University of Visvesvaraya College of Engineering

(First State Autonomous University On IIT Model)

K. R Circle, Bengaluru

Dept. of Computer Science and Engineering

Laboratory Record
for
Design and Analysis of Algorithms

Sunil Hegde
4th Semester ISE
U25UV22T064057

Certificate

This is to certify that **Sunil Hegde, U25UV22T064057** has successfully completed the course on **Practical Design and Analysis of Algorithms** as part of the curriculum for the **4th semester** of the **Information Science and Engineering** program at **UVCE, Bengaluru** during the academic year **2023 - 24**.

This manual includes various algorithms and their implementations, demonstrating a comprehensive understanding of the subject matter.

Date:

Dr. Thriveni J,	**Ms. Deepa Ashok Patil**,
Chairperson,	Instructor,
Dept. of CSE,	Dept. of CSE,
UVCE, Bengaluru	UVCE, Bengaluru

Index

**Time complexities are for average cases.*

1. Merge Sort

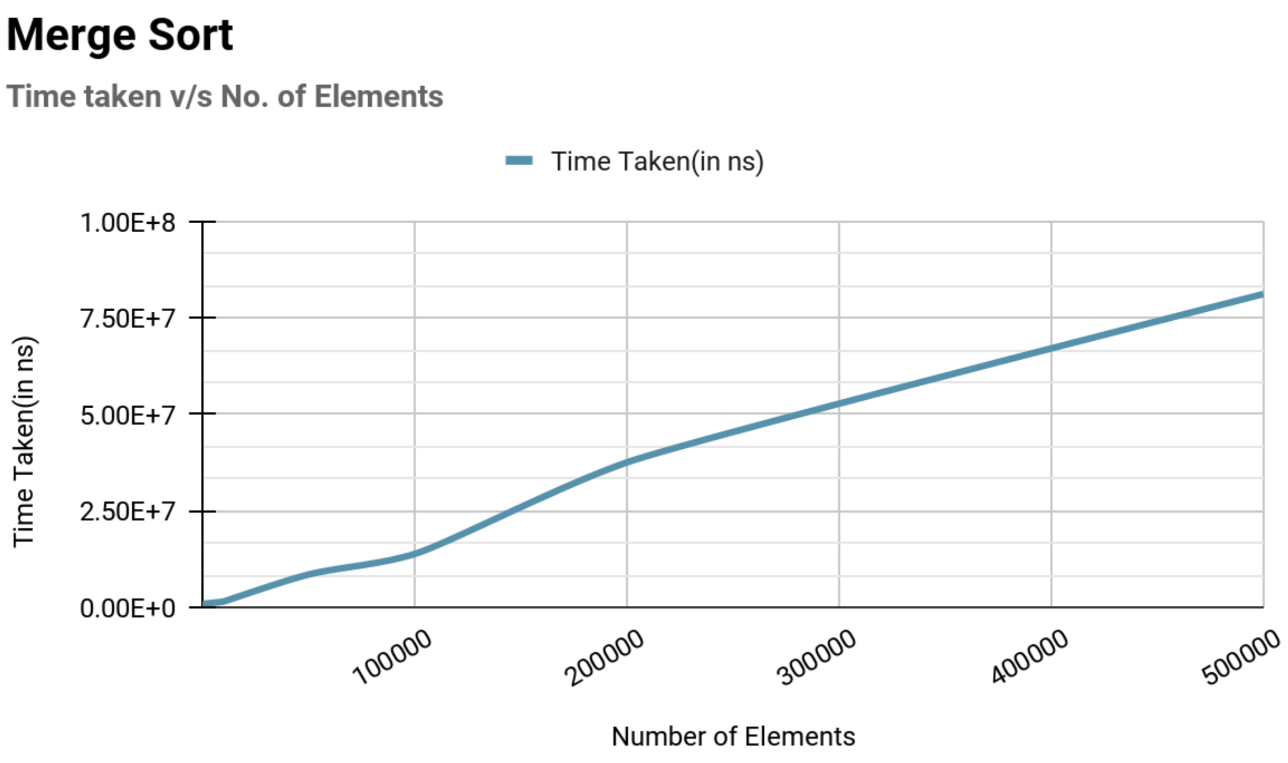
Sort a given set of elements using **Merge Sort** and determine the time required to sort the elements. Repeat the experiment for different values of `n`, the number of elements in the list to be sorted, and **plot a graph** of the **time taken versus the number of elements**. The elements can be read from a file or generated using a random number generator.

```java
import java.util.Random;

public class MergeSort{
    // Merge Sort implementation
    public static void mergeSort(int[] array, int left, int right) {
        if (left < right) {
            int middle = (left + right) / 2;
            // Sort the first and second halves
            mergeSort(array, left, middle);
            mergeSort(array, middle + 1, right);
            // Merge the sorted halves
            merge(array, left, middle, right);
        }
    }
    public static void merge(int[] array, int left, int middle, int right) {
        int n1 = middle - left + 1;
        int n2 = right - middle;
