VISHNU INSTITUTE OF TECHNOLOGY (AUTONOMOUS) :: BHIMAVARAM

Department of CSE(Artificial Intelligene and Data Science)

Advanced Data Structurs And Algorithm Analysis

UNIT 1 Topics

:

1. What is an Algorithm
2. Algorithm Specifications
    a. Pseudocode Conventions
    b. Recursive Algorithms
3. Performance Analysis
    a. Space Complexity
    b. Time Complexity
    c. Amortized Complexity
    d. Asymptotic Notation
    e. Practical Complexities
    f. Performance Measurement

1. What is an Algorithm

An algorithm is a step-by-step procedure for accomplishing a particular task. It is a sequence of welldefined instructions or rules that specify how to perform computations or process data. Algorithms are fundamental to computer science and are used in various fields to automate tasks, solve complex problems, and process information efficiently.

Key Characteristics of an Algorithm:

1. **Input**: An algorithm has zero or more inputs that are externally supplied.
2. **Finiteness**: An algorithm must always terminate after a finite number of steps.
3. **Definiteness**: Each step of the algorithm must be precisely defined; the instructions should be clear and unambiguous.
4. **Effective**: The steps of an algorithm must be basic enough to be carried out, in principle, by a person using only paper and pencil.
5. **Output**: An algorithm produces at least one output, which is the result of the computations.

Components of an Algorithm:

- **Input**: The data that the algorithm processes.
- **Output**: The result produced by the algorithm.
- **Steps**: The sequence of instructions that transform the input into the output.

Examples of Algorithms:

1. **Sorting Algorithms**: ○ **Bubble Sort**: A simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.
    - **Merge Sort**: A divide-and-conquer algorithm that divides the list into smaller sublists, sorts them, and then merges them back together.
2. **Searching Algorithms**: ○ **Linear Search**: A straightforward search algorithm that checks each element of a list sequentially until the desired element is found or the list ends.
    - **Binary Search**: A more efficient search algorithm that divides a sorted list in half to determine whether the desired element is in the left or right half, then repeats the process.
3. **Graph Algorithms**: ○ **Dijkstra's Algorithm**: Finds the shortest path from a single source node to all other nodes in a weighted graph.
    - **Depth-First Search (DFS)**: Explores a graph by starting at the root node and exploring as far as possible along each branch before backtracking.

Example: Algorithm for Adding Two Numbers

Here is a simple example of an algorithm to add two numbers:

1. **Input**: Two numbers, a and b.
2. **Step 1**: Start.
3. **Step 2**: Read the values of a and b.
4. **Step 3**: Calculate the sum of a and b and store it in a variable sum.

5. **Step 4**: Output the value of sum.
6. **Step 5**: End.

In pseudocode, this algorithm can be represented as:

Algorithm : AddNumbers(a, b)

Start Read a, b sum = a + b Print sum End

Importance of Algorithms:

- **Efficiency**: Algorithms help in optimizing tasks by reducing time complexity and resource usage.
- **Automation**: Algorithms allow for the automation of repetitive tasks, increasing productivity.
- **Problem-Solving**: They provide a structured approach to solving complex problems in various fields such as computer science, mathematics, and engineering.
- **Foundation of Programming**: Understanding algorithms is essential for writing effective and efficient code.

In summary, an algorithm is a precise set of instructions designed to perform a specific task or solve a particular problem. The study and development of algorithms are crucial for the advancement of technology and various scientific disciplines.

2. Algorithm Specificatins

2.1 Pseudocode Conventions

Pseudocode is a high-level description of an algorithm that uses the structural conventions of programming languages but is intended for human reading rather than machine reading. Here are some common conventions used in writing pseudocode:

Pseudocode Conventions

1. **Structure and Indentation**:
   - **Indentation**: Use consistent indentation to show the hierarchy and nesting of statements. This helps in understanding the flow of control.
   - **Blocks**: Represent blocks of code (like loops or conditionals) using indentation or explicit block markers (e.g., BEGIN...END).
2. **Variables and Constants**: ○ **Variables**: Use descriptive names for variables to indicate their purpose.
   - **Constants**: Use uppercase letters for constants to differentiate them from variables.
3. **Data Types**: ○ Pseudocode often omits specific data types for simplicity, but when needed, use descriptive names (e.g., integer, string).
4. **Assignment**:
   - Use the = symbol for assignment. ○ Example: sum = a + b
5. **Input/Output**:
   - Use READ or INPUT for input operations. ○ Use WRITE, PRINT, or OUTPUT for output operations. ○

Example: READ a, b or PRINT sum

6. **Control Structures**:
   - **If-Then-Else**: IF condition THEN
     statements
     ELSE
       statements
     ENDIF
   - **For Loop**:
     FOR i = start TO end DO
       Statements
     ENDFOR ○
     While Loop: WHILE condition DO
       statements
     ENDWHILE
   - **Repeat-Until Loop**: REPEAT
       statements UNTIL
     condition
7. **Procedures and Functions**:
   - **Procedure**:
     PROCEDURE ProcedureName(parameters)
     statements
     ENDPROCEDURE ○
     Function:
     FUNCTION FunctionName(parameters) RETURNS returnType
     statements
       RETURN value
     ENDFUNCTION
8. **Comments**: ○ Use comments to explain the purpose of complex sections of the pseudocode.
   - Example: // This is a comment **Example Pseudocode**

Example of pseudocode for finding the maximum number in a list:

```
PROCEDURE FindMax(numbers)     max = numbers[0]
  FOR i = 1 TO LENGTH(numbers) - 1 DO
    IF numbers[i] > max THEN
max = numbers[i]
    ENDIF
  ENDFOR
  RETURN max
END  PROCEDURE
```

Explanation of the Example:

- **Procedure Definition**: The FindMax procedure is defined to take a list of numbers.
- **Initialization**: The variable max is initialized with the first element of the list.
- **For Loop**: The loop iterates through the list starting from the second element.
- **If Statement**: Inside the loop, each element is compared with max, and if it is greater, max is updated.
- **Return Statement**: After the loop, max is returned as the largest number in the list.

2.2 Recursive Algorithms

Recursive algorithms are those that solve problems by calling themselves with modified parameters. They are particularly useful for problems that can be divided into similar subproblems. Each recursive call should simplify the problem, eventually reaching a base case that terminates the recursion. **Characteristics of Recursive Algorithms**

1. **Base Case**: The condition under which the recursion stops.
2. **Recursive Case**: The part of the function that calls itself with modified parameters to simplify the problem.

Advantages of Recursion

- **Simplicity**: Recursive solutions can be more straightforward and easier to understand.
- **Elegance**: They often provide a clean and elegant way to solve problems that involve repetitive, nested, or hierarchical structures (e.g., tree traversal).

Disadvantages of Recursion

- **Performance**: Recursive algorithms can be less efficient in terms of time and space due to function call overhead and stack usage.
- **Stack Overflow**: Too many recursive calls can lead to a stack overflow if the recursion depth exceeds the system's limit.

Examples of Recursive Algorithms

1. Factorial of a given Number : The factorial of a non-negative integer n is the product of all positive integers less than or equal to n. It can be defined recursively as:

```
FUNCTION factorial(n)
IF n == 0 THEN
    RETURN 1
ELSE
    RETURN n * factorial(n - 1)
ENDIF
END FUNCTION
```

3. Performance Analysis

3.1 Space Complexity

Space complexity refers to the amount of memory space required by an algorithm to execute as a function of the size of the input. It includes both the memory needed to hold the input data and the memory required for the algorithm's operations, such as temporary variables, data structures, and function call stacks in the case of recursion. Space complexity helps in understanding how an algorithm's memory consumption grows with the input size and is crucial for optimizing and ensuring that an algorithm runs efficiently within the available memory. **Components of Space Complexity**

1. **Fixed Part**: The space required for constants, simple variables, fixed-size data structures, etc. This does not depend on the input size and is usually a constant O(1)O(1)O(1).
2. **Variable Part**: The space required for dynamic data structures, temporary variables, and recursion stacks. This depends on the input size and often determines the overall space complexity.

Example: Space Complexity of Simple Algorithms

1. Constant Space Complexity

Consider a function that calculates the sum of the first n natural numbers: def

```
sum_natural_numbers(n):
   sum = 0     for i in
range(1, n + 1):
sum += i     return sum
```

- **Space Complexity**: O(1)
- The space complexity is O(1) because the amount of memory used by the variables sum and i is constant and does not depend on the input size n.

2. Linear Space Complexity

Consider a function that creates a list of the first n natural numbers:

```
def create_list(n):    numbers = []    for i in range(1, n +
1):        numbers.append(i)
  return numbers
```

- **Space Complexity**: O(n)

The space complexity is O(n) because the size of the list numbers grows linearly with the input size n.

3. **Recursive Algorithm Space Complexity**

Consider the space complexity of the recursive function to calculate the factorial of a number:

```
def factorial(n):
if n == 0:
return 1    else:
    return n * factorial(n - 1)
```

- **Space Complexity**: O(n)

The space complexity is O(n) because each recursive call adds a new frame to the call stack. The maximum depth of the recursion is n, so the space required grows linearly with the input size n.

Analyzing Space Complexity with an Example :

Example: Merging Two Sorted Lists

Suppose you have two sorted lists and you want to merge them into a single sorted list:

```
def merge_sorted_lists(list1, list2):    merged_list = []    i, j = 0, 0    while i < len(list1)
and j < len(list2):        if list1[i] < list2[j]:
        merged_list.append(list1[i])
i          +=          1
else:
        merged_list.append(list2[j])
j          +=          1
while i < len(list1):
      merged_list.append(list1[i])
i          += 1    while j <
len(list2):
merged_list.append(list2[j
])
j          += 1    return
merged_list
```

- **Input Space**: The input space is determined by the size of the input lists, list1 and list2.
- **Auxiliary Space**: The auxiliary space is the additional space used by the algorithm, excluding the input size. Here, the auxiliary space is the space used by merged_list, which is O(n+m)O(n + m)O(n+m) where n and m are the lengths of list1 and list2, respectively.
- **Space Complexity**: O(n+m)
- In this example, the space complexity is O(n+m) because the size of the merged_list depends on the sizes of the input lists list1 and list2.

Importance of Space Complexity

- **Memory Efficiency**: Understanding space complexity helps in designing algorithms that use memory efficiently.
- **Scalability**: Algorithms with lower space complexity are more scalable and can handle larger input sizes without running out of memory.
- **Performance**: Optimizing space complexity can improve the overall performance of an algorithm, especially in memory-constrained environments.

By analyzing and optimizing space complexity, you can ensure that your algorithms run efficiently and effectively within the available memory resources.

3.2 Time Complexity

Time complexity is a measure of the amount of time an algorithm takes to complete as a function of the length of the input. It gives an idea of the efficiency of an algorithm by describing how the runtime grows as the input size increases.

Big O Notation

Big O notation is used to classify algorithms according to how their run time or space requirements grow as the input size grows. It provides an upper bound on the time complexity.

Common Time Complexities

1. **Constant Time – O(1)**: ○ The run time of the algorithm is constant, irrespective of the input size.
   - ○ Example: Accessing an element in an array by index.
2. **Logarithmic Time –O(log n)**: ○ The run time grows logarithmically as the input size increases.
   - ○ Example: Binary search in a sorted array.
3. **Linear Time – O(n)**: ○ The run time grows linearly with the input size.
   - ○ Example: Iterating through an array.
4. **Linearithmic Time – O(n logn)**:
   - ○ The run time grows in proportion to nlog⁡nn \log nnlogn.
   - ○ Example: Merge sort, quicksort in the average case.
5. **Quadratic Time – O(n^2)**: ○ The run time grows proportionally to the square of the input size.
   - ○ Example: Bubble sort, insertion sort.
6. **Cubic Time – O(n^3)**: ○ The run time grows proportionally to the cube of the input size.
   - ○ Example: Floyd-Warshall algorithm for shortest paths. **Analyzing Time Complexity**

To analyze the time complexity of an algorithm, follow these steps:

1. **Identify Basic Operations**: Determine the most time-consuming operations in the algorithm.
2. **Count the Operations**: Express the number of times each operation is executed as a function of the input size nnn.
3. **Classify the Complexity**: Use Big O notation to classify the overall time complexity.

Example of Time Complexity Analysis: Insertion Sort

Let's take a detailed look at the time complexity of the Insertion Sort algorithm. Insertion Sort is a simple sorting algorithm that builds the final sorted array one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort.

Insertion Sort Algorithm **Pseudocode**:

```
FUNCTION insertionSort(arr):
FOR i FROM 1 TO length(arr) - 1:
key = arr[i]        j = i - 1
      WHILE j >= 0 AND arr[j] > key:
         arr[j + 1] = arr[j]
         j = j - 1
      END WHILE
arr[j + 1] = key
   END FOR
END FUNCTION
```

Step-by-Step Analysis 1. **Initialization**: ○ The for loop runs from i = 1 to i = n - 1, where n is the number of elements in the array.
   - ○ This loop executes n−1n-1n−1 times.

2. **Inner Loop (While Loop)**: ○ For each iteration of the for loop, the while loop checks and shifts elements to the right until the correct position for the key is found.

- In the worst case (when the array is sorted in reverse order), the while loop executes i times for each i in 1 to n-1.

Worst-Case Time Complexity

In the worst-case scenario, each element has to be compared with all the previous elements and shifted, so the number of operations for each i is 1+2+3+…+n: The sum of the first n−1 integers is:

$n(n-1)/2$

Therefore, the time complexity for the worst-case scenario is $O(n^2)$

Best-Case Time Complexity

In the best-case scenario, the array is already sorted, so the while loop condition arr[j] > key is never true, and no shifting is required. The while loop runs 0 times for each i. The total number of operations is proportional to the number of elements, resulting in:

O(n)

Average-Case Time Complexity

In the average case, the elements are in random order. On average, each element needs to be compared with half of the previous elements, so the while loop runs about i/2 times for each i:

Therefore, the average-case time complexity is still $O(n^2)$

Summary

- **Worst-case time complexity**: $O(n^2)$
- **Best-case time complexity**: O(n)
- **Average-case time complexity**: $O(n^2)$

3.3 Amortized Complexity

Amortized complexity is a concept in algorithm analysis used to average the time or space complexity of a sequence of operations over a worst-case sequence of operations. It gives a better overall picture of the performance of an algorithm over time, rather than just considering the worst-case scenario for a single operation. This is particularly useful for data structures that have occasional expensive operations but are efficient on average. **Types of Amortized Analysis**

1. **Aggregate Method**: Calculate the total cost of a sequence of operations and divide by the number of operations to get the average cost per operation.
2. **Accounting Method**: Assign different costs to operations, such that some operations "overpay" while others "underpay." The total overpaid cost compensates for the occasional expensive operations.
3. **Potential Method**: Define a potential function that represents the "stored energy" or "potential" in the data structure. The amortized cost of an operation is its actual cost plus the change in potential.

3.4 Asymptotic Notation

Asymptotic notation is a mathematical concept used in computer science to describe the behavior of functions as the input size grows towards infinity. It provides a way to classify algorithms based on their running time or space requirements in terms of the size of their inputs. Asymptotic notations are used to give an upper bound, lower bound, or tight bound on the time complexity of algorithms, helping us understand their efficiency and scalability.

Types of Asymptotic Notation

1. **Big O Notation (O)**: Represents the upper bound of the time complexity. It gives the worst-case scenario of an algorithm's growth rate.
2. **Omega Notation (Ω\Omega)**: Represents the lower bound of the time complexity. It gives the best-case scenario of an algorithm's growth rate.
3. **Theta Notation (Θ\Theta)**: Represents the tight bound of the time complexity. It gives both the upper and lower bounds, meaning it captures the average-case scenario when the growth rate is both O and Ω. **Big O Notation (OOO)**

Big O notation describes the upper bound of an algorithm's running time. It tells us the maximum time an algorithm can take to complete. ***Definition:*** $f(n)=O(g(n))$

if and only if there exist positive constants c and n0 such that for all $n \geq n0$,

$0 \leq f(n) \leq c.g(n)$ ***Example:***
Consider a function $f(n)=3n^2+5n+2$
As n grows large, the n^2 term will dominate, so we can say: $f(n)=O(n^2)$ This means that the function grows at most as fast as n^2

Omega Notation (Ω\OmegaΩ)
Omega notation describes the lower bound of an algorithm's running time. It tells us the minimum time an algorithm will take to complete.
Definition:
$f(n)=\Omega(g(n))$
if and only if there exist positive constants c and n0 such that for all $n \geq n0$,
$0 \leq c.g(n) \leq f(n)$ ***Example:***
Consider a function $f(n)=3n^2+5n+2$
As n grows large, the n^2 term will dominate, so we can say: $f(n)=\Omega(n^2)$
This means that the function grows at least as fast as n^2
Theta Notation (Θ\ThetaΘ)
Theta notation describes the tight bound of an algorithm's running time. It tells us that the algorithm's running time grows asymptotically at the same rate as the given function. ***Definition:***
$f(n)=\Theta(g(n))$
if and only if there exist positive constants c1, c2, and n0 such that for all $n \geq n0$, $0 \leq c1.g(n) \leq f(n) \leq c2.g(n)$ ***Example:***
Consider a function $f(n)=3n^2+5n+2$
As n grows large, the n^2 term will dominate, so we can say: $f(n)=\Theta(n2)$ This means that the function grows exactly as fast as n2n^2n2.
Summary of Asymptotic Notations

1. **Big O (O)**: Upper bound - Worst-case
2. **Omega (Ω\OmegaΩ)**: Lower bound - Best-case
3. **Theta (Θ\ThetaΘ)**: Tight bound - Average-case **Example of Using Asymptotic Notation**

Consider the following pseudocode for a function that prints all pairs in an array:

```
FUNCTION printAllPairs(arr):
  FOR i FROM 0 TO length(arr) - 1:
    FOR j FROM 0 TO length(arr) - 1:
      PRINT arr[i], arr[j]
END FUNCTION
```

Time Complexity Analysis

1. **Outer loop** runs nnn times, where nnn is the length of the array.
2. **Inner loop** runs nnn times for each iteration of the outer loop. Total number of iterations = $n \times n = n^2$ ***Asymptotic Notation:***

- **Big O Notation**: $O(n^2)$ (Worst-case scenario: every pair is printed)
- **Omega Notation**: $\Omega(n^2$ (Best-case scenario: every pair is printed)
- **Theta Notation**: $\Theta(n^2)$ (Average-case scenario: every pair is printed)

Importance of Asymptotic Notation

- **Scalability**: Helps in understanding how an algorithm will perform as the input size grows.
- **Comparison**: Enables comparison of different algorithms based on their time and space complexities.
- **Optimization**: Guides in identifying performance bottlenecks and optimizing algorithms.

Understanding asymptotic notation is crucial for designing efficient algorithms and ensuring that they perform well even with large inputs.

3.5 Practical Complexitites in Algorithm Analysis

Algorithm analysis is critical for understanding the efficiency and feasibility of algorithms in real-world applications. However, it comes with a host of practical challenges that can complicate the process. Here are some of the key practical challenges along with strategies to address them.

A. Real-world data can be messy, large, and varied, often not fitting the neat theoretical assumptions made during algorithm design and analysis.
B. Algorithms that perform well on small datasets may struggle with large datasets due to increased time and space complexity.
C. Performance can be significantly affected by the hardware on which the algorithm runs, including CPU speed, memory capacity, and I/O performance.
D. Some applications require algorithms to produce results within strict time constraints, such as in embedded systems or real-time processing.
E. For battery-powered or energy-sensitive applications, the energy consumption of algorithms can be a critical factor.
F. Algorithms may need to adapt to changing conditions, such as varying input sizes or types, or evolving requirements.
G. Balancing trade-offs between competing factors such as time complexity, space complexity, accuracy, and simplicity can be difficult.
H. Complex algorithms can be difficult to implement correctly, debug, and maintain, especially in large and collaborative projects.

