 - 8/20 = -2/5$

4) $1 - \frac{8 \times 0}{20} = 1 - \frac{0}{20} = 1 - 0 = 1$

5) $80 - \frac{8 \times 120}{20} = 80 - \frac{960}{20} = 80 - 48 = 32$

Now we have got the final values for Ist row and 2nd row.

Page No.
Date

- for the first row we have divided the key element with all the value.
- for second row we have applied the formula.
- Now you need to find Z_j.
- To find the Z_j value

1) $CB_i \times C_j = 16 \times 1/_3 + 0 \times 4 = 8$

2) $16 \times 1 + 0 \times 0 = 16$, 3) $16 \times 1/_{20} + 0 \times -2/5 = 4/5$

4) $16 \times 0 = 0$

5) $16 \times 0 + 0 \times 1 = 0$

- After finding z_j values you need to compare C_j values with Z_j,

So $C_j - Z_j = 12 - 8 = 4$, ---

check the optimality condition
$C_j - Z_j \leq 0$ or negative values.
in the first iteration we find find some positive values and some negative values so the optimality condition fails so we need to check with second iteration.

- To to proceed next select the max. positive value from iteration - I and find out the ratio

- You need to find ratio with solution value and key column value.

- After the select key row and that will be selected based on minimum value of ratio.

- Row variable is leaving variable and column variable will be entering variable.

Iteration - II

CBi	Cj / Basic value	12 x_1	16 x_2	0 S_1	0 S_2	Solution
16	x_2	0	1	$1/10$	$-1/8$	2
12	x_1	1	0	$-1/10$	$1/4$	8
	Z_j	12	16	$2/5$	1	128
	$C_j - Z_j$	12-12 0	16-16 0	$0-2/5$ $2/5$	0-1=-1	

- For new value divide key element with old value in order find out new values in the IInd Iteration.

1) $\frac{4}{4} = 1$ 2) $0/4 = 0$ 3) $\frac{-2/5}{4} = \frac{-2}{20} = -1/10$

4) $\frac{1}{4} = 1/4$

- To find out the values of first row you need to apply the formula

new value = old value − $\frac{\text{corresponding key column value} \times \text{corresponding key row value}}{\text{Key element}}$

1) $\frac{1/2 - (1/2 \times 4)}{4} = \frac{1}{2} - \frac{(2)}{4} = \frac{1}{2} - \frac{1}{2} = 0$

2) $\frac{1 - (\frac{1}{2} \times 0)}{4} = 1 - (\frac{0}{4}) = 1 - 0 = 1$

Date

5) $= \frac{1}{20} - \frac{\left(\frac{1}{2} \times -\frac{2}{5}\right)}{4} = ~~\frac{1}{20} - \left(\frac{5}{4}\right) = \frac{1}{20} + \frac{5}{5}~~ =$

~~$= \frac{1}{20} - \frac{\left(\frac{1}{2} \times -\frac{2}{5}\right)}{4} \quad = \frac{1}{20} - \left(\frac{4}{2} \times -\frac{8}{5}\right)$~~

~~$= \frac{1}{20} - \left(2 \times -\frac{8}{5}\right)$~~

~~$= \frac{1}{20} - \left(-\frac{16}{5}\right)$~~

~~$= \frac{1}{20} + \frac{16}{5} = \frac{325}{100}$~~

~~$= \frac{1}{20} - \left(\frac{1/2 \times -2/5}{4}\right) = \frac{1}{20} - \left(\frac{\frac{5-4}{10}}{4}\right) = \frac{1}{20} - \left(\frac{\frac{1}{10}}{4}\right)$~~

~~$= \frac{1}{20} - \left(\frac{4}{10}\right)$~~

$= \frac{1}{20} - \left(\frac{1/2 \times -2/5}{4}\right) = \frac{1}{10}$

6) $0 - \frac{\left(\frac{1}{2} \times 1\right)}{4} = -1/8$

7) $6 - \left(\frac{1/2 \times 32}{4}\right) = 2$

After find corresponding row values and coloumn values find out the values of Z_j