        // Temporary arrays
        int[] L = new int[n1];
        int[] R = new int[n2];
        // Copy data to temp arrays
        System.arraycopy(array, left, L, 0, n1);
        System.arraycopy(array, middle + 1, R, 0, n2);
        // Merge the temp arrays
        int i = 0, j = 0;
        int k = left;
        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
                array[k] = L[i];
                i++;
            } else {

                array[k] = R[j];
                j++;
            }
            k++;
        }
        // Copy remaining elements of L[] if any
        while (i < n1) {
            array[k] = L[i];
            i++;
            k++;
        }
        // Copy remaining elements of R[] if any
        while (j < n2) {
            array[k] = R[j];
            j++;
            k++;
        }
    }
    // Generate an array with random integers
    public static int[] generateRandomArray(int size) {
        Random random = new Random();
        int[] array = new int[size];
        for (int i = 0; i < size; i++) {
            array[i] = random.nextInt(10000);
        }
        return array;
    }
    public static void main(String[] args) {
        // Define the sizes of arrays to test
        int[] sizes = {100, 1000, 5000, 10000, 50000, 100000, 200000, 500000};
        for (int size : sizes) {
            int[] array = generateRandomArray(size);
            // Record start time in nanoseconds
            long startTime = System.nanoTime();
            // Sort the array using merge sort
            mergeSort(array, 0, array.length - 1);
            // Record end time in nanoseconds
            long endTime = System.nanoTime();
            // Calculate and display the time taken in nanoseconds
            long timeTaken = endTime - startTime;
            System.out.println("Size: " + size + ", Time taken: "
                                                  + timeTaken + " ns");
        }
    }
}
```