3.6 Performance Measurements

Performance measurement is a crucial aspect of software engineering and computer science. It involves assessing the efficiency of algorithms and systems to ensure they meet performance requirements. The primary goals are to understand how algorithms scale with input size, to identify bottlenecks, and to optimize the system for better performance. Here are some key aspects of performance measurement: **Key Metrics**

1. **Time Complexity**: Measures how the runtime of an algorithm grows with the size of the input.
2. **Space Complexity**: Measures the amount of memory an algorithm uses as the input size grows.
3. **Throughput**: The amount of work done in a given period of time.
4. **Latency**: The time taken to complete a single operation from start to finish.
5. **Efficiency**: The ratio of useful work performed by the system to the resources used. **Tools for Performance Measurement**

1. **Profilers**: Tools that analyze the program and report the time spent in each part of the code. Examples include:
   - **gprof** (GNU profiler) for C/C++ programs. ○ **VisualVM** for Java applications.
   - **cProfile** for Python programs.
2. **Benchmarking Tools**: Tools that run a set of standard tests and report the performance. Examples include:
   - **SPEC CPU** benchmarks for CPU performance. ○ **JMH (Java Microbenchmark Harness)** for Java.
3. **Logging and Monitoring Tools**: Tools that track performance metrics over time. Examples include:
   - **Prometheus** for monitoring and alerting.
   - **Grafana** for visualization of performance metrics.

VISHNU INSTITUTE OF TECHNOLOGY :: BHIMAVARAM
Department of CSE(Artificial Intelligence and Data Science)
Advanced Data Structures And Algorithm Analysis
Design and Analysis of Algorithms UNIT
2 : Divide and Conquer

Topics:

1. **Divide and Conquer : General Method**
2. **Defective Chessboard**
3. **Binary Search**
4. **Finding the minimum and maximum**
5. **Merge Sort**
6. **Quick Sort**
7. **Performance Analysis in Divide and Conquer**

The divide and conquer algorithm is a fundamental paradigm for designing algorithms, especially for solving complex problems efficiently. It works by recursively breaking down a problem into two or more sub-problems of the same or related type, solving these sub-problems independently, and then combining their solutions to solve the original problem.

General Method

The divide and conquer approach generally involves three main steps:

1. **Divide**:
   - The original problem is divided into smaller sub-problems. These sub-problems are usually of the same type as the original problem. The division continues until the sub-problems are simple enough to be solved directly.
2. **Conquer**:
   - Each sub-problem is solved recursively. If the sub-problems are small enough, they are solved directly without further division.
3. **Combine**:
   - The solutions to the sub-problems are combined to form the solution to the original problem.

Example of Divide and Conquer: Merge Sort

To illustrate the divide and conquer method, let's consider the merge sort algorithm:

1. **Divide**:
   - Split the array into two halves. This is done recursively until each sub-array contains a single element.
2. **Conquer**:
   - Sort each sub-array. Since each sub-array contains only one element at this stage, they are trivially sorted.

3. **Combine**:
    - Merge the sorted sub-arrays to produce new sorted arrays until the entire array is sorted.

Here is a simplified pseudocode for merge sort:

```
function mergeSort(array)     if length
of array ≤ 1          return array
middle = length of array / 2     left =
mergeSort(array[0:middle])     right =
mergeSort(array[middle:end])     return
merge(left, right)

function merge(left, right)
result = []
    while left is not empty and right is not empty
if left[0] ≤ right[0]              append left[0]
to result              remove left[0] from left
else
            append     right[0]     to     result
remove right[0] from right      append remaining
elements of left to result      append remaining
elements of right to result       return result
```

Benefits and Drawbacks

Benefits:

- **Efficiency**: Divide and conquer algorithms can greatly reduce the time complexity of certain problems. For example, merge sort has a time complexity of O(nlogn)O(n \log n)O(nlogn).
- **Simplicity**: By breaking down a problem into simpler sub-problems, the overall algorithm can become easier to design and understand.
- **Parallelism**: The independent sub-problems can often be solved in parallel, taking advantage of multi-core processors and distributed computing systems.

Drawbacks:

- **Overhead**: The recursive division and combination steps can introduce overhead, both in terms of time and space (e.g., additional memory for recursion stack and intermediate results).
- **Not Always Suitable**: Divide and conquer is not the best approach for all problems. Some problems do not naturally break down into smaller sub-problems, or the overhead of combining solutions can outweigh the benefits.

VISHNU INSTITUTE OF TECHNOLOGY :: BHIMAVARAM

Department of CSE(Artificial Intelligence and Data Science)

Advanced Data Structures And Algorithm Analysis

Design and Analysis of Algorithms UNIT

2 : Divide and Conquer

The divide and conquer algorithm is a powerful technique for solving complex problems by breaking them down into more manageable sub-problems. When appropriately applied, it can lead to efficient and elegant solutions.

Example 1 : Defective Chessboard Tiling

Defective chessboard tiling is a classic problem that can be efficiently solved using the divide and conquer algorithm. The problem involves tiling an n×n chessboard (where n is a power of 2) with L-shaped tiles, given that one square is defective (i.e., it cannot be used for tiling).

Problem Statement

Given an n×n chessboard with one defective square (a square that cannot be covered by a tile), tile the entire board using L-shaped tiles, which cover exactly three squares.

Solution using Divide and Conquer

The divide and conquer approach to this problem involves recursively breaking down the chessboard into smaller sub-boards and tiling them. Here's a step-by-step explanation:

1. **Divide**: ○ Divide the n×n board into four n/2×n/2 sub-boards.
2. **Conquer**:
    - Each sub-board is either:
        - The sub-board that contains the defective square.
        - One of the three other sub-boards that will be marked with a new "defective" square to balance the L-shaped tiles.
3. **Combine**:
    - Place an L-shaped tile in the center of the n×n board such that it covers one square in each of the three n/2×n/2 sub-boards that do not contain the original defective square. ○ Recursively apply the algorithm to each of the four n/2×n/2 sub-boards.

Pseudocode

VISHNU INSTITUTE OF TECHNOLOGY :: BHIMAVARAM
Department of CSE(Artificial Intelligence and Data Science)
Advanced Data Structures And Algorithm Analysis
Design and Analysis of Algorithms UNIT
2 : Divide and Conquer

Here is the pseudocode for the defective chessboard tiling algorithm:

```
function tileDefectiveChessboard(board, top, left, defect_row, defect_col, size)
if size == 2
        tile the 2x2 board directly, taking into account the defective square
return

    half = size / 2

    # Determine the coordinates of the centers of the four sub-boards
mid_row = top + half     mid_col = left + half

    # Determine which sub-board contains the defective square
if defect_row < mid_row and defect_col < mid_col
top_left_defective = true
    else if defect_row < mid_row and defect_col >= mid_col
top_right_defective = true
    else if defect_row >= mid_row and defect_col < mid_col
bottom_left_defective = true     else
        bottom_right_defective = true

    # Place the L-shaped tile in the center of the board
if not top_left_defective
        mark the bottom-right square of the top-left sub-board as defective
if not top_right_defective
        mark the bottom-left square of the top-right sub-board as defective
if not bottom_left_defective
        mark the top-right square of the bottom-left sub-board as defective
if not bottom_right_defective
        mark the top-left square of the bottom-right sub-board as defective
    # Recursively tile the four sub-boards
    tileDefectiveChessboard(board, top, left, defect_row, defect_col, half)
tileDefectiveChessboard(board, top, mid_col, defect_row, defect_col, half)
tileDefectiveChessboard(board, mid_row, left, defect_row, defect_col, half)
tileDefectiveChessboard(board, mid_row, mid_col, defect_row, defect_col, half)
```

Explanation of the Pseudocode

- **Base Case**: If the size of the board is 2x2, tile it directly, taking into account the defective square.
- **Dividing the Board**: The board is divided into four sub-boards. The coordinates of the centers of these sub-boards are determined.
- **Identifying the Defective Sub-board**: The sub-board that contains the original defective square is identified.
- **Placing the L-shaped Tile**: An L-shaped tile is placed in the center of the board, covering one square in each of the three sub-boards that do not contain the original defective square. This marks new defective squares in these sub-boards.
- **Recursive Calls**: The algorithm is recursively applied to each of the four sub-boards.

VISHNU INSTITUTE OF TECHNOLOGY :: BHIMAVARAM

Department of CSE(Artificial Intelligence and Data Science)

Advanced Data Structures And Algorithm Analysis

Design and Analysis of Algorithms UNIT

2 : Divide and Conquer

Working

Given a n by n board where n is of form 2^k where k >= 1 (Basically n is a power of 2 with minimum value as 2). The board has one missing cell (of size 1 x 1). Fill the board using L shaped tiles. A L shaped tile is a 2 x 2 square with one cell of size 1×1 missing.

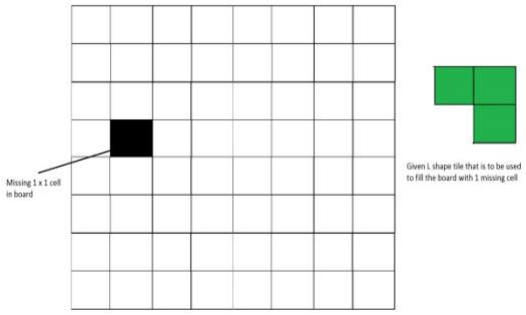


Figure 1: An example input

This problem can be solved using Divide and Conquer. Below is the recursive algorithm.

```
// n is size of given square, p is location of missing cell Tile(int
n, Point p)

1)   Base case: n = 2, A 2 x 2 square with one cell missing is nothing
but a tile and can be filled with a single tile.
2)   Place a L shaped tile at the center such that it does not cover
the n/2 * n/2 subsquare that has a missing square. Now all four
subsquares of size n/2 x n/2 have a missing cell (a cell that doesn't   need
to be filled).  See figure 2 below.

3)   Solve the problem recursively for following four. Let p1, p2,
p3 and    p4 be positions of the 4 missing cells in 4 squares.    a)
Tile(n/2, p1)
b) Tile(n/2, p2)
c) Tile(n/2, p3)
d) Tile(n/2, p3)
```

The below diagrams show working of above algorithm

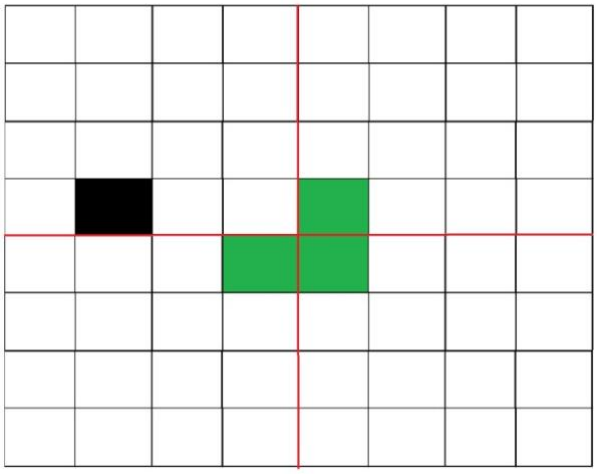


Figure 2: After placing the first tile

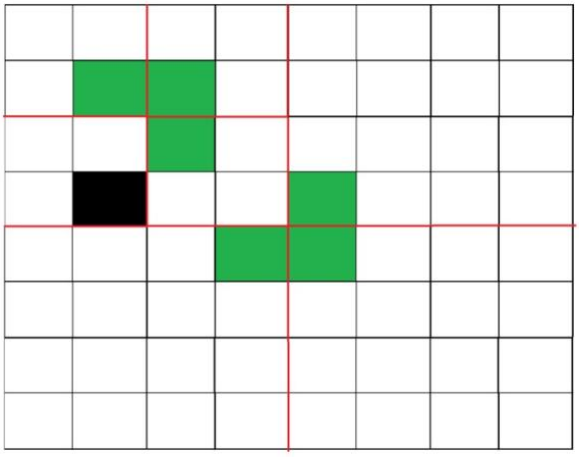


Figure 3: Recurring for the first subsquare.

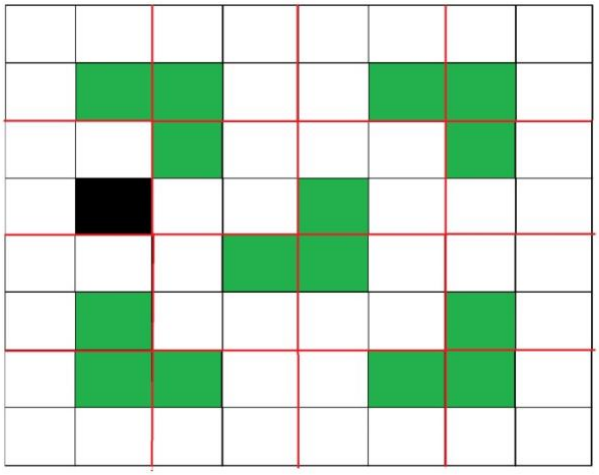


Figure 4: Shows the first step in all four subsquares.

Example 2 : Binary Search

Binary Search is a classic example of the divide-and-conquer algorithmic technique. It works on sorted arrays and helps in finding the position of a target element efficiently. The basic idea is to divide the array into halves and then decide which half to discard based on the comparison of the target element with the middle element. This reduces the search space by half at every step, leading to a time complexity of O(logn).

Here's a detailed explanation of the Binary Search algorithm using the divide-and-conquer approach, followed by its implementation in C.

Binary Search Algorithm

1. **Divide:**
   - Find the middle element of the array.
   - Compare the middle element with the target element.
2. **Conquer:**
   - If the middle element is equal to the target element, return the index of the middle element.
   - If the target element is less than the middle element, repeat the search in the left half of the array. ○ If the target element is greater than the middle element, repeat the search in the right half of the array.
3. **Combine:**
   - Since the problem is divided into two sub-problems and only one sub-problem is recursively solved, there is no need to combine results from sub-problems.

Implementation in C

Here's how you can implement Binary Search in C using the divide-and-conquer approach:

```
#include <stdio.h>

// Function prototype for binary search
int binarySearch(int arr[], int low, int high, int target);

int main() {     //
Sorted array
    int arr[] = {2, 3, 4, 10, 40};
int n = sizeof(arr) / sizeof(arr[0]);
int target = 10;
```

```
    // Perform binary search
    int result = binarySearch(arr, 0, n - 1, target);

    // Print the result
    if (result == -1)
        printf("Element not present in array\n");
else
        printf("Element is present at index %d\n", result);

    return 0;
}

// Binary search function using divide and conquer int
binarySearch(int arr[], int low, int high, int target) {
if (high >= low) {
        int mid = low + (high - low) / 2;

        // Check if the target element is at the middle
if (arr[mid] == target)             return mid;

        // If the target is smaller than the middle element, it must be in
the left sub-array         if (arr[mid] > target)
            return binarySearch(arr, low, mid - 1, target);

        // Otherwise, the target must be in the right sub-array
return binarySearch(arr, mid + 1, high, target);
    }

    // Target element is not present in the array
return -1; }
```

Explanation

- The `binarySearch` function takes an array `arr`, the low and high indices of the array, and the `target` element as inputs.
- It calculates the middle index and compares the middle element with the target.
- If the middle element matches the target, the function returns the middle index.
- If the target is smaller than the middle element, the search continues recursively in the left half of the array.
- If the target is greater than the middle element, the search continues recursively in the right half of the array.
- If the search range becomes invalid (i.e., `high < low`), it means the target is not present in the array, and the function returns -1.

This implementation demonstrates how binary search efficiently reduces the problem size by half in each step, adhering to the divide-and-conquer strategy.

Example 3 : Finding the Maximum and Minimum

Finding the minimum and maximum elements in an array using the divide-and-conquer approach can be more efficient than a simple linear scan. Instead of scanning the array in a single pass, we can divide the array into smaller sub-arrays, find the minimum and maximum in these subarrays, and then combine the results to get the overall minimum and maximum.

Here's a detailed explanation of how to implement this using the divide-and-conquer strategy, followed by a C implementation.

Divide and Conquer Strategy 1.

Divide:

- If the array contains only one element, that element is both the minimum and maximum.
- If the array contains two elements, compare them and decide the minimum and maximum.
- If the array contains more than two elements, split the array into two halves.

2. **Conquer:**
   - Recursively find the minimum and maximum in each half of the array.
3. **Combine:**
   - Compare the minimums from both halves to get the overall minimum.
   - Compare the maximums from both halves to get the overall maximum.

C Implementation

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

// A structure to hold the min and max values
struct MinMax {     int min;     int max;
};

// Function to find the minimum and maximum using divide and conquer
struct MinMax findMinMax(int arr[], int low, int high) {     struct
MinMax minmax, leftMinMax, rightMinMax;     int mid;
```

```
    // If there is only one element
if (low == high) {
minmax.min = arr[low];
minmax.max = arr[low];
return minmax;
    }

    // If there are two elements
if (high == low + 1) {         if
(arr[low] < arr[high]) {
minmax.min = arr[low];
minmax.max = arr[high];
        } else {
minmax.min = arr[high];
minmax.max = arr[low];
        }
        return minmax;
    }

    // If there are more than two elements
mid = (low + high) / 2;     leftMinMax =
findMinMax(arr, low, mid);     rightMinMax =
findMinMax(arr, mid + 1, high);

    // Combine the results     if
(leftMinMax.min < rightMinMax.min) {
minmax.min = leftMinMax.min;
    } else {         minmax.min =
rightMinMax.min;
    }      if (leftMinMax.max >
rightMinMax.max) {         minmax.max =
leftMinMax.max;
    } else {         minmax.max =
rightMinMax.max;
    }      return
minmax;
}  int
main() {
    int arr[] = {1000, 11, 445, 1, 330, 3000};     int
arr_size = sizeof(arr)/sizeof(arr[0]);     struct MinMax
minmax = findMinMax(arr, 0, arr_size - 1);
     printf("Minimum element is %d\n", minmax.min);
printf("Maximum element is %d\n", minmax.max);

    return 0;
}
```

Explanation

1. **Base Cases:**
   - If the sub-array has only one element, it is both the minimum and maximum.
   - If the sub-array has two elements, compare them to determine the minimum and maximum.
2. **Recursive Case:**
   - Split the array into two halves.
   - Recursively find the minimum and maximum of each half.
3. **Combining Results:**
   - The minimum of the entire array is the lesser of the minimums of the two halves. ○ The maximum of the entire array is the greater of the maximums of the two halves.

Time Complexity

The time complexity of this divide-and-conquer approach is O(n), where nnn is the number of elements in the array. This is because, in each step, the array is divided into two halves, and the minimum and maximum of each half are found recursively. Each level of recursion processes all elements exactly once, leading to a linear time complexity.

By using this divide-and-conquer strategy, the number of comparisons is reduced compared to a straightforward linear scan, making it more efficient, especially for large arrays.

Example 4 : Merge Sort

Mergesort is a classic example of the divide and conquer algorithmic paradigm. Here's how it works:

1. **Divide**: Split the array into two halves.
2. **Conquer**: Recursively sort each half.
3. **Combine**: Merge the two sorted halves to produce a single sorted array.

Here's a step-by-step explanation and a Python implementation of Mergesort:

Step-by-Step Explanation

1. **Divide**: ○ Find the middle point of the array to divide it into two halves.
2. **Conquer**: ○ Recursively apply the same procedure to both halves.
3. **Combine**:

o Merge the two sorted halves. To merge, compare the elements of both halves one by one and place the smaller element into the result array until all elements are processed. **Python Implementation**

```
def merge_sort(arr):
if len(arr) <= 1:
return arr

    # Find the middle point to divide the array into two halves
mid = len(arr) // 2

    # Call merge_sort recursively on both halves
left_half = merge_sort(arr[:mid])     right_half
= merge_sort(arr[mid:])

    # Merge the sorted halves
return merge(left_half, right_half)
 def merge(left,
right):
    result = []
i = j = 0

    # Traverse both arrays and copy the smallest element to result
while i < len(left) and j < len(right):         if left[i] <
right[j]:             result.append(left[i])             i += 1
else:             result.append(right[j])             j += 1