1) $16 \times 0 + 12 \times 1 = 0 + 12 = 12$

2) $16 \times 1 + 12 \times 0 = 16 + 0 = 16$

3) $16 \times \frac{1}{10} + 12 \times -\frac{1}{10} = \frac{16}{10} + \left(-\frac{12}{10}\right)$

$= \frac{8}{5} + \left(-\frac{6}{5}\right)$

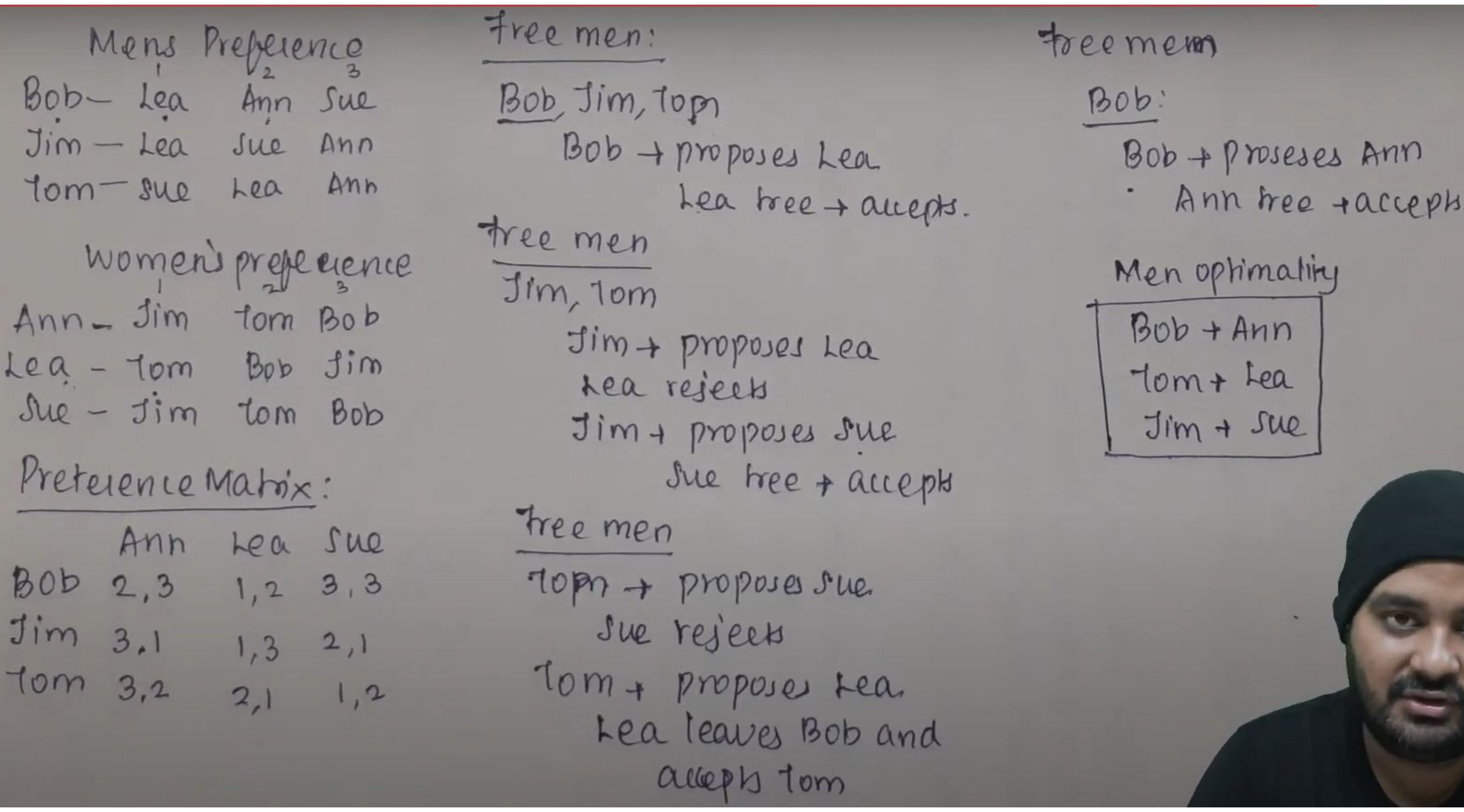
$= \frac{2}{5}$

4) $16 \times -\frac{1}{8} + 12 \times \frac{1}{4} = 1$

5) $16 \times 2 + 12 \times 8 = 128$

After getting the value of Z_j we need to find out the values of $C_j - Z_j$.