Output:

```
sunil@sunil:~/daaLabManual/program01$ java MergeSort.java
Size: 100, Time taken: 90745 ns
Size: 1000, Time taken: 853677 ns
Size: 5000, Time taken: 1155572 ns
Size: 10000, Time taken: 1493659 ns
Size: 50000, Time taken: 8477575 ns
Size: 100000, Time taken: 13802170 ns
Size: 200000, Time taken: 37517808 ns
Size: 500000, Time taken: 81148003 ns
```

Graph:



Time complexity for **Merge Sort** is **O(n*logn).**

Viva Questions:

What is the main idea behind Merge Sort?
It divides the array into halves, sorts them, and merges the sorted halves.

What is the time complexity of Merge Sort?
O(n log n) in average and worst cases.

Can Merge Sort be implemented iteratively?
Yes, using a bottom-up approach to merge sorted pairs progressively.

What is the space complexity of Merge Sort?
O(n) for temporary arrays used in merging.

What are the advantages of Merge Sort?
Efficient for large datasets, stable, and works well with linked lists.

2. Quick Sort

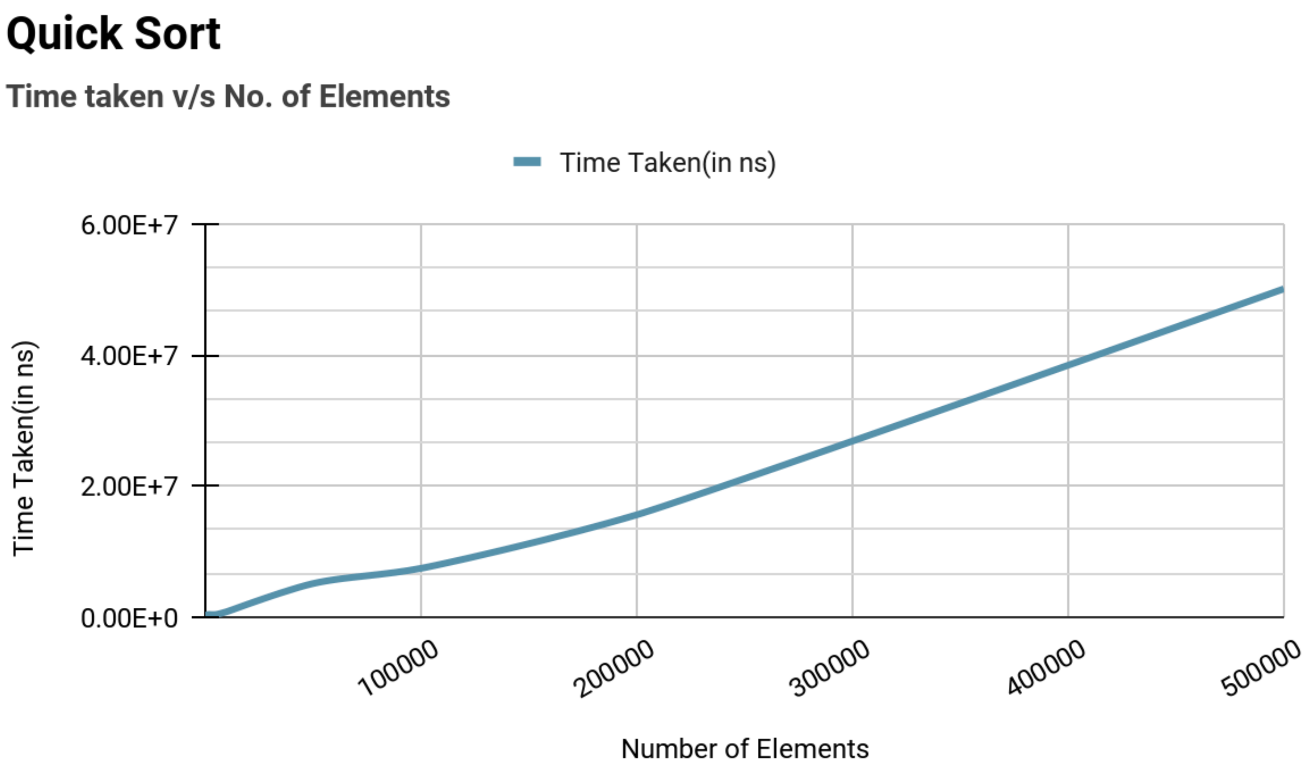
Sort a given set of elements using **Quick sort** and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted and **plot a graph** of the **time taken versus number of elements**. The elements can be read from a file or generated using a random number generator.

```java
import java.util.Random;

public class QuickSort {
    // Quick Sort implementation
    public static void quickSort(int[] array, int low, int high) {
        if (low < high) {
            int pi = partition(array, low, high);
            // Recursively sort elements before and after partition
            quickSort(array, low, pi - 1);
            quickSort(array, pi + 1, high);
        }
    }

    public static int partition(int[] array, int low, int high) {
        int pivot = array[high];
        int i = (low - 1); // Index of smaller element
        for (int j = low; j < high; j++) {
            // If current element is smaller than or equal to pivot
            if (array[j] <= pivot) {
                i++;
                // Swap array[i] and array[j]
                int temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
        }
        // Swap array[i + 1] and array[high] (or pivot)
        int temp = array[i + 1];
        array[i + 1] = array[high];
        array[high] = temp;

        return i + 1;
    }
    // Generate an array with random integers
    public static int[] generateRandomArray(int size) {
        Random random = new Random();
        int[] array = new int[size];
        for (int i = 0; i < size; i++) {
            array[i] = random.nextInt(10000)
        }
        return array;
    }

    public static void main(String[] args) {
        // Define the sizes of arrays to test
        int[] sizes = {100, 1000, 5000, 10000, 50000, 100000, 200000, 500000};
        for (int size : sizes) {
            int[] array = generateRandomArray(size);
            // Record start time in nanoseconds
            long startTime = System.nanoTime();
            // Sort the array using quick sort
            quickSort(array, 0, array.length - 1);
            // Record end time in nanoseconds
            long endTime = System.nanoTime();
            // Calculate and display the time taken in nanoseconds
            long timeTaken = endTime - startTime;
            System.out.println("Size: " + size + ", Time taken: "
                                                 + timeTaken + " ns");
        }
    }
}
```