    # Collect the remaining elements
result.extend(left[i:])
result.extend(right[j:])

    return result

# Example usage
arr = [38, 27, 43, 3, 9, 82, 10]
sorted_arr = merge_sort(arr)
print("Sorted array is:", sorted_arr)
```

Key Points

- **Time Complexity**: Mergesort has a time complexity of O(n logn), where n is the number of elements in the array. This is because the array is split in half (logn divisions) and merging takes linear time in the size of the array O(n).

- **Space Complexity**: Mergesort requires additional space proportional to the size of the array O(n) for the temporary arrays used during merging.
- **Stability**: Mergesort is a stable sort, meaning that it maintains the relative order of equal elements.

Mergesort is efficient and particularly useful for sorting large datasets that do not fit entirely in memory, as its performance is consistent and predictable.

Example 5 : Quick Sort

Quicksort is one of the efficient sorting algorithm that follows the divide and conquer paradigm. Unlike Mergesort, which divides the array into two halves, Quicksort divides the array based on a pivot element.

Step-by-Step Explanation

1. **Divide**:
   - Choose a pivot element from the array.
   - Partition the array such that elements less than the pivot are on the left, elements greater than the pivot are on the right.
2. **Conquer**: o Recursively apply the same procedure to the left and right subarrays.
3. **Combine**:
   - Since the array is sorted in place, no additional work is needed to combine the results.

Python implementation of Quicksort:

```
def quicksort(arr):
if len(arr) <= 1:
return arr

    # Choose a pivot element
pivot = arr[len(arr) // 2]

    # Partition the array into three parts
left = [x for x in arr if x < pivot]     middle
= [x for x in arr if x == pivot]     right = [x
for x in arr if x > pivot]

    # Recursively apply the same procedure to the left and right parts
return quicksort(left) + middle + quicksort(right)

# Example usage
```

```
arr = [38, 27, 43, 3, 9, 82, 10]
sorted_arr = quicksort(arr) print("Sorted
array is:", sorted_arr)
```

Key Points

- **Time Complexity**: Quicksort has an average time complexity of O(n logn). However, in the worst case (e.g., when the smallest or largest element is always chosen as the pivot), it can degrade to $O(n^2)$. This can be mitigated by using randomized pivot selection or the "median-of-three" method.
- **Space Complexity**: Quicksort has a space complexity of O(logn) due to the recursive call stack.
- **In-place Sorting**: Quicksort can be implemented as an in-place sort, requiring only a small, constant amount of extra storage.
- **Unstable**: Quicksort is not a stable sort. Equal elements may not retain their relative order after sorting.

Performance Measurement in Divide and Conquer

Performance measurement in divide and conquer algorithms typically involves analyzing time complexity and space complexity. Let's break this down using the example of Mergesort and Quicksort:

1. Time Complexity

Divide and Conquer Analysis Framework:

For a divide and conquer algorithm, the time complexity T(n) can often be expressed using the recurrence relation:

$$T(n)=aT(nb)+f(n)$$

where:

- n is the size of the problem.
- a is the number of subproblems in the recursion.
- n/b is the size of each subproblem (assuming they are equal).
- f(n) is the cost outside the recursive calls, often the time to divide the problem and combine the results.

Mergesort:

- **Divide**: Splits the array into two halves. Cost: O(1)
- **Conquer**: Recursively sort two halves. Cost: 2T(n/2) **Combine**: Merging two sorted halves. Cost: O(n)

Recurrence relation: $T(n)=2T(n/2)+O(n)$

Using the Master Theorem, we find: $T(n)=O(n\log n)$

2. Space Complexity

Mergesort:

- Requires O(n) additional space for the temporary arrays used during merging. Total space complexity: O(n).

3. Practical Performance Considerations

- **Cache Efficiency**: Quicksort often performs better in practice due to better cache performance, especially with in-place sorting.
- **Overhead**: Recursive calls and function overhead can affect actual runtime, even if the asymptotic complexity is good.

- **Input Characteristics**: Performance can be affected by the characteristics of the input (e.g., nearly sorted data, repeating elements).

Design and Analysis of Algorithms

UNIT 3 Greedy Method

Topics :

1. **Introduction, General Method**
2. **Knapsack Problem**
3. **Job Sequencing with Dearlines**
4. **Krushkal's Algorithm**
5. **Prims Algorithm**
6. **Djkstra's Algorithm**

Introduction

Among all the algorithmic approaches, the simplest and straightforward approach is the Greedy method. In this approach, the decision is taken on the basis of current available information without worrying about the effect of the current decision in future.

Greedy algorithms build a solution part by part, choosing the next part in such a way, that it gives an immediate benefit. This approach never reconsiders the choices taken previously. This approach is mainly used to solve optimization problems. Greedy method is easy to implement and quite efficient in most of the cases. Hence, we can say that Greedy algorithm is an algorithmic paradigm based on heuristic that follows local optimal choice at each step with the hope of finding global optimal solution.

In many problems, it does not produce an optimal solution though it gives an approximate (near optimal) solution in a reasonable time.

Components of Greedy Algorithm

Greedy algorithms have the following five components −

- **A candidate set** − A solution is created from this set.
- **A selection function** − Used to choose the best candidate to be added to the solution. • **A feasibility function** − Used to determine whether a candidate can be used to contribute to the solution. • **An objective function** − Used to assign a value to a solution or a partial solution. • **A solution function** − Used to indicate whether a complete solution has been reached.

Areas of Application

Greedy approach is used to solve many problems, such as

- Finding the shortest path between two vertices using Dijkstra's algorithm.

- Finding the minimal spanning tree in a graph using Prim's /Kruskal's algorithm, etc.

Where Greedy Approach Fails

In many problems, Greedy algorithm fails to find an optimal solution; moreover it may produce a worst solution. Problems like Travelling Salesman and Knapsack cannot be solved using this approach.

Knapsack Problem

The Greedy algorithm could be understood very well with a well-known problem referred to as Knapsack problem. Although the same problem could be solved by employing other algorithmic approaches, Greedy approach solves Fractional Knapsack problem reasonably in a good time. Let us discuss the Knapsack problem in detail.

Given a set of items, each with a weight and a value, determine a subset of items to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

The knapsack problem is in combinatorial optimization problem. It appears as a subproblem in many, more complex mathematical models of real-world problems. One general approach to difficult problems is to identify the most restrictive constraint, ignore the others, solve a knapsack problem, and somehow adjust the solution to satisfy the ignored constraints.

Applications

In many cases of resource allocation along with some constraint, the problem can be derived in a similar way of Knapsack problem. Following is a set of example.

- Finding the least wasteful way to cut raw materials
- portfolio optimization
- Cutting stock problems **Problem Scenario**

A thief is robbing a store and can carry a maximal weight of W into his knapsack. There are n items available in the store and weight of i^{th} item is w_i and its profit is p_i. What items should the thief take?

In this context, the items should be selected in such a way that the thief will carry those items for which he will gain maximum profit. Hence, the objective of the thief is to maximize the profit.

Based on the nature of the items, Knapsack problems are categorized as

- Fractional Knapsack
- Knapsack

Fractional Knapsack

In this case, items can be broken into smaller pieces, hence the thief can select fractions of items.

According to the problem statement, •

There are **n** items in the store

- Weight of **i**th item $w_i>0$

Profit for **i**th item $p_i>0$

- and
- Capacity of the Knapsack is **W**

In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction x_i of **i**th item.

$$0 \leqslant x_i \leqslant 1$$

The **i**th item contributes the weight $x_i.w_i$ to the

total weight in the knapsack and profit $x^i.p_i$ to the

total profit.

Hence, the objective of this algorithm is to

$$maximize \sum_{n=1}^{n}(x_i.pi)$$

subject to constraint,

$$\sum_{n=1}^{n}(x_i.wi) \leqslant W$$

It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit.

Thus, an optimal solution can be obtained by

$$\sum_{n=1}^{n}(x_i.wi) = W$$

In this context, first we need to sort those items according to the value of $p_i w_i$

, so that $p_{i+1} w_{i+1} \leq p_i w_i$

. Here, x is an array to store the fraction of items.

Algorithm: Greedy-Fractional-Knapsack (w[1..n], p[1..n], W)

{

for i = 1 to n do

x[i] = 0 weight

= W

for i = 1 to n

```
{
if w[i] > W then   x[i]
= 1
weight-weight-w[i];
}
if(i<=n) then x[i]:=weight/w[i];
}
```

Analysis

If the provided items are already sorted into a decreasing order of $\mathbf{p_i.w_i}$, then the whileloop takes a time in ***O(n)***; Therefore, the total time including the sort is in ***O(n logn)***.

Example

Let us consider that the capacity of the knapsack $W = 60$ and the list of provided items are shown in the following table −

Item	A	B	C	D
Profit	280	100	120	120
Weight	40	10	20	24
Ratio $(p_i w_i)$	7	10	6	5

The provided items are not sorted based on $\mathbf{p_i w_i}$. After sorting, the items are as shown in the following table.

Item	B	A	C	D
Profit	100	280	120	120
Weight	10	40	20	24
Ratio $(p_i w_i)$	10	7	6	5

Solution

After sorting all the items according to $p_i w_i$

First all of ***B*** is chosen as weight of ***B*** is less than the capacity of the knapsack. Next, item ***A*** is chosen, as the available capacity of the knapsack is greater than the weight of ***A***. Now, ***C*** is chosen as the next item. However, the whole item cannot be chosen as the remaining capacity of the knapsack is less than the weight of ***C***.

Hence, fraction of ***C*** (i.e. (60 − 50)/20) is chosen.

Now, the capacity of the Knapsack is equal to the selected items. Hence, no more item can be selected.

The total weight of the selected items is **10 + 40 + 20 * (10/20) = 60**

And the total profit is **100 + 280 + 120 * (10/20) = 380 + 60 = 440**

This is the optimal solution. We cannot gain more profit selecting any different combination of items.

Job Sequencing with Deadlines

In job sequencing problem, the objective is to find a sequence of jobs, which is completed within their deadlines and gives maximum profit.

Solution

Let us consider, a set of ***n*** given jobs which are associated with deadlines and profit is earned, if a job is completed by its deadline. These jobs need to be ordered in such a way that there is maximum profit.

It may happen that all of the given jobs may not be completed within their deadlines.

Assume, deadline of **ith** job J_i is d_i and the profit received from this job is p_i. Hence, the optimal solution of this algorithm is a feasible solution with maximum profit. Thus,

$D(i)>0$ for $1 \leqslant i \leqslant n$

.

Initially, these jobs are ordered according to profit, i.e. $p1 \geqslant p2 \geqslant p3 \geqslant \ldots \geqslant pn$

.