- Now check the optimality condition.
The optimality condition says that all the $C_j - Z_j$ values should be either less than zero or negative. So the optimality will be reached.
- So by observing IInd Iteration we have fullfilled the condition and and reached the optimality.
- so $X_1 = 12$, $X_2 = 16$, $Z(opt) = 128$.



Mens Preference

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

Women's preference

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

Preterence 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

Free men:

Bob, Jim, Tom

Bob → proposes Lea

Lea free → accepts.

Free men

Jim, Tom

Jim → proposes Lea

Lea rejects

Jim → proposes Sue

Sue free → accepts

Free men

Tom → proposes Sue.

Sue rejects

Tom → proposes Lea.

Lea leaves Bob and accepts Tom

Free men

Bob:

Bob → Proseses Ann

Ann free → accepts

Men optimality

$$\boxed{\begin{array}{l}\text{Bob + Ann}\\ \text{Tom + Lea}\\ \text{Jim + Sue}\end{array}}$$

Stable Marriage Problem

The stable marriage problem states that given 'n' men and 'n' women, where each person has ranked all ~~other~~ members of the opposite sex in order of preference, marry the men and women together such that there are no two people of opposite sex who would both rather have each other than their current partners. If there are no such people, all the marriages are stable.
It is always possible to form stable marriges from list of preferences.
The Gale-shapely algorithm to find a stable matching - The Idea is to iterate through all free man available. every free man goes to ~~every~~ all women in his preference list according to the order.
for every women he goes, he checks the women is free, if yes they both become engaged.
If woman is not free, the woman chooses either says not to him or dump her current engagement according to her preference list.
so engagement done once can be broken if a woman gets better option.

xample

Mens	Preferences			Womens	Preferences		
Bob -	Lea	Ann	Sue	Ann -	Jim	Tom	Bob
Jim -	Lea	Sue	Ann	Lea -	Tom	Bob	Jim
Tom -	Sue	Lea	Ann	Sue -	Jim	Tom	Bob

Page No.
Date

Preference

- Initially we construct a matrix for the given preferences.

<u>Preference ~~Mator~~ Matrix</u> :

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

- List out the free Men

<u>Free Men</u> :

<u>Bob</u>, Jim, Tom

Bob → proposes Lea
Lea free → Accepts.

<u>Free Men</u> :
<u>Jim</u> Tom

Jim → Proposes Lea
Lea rejects

Jim → Proposes Sue
Sue → free → Accepts.

<u>Free Man</u> :
Tom → Proposes Sue
Sue rejects

Page No.
Date

Tom → proposes Lea
Lea leaves Bob and
Accepts Tom.

<u>Free Men</u>
Bob → Proposes Ann
Ann → Accepts.

- The pair we identified are
  Bob → Ann
  Tom → Lea
  Jim → Sea.

- Based upon men optimality we just identifie
stable pairs for Marriage.

Pimpri Chinchwad Education Trust's

Pimpri Chinchwad University

School of Engineering and Technology

MCA Department

Unit 05: Coping with Limitations of Algorithm Power

Lower Bound Arguments:

Lower Bound Theory Concept is based upon the calculation of minimum time that is required to execute an algorithm is known as a lower bound theory or Base Bound Theory.

Lower Bound Theory uses a number of methods/techniques to find out the lower bound.

Concept/Aim: The main aim is to calculate a minimum number of comparisons required to execute an algorithm.

Techniques:

The techniques which are used by lower Bound Theory are:

1. Comparisons Trees.
2. Oracle and adversary argument
3. State Space Method

1. Comparison trees:

In a comparison sort, we use only comparisons between elements to gain order information about an input sequence (a1; a2......an).

Given a_i,a_j from (a_1, a_2.....a_n)We Perform One of the Comparisons

- $a_i < a_j$ less than
- $a_i \leq a_j$ less than or equal to
- $a_i > a_j$ greater than
- $a_i \geq a_j$ greater than or equal to
- $a_i = a_j$ equal to

To determine their relative order, if we assume all elements are distinct, then we just need to consider $a_i \leq a_j$ '=' is excluded &, ≥,≤,>,< are equivalent.