Output:

```
sunil@sunil:~/daaLabManual/program02$ java QuickSort.java
Size: 100, Time taken: 31928 ns
Size: 1000, Time taken: 390821 ns
Size: 5000, Time taken: 418341 ns
Size: 10000, Time taken: 836881 ns
Size: 50000, Time taken: 5127389 ns
Size: 100000, Time taken: 7445483 ns
Size: 200000, Time taken: 15604122 ns
Size: 500000, Time taken: 50061383 ns
```

Graph:



Time complexity for **Quick Sort** is **O(n*logn).**

Viva Questions:

What is the basic idea behind Quick Sort?
It selects a pivot, partitions the array into elements less than and greater than the pivot, and recursively sorts the partitions.

What is the average time complexity of Quick Sort?
O(n log n) in average cases.

What is the worst-case time complexity of Quick Sort, and how can it be avoided?
O(n^2); avoid it by using random pivots or the median-of-three method.

Is Quick Sort a stable sorting algorithm?
No, it is not stable.

What is the space complexity of Quick Sort?
O(log n) for the recursion stack in the average case.

3. Binary Search Tree

Write a program to perform **insert** and **delete** operations in **Binary Search Tree**.

```
class BinarySearchTree {
    // Node class representing a node in the BST
    class Node {
        int key;
        Node left, right;
        public Node(int item) {
            key = item;
            left = right = null;
        }
    }
    // Root of the BST
    Node root;
    // Constructor
    BinarySearchTree() {
        root = null;
    }
    // Insert a new key in the BST
    void insert(int key) {
        root = insertRec(root, key);
    }
    // A recursive function to insert a new key in the BST
    Node insertRec(Node root, int key) {
        // If the tree is empty, return a new node
        if (root == null) {
            root = new Node(key);
            return root;
        }
        // Otherwise, recur down the tree
        if (key < root.key) {
            root.left = insertRec(root.left, key);
        } else if (key > root.key) {
            root.right = insertRec(root.right, key);
        }
        // Return the unchanged node pointer
        return root;
    }

    // Delete a key from the BST
    void deleteKey(int key) {
        root = deleteRec(root, key);
    }
    // A recursive function to delete a key in the BST
    Node deleteRec(Node root, int key) {
        // Base case: If the tree is empty
        if (root == null) {
            return root;
        }
        // Otherwise, recur down the tree
        if (key < root.key) {
            root.left = deleteRec(root.left, key);
        } else if (key > root.key) {
            root.right = deleteRec(root.right, key);
        } else {
            // Node with only one child or no child
            if (root.left == null) {
                return root.right;
            } else if (root.right == null) {
                return root.left;
            }
            // Node with two children:
           // Get the inorder successor (smallest in the right subtree)
            root.key = minValue(root.right);
            // Delete the inorder successor
            root.right = deleteRec(root.right, root.key);
        }
        return root;
    }
    // A utility function to find the minimum value node in a given BST
    int minValue(Node root) {
        int minValue = root.key;
        while (root.left != null) {
            minValue = root.left.key;
            root = root.left;
        }
        return minValue;
    }
    // This method calls the in-order traversal of the BST
    void inorder() {
        inorderRec(root);
    }
    // Function to do in-order traversal of the BST
```

```java
    void inorderRec(Node root) {
        if (root != null) {
            inorderRec(root.left);
            System.out.print(root.key + " ");
            inorderRec(root.right);
        }
    }
    public static void main(String[] args) {
        BinarySearchTree bst = new BinarySearchTree();
        int[] nodes = {50, 30, 20, 40, 70, 60, 80};
        // Insert nodes
        for(int node : nodes){
            bst.insert(node);
        }
        // Print the in-order traversal of the BST
        System.out.println("In-order traversal of the given tree:");
        bst.inorder();
        System.out.println();
        // Delete nodes
        int[] deleteNodes = {20, 30, 50};
        for(int node : deleteNodes){
            bst.deleteKey(node);
            System.out.println("In-order traversal after deleting "
                                                    + node + ": ");
            bst.inorder();
            System.out.println();
        }
    }
}
```