```
Algorithm: Job-Sequencing-With-Deadline (D, J, n,
k)  D(0) := J(0) := 0  k := 1
J(1) := 1   // means first job is selected  for
i = 2 … n do
   r := k
  while D(J(r)) > D(i) and D(J(r)) ≠ r do
r := r − 1
  if D(J(r)) ≤ D(i) and D(i) > r then
       for l = k … r + 1 by -1 do
          J(l + 1) := J(l)
          J(r + 1) := i
k := k + 1
```

Analysis

In this algorithm, we are using two loops, one is within another. Hence, the complexity of this algorithm is $O(n^2)$

Example

Let us consider a set of given jobs as shown in the following table. We have to find a sequence of jobs, which will be completed within their deadlines and will give maximum profit. Each job is associated with a deadline and profit.

Job	J_1	J_2	J_3	J_4	J_5
Deadline	2	1	3	2	1
Profit	60	100	20	40	20

Solution

To solve this problem, the given jobs are sorted according to their profit in a descending order. Hence, after sorting, the jobs are ordered as shown in the following table.

Job	J_2	J_1	J_4	J_3	J_5
Deadline	1	2	2	3	1
Profit	100	60	40	20	20

From this set of jobs, first we select J_2, as it can be completed within its deadline and contributes maximum profit.

- Next, J_1 is selected as it gives more profit compared to J_4.
- In the next clock, J_4 cannot be selected as its deadline is over, hence J_3 is selected as it executes within its deadline.
- The job J_5 is discarded as it cannot be executed within its deadline.

Thus, the solution is the sequence of jobs (J_2, J_1, J_3), which are being executed within their deadline and gives maximum profit.

Total profit of this sequence is **100 + 60 + 20 = 180**.

Krushkal's Minimum Spanning Tree – Greedy Algorithm

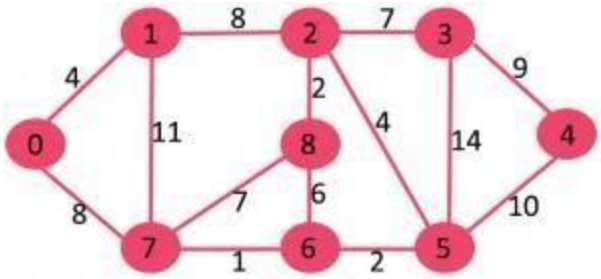
Given a connected and undirected graph, a *spanning tree* of graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees.

A *minimum spanning tree (MST)* or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree. A minimum spanning tree has (V – 1) edges where V is the number of vertices in the given graph.

Algorithm

1. **Sort all the edges in non-decreasing order of their weight.**
2. **Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.**
3. **Repeat step#2 until there are (V-1) edges in the spanning tree.**

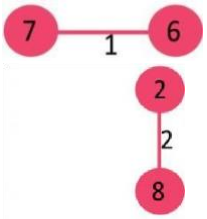
The algorithm is a Greedy Algorithm. The Greedy Choice is to pick the smallest weight edge that does not cause a cycle in the MST constructed so far. Let us understand it with an example: Consider the below input graph.



The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.

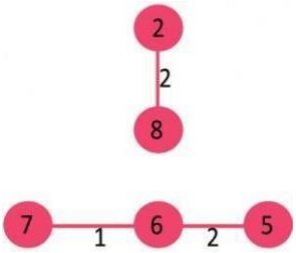
```
After sorting:
Weight   Src    Dest
1          7      6
2          8      2
2          6      5
4          0      1
4          2      5
6          8      6
7          2      3
7          7      8
8          0      7
8          1      2
9          3      4
10         5      4
11         1      7
14         3      5
```

Now pick all edges one by one from sorted list of edges **1.**
Pick edge 7-6: No cycle is formed, include it.

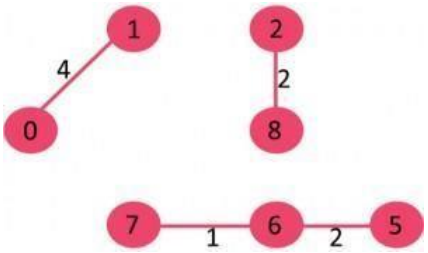


2. *Pick edge 8-2:* No cycle is formed, include it.

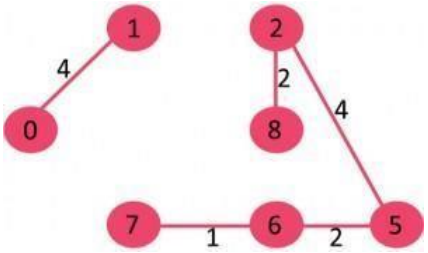
3. *Pick edge 6-5:* No cycle is formed, include it.



4. *Pick edge 0-1:* No cycle is formed, include it.

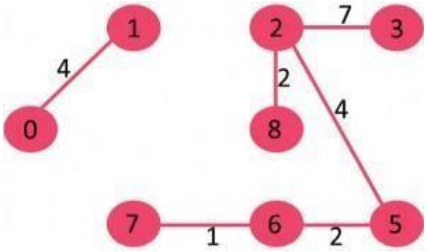


5. *Pick edge 2-5:* No cycle is formed, include it.



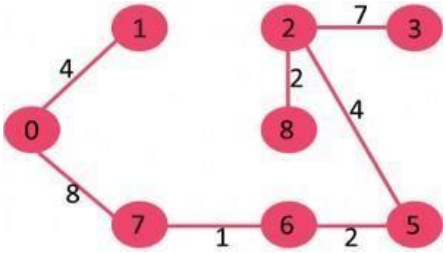
6. *Pick edge 8-6:* Since including this edge results in cycle, discard it.

7. *Pick edge 2-3:* No cycle is formed, include it.



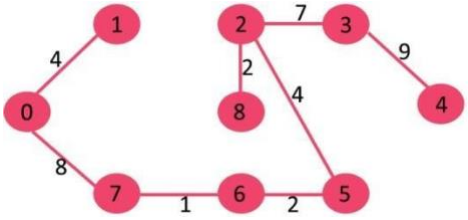
8. *Pick edge 7-8:* Since including this edge results in cycle, discard it.

9. *Pick edge 0-7:* No cycle is formed, include it.



10. *Pick edge 1-2:* Since including this edge results in cycle, discard it.

11. *Pick edge 3-4:* No cycle is formed, include it.



Since the number of edges included equals (V – 1), the algorithm stops here.

Prim's Minimum Spanning Tree – Greedy Algorithm

Prim's starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.A group of edges that connects two set of vertices in a graph is called cut in graph theory. *So, at every step of Prim's algorithm, we find a cut (of two sets, one contains the vertices already included in MST and other contains rest of the verices), pick the minimum weight edge from the cut and include this vertex to MST Set (the set that contains already included vertices).*

The idea behind Prim's algorithm is simple; a spanning tree means all vertices must be connected. So the two disjoint subsets (discussed above) of vertices must be connected to make a *Spanning* Tree. And they must be connected with the minimum weight edge to make it a *Minimum* Spanning Tree.

Algorithm

1) Create a set *mstSet* that keeps track of vertices already included in MST.

2) Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.

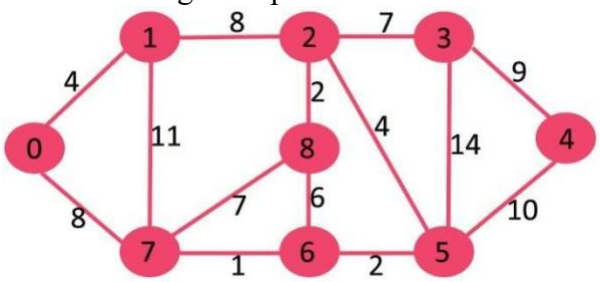
3) While mstSet doesn't include all vertices

A. **Pick a vertex *u* which is not there in *mstSet* and has minimum key value.** B. **Include *u* to mstSet.**

C. **Update key value of all adjacent vertices of *u*. To update the key values, iterate through all adjacent vertices. For every adjacent vertex *v*, if weight of edge *u*-*v* is less than the previous key value of *v*, update the key value as weight of *u*-*v***

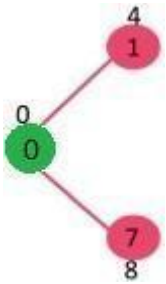
The idea of using key values is to pick the minimum weight edge from cut. The key values are used only for vertices which are not yet included in MST, the key value for these vertices indicate the minimum weight edges connecting them to the set of vertices included in MST.

Let us understand with the following example:

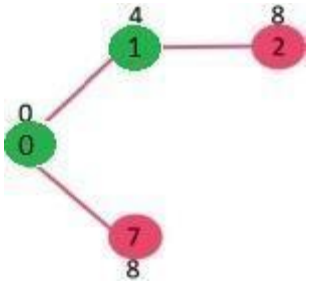


The set *mstSet* is initially empty and keys assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum key value. The vertex 0 is picked, include it in *mstSet*. So *mstSet* becomes {0}. After including to *mstSet*, update key values of adjacent vertices. Adjacent vertices of 0 are 1 and 7. The key values of 1

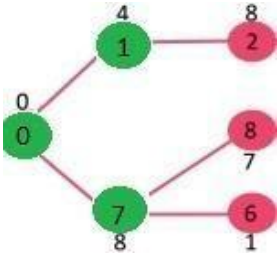
and 7 are updated as 4 and 8. Following subgraph shows vertices and their key values, only the vertices with finite key values are shown. The vertices included in MST are shown in green color.



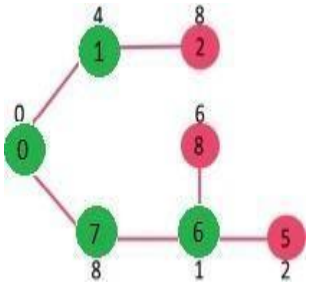
Pick the vertex with minimum key value and not already included in MST (not in mstSET). The vertex 1 is picked and added to mstSet. So mstSet now becomes {0, 1}. Update the key values of adjacent vertices of 1. The key value of vertex 2 becomes 8.



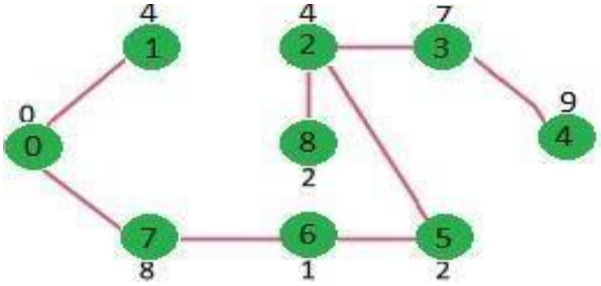
Pick the vertex with minimum key value and not already included in MST (not in mstSET). We can either pick vertex 7 or vertex 2, let vertex 7 is picked. So mstSet now becomes {0, 1, 7}. Update the key values of adjacent vertices of 7. The key value of vertex 6 and 8 becomes finite (7 and 1 respectively).



Pick the vertex with minimum key value and not already included in MST (not in mstSET). Vertex 6 is picked. So mstSet now becomes {0, 1, 7, 6}. Update the key values of adjacent vertices of 6. The key value of vertex 5 and 8 are updated.



We repeat the above steps until *mstSet* includes all vertices of given graph. Finally, we get the following graph.



Single Source Shortest Paths Problem – DjkStra's Algorithm

Dijkstra's algorithm is very similar to Prim's algorithm for minimum spanning tree. Like Prim's MST, we generate a *SPT (shortest path tree)* with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source. Below are the detailed steps used in Dijkstra's algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

Algorithm

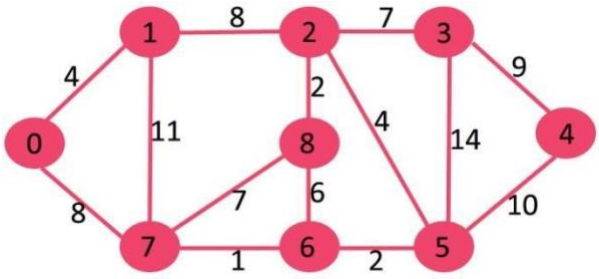
1) Create a set *sptSet* (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.

2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.

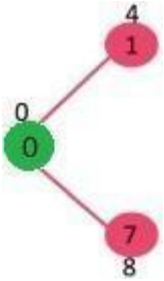
3) While *sptSet* doesn't include all vertices

- **A. Pick a vertex u which is not there in *sptSet* and has minimum distance value.**
- **B. Include u to *sptSet*.**
- **C. Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.**

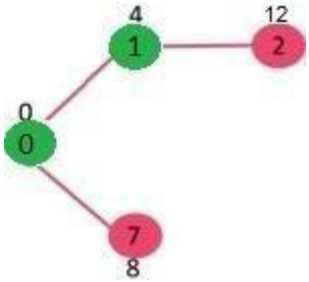
Let us understand Djkstra's Algorithm with the following example:



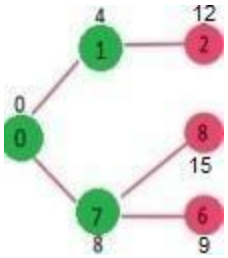
The set *sptSet* is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum distance value. The vertex 0 is picked, include it in *sptSet*. So *sptSet* becomes {0}. After including 0 to *sptSet*, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green colour.



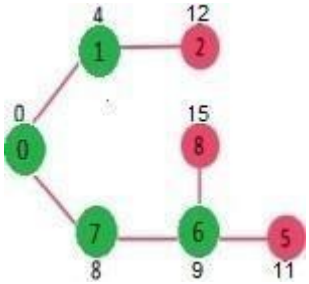
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). The vertex 1 is picked and added to sptSet. So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.



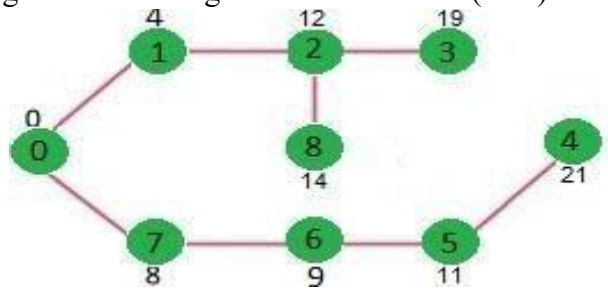
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).



Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 6 is picked. So sptSet now becomes {0, 1, 7, 6}. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



We repeat the above steps until *sptSet* doesn't include all vertices of given graph. Finally, we get the following Shortest Path Tree (SPT).



Design and Analysis of Algorithms

UNIT 4

Dynamic Programming

Topics

- **Introduction, General Method**
- **Multistage Graphs**
- **Single Source Shortest Paths : Bellman-Ford Algorithm**
- **All pairs shortest path problem : Floyd Warshall Algorithm**
- **0/1 Knapsack Problem**
- **Travelling Salesperson Problem**
- **Reliability Design**

Introduction

Dynamic Programming is also used in optimization problems. Like divideand-conquer method, Dynamic Programming solves problems by combining the solutions of subproblems. Moreover, Dynamic Programming algorithm solves each sub-problem just once and then saves its answer in a table, thereby avoiding the work of re-computing the answer every time.

Two main properties of a problem suggest that the given problem can be solved using Dynamic Programming. These properties are **overlapping sub-problems and optimal substructure**.

Overlapping Sub-Problems

Similar to Divide-and-Conquer approach, Dynamic Programming also combines solutions to sub-problems. It is mainly used where the solution of one sub-problem is needed repeatedly. The computed solutions are stored in a table, so that these don't have to be re-computed. Hence, this technique is needed where overlapping sub-problem exists.

For example, Binary Search does not have overlapping sub-problem. Whereas recursive program of Fibonacci numbers have many overlapping sub-problems.

Optimal Sub-Structure

A given problem has Optimal Substructure Property, if the optimal solution of the given problem can be obtained using optimal solutions of its sub-problems.

For example, the Shortest Path problem has the following optimal substructure property –

If a node **x** lies in the shortest path from a source node **u** to destination node **v**, then the shortest path from **u** to **v** is the combination of the shortest path from **u** to **x**, and the shortest path from **x** to **v**.

The standard All Pair Shortest Path algorithms like Floyd-Warshall and Bellman-Ford are typical examples of Dynamic Programming.

General Method

Dynamic Programming algorithm is designed using the following four steps –

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution, typically in a bottom-up fashion.
- Construct an optimal solution from the computed information.

Applications of Dynamic Programming Approach

- Matrix Chain Multiplication
- Longest Common Subsequence
- Travelling Salesman Problem

Multistage Graphs

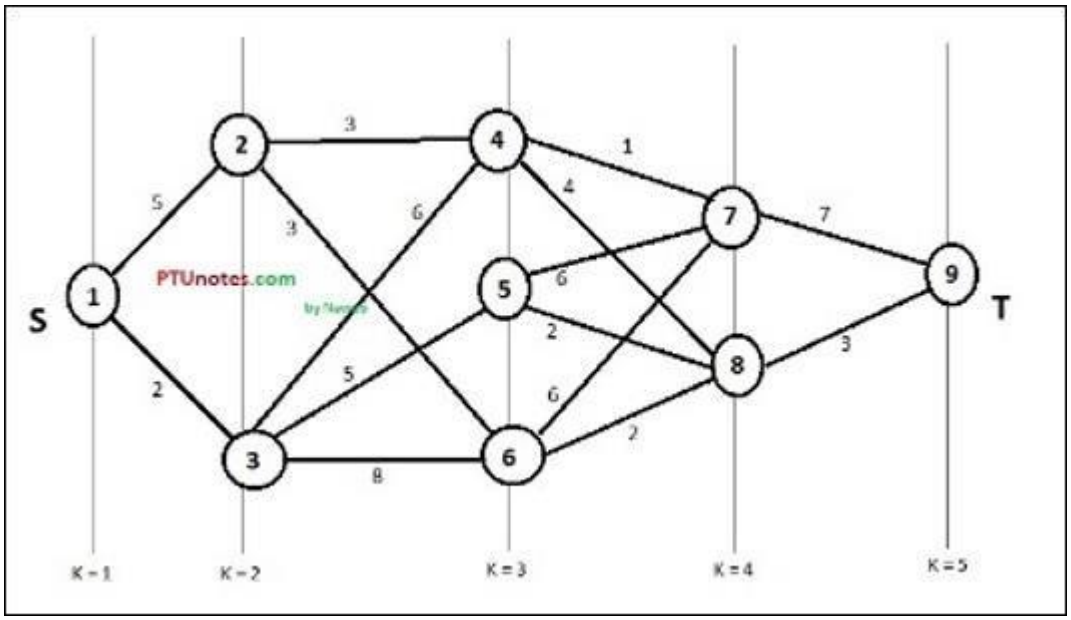
A multistage graph **G = (V, E)** is a directed graph where vertices are partitioned into **k** (where ***k* > 1**) number of disjoint subsets $S = \{s_1, s_2, \ldots, s_k\}$ such that edge *(u, v)* is in E, then $u \in s_i$ and $v \in s_{1+1}$ for some subsets in the partition and $|s_1| = |s_k| = 1$.

The vertex $s \in s_1$ is called the **source** and the vertex $t \in s_k$ is called **sink**. ***G*** is usually assumed to be a weighted graph. In this graph, cost of an edge *(i, j)* is represented by *c(i, j)*. Hence, the cost of path from source ***s*** to sink ***t*** is the sum of costs of each edges in this path.

The multistage graph problem is finding the path with minimum cost from source ***s*** to sink ***t***.

Example

Consider the following example to understand the concept of multistage graph.



According to the formula, we have to calculate the cost **(i, j)** using the following steps

Step-1: Cost (K-2, j)

In this step, three nodes (node 4, 5. 6) are selected as **j**. Hence, we have three options to choose the minimum cost at this step.

Cost(3, 4) = min {c(4, 7) + Cost(7, 9),c(4, 8) + Cost(8, 9)} = 7

Cost(3, 5) = min {c(5, 7) + Cost(7, 9),c(5, 8) + Cost(8, 9)} = 5

Cost(3, 6) = min {c(6, 7) + Cost(7, 9),c(6, 8) + Cost(8, 9)} = 5 Step-2: Cost (K-3, j)

Two nodes are selected as j because at stage k - 3 = 2 there are two nodes, 2 and 3. So, the value i = 2 and j = 2 and 3.

Cost(2, 2) = min {c(2, 4) + Cost(4, 8) + Cost(8, 9),c(2, 6) +

Cost(6, 8) + Cost(8, 9)} = 8

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

Step-3: Cost (K-4, j)

Cost (1, 1) = min {c(1, 2) + Cost(2, 6) + Cost(6, 8) + Cost(8, 9), c(1, 3) + Cost(3, 6) + Cost(6, 8 + Cost(8, 9))} = 13

Hence, the path having the minimum cost is **1**→ **2**→ **6**→ **8**→ **9**. **Single**

Source Shortest Paths

Bellman Ford Algorithm

This algorithm solves the single source shortest path problem of a directed graph ***G = (V, E)*** in which the edge weights may be negative. Moreover, this algorithm can be applied to find the shortest path, if there does not exist any negative weighted cycle.

```
Algorithm: Bellman-Ford-Algorithm (G, w, s)  for
each vertex v ∈ G.V
   v.d := ∞
   v.∏ := NIL
s.d := 0  for i = 1 to |G.V|
- 1     for each edge (u, v)
∈ G.E        if v.d > u.d +
w(u, v)           v.d := u.d
+w(u, v)
         v.∏ := u  for
each edge (u, v) ∈ G.E
if v.d > u.d + w(u, v)
return FALSE  return TRUE
```

Analysis

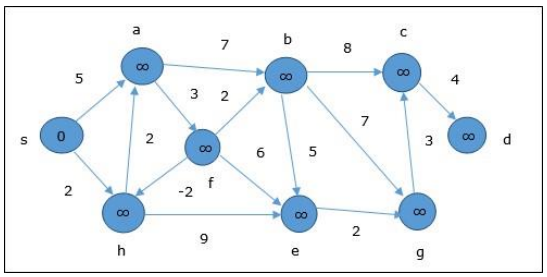
The first **for** loop is used for initialization, which runs in ***O(V)*** times. The next **for** loop runs |***V - 1***| passes over the edges, which takes ***O(E)*** times.

Hence, Bellman-Ford algorithm runs in ***O(V, E)*** time.

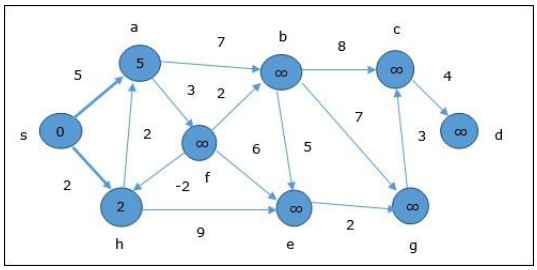
Example

The following example shows how Bellman-Ford algorithm works step by step. This graph has a negative edge but does not have any negative cycle, hence the problem can be solved using this technique.

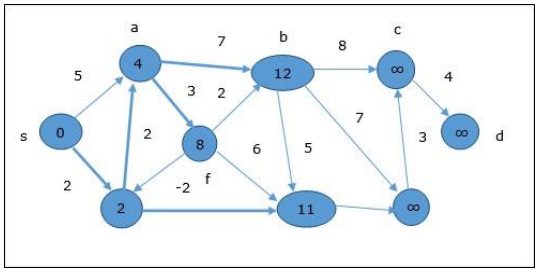
At the time of initialization, all the vertices except the source are marked by ∞ and the source is marked by **0**.



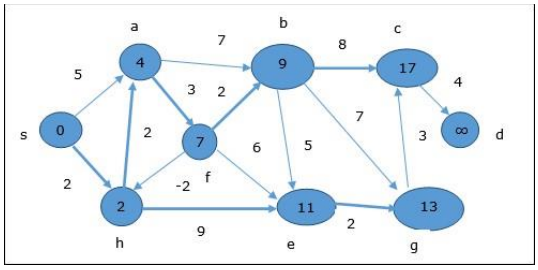
In the first step, all the vertices which are reachable from the source are updated by minimum cost. Hence, vertices ***a*** and ***h*** are updated.



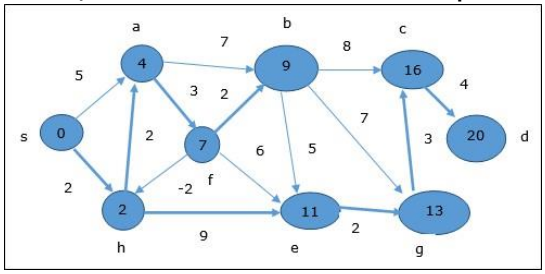
In the next step, vertices ***a, b, f*** and ***e*** are updated.



Following the same logic, in this step vertices ***b, f, c*** and ***g*** are updated.



Here, vertices ***c*** and ***d*** are updated.



Hence, the minimum distance between vertex **s** and vertex **d** is **20**.

Based on the predecessor information, the path is s→ h→ e→ g→ c→ d **All pairs Shortest Paths**

The Floyd Warshall Algorithm is for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph. Example:

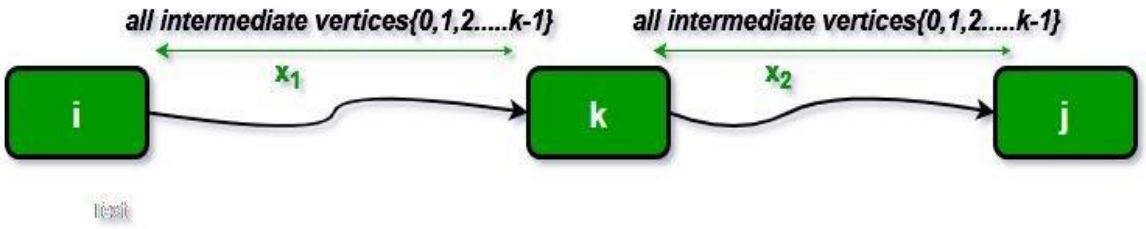
```
Input:       graph[][] = { {0,   5,  INF,
10},                       {INF,  0,   3,
INF},
                     {INF, INF, 0,    1},
{INF, INF, INF, 0} } which represents the
following graph
               10
       (0)------->(3)
|         /|\
      5 |          |
        |          | 1
        \|/        |
       (1)------->(2)
             3
Note that the value of graph[i][j] is 0 if i is equal to j
And graph[i][j] is INF (infinite) if there is no edge from vertex i to j.