Consider sorting three numbers a1, a2, and a3. There are 3! = 6 possible combinations:

1. (a1, a2, a3), (a1, a3, a2),
2. (a2, a1, a3), (a2, a3, a1)
3. (a3, a1, a2), (a3, a2, a1)

The Comparison based algorithm defines a decision tree.

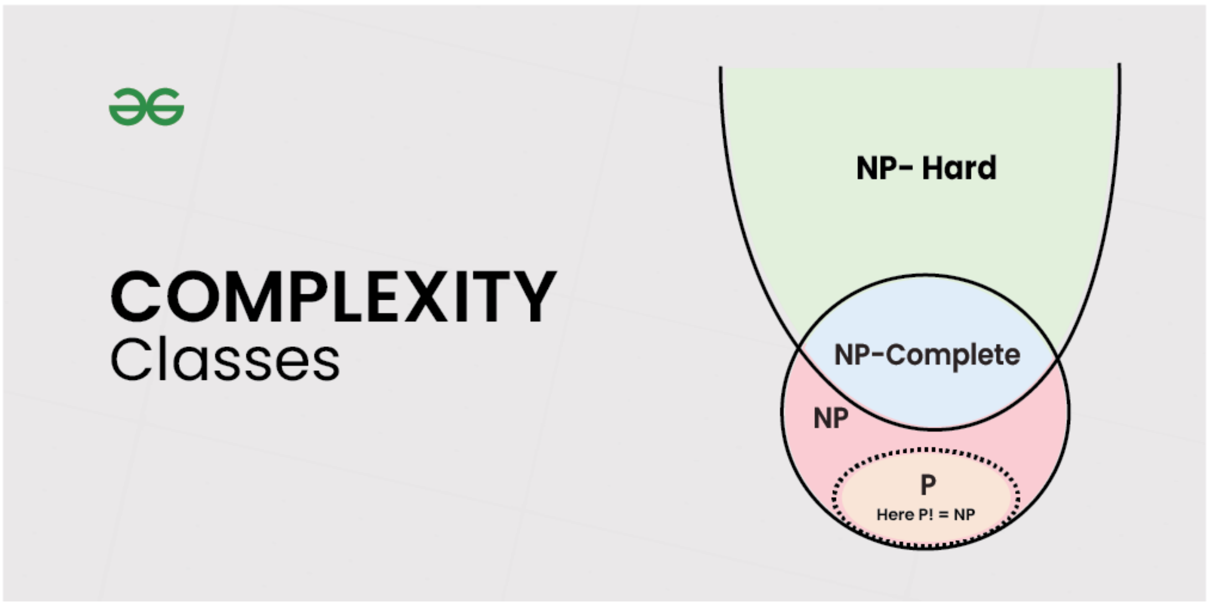
P, NP, NP hard and NP complete

In computer science, there exist some problems whose solutions are not yet found, the problems are divided into classes known as **Complexity Classes**. In complexity theory, a Complexity Class is a set of problems with related complexity. These classes help scientists to group problems based on how much time and space they require to solve problems and verify the solutions. It is the branch of the theory of computation that deals with the resources required to solve a problem.

The common resources are time and space, meaning how much time the algorithm takes to solve a problem and the corresponding memory usage.

- The time complexity of an algorithm is used to describe the number of steps required to solve a problem, but it can also be used to describe how long it takes to verify the answer.
- The space complexity of an algorithm describes how much memory is required for the algorithm to operate.

Complexity classes are useful in organising similar types of problems.



P Class

The P in the P class stands for Polynomial Time. It is the collection of decision problems (problems with a "yes" or "no" answer) that can be solved by a deterministic machine in polynomial time.

Features:

- The solution to P problems is easy to find.

- P is often a class of computational problems that are solvable and tractable. Tractable means that the problems can be solved in theory as well as in practice. But the problems that can be solved in theory but not in practice are known as intractable.

This class contains many problems:

1. Calculating the greatest common divisor.
2. Finding a maximum matching.
3. Merge Sort

NP Class

The NP in NP class stands for Non-deterministic Polynomial Time. It is the collection of decision problems that can be solved by a non-deterministic machine in polynomial time.