Output:

```
sunil@sunil:~/daaLabManual/program03$ java BinarySearchTree.java
In-order traversal of the given tree:
20 30 40 50 60 70 80
In-order traversal after deleting 20:
30 40 50 60 70 80
In-order traversal after deleting 30:
40 50 60 70 80
In-order traversal after deleting 50:
40 60 70 80
```

Viva Questions:

What is a Binary Search Tree?
A BST is a tree data structure where each node has at most two children, and the left child is less than the parent, while the right child is greater.

What are the average and worst-case time complexities for searching in a BST?
Average: O(log n); Worst: O(n) if unbalanced.

How do you insert a value into a BST?
Start at the root; if the value is less, go left; if greater, go right; repeat until finding a null position.

What is a balanced BST, and why is it important?
A balanced BST maintains height balance to ensure O(log n) time complexity for operations.

What is an in-order traversal of a BST?
Visiting the left subtree, then the node, and finally the right subtree, resulting in sorted order.

4. Breadth-First Search

Print all the nodes reachable from a given starting node in a **Di-graph** using the **Breadth-First Search** method.

```java
import java.util.*;

public class BFSDigraph {
    // Graph class to represent a directed graph using adjacency list
    static class Graph {
        private int vertices; // Number of vertices
        private LinkedList<Integer>[] adjList; // Adjacency list
        // Constructor
        Graph(int vertices) {
            this.vertices = vertices;
            adjList = new LinkedList[vertices];
            for (int i = 0; i < vertices; i++) {
                adjList[i] = new LinkedList<>();
            }
        }
        // Function to add an edge to the graph
        void addEdge(int v, int w) {
            adjList[v].add(w); // Add w to v's list
        }
        // Function to perform BFS from a given starting node
        void BFS(int start) {
            // Mark all the vertices as not visited
            boolean[] visited = new boolean[vertices];
            // Create a queue for BFS
            LinkedList<Integer> queue = new LinkedList<>();
            // Mark the current node as visited and enqueue it
            visited[start] = true;
            queue.add(start);
            while (queue.size() != 0) {
                // Dequeue a vertex from the queue and print it
                start = queue.poll();
                System.out.print(start + " ");
                // Get all adjacent vertices of the dequeued vertex
                // If an adjacent vertex has not been visited, mark it as
                //visited and enqueue it
                for (int n : adjList[start]) {
                    if (!visited[n]) {
                        visited[n] = true;
                        queue.add(n);
                    }
                }
            }
        }
    }
    public static void main(String[] args) {
        // Create a graph with 6 vertices
        Graph graph = new Graph(6);
        // Add edges to the digraph
        graph.addEdge(0, 1);
        graph.addEdge(0, 2);
        graph.addEdge(1, 2);
        graph.addEdge(2, 0);
        graph.addEdge(2, 3);
        graph.addEdge(3, 3);
        graph.addEdge(4, 5);
        // Print all nodes reachable from node 2
        System.out.println("Following are the nodes reachable from node 2:");
        graph.BFS(2);
        System.out.println();
    }
}
```

Output:

```
sunil@sunil:~/daaLabManual/program04$ java BFSDigraph.java

Following are the nodes reachable from node 2:
2 0 3 1
```

Viva Questions:

What is Breadth-First Search?
BFS is an algorithm for traversing or searching tree or graph data structures level by level.

What data structure is typically used to implement BFS?
A queue is used to keep track of nodes to be explored.

What is the time complexity of BFS?
O(V + E), where V is the number of vertices and E is the number of edges.

What is the main difference between BFS and Depth-First Search (DFS)?
BFS explores all neighbors at the present depth before moving to the next level, while DFS goes as deep as possible along one branch before backtracking.

Can BFS be used to find the shortest path in an unweighted graph?
Yes, BFS can find the shortest path in terms of the number of edges.