Output:
Shortest distance matrix
      0      5      8      9
INF       0      3      4
    INF    INF      0      1
    INF    INF    INF      0
```

Algorithm

We initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices 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. **1)** k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.

2) 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]

The following figure shows the above optimal substructure property in the all-pairs shortest path problem.



Note : Use Example of Floyd Warshall Algorithm discussed in full in class

0/1 Knapsack Problem

Greedy approach gives an optimal solution for Fractional Knapsack. In 01 Knapsack, items cannot be broken which means the thief should take the item as a whole or should leave it. This is reason behind calling it as 0-1 Knapsack.

Hence, in case of 0-1 Knapsack, the value of x_i can be either ***0*** or ***1***, where other constraints remain the same.

0-1 Knapsack cannot be solved by Greedy approach. Greedy approach does not ensure an optimal solution. In many instances, Greedy approach may give an optimal solution.

The following examples will establish our statement.

Example-1

Let us consider that the capacity of the knapsack is W = 25 and the items are as shown in the following table.

Item	A	B	C	D
Profit	24	18	18	10
Weight	24	10	10	7

Without considering the profit per unit weight (p_i/w_i), if we apply Greedy approach to solve this problem, first item ***A*** will be selected as it will contribute max$_i$mum profit among all the elements.

After selecting item ***A***, no more item will be selected. Hence, for this given set of items total profit is ***24***. Whereas, the optimal solution can be achieved by selecting items, ***B*** and C, where the total profit is 18 + 18 = 36.

Travelling Salesperson Problem

A traveler needs to visit all the cities from a list, where distances between all the cities are known and each city should be visited just once. What is the shortest possible route that he visits each city exactly once and returns to the origin city?

Solution

Travelling salesman problem is the most notorious computational problem. We can use brute-force approach to evaluate every possible tour and select the best one. For **n** number of vertices in a graph, there are ***(n - 1)!*** number of possibilities.

Instead of brute-force using dynamic programming approach, the solution can be obtained in lesser time, though there is no polynomial time algorithm.

Let us consider a graph ***G = (V, E)***, where ***V*** is a set of cities and ***E*** is a set of weighted edges. An edge ***e(u, v)*** represents that vertices ***u*** and ***v*** are connected. Distance between vertex ***u*** and ***v*** is ***d(u, v)***, which should be non-negative.

Suppose we have started at city ***1*** and after visiting some cities now we are in city ***j***. Hence, this is a partial tour. We certainly need to know ***j***, since this will determine which cities are most convenient to visit next. We also need to know all the cities visited so far, so that we don't repeat any of them. Hence, this is an appropriate sub-problem.
For a subset of cities ***S ∈ {1, 2, 3, … , n}*** that includes ***1***, and ***j ∈ S***, let ***C(S, j)*** be the length of the shortest path visiting each node in **S** exactly once, starting at ***1*** and ending at ***j***.

When |***S***| > 1, we define ***C(S, 1)*** = ∝ since the path cannot start and end at **1**.

Now, let express **C(S, j)** in terms of smaller sub-problems. We need to start at ***1*** and end at **j**. We should select the next city in such a way that

$$C(S,j)=\min C(S-\{j\},i)+d(i,j) \text{ where } i\in S \text{ and } i\neq j$$

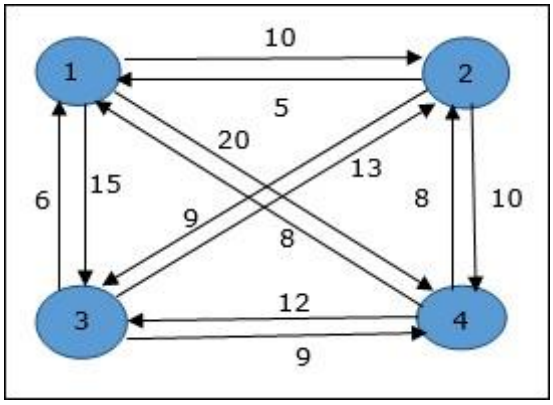
```
Algorithm: Traveling-Salesman-Problem
C ({1}, 1) = 0  for s = 2 to n do
   for all subsets S Є {1, 2, 3, … , n} of size s and containing 1
C (S, 1) = ∞
   for all j Є S and j ≠ 1
      C (S, j) = min {C (S – {j}, i) + d(i, j) for i Є S and i ≠ j}
Return minj C ({1, 2, 3, …, n}, j) + d(j, i)
```

Analysis

There are at the most $2^n.n2n.n$ sub-problems and each one takes linear time to solve. Therefore, the total running time is $O(2^n.n^2)O(2n.n2)$.

Example

In the following example, we will illustrate the steps to solve the travelling salesman problem.



From the above graph, the following weights table is prepared.

W	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

S = Φ

$$Cost(2,\Phi,1)=d(2,1)=5 Cost(2,\Phi,1)=d(2,1)=5$$

$$Cost(3,\Phi,1)=d(3,1)=6 Cost(3,\Phi,1)=d(3,1)=6$$

$$Cost(4,\Phi,1)=d(4,1)=8 Cost(4,\Phi,1)=d(4,1)=8$$

S = 1

$$Cost(i,s)=min\{Cost(j,s–(j))+d[i,j]\} Cost(i,s)=min\{Cost(j,s–(j))+d[i,j]\}$$

$$Cost(2,\{3\},1)=d[2,3]+Cost(3,\Phi,1)=9+6=15 Cost(2,\{3\},1)=d[2,3]+Cost(3,\Phi,1)=9+6=15$$

$$Cost(2,\{4\},1)=d[2,4]+Cost(4,\Phi,1)=10+8=18 Cost(2,\{4\},1)=d[2,4]+Cost(4,\Phi,1)=10+8=18$$

$$Cost(3,\{2\},1)=d[3,2]+Cost(2,\Phi,1)=13+5=18 Cost(3,\{2\},1)=d[3,2]+Cost(2,\Phi,1)=13+5=18$$

$$Cost(3,\{4\},1)=d[3,4]+Cost(4,\Phi,1)=12+8=20 Cost(3,\{4\},1)=d[3,4]+Cost(4,\Phi,1)=12+8=20$$

$$Cost(4,\{3\},1)=d[4,3]+Cost(3,\Phi,1)=9+6=15 Cost(4,\{3\},1)=d[4,3]+Cost(3,\Phi,1)=9+6=15$$

$$Cost(4,\{2\},1)=d[4,2]+Cost(2,\Phi,1)=8+5=13 Cost(4,\{2\},1)=d[4,2]+Cost(2,\Phi,1)=8+5=13$$

S = 2

$$Cost(2,\{3,4\},1)=\{d[2,3]+Cost(3,\{4\},1)=9+20=29d[2,4]+Cost(4,\{3\},1)=10+15=25=25 Cost(2,\{3,4\},1)=\{d[2,3]+Cost(3,\{4\},1)=9+20=29d[2,4]+Cost(4,\{3\},1)=10+15=25=25$$

$$Cost(3,\{2,4\},1)=\{d[3,2]+Cost(2,\{4\},1)=13+18=31d[3,4]+Cost(4,\{2\},1)=12+13=25=25 Cost(3,\{2,4\},1)=\{d[3,2]+Cost(2,\{4\},1)=13+18=31d[3,4]+Cost(4,\{2\},1)=12+13=25=25$$

$$Cost(4,\{2,3\},1)=\{d[4,2]+Cost(2,\{3\},1)=8+15=23d[4,3]+Cost(3,\{2\},1)=9+18=27=23 Cost(4,\{2,3\},1)=\{d[4,2]+Cost(2,\{3\},1)=8+15=23d[4,3]+Cost(3,\{2\},1)=9+18=27=23$$

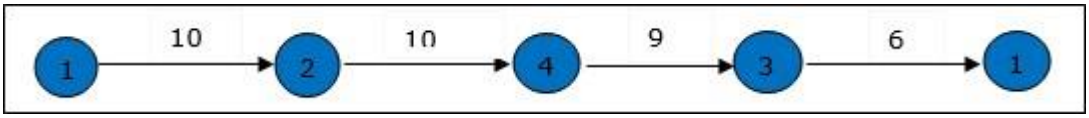
S = 3

$$Cost(1,\{2,3,4\},1)=\begin{cases} d[1,2]+Cost(2,\{3,4\},1)=10+25=35 \\ d[1,3]+Cost(3,\{2,4\},1)=15+25=40 \\ d[1,4]+Cost(4,\{2,3\},1)=20+23=43=35 \end{cases} Cost(1,\{2,3,4\},1)=\{d[1,2]+Cost(2,\{3,4\},1)=10+25=35d[1,3]+Cost(3,\{2,4\},1)=15+25=40d[1,4]+Cost(4,\{2,3\},1)=20+23=43=35$$

The minimum cost path is 35.

Start from cost **{1, {2, 3, 4}, 1}**, we get the minimum value for **d [1, 2]**. When **s = 3**, select the path from 1 to 2 (cost is 10) then go backwards. When **s = 2**, we get the minimum value for **d [4, 2]**. Select the path from 2 to 4 (cost is 10) then go backwards.

When **s = 1**, we get the minimum value for **d [4, 2]** but 2 and 4 is already selected. Therefore, we select **d [4, 3]** (two possible values are 15 for d [2, 3] and d [4, 3], but our last node of the path is 4). Select path 4 to 3 (cost is 9), then go to **s = Φ** step. We get the minimum value for **d [3, 1]** (cost is 6).



Reliability Design

Reliability design in dynamic programming focuses on ensuring that systems are not only efficient but also reliable throughout their operational life. This involves considering various factors, including system components, environmental conditions, and operational parameters. Here's a deeper look at the topic:

Key Concepts

1. **Dynamic Programming (DP):**
   - A mathematical optimization technique used to solve complex problems by breaking them down into simpler subproblems.
   - DP is particularly useful in problems involving decision-making over time, where the solution to a larger problem depends on solutions to smaller subproblems.
2. **Reliability:**
   - Refers to the probability that a system will perform its intended function without failure over a specified period.
   - Reliability can be influenced by design choices, component failure rates, and operational conditions.

Steps in Reliability Design Using Dynamic Programming

1. **Problem Formulation:**
   - Define the system model, including all components and their interdependencies.
   - Identify the objective function, often focusing on maximizing reliability or minimizing costs associated with failures.
2. **State Representation:**
   - Use state variables to represent the condition of the system. This may include the operational status of components, time, and other relevant factors.

    - Each state transition reflects changes due to operational decisions, maintenance actions, or failures.
3. **Transition Probabilities:**
    - Establish probabilities associated with state transitions, particularly those that involve failures and repairs.
    - This includes determining the failure rates of components and the impact of repairs on system reliability.
4. **Cost Function:**
    - Define a cost function that includes both direct costs (e.g., maintenance, replacement) and indirect costs (e.g., lost productivity due to failures).
    - The goal is to optimize this cost function while ensuring reliability.
5. **Dynamic Programming Recursion:**
    - Develop recursive relationships that relate the current state to possible future states, considering the decisions made at each step.
    - This often involves defining a Bellman equation that captures the trade-offs between immediate costs and future reliability.
6. **Solution Approach:**
    - Use backward induction or other methods to solve the dynamic programming model.
    - This involves calculating the optimal decisions at each state, starting from the final states and working backward to the initial state.
7. **Policy Extraction:**
    - Extract optimal policies from the dynamic programming solution, indicating the best actions to take at each state to maximize reliability.

Applications

- **Manufacturing Systems:**
    - Designing production systems that minimize downtime and maximize throughput while considering equipment reliability.
- **Transportation Networks:**
    - Developing reliable routing strategies in logistics, considering the probability of failures and delays.
- **Telecommunications:**

- Ensuring network reliability by optimizing resource allocation and redundancy strategies.

Challenges

- **Complexity:**
  - Real-world systems can have a vast number of states and transitions, making DP computationally intensive.
- **Data Availability:**
  - Reliable estimates of failure rates and transition probabilities are crucial for effective modeling.
- **Trade-offs:**
  - Balancing reliability with cost efficiency can lead to complex decision-making scenarios.

Design and Analysis of Algorithms

UNIT 5 Part 1

Back Tracking

Topics:

1. **Introduction**
2. **8 Queens Problem**
3. **Graph Coloring**
4. **Sum of Subsets**

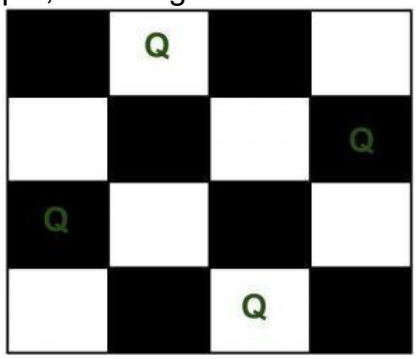
Introduction

Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time (by time, here, is referred to the time elapsed till reaching any level of the search tree).

Consider a situation that you have three boxes in front of you and only one of them has a gold coin in it but you do not know which one. So, in order to get the coin, you will have to open all of the boxes one by one. You will first check the first box, if it does not contain the coin, you will have to close it and check the second box and so on until you find the coin. This is what backtracking is, that is solving all sub-problems one by one in order to reach the best possible solution.

8 Queens Problem

Let us try to solve a standard Backtracking problem, **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. For example, following is a solution for 4 Queen problem.



The expected output is a binary matrix which has 1s for the blocks where queens are placed. For example, following is the output matrix for the above 4 queen solution.

```
{ 0,  1,  0,  0}
{ 0,  0,  0,  1}
{ 1,  0,  0,  0}
{ 0,  0,  1,  0}
```

Backtracking Algorithm: 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.

```
1) Start in the leftmost column
2)  If  all  queens  are  placed
return true
3) Try all rows in the current column.  Do following for every tried row.
a)     If the queen can be placed safely in this row then mark this [row,
column] as part of the solution and recursively check if placing          queen
here leads to a solution.
b)     If placing the queen in [row, column] leads to a solution then return
true.
c)     If placing queen doesn't lead to a solution then unmark this [row,
column] (Backtrack) and go to step (a) to try other rows.
3) If all rows have been tried and nothing worked, return false to trigger
backtracking.
```

Output: The 1 values indicate placements of queens

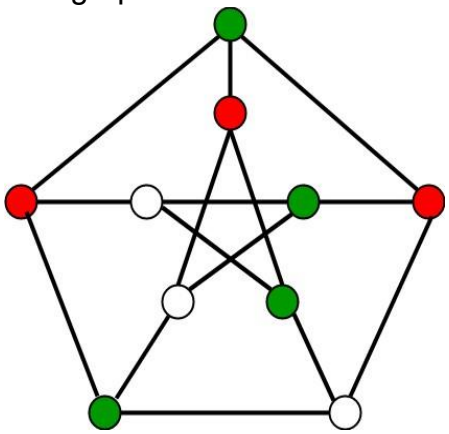
```
0  0  1  0
1  0  0  0
 0  0  0  1
 0  1  0  0
```

Graph Coloring

Given an undirected graph and a number m, determine if the graph can be colored with at most m colors such that no two adjacent vertices of the graph are colored with same color. Here coloring of a graph means assignment of colors to all vertices.

Input:
1) A 2D array graph[V][V] where V is the number of vertices in graph and graph[V][V] is adjacency matrix representation of the graph. A value graph[i][j] is 1 if there is a direct edge from i to j, otherwise graph[i][j] is 0.
2) An integer m which is maximum number of colors that can be used.
Output:
An array color[V] that should have numbers from 1 to m. color[i] should represent the color assigned to the ith vertex. The code should also return false if the graph cannot be colored with m colors.

Following is an example of graph that can be colored with 3 different colors.



Naive Algorithm

Generate all possible configurations of colors and print a configuration that satisfies the given constraints.

```
while there are untried conflagrations
{    generate the next configuration    if no adjacent
vertices are colored with same color
   {       print this
configuration;
   }
}
```

There will be V^m configurations of colors.

Backtracking Algorithm : The idea is to assign colors one by one to different vertices, starting from the vertex 0. Before assigning a color, we check for safety by considering already assigned colors to the adjacent vertices. If we find a color assignment which is safe, we mark the color assignment as part of solution. If we do not a find color due to clashes then we backtrack and return false. Output:

```
Solution Exists: Following are the assigned colors
1  2  3  2
```

Sum of Subsets

Subset sum problem is to find subset of elements that are selected from a given set whose sum adds up to a given number K. We are considering the set contains non-negative values. It is assumed that the input set is unique (no duplicates are presented).

Exhaustive Search Algorithm for Subset Sum

One way to find subsets that sum to K is to consider all possible subsets. A power set contains all those subsets generated from a given set. The size of such a power set is $2^{N.}$ Using exhaustive search we consider all subsets irrespective of whether they

satisfy given constraints or not. Backtracking can be used to make a systematic consideration of the elements to be selected.

Let, S = {S1 …. Sn} be a set of n positive integers, then we have to find a subset whose sum is equal to given positive integer d.It is always convenient to sort the set's elements in ascending order. That is, S1 ≤ S2 ≤…. ≤ Sn

Back Tracking Algorithm:

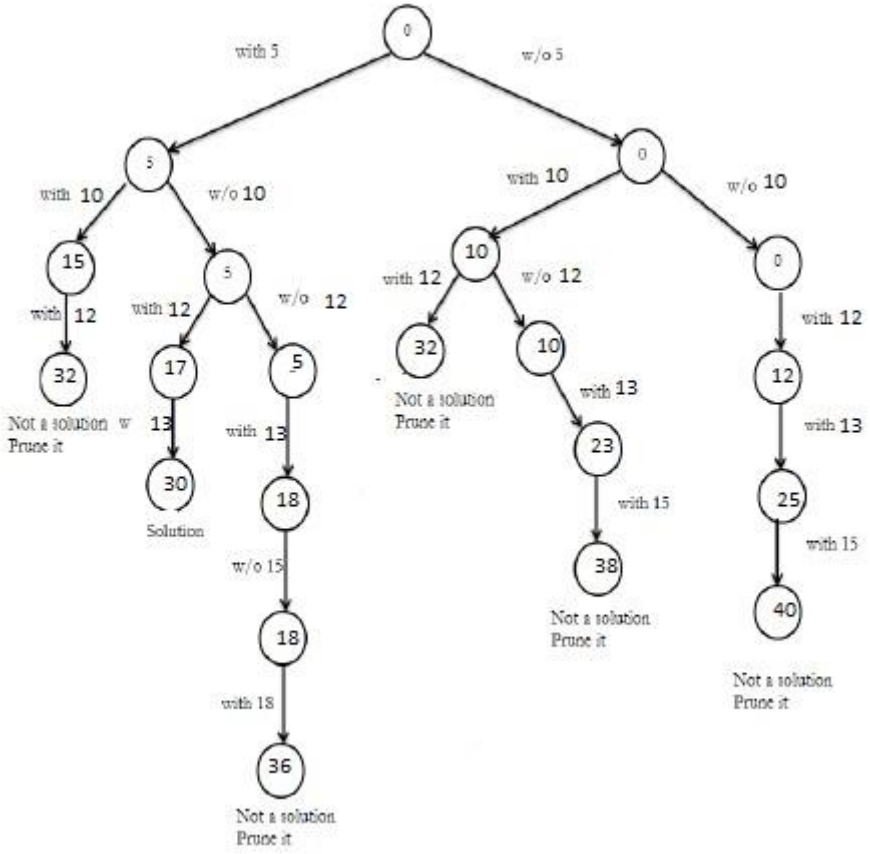
Let, S is a set of elements and m is the expected sum of subsets. Then:

1. Start with an empty set.
2. Add to the subset, the next element from the list.
3. If the subset is having sum m then stop with that subset as solution.
4. If the subset is not feasible or if we have reached the end of the set then backtrack through the subset until we find the most suitable value.
5. If the subset is feasible then repeat step 2.
6. If we have visited all the elements without finding a suitable subset and if no backtracking is possible then stop without solution.

Example: Solve following problem and draw portion of state space tree M=30,W ={5, 10, 12, 13, 15, 18} Solution:

Initially subset = {}	Sum = 0	Description
5	5	Then add next element.
5, 10	15 i.e. 15 < 30	Add next element.
5, 10, 12	27 i.e. 27 < 30	Add next element.
5, 10, 12, 13	40 i.e. 40 < 30	Sum exceeds M = 30. Hence backtrack.
5, 10, 12, 15	42	Sum exceeds M = 30. Hence backtrack.
5, 10, 12, 18	45	Sum exceeds M = 30. Hence backtrack.
5, 12, 13	30	Solution obtained as M = 30

The state space tree is shown as below in figure. {5, 10, 12, 13, 15, 18}


0
with 5
w/o 5
5
0
with 10
w/o 10
with 10
w/o 10
15
5
10
0
with 12
w/o 12
with 12
w/o 12
with 12
32
17
5
32
10
0
Not a solution
Prune it
with 13
w/o 13
with 13
Not a solution
Prune it
12
30
18
23
25
Solution
with 15
with 13
w/o 15
38
40
18
Not a solution
Prune it
Not a solution
Prune it
with 18
36
Not a solution
Prune it

UNIT 5 Part 2

Branch and Bound Topics

:

1. Introduction, General Method
2. Least Cost Search
3. FIFO Branch and Bound
4. LC Branch and Bound
5. 0/1 Knapsack Problem
6. Travelling Salesperson Problem

UNIT 5 Part 2

Branch and Bound

Topics :

1. Introduction, General Method
2. LIFO Branch and Bound Search
3. Least Cost Search
4. Bounding
5. 0/1 Knapsack Problem
6. FIFO Branch and Bound Solution
7. Travelling Salesperson Problem

Introduction:

Branch and Bound refers to all state space search methods in which all children of the E-Node are generated before any other live node becomes the E-Node.

Branch and Bound is the generalization of both graph search strategies, BFS and D-search.

- A BFS like state space search is called as FIFO (First in first out) search as the list of live nodes in a first in first out.
- A D-search like state space search is called as LIFO (last in first out) search as the list of live nodes in a last in first out list.

Live node is a node that has been generated but whose children have not yet been generated. **E-node** is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.

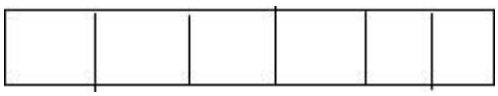
Dead node is a generated anode that is not be expanded or explored any further. All children of a dead node have already been expanded.

Here we will use 3 types of search strategies:

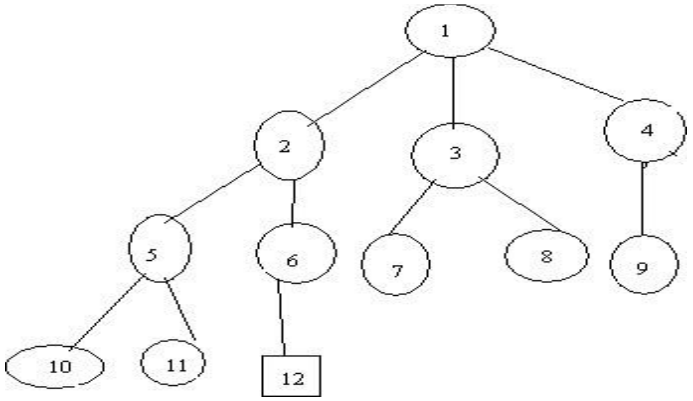
1. FIFO (First In First Out)
2. LIFO (Last In First Out)
3. LC (Least Cost) Search

FIFO Branch and Bound Search:

For this we will use a data structure called Queue. Initially Queue is empty.



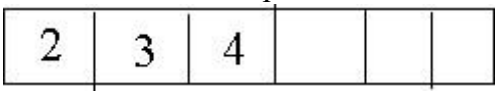
Example:



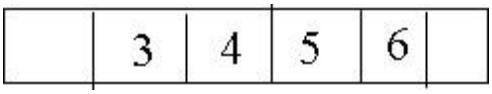
Assume the node 12 is an answer node (solution)

In FIFO search, first we will take E-node as a node 1.

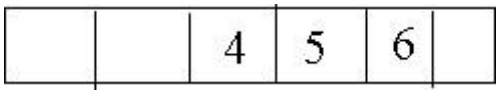
Next we generate the children of node 1. We will place all these live nodes in a queue.



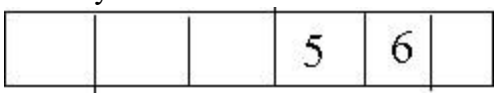
Now we will delete an element from queue, i.e. node 2, next generate children of node 2 and place in this queue.



Next, delete an element from queue and take it as E-node, generate the children of node 3, 7, 8 are children of 3 and these live nodes are killed by bounding functions. So we will not include in the queue.

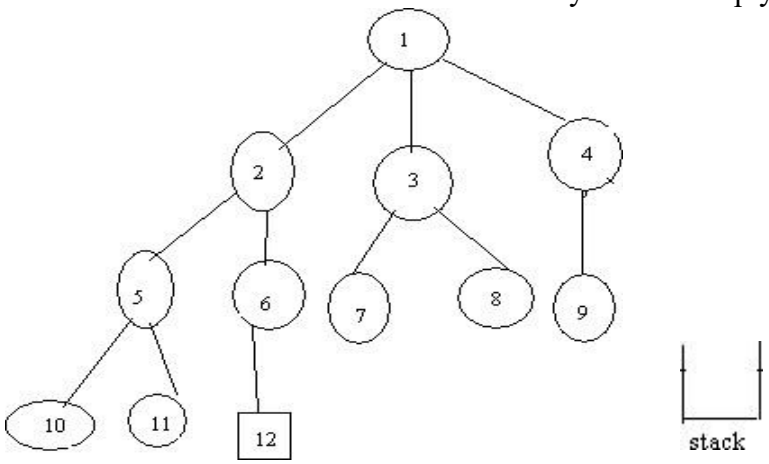


Again delete an element an from queue. Take it as E-node, generate the children of 4. Node 9 is generated and killed by boundary function.



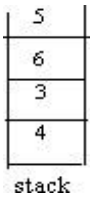
Next, delete an element from queue. Generate children of nodes 5, i.e., nodes 10 and 11 are generated and by boundary function, last node in queue is 6. The child of node 6 is 12 and it satisfies the conditions of the problem, which is the answer node, so search terminates. **LIFO Branch and Bound Search**

For this we will use a data structure called stack. Initially stack is empty. **Example:**

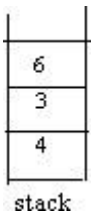


Generate children of node 1 and place these live nodes into stack.

Remove element from stack and generate the children of it, place those nodes into stack. 2 is removed from stack. The children of 2 are 5, 6. The content of stack is,



Again remove an element from stack, i.,e node 5 is removed and nodes generated by 5 are 10, 11 which are killed by bounded function, so we will not place 10, 11 into stack.

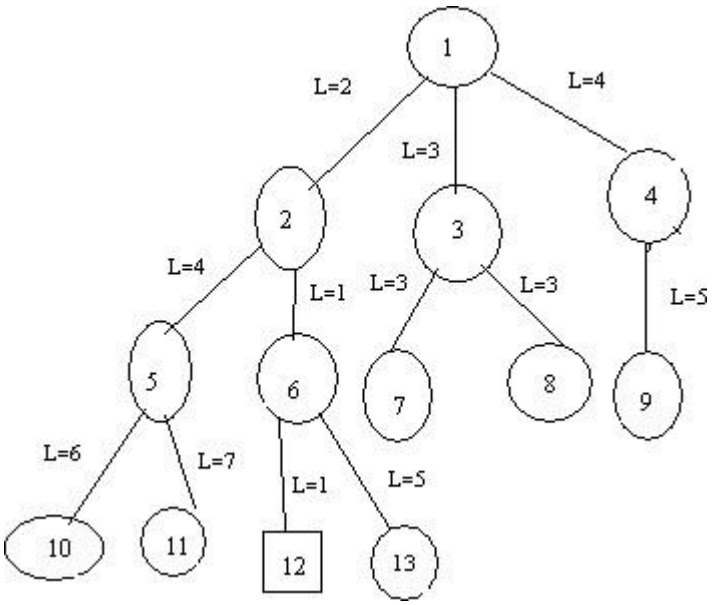


Delete an element from stack, i.,e node 6. Generate child of node 6, i.,e 12, which is the answer node, so search process terminates.

LC (Least Cost) Branch and Bound Search

In both FIFO and LIFO Branch and Bound the selection rules for the next E-node in rigid and blind. The selection rule for the next E-node does not give any preferences to a node that has a very good chance of getting the search to an answer node quickly.

In this we will use ranking function or cost function. We generate the children of E-node, among these live nodes; we select a node which has minimum cost. By using ranking function we will calculate the cost of each node.



Initially we will take node 1 as E-node. Generate children of node 1, the children are 2, 3, 4. By using ranking function we will calculate the cost of 2, 3, 4 nodes is ĉ =2, ĉ =3, ĉ =4 respectively. Now we will select a node which has minimum cost i.,e node 2. For node 2, the children are 5, 6. Between 5 and 6 we will select the node 6 since its cost minimum. Generate children of node 6 i.,e 12 and 13. We will select node 12 since its cost (ĉ =1) is minimum. More over 12 is the answer node. So, we terminate search process.

Control Abstraction for LC-search

Let **t** be a state space tree and c() a cost function for the nodes in t. If x is a node in t, then c(x) is the minimum cost of any answer node in the sub tree with root x. Thus, c(t) is the cost of a minimum-cost answer node in t.

LC search uses ĉ to find an answer node. The algorithm uses two functions

1. Least-cost() 2.

Add_node().

Least-cost() finds a live node with least c(). This node is deleted from the list of live nodes and returned.
Add_node() to delete and add a live node from or to the list of live nodes.
Add_node(x)adds the new live node x to the list of live nodes. The list of live nodes be implemented as a min-heap.

BOUNDING

- A branch and bound method searches a state space tree using any search mechanism in which all the children of the E-node are generated before another node becomes the E-node.
- A good bounding helps to prune (reduce) efficiently the tree, leading to a faster exploration of the solution space. Each time a new answer node is found, the value of upper can be updated.
- Branch and bound algorithms are used for optimization problem where we deal directly only with minimization problems. A maximization problem is easily converted to a minimization problem by changing the sign of the objective function.

APPLICATION: 0/1 KNAPSACK PROBLEM (LCBB)

There are n objects given and capacity of knapsack is M. Select some objects to fill the knapsack in such a way that it should not exceed the capacity of Knapsack and maximum profit can be earned. The Knapsack problem is maximization problem. It means we will always seek for maximum p_1x_1 (where p_1 represents profit of object x_1).

A branch bound technique is used to find solution to the knapsack problem. But we cannot directly apply the branch and bound technique to the knapsack problem. Because the branch bound deals only the minimization problems. We modify the knapsack problem to the minimization problem. The modifies problem is,

$$\text{minimize } -\sum_{i=1}^{n} p_i x_i$$

$$\text{subject to } \sum_{i=1}^{n} w_i x_i \leq m$$

$$x_i = 0 \text{ or } 1, \quad 1 \leq i \leq n$$