Features:

- The solutions of the NP class are hard to find since they are being solved by a non-deterministic machine but the solutions are easy to verify.
- Problems of NP can be verified by a Turing machine in polynomial time.

Example:

Let us consider an example to better understand the NP class. Suppose there is a company having a total of 1000 employees having unique employee IDs. Assume that there are 200 rooms available for them. A selection of 200 employees must be paired together, but the CEO of the company has the data of some employees who can't work in the same room due to personal reasons.

This is an example of an NP problem. Since it is easy to check if the given choice of 200 employees proposed by a coworker is satisfactory or not i.e. no pair taken from the coworker list appears on the list given by the CEO. But generating such a list from scratch seems to be so hard as to be completely impractical.

It indicates that if someone can provide us with the solution to the problem, we can find the correct and incorrect pair in polynomial time. Thus for the NP class problem, the answer is possible, which can be calculated in polynomial time.

This class contains many problems that one would like to be able to solve effectively:

1. Boolean Satisfiability Problem (SAT).
2. Hamiltonian Path Problem.
3. Graph coloring.

NP-hard class

An NP-hard problem is at least as hard as the hardest problem in NP and it is a class of problems such that every problem in NP reduces to NP-hard.

Features:

- All NP-hard problems are not in NP.
- It takes a long time to check them. This means if a solution for an NP-hard problem is given then it takes a long time to check whether it is right or not.
- A problem A is in NP-hard if, for every problem L in NP, there exists a polynomial-time reduction from L to A.

Some of the examples of problems in Np-hard are:

1. Halting problem.
2. Qualified Boolean formulas.
3. No Hamiltonian cycle.

NP-complete class

A problem is NP-complete if it is both NP and NP-hard. NP-complete problems are the hard problems in NP.

Features:

- NP-complete problems are special as any problem in NP class can be transformed or reduced into NP-complete problems in polynomial time.
- If one could solve an NP-complete problem in polynomial time, then one could also solve any NP problem in polynomial time.

Some example problems include:

1. Hamiltonian Cycle.
2. Satisfiability.
3. Vertex cover.

Complexity Class	**Characteristic feature**
P	Easily solvable in polynomial time.

NP	Yes, answers can be checked in polynomial time.
Co-NP	No, answers can be checked in polynomial time.
NP-hard	All NP-hard problems are not in NP and it takes a long time to check them.
NP-complete	A problem that is NP and NP-hard is NP-complete.

Backtracking:

Backtracking algorithms are like problem-solving strategies that help explore different options to find the best solution. They work by trying out different paths and if one doesn't work, they backtrack and try another until they find the right one. It's like solving a puzzle by testing different pieces until they fit together perfectly.

What is Backtracking Algorithm?

Backtracking is a problem-solving algorithmic technique that involves finding a solution incrementally by trying **different options** and **undoing** them if they lead to a **dead end**.

It is commonly used in situations where you need to explore multiple possibilities to solve a problem, like searching for a path in a maze or solving puzzles like **Sudoku**. When a dead end is reached, the algorithm backtracks to the previous decision point and explores a different path until a solution is found or all possibilities have been exhausted.

How Does a Backtracking Algorithm Work?

A **backtracking algorithm** works by recursively exploring all possible solutions to a problem. It starts by choosing an initial solution, and then it explores all possible extensions of that solution. If an extension leads to a solution, the algorithm returns that solution. If an extension does not lead to a solution, the algorithm backtracks to the previous solution and tries a different extension.

The following is a general outline of how a backtracking algorithm works:

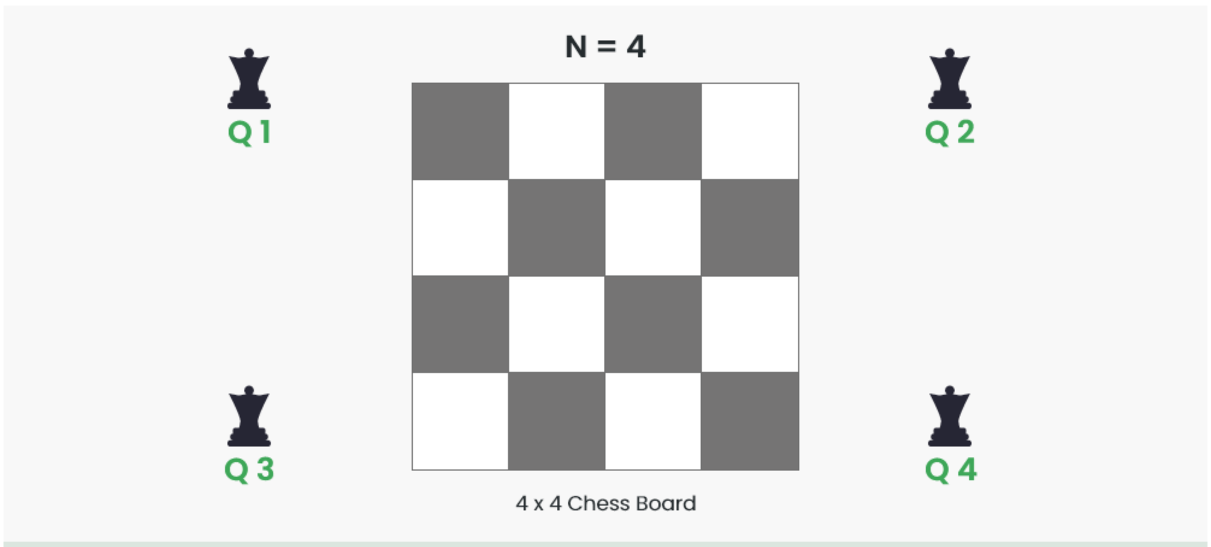
1. Choose an initial solution.
2. Explore all possible extensions of the current solution.
3. If an extension leads to a solution, return that solution.

4. If an extension does not lead to a solution, backtrack to the previous solution and try a different extension.
5. Repeat steps 2-4 until all possible solutions have been explored.

N-Queen Problem:

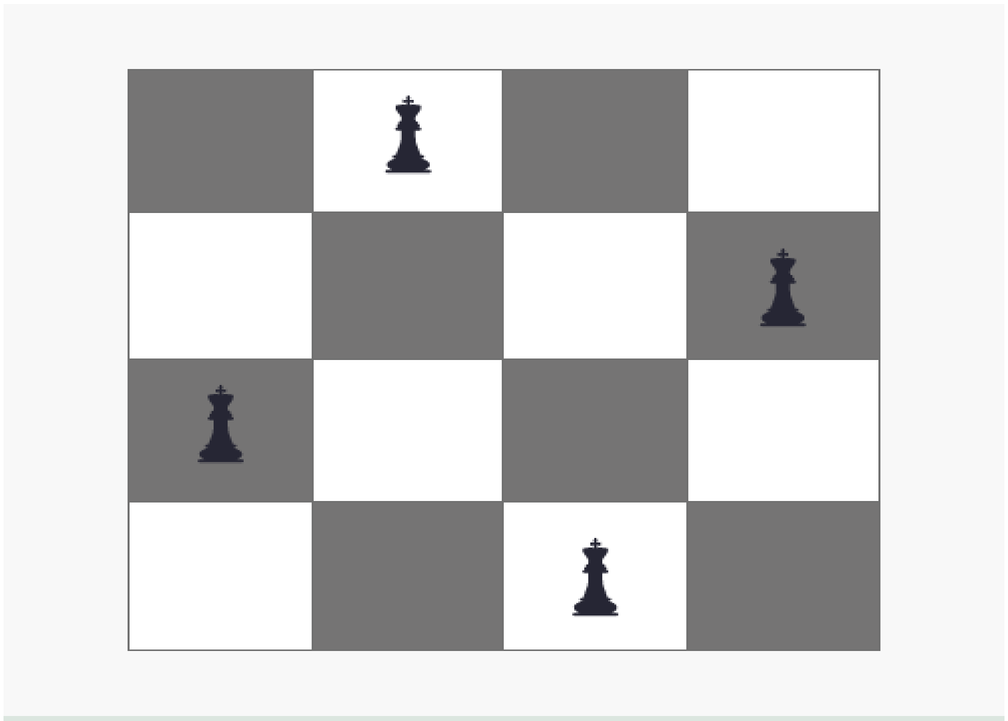
What is N-Queen problem?