5. Topological Sort

Obtain the **Topological ordering** of vertices in a given digraph.

```java
import java.util.*;

public class TopologicalSort {
    private int V; // Number of vertices
    private List<Integer>[] adjList; // Adjacency list
    // Constructor
    @SuppressWarnings("unchecked")
    public TopologicalSort(int v) {
        V = v;
        adjList = new LinkedList[v];
        for (int i = 0; i < v; ++i)
            adjList[i] = new LinkedList<>();
    }
    // Function to add an edge to the graph
    private void addEdge(int v, int w) {
        adjList[v].add(w);
    }
    // A recursive function used by topologicalSort
    private void topologicalSortUtil(int v, boolean[] visited, Stack<Integer>
stack) {
        // Mark the current node as visited
        visited[v] = true;
        // Recur for all the vertices adjacent to this vertex
        for (Integer neighbor : adjList[v]) {
            if (!visited[neighbor])
                topologicalSortUtil(neighbor, visited, stack);
        }
        // Push the current vertex to the stack which stores the result
        stack.push(v);
    }
    // The function to do Topological Sort
    private void topologicalSort() {
        Stack<Integer> stack = new Stack<>();
        // Mark all the vertices as not visited
        boolean[] visited = new boolean[V];
        Arrays.fill(visited, false);
        // Call the recursive helper function to store Topological Sort starting
        // from all vertices one by one
        for (int i = 0; i < V; i++) {
            if (!visited[i])
                topologicalSortUtil(i, visited, stack);
        }
        // Print the contents of the stack
        System.out.println("Topological Sort:");
        while (!stack.isEmpty())
            System.out.print(stack.pop() + " ");
    }
    // Main method to test the Topological Sort
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the number of vertices: ");
        int V = scanner.nextInt();
        TopologicalSort g = new TopologicalSort(V);
        System.out.println("Enter the adjacency matrix:");
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                if (scanner.nextInt() == 1) {
                    g.addEdge(i, j);
                }
            }
        }
        g.topologicalSort();
        scanner.close();
        System.out.println();
    }
}
```

Output:

```
sunil@sunil:~/daaLabManual/program05$ java TopologicalSort.java
Enter the number of vertices: 5
Enter the adjacency matrix:
0 5 0 0 0
0 0 7 10 0
0 0 0 3 0
0 0 0 0 6
0 2 0 0 0
Topological Sort:
4 3 2 1 0
```

Viva Questions:

What is topological sorting?

Topological sorting is the linear ordering of vertices in a directed acyclic graph (DAG) such that for every directed edge (u, v), vertex u comes before vertex v.

What are the main applications of topological sorting?

It is commonly used in scheduling tasks, resolving dependencies, and in the representation of directed acyclic graphs.

What is the time complexity of topological sort using Kahn's algorithm?

O(V + E), where V is the number of vertices and E is the number of edges.

Can topological sorting be applied to graphs with cycles?

No, topological sorting is only applicable to directed acyclic graphs (DAGs).

What data structures can be used to implement topological sort?

A queue can be used for Kahn's algorithm, and a stack can be used for the depth-first search (DFS) based approach.

6. Warshall’s Algorithm

Compute the transitive closure of a given directed graph using **Warshall’s Algorithm**.

```
import java.util.Scanner;

public class WarshallsAlgorithm {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the number of vertices: ");
        int vertices = scanner.nextInt();
        // Initialize the adjacency matrix
        int[][] graph = new int[vertices][vertices];
        // Get the adjacency matrix from the user
        System.out.println("Enter the adjacency matrix (0 for no edge, 1 for
                            edge):");
        for (int i = 0; i < vertices; i++) {
            for (int j = 0; j < vertices; j++) {
                graph[i][j] = scanner.nextInt();
            }
        }
        // Find the transitive closure using Warshall's Algorithm
        for (int k = 0; k < vertices; k++) {
            for (int i = 0; i < vertices; i++) {
                for (int j = 0; j < vertices; j++) {
                    graph[i][j] = graph[i][j] | (graph[i][k] & graph[k][j]);
                }
            }
        }
        // Display the transitive closure matrix
        System.out.println("Transitive Closure:");
        for (int i = 0; i < vertices; i++) {
            for (int j = 0; j < vertices; j++) {
                System.out.print(graph[i][j] + " ");
            }
            System.out.println();
        }
        scanner.close();
    }
}
```