```
Algorithm Reduce(p, w, n, m, I1, I2)
// Variables are as described in the discussion.
// p[i]/w[i] ≥ p[i + 1]/w[i + 1], 1 ≤ i < n.
{
    I1 := I2 := ∅;
    q := Lbb(∅, ∅);
    k := largest j such that w[1] + ··· + w[j] < m;
    for i := 1 to k do
    {
        if (Ubb(∅, {i}) < q) then I1 := I1 ∪ {i};
        else if (Lbb(∅, {i}) > q) then q := Lbb(∅, {i});
    }
    for i := k + 1 to n do
    {
        if (Ubb({i}, ∅) < q) then I2 := I2 ∪ {i};
        else if (Lbb({i}, ∅) > q) then q := Lbb({i}, ∅);
    }
}
```

Algorithm: KNAPSACK PROBLEM

Example: Consider the instance M=15, n=4, (p_1, p_2, p_3, p_4) = 10, 10, 12, 18 and (w_1, w_2, w_3, w_4)=(2, 4, 6, 9).

Solution: knapsack problem can be solved by using branch and bound technique. In this problem we will calculate lower bound and upper bound for each node.

Arrange the item profits and weights with respect of profit by weight ratio. After that, place the first item in the knapsack. Remaining weight of knapsack is 15-2=13. Place next item w_2 in knapsack and the remaining weight of knapsack is 13-4=9. Place next item w_3, in knapsack then the remaining weight of knapsack is 9-6=3. No fraction are allowed in calculation of upper bound so w_4, cannot be placed in knapsack.

Profit= p_1+p_2+ p_3,=10+10+12

So, Upper bound=32

To calculate Lower bound we can place w_4 in knapsack since fractions are allowed in calculation of lower bound.

Lower bound=10+10+12+ (3/9*18)=32+6=38

Knapsack is maximization problem but branch bound technique is applicable for only minimization problems. In order to convert maximization problem into minimization problem we have to take negative sign for upper bound and lower bound.

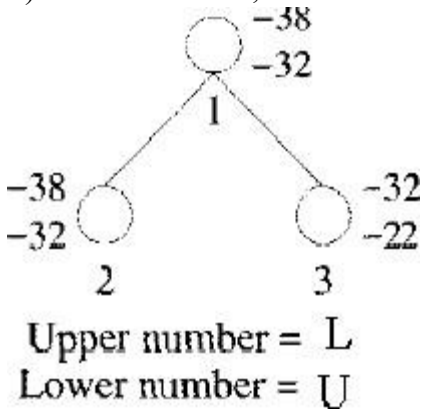
Therefore, upper bound (U) =-32

Lower bound (L)=-38

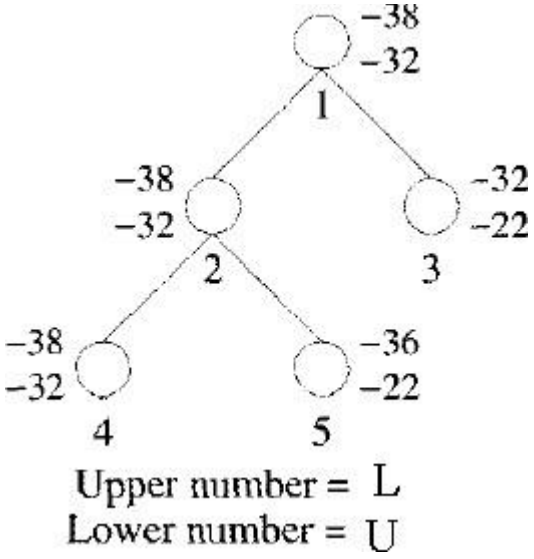
We choose the path, which has minimized difference of upper bound and lower bound. If the difference is equal then we choose the path by comparing upper bounds and we discard node with maximum upper bound.

Now we will calculate upper bound and lower bound for nodes 2, 3

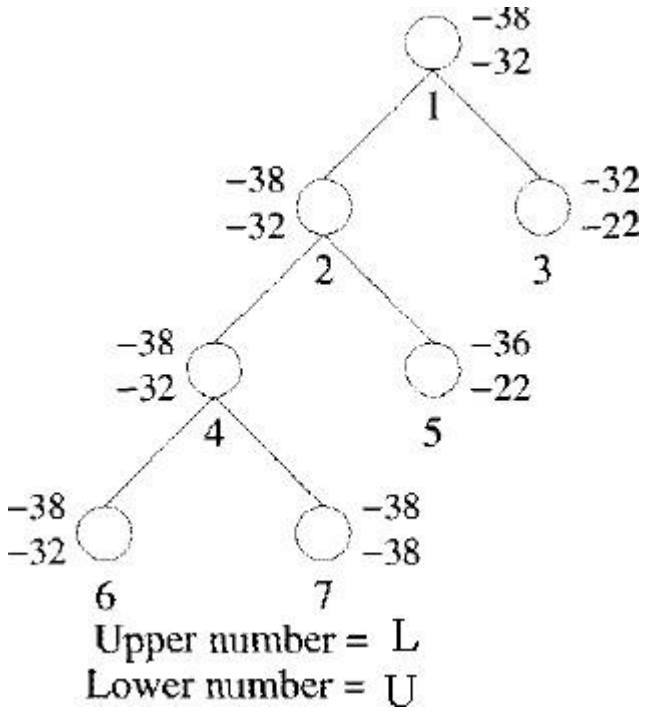
For node 2, x_1=1, means we should place first item in the knapsack. U=10+10+12=32,
make it as -32
L=10+10+12+ (3/9*18) = 32+6=38, we make it as -38
For node 3, x_1=0, means we should not place first item in the knapsack. U=10+12=22,
make it as -22
L=10+12+ (5/9*18) = 10+12+10=32, we make it as -32



Next we will calculate difference of upper bound and lower bound for nodes 2, 3
For node 2, U-L=-32+38=6
For node 3, U-L=-22+32=10
Choose node 2, since it has minimum difference value of 6.



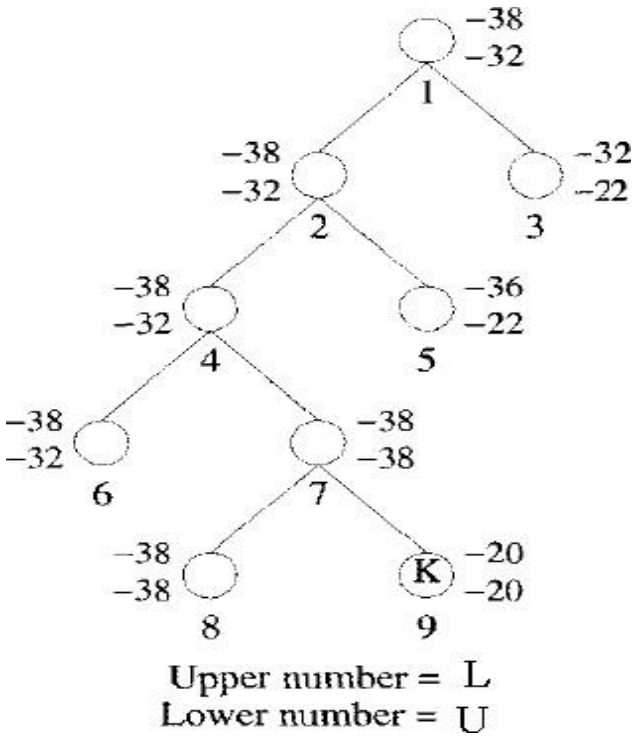
Now we will calculate lower bound and upper bound of node 4 and 5. Calculate difference of lower and upper bound of nodes 4 and 5.
For node 4, U-L=-32+38=6
For node 5, U-L=-22+36=14
Choose node 4, since it has minimum difference value of 6



Now we will calculate lower bound and upper bound of node 6 and 7. Calculate difference of lower and upper bound of nodes 6 and 7. For node 6, U-L=-32+38=6

For node 7, U-L=-38+38=0

Choose node 7, since it has minimum difference value of 0.



Now we will calculate lower bound and upper bound of node 8 and 9. Calculate difference of lower and upper bound of nodes 8 and 9.

For node 8, U-L=-38+38=0

For node 9, U-L=-20+20=0

Here, the difference is same, so compare upper bounds of nodes 8 and 9. Discard the node, which has maximum upper bound. Choose node 8, discard node 9 since, it has maximum upper bound.

Consider the path from 1→2→4→7→8

$X_1=1$

$X_2=1$

$X_3=0$

$X_4=1$

The solution for 0/1 knapsack problem is (x_1, x_2, x_3, x_4)=(1, 1, 0, 1) Maximum profit is:

$\sum pixi=10*1+10*1+12*0+18*1$

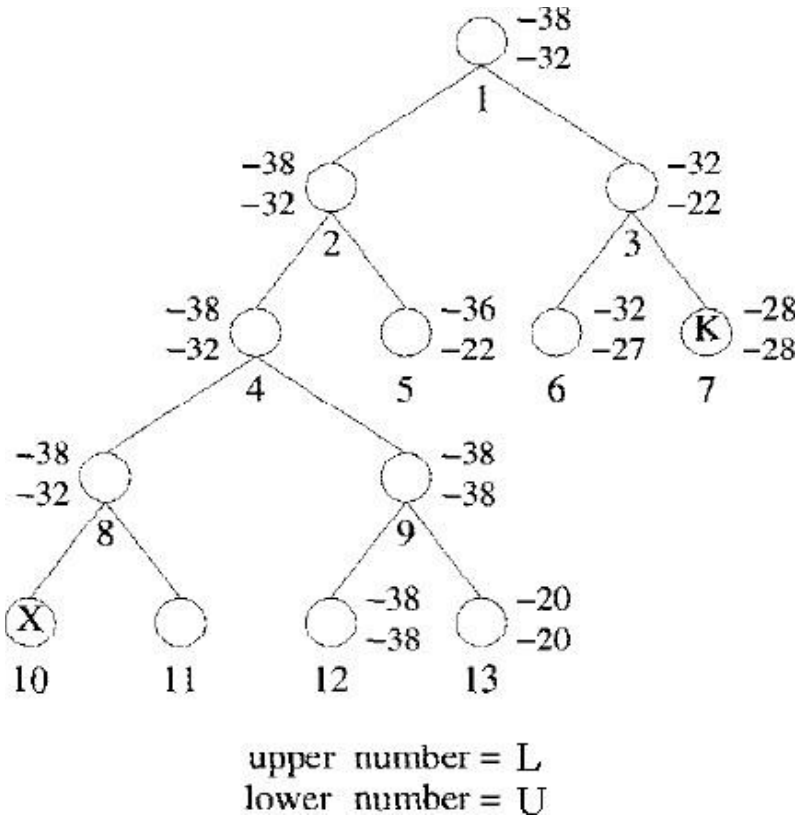
10+10+18=38.

FIFO Branch-and-Bound Solution

Now, let us trace through the FIFOBB algorithm using the same knapsack instance as in above Example. Initially the root node, node 1 of following Figure, is the E-node and the queue of live nodes is empty. Since this is not a solution node, upper is initialized to u(l) = -32. We assume the children of a node are generated left to right. Nodes 2 and 3 are generated and added to the queue (in that order). The value of upper remains unchanged. Node 2 becomes the next E- node. Its children, nodes 4 and 5, are generated and added to the queue.

Node 3, the next-node, is expanded. Its children nodes are generated; Node 6 gets added to the queue. Node 7 is immediately killed as L (7) > upper. Node 4 is expanded next. Nodes 8 and 9 are generated and added to the queue. Then Upper is updated to u(9) = -38, Nodes 5 and 6 are the next two nodes to become B-nodes. Neither is expanded as for each, L > upper. Node 8 is the next E-node. Nodes 10 and 11 are generated; Node 10 is infeasible and so killed. Node 11 has L (11) > upper and so is also killed. Node 9 is expanded next.

When node 12 is generated, 'Upper and ans are updated to -38 and 12 respectively. Node 12 joins the queue of live nodes. Node 13 is killed before it can get onto the queue of live nodes as L (13) > upper. The only remaining live node is node 12. It has no children and the search terminates. The value of upper and the path from node 12 to the root is output. So solution is $X_1=1$, X2=1, X3=0, X4=1.



APPLICATON: TRAVELLING SALES PERSON PROBLEM

Let G = (V', E) be a directed graph defining an instance of the traveling salesperson problem. Let Cij equal the cost of edge (i, j), Cij = ∞ if (i, j) != E, and let IVI = n, without loss of generality, we can assume that every tour starts and ends at vertex 1.

Procedure for solving travelling sales person problem

1. Reduce the given cost matrix. A matrix is reduced if every row and column is reduced. This can be done as follows: **Row Reduction:**
   a) Take the minimum element from first row, subtract it from all elements of first row, next take minimum element from the second row and subtract it from second row. Similarly apply the same procedure for all rows.
   b) Find the sum of elements, which were subtracted from rows.
   c) Apply column reductions for the matrix obtained after row reduction.
   Column Reduction:
   d) Take the minimum element from first column, subtract it from all elements of first column, next take minimum element from the second column and subtract it from second column. Similarly apply the same procedure for all columns.
   e) Find the sum of elements, which were subtracted from columns.
   f) Obtain the cumulative sum of row wise reduction and column wise reduction. Cumulative reduced sum=Row wise reduction sum + Column wise reduction sum. Associate the cumulative reduced sum to the starting state as lower bound and α as upper bound.
2. Calculate the reduced cost matrix for every node.

a) If path (i,j) is considered then change all entries in row **i** and column **j** of A to α. b) Set A(j,1) to α.

c) Apply row reduction and column reduction except for rows and columns containing only α. Let **r** is the total amount subtracted to reduce the matrix.

d) Find ĉ(S)= ĉ(R)+A(i,j)+r.

Repeat step 2 until all nodes are visited.

Example: Find the LC branch and bound solution for the travelling sales person problem whose cost matrix is as follows.

The cost matrix is

∞	20	30	10	11
15	∞	16	4	2
3	5	∞	2	4
19	6	18	∞	3
16	4	7	16	∞

Step 1: Find the reduced cost matrix

Apply now reduction method:

Deduct 10 (which is the minimum) from all values in the 1st row. Deduct 2 (which is the minimum) from all values in the 2nd row.

Deduct 2 (which is the minimum) from all values in the 3rd t row.

Deduct 3 (which is the minimum) from all values in the 4th row.

Deduct 4 (which is the minimum) from all values in the 5th row.

The resulting row wise reduced cost matrix =

∞	10	20	0	1
13	∞	14	2	0
1	3	∞	0	2
16	3	15	∞	
0	12	0	3	
12		∞		

Row wise reduction sum = 10+2+2+3+4=21.

Now apply column reduction for the above matrix:

Deduct 1 (which is the minimum) from all values in the 1st column.

Deduct 3 (which is the minimum) from all values in the 2nd column.

The resulting column wise reduced cost matrix (A) =

∞	10	17	0	1
12	∞	11	2	0
0	3	∞	0	2
15	3	12	∞	
0	11	0	3	
12	∞			

Column wise reduction sum = 1+0+3+0+0=4.

Cumulative reduced sum = row wise reduction + column wise reduction sum. =21+ 4 =25.

This is the cost of a root i.e. node 1, because this is the initially reduced cost matrix.

The lower bound for node is 25 and upper bound is ∞.

Starting from node 1, we can next visit 2, 3, 4 and 5 vertices. So, consider to explore the paths (1, 2), (1,3), (1, 4), (1,5).

The tree organization up to this as follows; Variable **i** indicate the next node to visit.

Step 2:

Now consider the path (1, 2)

Change all entries of row 1 and column 2 of A to ∞ and also set A (2, 1) to ∞.

∞	∞	∞	∞	∞
∞	∞	11	2	0
0	∞	∞	0	2
15	∞	12	∞	0
11	∞	0	12	∞

Apply row and column reduction for the rows and columns whose rows and column are not completely ∞. Then the resultant matrix is

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$$

Row reduction sum = 0 + 0 + 0 + 0 = 0

Column reduction sum = 0 + 0 + 0 + 0= 0

Cumulative reduction(r) = 0 + 0=0

Therefore, as ĉ(S)= ĉ(R)+A(1,2)+r ➔ ĉ(S)= 25 + 10 + 0 = 35.

Now consider the path (1, 3)

Change all entries of row 1 and column 3 of A to ∞ and also set A (3, 1) to ∞.

∞	∞	∞	∞	∞
12	∞	∞	2	0
∞	3	∞	0	2
15	3	∞	∞	0
11	0	∞	12	∞

Apply row and column reduction for the rows and columns whose rows and column are not completely ∞

Then the resultant matrix is =

∞	∞	∞	∞	∞
1	∞	∞	2	0
∞	3	∞	0	2
4	3	∞	∞	0
0	0	∞	12	∞

Row reduction sum = 0

Column reduction sum = 11

Cumulative reduction(r) = 0 +11=11 Therefore,

as ĉ(S)= ĉ(R)+A(1,3)+r

ĉ(S)= 25 + 17 +11 = 53.

Now consider the path (1, 4)

Change all entries of row 1 and column 4 of A to ∞ and also set A(4,1) to ∞.

∞	∞	∞	∞	∞
12	∞	11	∞	0
0	3	∞	∞	2
∞	3	12	∞	0
11	0	0	∞	∞

Apply row and column reduction for the rows and columns whose rows and column are not completely ∞

Then the resultant matrix is =

∞	∞	∞	∞	∞
12	∞	11	∞	0
0	3	∞	∞	2
∞	3	12	∞	0
11	0	0	∞	∞

Row reduction sum = 0

Column reduction sum = 0

Cumulative reduction(r) = 0 +0=0

Therefore, as ĉ(S)= ĉ(R)+A(1,4)+r ĉ(S)=

25 + 0 +0 = 25.

Now Consider the path (1, 5)

Change all entries of row 1 and column 5 of A to ∞ and also set A(5,1) to ∞.

∞	∞	∞	∞	∞
12	∞	11	2	∞
0	3	∞	0	∞
15	3	12	∞	∞
∞	0	0	12	∞

Apply row and column reduction for the rows and columns whose rows and column are not completely ∞

	∞	∞	∞	∞	∞
		∞			∞
=	0	3	∞	0	∞

10

9 0

Then the resultant matrix is

12	0	9	∞	∞
∞	0	0	12	∞

Row reduction sum = 5

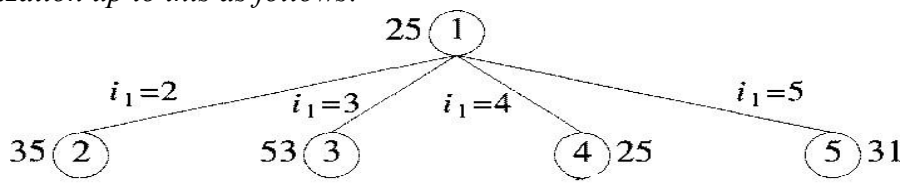
Column reduction sum = 0

Cumulative reduction(r) = 5 +0=0

Therefore, as ĉ(S)= ĉ(R)+A(1,5)+r

ĉ(S)= 25 + 1 +5 = 31.

The tree organization up to this as follows:



Numbers outside the node are $\hat{c}$ values

The cost of the between (1, 2) = 35, (1, 3) = 53, (1, 4) = 25, (1, 5) = 31. The cost of the path between (1, 4) is minimum. Hence the matrix obtained for path (1, 4) is considered as reduced cost matrix.

$$A = \begin{matrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{matrix}$$

The new possible paths are (4, 2), (4, 3) and (4, 5).

Now consider the path (4, 2)

Change all entries of row 4 and column 2 of A to ∞ and also set A(2,1) to ∞.

$$\begin{matrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{matrix}$$

Apply row and column reduction for the rows and columns whose rows and column are not completely ∞

Then the resultant matrix is =

$$\begin{matrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{matrix}$$

Row reduction sum = 0

Column reduction sum = 0

Cumulative reduction(r) = 0 +0=0

Therefore, as ĉ(S)= ĉ(R)+A(4,2)+r

ĉ(S)= 25 + 3 +0 = 28.

Now consider the path (4, 3)

Change all entries of row 4 and column 3 of A to ∞ and also set A(3,1) to ∞.

$$\begin{matrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ \infty & 3 & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & 0 & \infty & \infty & \infty \end{matrix}$$

Apply row and column reduction for the rows and columns whose rows and column are not completely ∞

Then the resultant matrix is =

∞	∞	∞	∞	∞
1	∞	∞	∞	0
∞	1	∞	∞	0
∞	∞	∞	∞	∞
0	0	∞	∞	∞

Row reduction sum = 2

Column reduction sum = 11

Cumulative reduction(r) = 2 +11=13

Therefore, as ĉ(S)= ĉ(R)+A(4,3)+r ĉ(S)=

25 + 12 +13 = 50.

.Now consider the path (4, 5)

Change all entries of row 4 and column 5 of A to ∞ and also set A(5,1) to ∞.

∞	∞	∞	∞	∞
12	∞	11	∞	∞
0	3	∞	∞	∞
∞	∞	∞	∞	∞
∞	0	0	∞	∞

Apply row and column reduction for the rows and columns whose rows and column are not completely ∞

Then the resultant matrix is =

∞	∞	∞	∞	∞
1	∞	0	∞	∞
0	3	∞	∞	∞
∞	∞	∞	∞	∞
∞	0	0	∞	∞

Row reduction sum =11

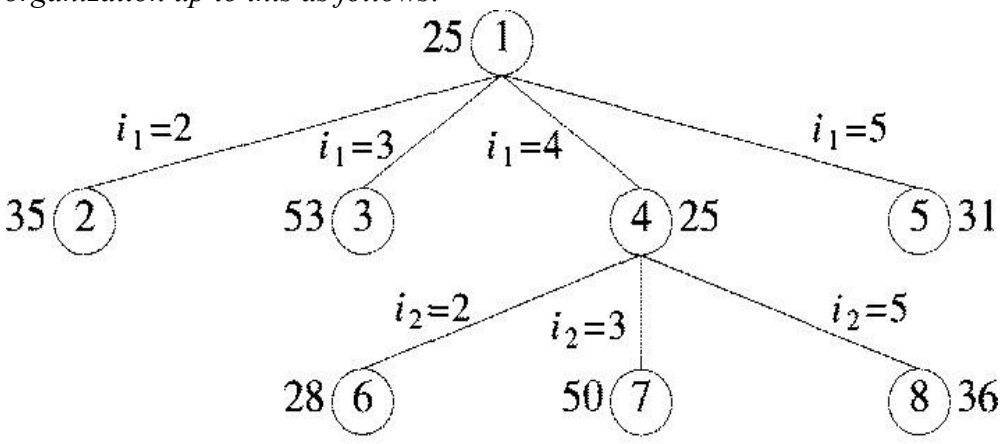
Column reduction sum = 0

Cumulative reduction(r) = 11 +0=11

Therefore, as ĉ(S)= ĉ(R)+A(4,5)+r

ĉ(S)= 25 + 0 +11 = 36.

The tree organization up to this as follows:



Numbers outside the node are $\hat{c}$ values

The cost of the between (4, 2) = 28, (4, 3) = 50, (4, 5) = 36. The cost of the path between (4, 2) is minimum. Hence the matrix obtained for path (4, 2) is considered as reduced cost matrix.

$$A = \begin{matrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{matrix}$$

(2, 3) and (2, 5).

2

The new possible paths are

Now Consider the path (2, 3):

Change all entries of row 2 and column 3 of A to ∞ and also set A(3,1) to ∞.

$$\begin{matrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & \infty & \infty & \infty \end{matrix}$$

Apply row and column reduction for the rows and columns whose rows and column are not completely ∞

Then the resultant matrix is =

$$\begin{matrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{matrix}$$

Row reduction sum =13
Column reduction sum = 0
Cumulative reduction(r) = 13 +0=13
Therefore, as ĉ(S)= ĉ(R)+A(2,3)+r
ĉ(S)= 28 + 11 +13 = 52.

Now Consider the path (2, 5):

Change all entries of row 2 and column 5 of A to ∞ and also set A(5,1) to ∞.

$$\begin{matrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{matrix}$$

Apply row and column reduction for the rows and columns whose rows and column are not completely ∞

$$\begin{matrix} \infty & \infty & \infty & \infty & \infty \end{matrix}$$

Then the resultant matrix is =

∞	∞	∞	∞	∞
0	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	0	∞	∞

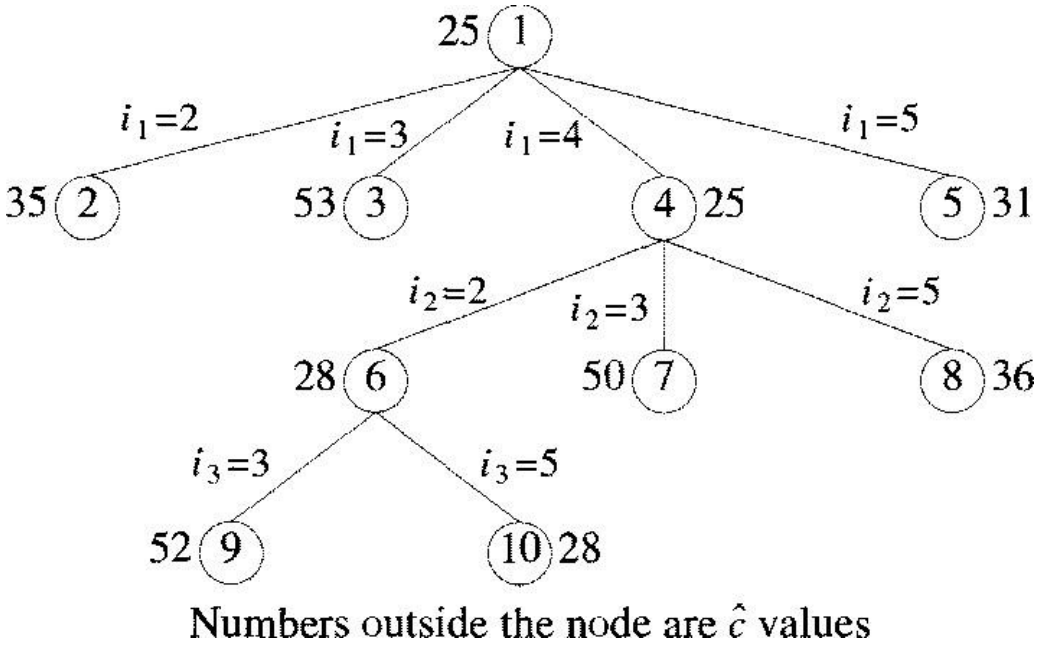
Row reduction sum =0

Column reduction sum = 0

Cumulative reduction(r) = 0 +0=0

Therefore, as ĉ(S)= ĉ(R)+A(2,5)+r ➔ ĉ(S)= 28 + 0 +0 = 28.

The tree organization up to this as follows:



Numbers outside the node are $\hat{c}$ values

The cost of the between (2, 3) = 52 and (2, 5) = 28. The cost of the path between (2, 5) is minimum. Hence the matrix obtained for path (2, 5) is considered as reduced cost matrix.

A =

∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
0	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	0	∞	∞

The new possible path is (5, 3).

Now c***onsider the path (5, 3):***

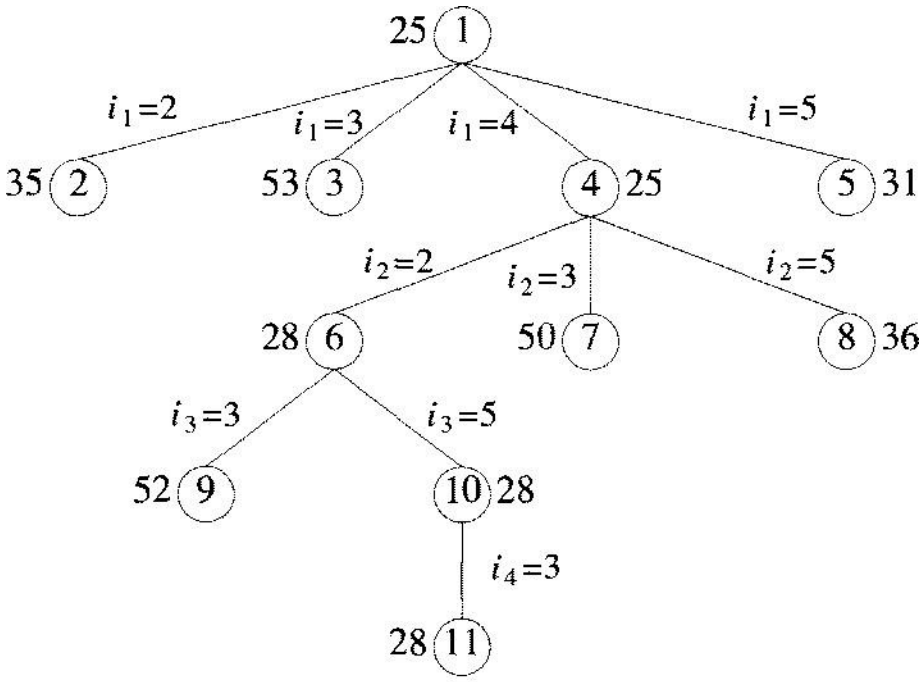
Change all entries of row 5 and column 3 of A to ∞ and also set A(3,1) to ∞. Apply row and column reduction for the rows and columns whose rows and column are not completely ∞

Then the resultant matrix is =

∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞

∞	∞	∞	∞	∞
∞	∞	∞	∞	∞

Row reduction sum =0
Column reduction sum = 0
Cumulative reduction(r) = 0 +0=0
Therefore, as ĉ(S)= ĉ(R)+A(5,3)+r ĉ(S)=
28 + 0 +0 = 28.
The path travelling sales person problem is:
1□4□2□5□3□1:
The minimum cost of the path is: 10+2+6+7+3=28.
The overall tree organization is as follows:



Numbers outside the node are $\hat{c}$ values

UNIT-5

5. Explain Cook's Theorem

The Cook–Levin Theorem: SAT is NP-Complete

The **Cook–Levin theorem**, often called **Cook's theorem**, is a fundamental concept in computer science, particularly in computational complexity theory. The theorem states that the **Boolean satisfiability problem** (SAT) is **NP-complete**.

NP-complete means two things:

1. SAT is a problem in **NP**. NP (nondeterministic polynomial time) is a class of problems for which any solution, once found, can be quickly checked to see if it's correct.
2. Any problem in NP can be transformed or "reduced" to SAT in **polynomial time** (meaning the time taken grows reasonably with input size, not exponentially).

What is the SAT Problem?

The **Boolean satisfiability problem (SAT)** is about finding out if we can assign true or false values to variables in a Boolean expression so that the expression becomes true.

example of a Boolean expression:
F=(x1Vx2barVx3)Λ(x1Vx2)

- Here, x1, x2,x3 are **Boolean variables** (each can be true or false).
- The symbol V means **OR** (the expression is true if at least one part is true).

- The symbol ∧ means **AND** (the expression is only true if all parts are true).
- The symbol x2bar means **NOT** x2x_2x2 (the opposite of whatever x2 is).

Important Terminology in Boolean Expressions

- **Boolean Variable**: A variable like xxx that can be either true or false.
- **Literal**: A variable (like x) or its negation (like xbar).
    - If it's not negated, it's a **positive literal**.
    - If it's negated, it's a **negative literal**.
- **Clause**: A group of literals connected by **OR** (∨), such as (x1∨x2 ∨x3bar).
- **Expression**: Multiple clauses combined using **AND** (∧).
- **Conjunctive Normal Form (CNF)**: An expression in CNF has clauses joined by AND, where each clause is a series of literals joined by OR.
    - Example of CNF: (x1∨x2)∧(x1bar∨x3).
- **3-CNF**: A CNF expression where each clause has exactly three literals.

Why SAT is Important and Difficult