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other.



N Queen Problem

For example, the following is a solution for the 4 Queen problem.



Solution Of 4 Queen Problem

The expected output is in the form of a matrix that has 'Q's for the blocks where queens are placed and the empty spaces are represented by '.' . For example, the following is the output matrix for the above 4-Queen solution.

```
. Q . .
. . . Q
Q . . .
. . Q .
```

N Queen Problem using Backtracking:

The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes, then we backtrack and return false.

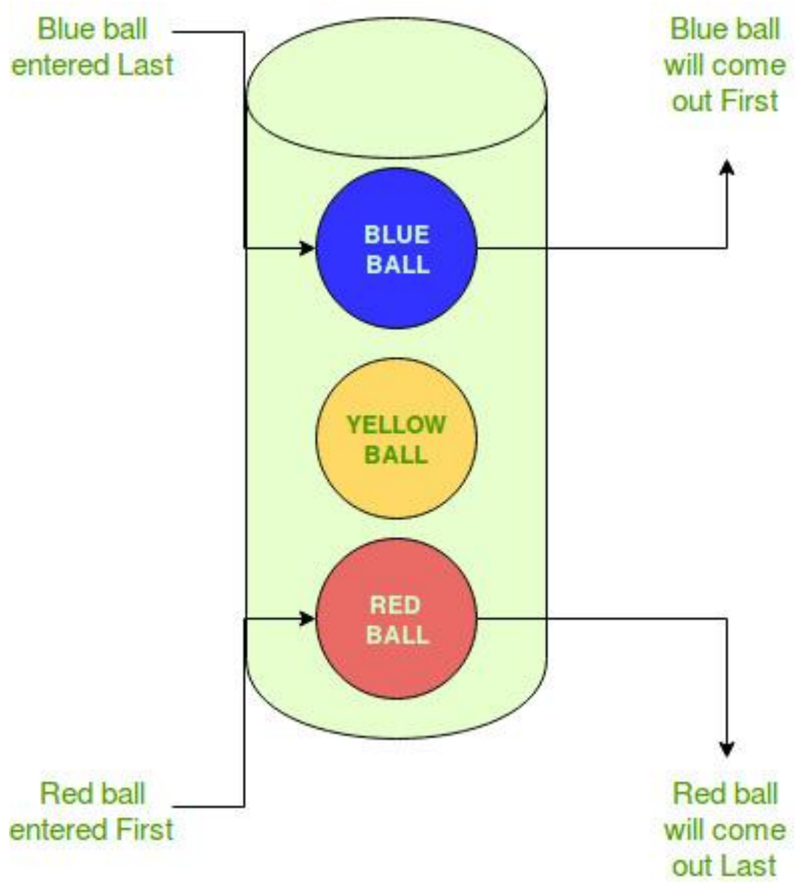
Branch and Bound

The Branch and Bound Algorithm is a method used in combinatorial optimization problems to systematically search for the best solution. It works by dividing the problem into smaller subproblems, or branches, and then eliminating certain branches based on bounds on the optimal solution. This process continues until the best solution is found or all branches have been explored. Branch and Bound is commonly used in problems like the traveling salesman and job scheduling.

LIFO Search

LIFO is an abbreviation for last in, first out. It is a method for handling data structures where the first element is processed last and the last element is processed first.

Real-life example:



In this example, following things are to be considered:

There is a bucket that holds balls.

- Different types of balls are entered into the bucket.
- The ball to enter the bucket last will be taken out first.
- The ball entering the bucket next to last will be taken out after the ball above it (the newer one).
- In this way, the ball entering the bucket first will leave the bucket last.
- Therefore, the Last ball (Blue) to enter the bucket gets removed first and the First ball (Red) to enter the bucket gets removed last.
- This is known as Last-In-First-Out approach or LIFO.