Output:

```
sunil@sunil:~/daaLabManual/program06$ java WarshallsAlgorithm.java
Enter the number of vertices: 5
Enter the adjacency matrix (0 for no edge, 1 for edge):
0 1 0 0 0
0 0 1 1 0
0 0 0 1 0
0 0 0 0 1
0 1 0 0 0
Transitive Closure:
0 1 1 1 1
0 1 1 1 1
0 1 1 1 1
0 1 1 1 1
0 1 1 1 1
```

Viva Questions:

What is Warshall's algorithm used for?
It is used to compute the transitive closure of a directed graph, indicating which vertices are reachable from each other.

What is the time complexity of Warshall's algorithm?
$O(V^3)$, where V is the number of vertices in the graph.

What is the time complexity of topological sort using Kahn's algorithm?
It iteratively updates the matrix by checking if there is an indirect path through an intermediate vertex.

What is the input and output of Warshall's algorithm?
Input: An adjacency matrix of a directed graph; Output: A reachability matrix indicating the transitive closure.

What does the transitive closure of a graph represent?
It represents a matrix where each entry (i, j) is true if there is a path from vertex i to vertex j.

7. Heap Sort

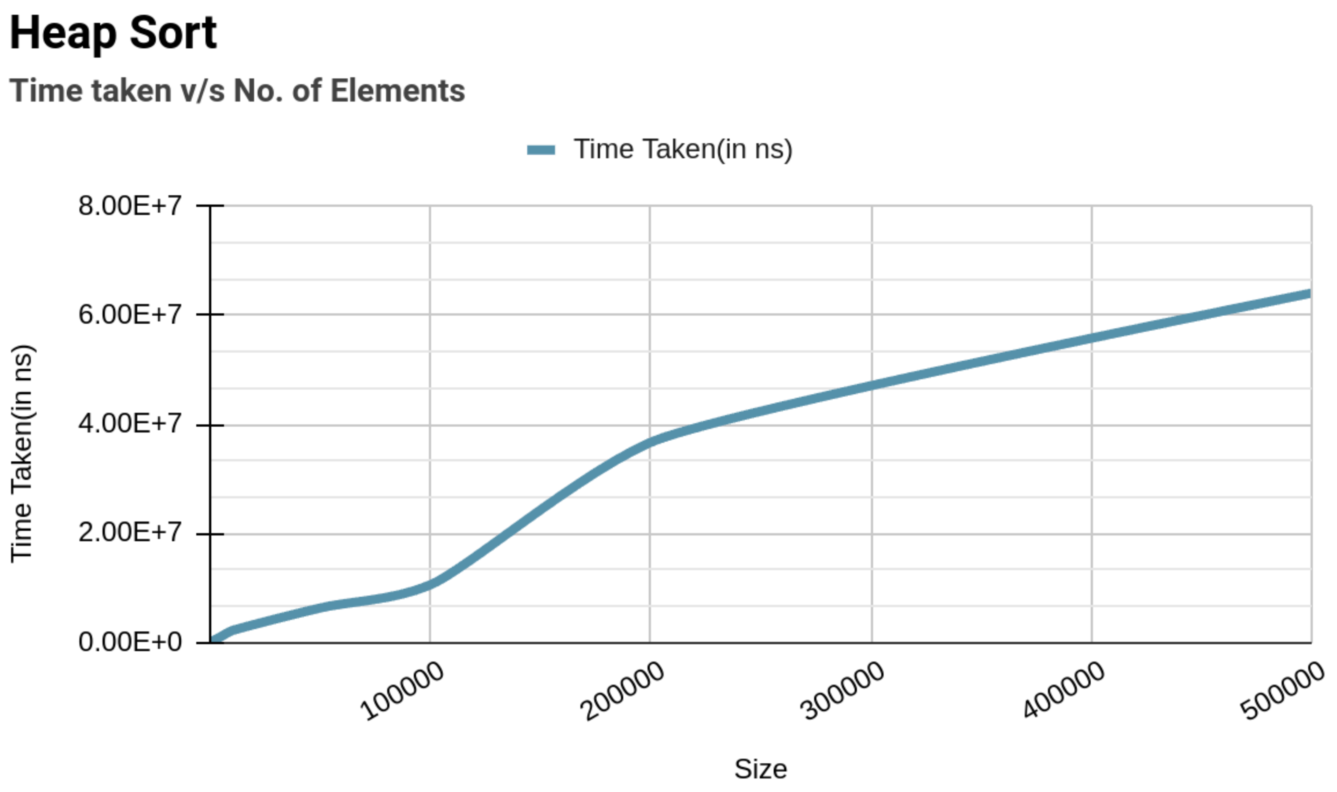
Sort a given set of elements using **Heap Sort** and determine the time required to sort the elements. Repeat the experiment for different values of `n`, the number of elements in the list to be sorted, and **plot a graph** of the **time taken versus the number of elements**. The elements can be read from a file or generated using a random number generator.

```
import java.util.Random;

public class HeapSort{
    public static void heapSort(int[] array) {
        int n = array.length;
        // Build a max heap
        for (int i = n / 2 - 1; i >= 0; i--) {
            heapify(array, n, i);
        }
        // One by one extract an element from the heap
        for (int i = n - 1; i > 0; i--) {
            // Move current root to end
            int temp = array[0];
            array[0] = array[i];
            array[i] = temp;
            // Call max heapify on the reduced heap
            heapify(array, i, 0);
        }
    }
    // To heapify a subtree rooted with node i
    public static void heapify(int[] array, int n, int i) {
        int largest = i; // Initialize largest as root
        int left = 2 * i + 1; // left child
        int right = 2 * i + 2; // right child
        // If left child is larger than root
        if (left < n && array[left] > array[largest]) {
            largest = left;
        }
        // If right child is larger than largest so far
        if (right < n && array[right] > array[largest]) {
            largest = right;
        }