SAT is significant because it is one of the **hardest problems in computer science**. For SAT, there is no fast algorithm that works in polynomial time, and the only general approach is a brute-force method. The brute-force approach tries every possible assignment of true or false to each variable, which quickly becomes impractical for larger expressions since

there are 2^n possible combinations for n variables (doubling with each additional variable).

Contributions of Stephen Cook and Leonid Levin

- **Stephen Cook** introduced SAT as an NP-complete problem in his 1971 paper, *"The Complexity of Theorem Proving Procedures."* He showed how SAT could be used to understand the complexity of proving mathematical theorems by converting or "reducing" other problems to SAT. He proved that other forms of SAT, like **Circuit-SAT** and **3-CNF-SAT**, are also as difficult as SAT.
- **Leonid Levin**, a mathematician working independently in the Soviet Union, reached similar conclusions around the same time. Together, Cook and Levin's work laid the foundation for the field of NP-completeness.
- **Richard Karp**, another computer scientist, later used their work to show that 21 well-known problems could also be reduced to SAT. These problems included challenges like finding a Hamiltonian tour (a specific kind of path in a graph), a vertex cover, and a clique. Karp's work expanded the importance of SAT, making it clear that solving SAT efficiently would mean that many other hard problems could also be solved efficiently.

Types of SAT Problems

There are different versions of SAT, each with specific restrictions:

1. **Circuit-SAT**: Given a circuit with logic gates (like AND, OR, and NOT), Circuit-SAT asks if there's a way to assign values to the inputs so that the circuit outputs true. This problem is challenging because for n inputs, you may need to check 2^n combinations.
2. **CNF-SAT**: This is a restricted form of SAT where the Boolean expression is in CNF form (clauses joined by ANDs, with each clause

a series of literals joined by ORs). Like with general SAT, CNF-SAT is about finding an assignment that makes the expression true.

3. **3-CNF-SAT (3-SAT)**: A further restriction where each clause in the CNF form has exactly three literals. This version of SAT is still NP-complete and is commonly studied because it's both structured and difficult, representing the complexity of SAT in a manageable form.

The **Cook–Levin theorem** was a breakthrough that established SAT as an NP-complete problem. Cook and Levin proved that solving SAT quickly (in polynomial time) would mean that every other NP problem could also be solved quickly. This realization has had a major impact on fields like cryptography, optimization, and algorithms, as SAT continues to be a central problem for understanding computational difficulty.

6. Explain Satisfiability

The **2-Satisfiability (2-SAT) Problem** is a simpler form of the general Boolean Satisfiability Problem (SAT), specifically where each clause in a Boolean formula has exactly two terms or variables.

1. What is the Satisfiability (SAT) Problem?

- In the SAT problem, we have a Boolean formula made up of variables (like A and B) that can be either **TRUE** or **FALSE**.
- The goal is to find out if there's a way to assign values to these variables (TRUE or FALSE) so that the whole formula evaluates to TRUE.

- **Satisfiable**: If there's a way to make the formula TRUE, the formula is called "satisfiable."
- **Unsatisfiable**: If it's impossible to make the formula TRUE with any assignment, the formula is "unsatisfiable."

For example:

- F=*A*∧*Bbar* is satisfiable because
  setting A=TRUE and B=FALSE makes F=TRUE.
- G=A∧Abar is unsatisfiable because *A* cannot be both TRUE and FALSE at the same time.

2. What is the 2-SAT Problem?

- The 2-SAT problem is a special case of SAT where each clause has exactly **two variables**.
- Each clause is a pair of variables or their negations combined with OR (e.g., A∨B or Abar∨B).
- The goal of 2-SAT is to figure out if there's a way to assign TRUE or FALSE values to all the variables to make the whole formula TRUE.

Example:

- Suppose we have a formula F=$(x_1 \vee x_2) \wedge (x_2 \vee x_{1bar}) \wedge (x_{1bar} \vee x_{2bar})$.
- We want to see if we can assign values to x1 and x2 so that F is TRUE.

3. Understanding Conjunctive Normal Form (CNF)

- **CNF (Conjunctive Normal Form)**: A formula is in CNF if it's a series of clauses joined by ANDs (meaning each clause has to be TRUE).
- **2-CNF**: In the 2-SAT problem, each clause has only two literals (like$x_1 \vee x_2$ or x2∨x1bar).

4. Approach to Solving 2-SAT with Implication Graphs

To solve 2-SAT, we can turn the formula into something called an **Implication Graph**. This graph helps us see how the values of variables depend on each other. Here's how we do it:

- **Implications in a Clause**:
  - Each clause like (A∨B)can be rewritten as implications.
    - If A is FALSE, *B* must be TRUE: Abar⇒*B*.
    - If B is FALSE, A must be TRUE: Bbar⇒*A*.
- **Building the Implication Graph**:
  - For each clause (A∨B), we add two edges in a graph:
    - An edge from A*bar* to *B*.
    - An edge from Bbar to *A*.
  - Each variable and its negation are nodes in the graph.

5. Using the Implication Graph to Find Satisfiability

- **Cycles and Contradictions**:
  - If there's a path from a variable to its own negation (e.g., X→Xbar and Xbar→*X*), this means that the formula is unsatisfiable because X would have to be both TRUE and FALSE.
- **Strongly Connected Components (SCCs)**:
  - To identify these contradictions, we use an algorithm to find **Strongly Connected Components (SCCs)** in the graph.
  - If any SCC contains both a variable X and its negation Xbar, then the formula is unsatisfiable.

6. Solving 2-SAT Efficiently

- Use an algorithm like **Tarjan's Algorithm** to find SCCs in linear time.
- Check each SCC:
    - If any SCC has both *X* and *Xbar*, the formula is unsatisfiable.
    - Otherwise, it's satisfiable, and we can find a solution by assigning values to the variables based on the SCC structure.

Example Walkthrough

Consider a formula:

- $F=(x1 \vee x2) \wedge (x2 \vee \bar{x1}) \wedge (\bar{x1} \vee \bar{x2})$

1. **Convert to Implication Graph**:
    - From $(x1 \vee x2)$: $\bar{x1} \rightarrow x2$ and $\bar{x2} \rightarrow x1$.
    - From $(x2 \vee \bar{x1})$: $\bar{x2} \rightarrow \bar{x1}$ and $x1 \rightarrow x2$.
    - From $(\bar{x1} \vee \bar{x2})$: $x1 \rightarrow \bar{x2}$ and $x2 \rightarrow \bar{x1}$.
2. **Identify SCCs**:
    - Using an SCC algorithm, check for cycles. Here, if there's no SCC containing both a variable and its negation, it's satisfiable.

The 2-SAT problem allows us to determine if there's a way to make a simple Boolean formula true. We use **implication graphs** to convert each clause into implications and analyze the graph's structure to find contradictions. If we don't find any contradictions, the formula is satisfiable, and we can assign values to variables that satisfy it. This approach makes 2-SAT solvable in **polynomial time**, even though the general SAT problem is much more complex.

7.Explain NP Hard and NP Complete Problems

1. NP Problems (Nondeterministic Polynomial Time)

- Definition: NP stands for Nondeterministic Polynomial Time. Problems in this category have a unique property: the solution may be hard to find, but once we have a potential solution, we can verify it quickly (in polynomial time). The reason they're called "nondeterministic" is that, theoretically, we could use a machine that guesses solutions and checks them instantly to find a correct answer.
- Verification Process: Suppose we have a solution to an NP problem. The verification process would involve checking whether this solution actually satisfies the problem requirements. If it does, we can confirm it as correct, and this checking process doesn't take too long (it can be done in polynomial time).
- NP problems may have many possible solutions, but not all are correct. If someone provides a correct answer, we can quickly verify it, even if finding the answer initially is very challenging.
- Examples:
    - Subset Sum Problem: Imagine a set of numbers and a target number. The question is: Is there a subset of these numbers that adds up exactly to the target? Without a solution, finding one can be hard. But if someone gives us a subset, we can just add the numbers quickly to see if it matches the target.
    - Traveling Salesman Problem (Decision Version): Given several cities and distances between each pair of cities, the problem asks if there's a route that visits each city exactly once with a

total distance under a specified value. If a specific route is given, we can quickly calculate its distance to see if it meets the requirement. Finding such a route without any hints, however, is very complex.

2. NP-Hard Problems

- Definition: NP-Hard problems are at least as difficult as the hardest problems in NP. A problem is considered NP-Hard if any problem in NP can be transformed or "reduced" to it in polynomial time. Essentially, if we could solve this NP-Hard problem quickly, then we could also solve all NP problems quickly.
- Characteristics:
    - NP-Hard problems might not be in NP. This means that while they're at least as hard as NP problems, they don't have to be easily verifiable in polynomial time. They may require resources or time that grows exponentially with the problem size, making them potentially impossible to solve efficiently.
    - NP-Hard includes both decision problems (yes/no questions) and optimization problems (problems looking for the best solution), as well as other problems that might not fit neatly in NP.
- Implications: Solving an NP-Hard problem efficiently would revolutionize problem-solving in computer science because we could then tackle any NP problem in polynomial time as well. However, NP-Hard problems can often be beyond reach for fast, exact solutions.
- Examples:

- Hamiltonian Cycle Problem: This asks whether there is a cycle that visits every node in a graph exactly once and then returns to the starting point. Finding such a cycle is complex, and because it's NP-Hard, finding an efficient solution for it would mean we could solve all NP problems efficiently as well.
- Optimization Problem: Consider a variant of the Traveling Salesman Problem, where we don't just want to find a feasible route but the shortest possible route. This optimization version is NP-Hard because it's challenging to find the most optimal solution in a reasonable amount of time.
- Halting Problem: This is a famous theoretical problem in computer science that asks if we can determine whether any given computer program will eventually stop running or will run forever. This problem is NP-Hard but not in NP, as we can't even check a solution for it in polynomial time.

3. NP-Complete Problems

- Definition: NP-Complete problems represent the intersection of NP and NP-Hard. This means they are:
    1. In NP: Their solutions can be verified in polynomial time.
    2. NP-Hard: They are as hard as any problem in NP, meaning that any NP problem can be converted into this NP-Complete problem in polynomial time.
- Why NP-Complete Problems are Important: NP-Complete problems are the most challenging problems within the NP class. If someone finds a fast (polynomial-time) solution to any NP-Complete problem, it would prove that all NP problems can be solved in

polynomial time. This is because every NP problem can be transformed into an NP-Complete problem, so solving any one NP-Complete problem efficiently would mean we could efficiently solve all NP problems.

- Implications: Solving an NP-Complete problem in polynomial time would be a breakthrough in computer science, proving that P = NP (i.e., all problems that can be verified in polynomial time can also be solved in polynomial time). However, no such solution has been found, and it's an open question whether this is even possible.
- Examples:
    - 3-SAT (3-Satisfiability Problem): Given a logical expression in which each clause has exactly three literals (variables or their negations), can we assign truth values to the variables so that the entire expression is true? For example, if the expression is (A or B or C) and (not A or not B or D), is there a way to assign values to A, B, C, and D that makes this true? Checking a solution is easy, but finding one is difficult.
    - Knapsack Problem (Decision Version): Given items with specific weights and values, can we select a subset of items to fit within a maximum weight limit while achieving at least a certain value? If a subset is provided, we can quickly verify whether it meets the weight and value requirements, but finding the subset is challenging.
    - Graph Coloring (Decision Version): Given a graph, can we assign colors to each node such that no two adjacent nodes have the same color, using only a specified number of colors? Verifying a coloring is easy, but finding a valid coloring is hard.

-If we solve an **NP-Complete** problem in polynomial time, all **NP** problems would become solvable in polynomial time.

-NP-Hard problems are more general and can be harder than **NP** problems, and solving them efficiently isn't always possible with current methods.

11. Explain Sum of Subsets problem

The *Sum of Subsets* problem, also called the *Subset Sum Problem*, is a well-known combinatorial problem in algorithm design and analysis. In this problem, the goal is to find all possible subsets of a given set of positive integers that add up to a specific target sum. This problem is often used in fields like resource allocation, cryptography, and clustering due to its complexity and practical applications.

Problem Definition

The Sum of Subsets problem is defined as follows:

- **Input**: You have a set S={s1,s2,...,sn} containing *n* positive integers and a target sum *T*.
- **Output**: You need to find all subsets of S where the sum of the elements is exactly equal to *T*.

For example, suppose you have the set S={3,34,4,12,5,2} and the target sum T=9. The subsets of S*S* that add up to T are {3,4,2} and {5,4}.

Solution Approaches

1. Recursive Backtracking

Algorithm: In backtracking, we explore each subset by making two choices for each element: either include it in the subset or exclude it. If a subset's sum matches *T*, we print or store that subset. If the subset's sum exceeds *T*, we abandon further exploration of that subset (i.e., "backtrack").

- **Complexity**: The backtracking method has a time complexity of O(2^n), as it checks all possible subsets of the set. The worst case is when it has to explore every subset of the set, making this an exponential-time algorithm.
- **Implementation**: This approach can be implemented using a recursive function that explores each subset by adding or skipping elements in the set. When the subset sum reaches *T*, it prints the subset and backtracks.

2. Dynamic Programming (DP)

Algorithm: The dynamic programming approach optimizes the process when only one solution or a count of possible solutions is needed. It builds a boolean table, dp[i][j], where i represents the elements in the set considered so far, and j represents a possible subset sum up to the target *T*. Each cell in this table indicates whether a subset with sum *j* can be formed using the first *i* elements.

- **Complexity**: This approach has a time complexity of O(n·T), where n is the number of elements in the set and *T* is the target

sum. It's more efficient than backtracking when determining if there is at least one subset sum that matches T but does not generate all possible subsets that sum to T.

- **Limitations**: The DP approach is effective for deciding whether there exists any subset that sums to *T* or counting such subsets, but it does not return the actual subsets themselves. Additional modifications are needed to extract each subset.

3. Branch and Bound

Algorithm: Branch and bound is a modified form of the backtracking method. It uses a decision tree to explore all subsets, where each node in the tree represents including or excluding an element in the subset. Branches of the tree are pruned (cut off) if they exceed the target sum *T*, making this approach faster than regular backtracking in many cases.

- **Complexity**: The time complexity for branch and bound varies, depending on how effectively branches are pruned. In the best case, if many subsets can be discarded early, this method can be faster than O(2^n), though it is still exponential in the worst case.
- **Implementation**: Using a priority queue or custom bounds can help choose promising branches to explore first. Nodes are pruned when it's clear they cannot lead to a solution (e.g., if the subset sum exceeds *T*).

Optimization and Analysis

- **Time Complexity**: The backtracking approach and branch and bound both have exponential complexity O(2^n) in the worst case due to the combinatorial number of subsets. The dynamic programming approach reduces this to O(n·T), but only applies

when finding the presence or number of subsets without listing them.

- **Space Complexity**: Recursive methods like backtracking and branch and bound use O(n) space for recursion stacks or tree nodes. The DP approach requires O(n·T) space to store the table used for subset-sum validation.

Applications

- **Resource Allocation**: Helps in deciding optimal allocations when resources have a fixed quantity or target.
- **Cryptography**: Useful in situations requiring combinations or partitioning of sums, such as breaking down values in secure communication.
- **Subset Partitioning**: Applies to tasks like clustering, scheduling, or bin packing, where elements need to be grouped based on specific sum targets.

Simple Backtracking Algorithm (Pseudocode)