Where is LIFO used:

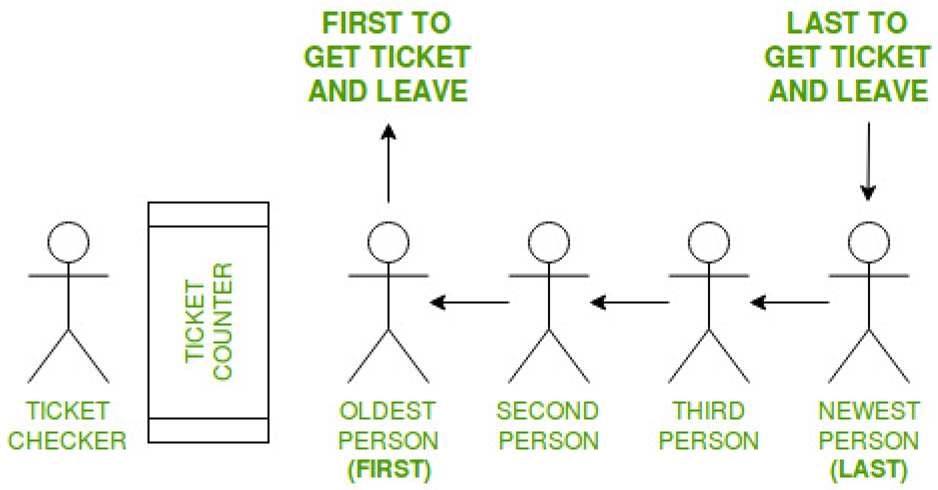
1. **Data Structures:** Certain data structures like Stacks and other variants of Stacks use LIFO approach for processing data.

2. **Extracting latest information:** Sometimes computers use LIFO when data is extracted from an array or data buffer. When it is required to get the most recent information entered, the LIFO approach is used.

FIFO Search

FIFO is an abbreviation for first in, first out. It is a method for handling data structures where the first element is processed first and the newest element is processed last.

Real Life Example



In this example, following things are to be considered:

- There is a ticket counter where people come, take tickets and go.
- People enter a line (queue) to get to the Ticket Counter in an organized manner.
- The person to enter the queue first, will get the ticket first and leave the queue.
- The person entering the queue next will get the ticket after the person in front of him
- In this way, the person entering the queue last will the tickets last
- Therefore, the First person to enter the queue gets the ticket first and the Last person to enter the queue gets the ticket last.

This is known as First-In-First-Out approach or FIFO.

Where is FIFO used:

1. **Data Structures:**
    - Certain data structures like Queue and other variants of Queue uses FIFO approach for processing data.
2. **Disk scheduling:**

    - Disk controllers can use the FIFO as a disk scheduling algorithm to determine the order in which to service disk I/O requests.

3. **Communications and networking"**
    - Communication network bridges, switches and routers used in computer networks use FIFOs to hold data packets en route to their next destination.

Hamiltonian Circuit Path

What is Hamiltonian Cycle?

Hamiltonian Cycle or Circuit in a graph G is a cycle that visits every vertex of G exactly once and returns to the starting vertex.

- If graph contains a Hamiltonian cycle, it is called Hamiltonian graph otherwise it is non-Hamiltonian.
- Finding a Hamiltonian Cycle in a graph is a well-known NP-complete problem, which means that there's no known efficient algorithm to solve it for all types of graphs. However, it can be solved for small or specific types of graphs.
- The Hamiltonian Cycle problem has practical applications in various fields, such as logistics, network design, and computer science.

What is Hamiltonian Path?

Hamiltonian Path in a graph G is a path that visits every vertex of G exactly once and Hamiltonian Path doesn't have to return to the starting vertex. It's an open path.

- Similar to the Hamiltonian Cycle problem, finding a Hamiltonian Path in a general graph is also NP-complete and can be challenging. However, it is often a more easier problem than finding a Hamiltonian Cycle.
- Hamiltonian Paths have applications in various fields, such as finding optimal routes in transportation networks, circuit design, and graph theory research.

Subset Sum Problem

Given a set of non-negative integers and a value sum, the task is to check if there is a subset of the given set whose sum is equal to the given sum.

Examples:

Input: set[] = {3, 34, 4, 12, 5, 2}, sum = 9

Output: True

Explanation: There is a subset (4, 5) with sum 9.

Input: set[] = {3, 34, 4, 12, 5, 2}, sum = 30

Output: False

Explanation: There is no subset that add up to 30.