```
function subsetSum(arr, target, partial=[]):
    if sum(partial) == target:
        print("Subset found:", partial)
        return
    if sum(partial) > target:
        return

    for i in range(len(arr)):
        n = arr[i]
```

```
remaining = arr[i+1:]
subsetSum(remaining, target, partial + [n])
```

1. Recursively builds up subsets by including each element in turn.
2. Prints each subset if the sum matches T*T*.
3. Stops exploring further when the sum exceeds T*T*.

12. Explain Clique Decision Problem(CDP) and Chromatic Number Decision Problem (CNDP).

These problems are both related to graph theory, which is a way of studying relationships between objects by representing them as "graphs." A graph is a set of *vertices* (points) connected by *edges* (lines).

Clique Decision Problem (CDP)

The Clique Decision Problem is about finding a tightly connected subset of points (called a "clique") in a graph.

- **Problem Statement**: Imagine you have a graph G=(V,E), where V is a set of vertices (points) and E*E* is a set of edges (connections between points). You're given an integer *k*, and you need to figure out if there's a group of *k* vertices in the graph where every pair of vertices in this group is connected to each other. This group is called a *k-clique*.
- **Definition**: A *clique* is a group of vertices that are all connected to each other. If there are 5 vertices, and each vertex is connected to

every other vertex, it's called a 5-clique. This is like a very close-knit friend group where everyone is directly friends with everyone else.

- **Decision Problem**: Given a graph and an integer *k*, we simply ask, "Does this graph have a group of *k* vertices where every two vertices are connected?" If yes, the answer is "yes," and if not, it's "no."
- **Why It's Hard (NP-Complete)**: The Clique Decision Problem is classified as an NP-complete problem. This means there's no known fast way to solve it for all graphs, especially as the size of the graph grows. In simple terms, for large graphs, there's no quick method to find cliques, and checking all possible groups takes too much time as the graph size increases.

  Example: Imagine a graph with 4 people: V={A,B,C,D}, and connections E={(A,B),(B,C),(C,D),(D,A),(A,C)}. Suppose k=3. We need to check if there are three people who are all connected (friends with each other). In this example, {A,B,C} forms a 3-clique because each person is connected to the others.

  Chromatic Number Decision Problem (CNDP)

  The Chromatic Number Decision Problem deals with coloring the vertices of a graph in such a way that no two connected vertices have the same color.

- **Problem Statement**: Given a graph G=(V,E) and an integer *k*, the question is whether you can color the vertices of the graph using k or fewer colors in a way that no two connected vertices share the same color.
- **Definition**: The minimum number of colors needed to color a graph this way is called the *chromatic number*, denoted χ(G). Each color represents a different "category" or "group," and each connected

vertex must be in a different group. Think of it like trying to assign teams where each person connected to another is on a different team.

- **Decision Problem**: Here, we are simply asking, “Can I color this graph using *k* colors or fewer?” If the answer is yes, then the answer to CNDP is "yes"; otherwise, it’s "no."
- **Why It’s Hard (NP-Complete)**: Like CDP, CNDP is also NP-complete. For large graphs, checking all possible ways to color them without breaking the rule (no two connected vertices should share the same color) quickly becomes computationally unfeasible.

  Example: Consider a graph with 4 vertices V={A,B,C,D} and edges E={(A,B),(B,C),(C,D),(D,A)}. Suppose k=2. We’re asked if it’s possible to color this graph using only two colors such that no connected vertices share the same color. Here, you could color vertices A and C with one color, and B and D with the other. This solution is possible, so for this graph, the answer is "yes."

Both the **Clique Decision Problem** and **Chromatic Number Decision Problem** play a central role in understanding graph properties and serve as classic examples of hard computational problems.

3.Write about 4 Queens problem in detail.

Introduction

The 4 Queens Problem is a classic example of a constraint satisfaction problem in computer science, specifically in the field of combinatorial optimization and backtracking algorithms. The

challenge is to place four queens on a standard 8x8 chessboard such that no two queens threaten each other. In chess, a queen can move any number of squares vertically, horizontally, or diagonally, making this problem a fascinating exploration of permutations and combinations.

Detailed Explanation

The goal of the 4 Queens Problem is to find all possible configurations where four queens can be positioned on the board without any queen being in a position to attack another. This requires ensuring that no two queens share the same row, column, or diagonal.

To solve this problem, we can use a backtracking algorithm, which incrementally builds candidates for solutions and abandons those candidates ("backtracks") as soon as it determines that they cannot lead to a valid solution.

Key Concepts:

1. **Chessboard Representation**: We represent the chessboard as a 2D array or a 1D array. Each index represents a row, and the value at that index represents the column position of a queen in that row.
2. **Constraints**:
    - No two queens can occupy the same row (which is inherently handled by placing one queen per row).
    - No two queens can occupy the same column.
    - No two queens can occupy the same diagonal, which can be tracked using two auxiliary arrays: one for the main diagonals (difference of row and column indices) and another for the anti-diagonals (sum of row and column indices).

Algorithm

1. **Initialization**: Set up an empty board and auxiliary arrays to keep track of the columns and diagonals that are already occupied.
2. **Recursive Backtracking Function**:
    - Start from the first row and attempt to place a queen in each column.
    - For each column, check if placing a queen there is valid (i.e., check the column and both diagonals).
    - If valid, place the queen, mark the column and diagonals as occupied, and recursively attempt to place queens in the next row.
    - If placing a queen leads to a dead end (i.e., no valid positions in subsequent rows), backtrack by removing the queen and unmarking the occupied positions.
    - If all queens are successfully placed, store or print the solution.
3. **Termination**: The algorithm terminates when all rows have been processed.

Example

1. **Initialization**: We have an empty board:

. . . .

. . . .

. . . .

. . . .

2. **Placing Queens**:

- Start with the first row (row 0) and place a queen in column 0

  Q . . .

  . . . .

  . . . .

  . . . .

Move to row 1 and try placing a queen. We can place it in column 2:

Q . . .

. . Q .

. . . .

. . . .

Move to row 2. The possible placements are column 1:

Q . . .

. . Q .

. Q . .

. . . .

Finally, in row 3, we can place the last queen in column 3:

Q . . .

. . Q .

. Q . .

. . . Q

- This configuration is valid.

3. **Backtracking**: If we cannot place a queen in a particular configuration, we backtrack to the last placed queen and try the next column.

Conclusion

The 4 Queens Problem is not only a fundamental exercise in algorithm design but also a way to introduce students to the concept of backtracking. Through recursive strategies and careful tracking of constraints, we can efficiently explore possible configurations and find all valid solutions to the problem. This method can be extended to the N Queens Problem, where N can be any number, making it a versatile technique in algorithmic problem-solving.