        // If largest is not root
        if (largest != i) {
            int swap = array[i];
            array[i] = array[largest];
            array[largest] = swap;
            // Recursively heapify the affected sub-tree
            heapify(array, n, largest);
        }
    }
    // Generate an array with random integers
    public static int[] generateRandomArray(int size) {
        Random random = new Random();
        int[] array = new int[size];
        for (int i = 0; i < size; i++) {
            array[i] = random.nextInt(10000);
        }
        return array;
    }
    public static void main(String[] args) {
        int[] sizes = {100, 1000, 5000,
                       10000, 50000, 100000,
                       200000, 500000};
        for (int size : sizes) {
            int[] array = generateRandomArray(size);
            // Record start time in nanoseconds
            long startTime = System.nanoTime();
            // Sort the array using heap sort
            heapSort(array);
            // Record end time in nanoseconds
            long endTime = System.nanoTime();
            // Calculate and display the time taken in nanoseconds
            long timeTaken = endTime - startTime;
            System.out.println("Size: " + size + ", Time taken: "
                                      + timeTaken + " ns");
        }
    }
}
```

Output:

```
sunil@sunil:~/daaLabManual/program07$ java HeapSort.java
Size: 100, Time taken: 168116 ns
Size: 1000, Time taken: 219854 ns
Size: 5000, Time taken: 1090572 ns
Size: 10000, Time taken: 2167278 ns
Size: 50000, Time taken: 6318440 ns
Size: 100000, Time taken: 10556163 ns
Size: 200000, Time taken: 36708585 ns
Size: 500000, Time taken: 64081103 ns
```

Graph:



Time complexity for **Heap Sort** is **O(n*logn).**

Viva Questions:

What is heap sort?
Heap sort is a comparison-based sorting algorithm that uses a binary heap data structure to sort elements.

What are the two main phases of heap sort?
The two main phases are building the max heap and then repeatedly extracting the maximum element to sort the array.

What is the time complexity of heap sort?
The time complexity is O(n log n) for the average and worst cases.

How does heap sort ensure the elements are sorted?
By repeatedly removing the maximum element from the heap and placing it at the end of the array, ensuring sorted order.

What is the space complexity of heap sort?
The space complexity is O(1) because it sorts the array in place.

8. Horspool String Matching Algorithm

Search for a pattern string in a given text using **Horspool String Matching algorithm**.

```
public class HorspoolStringMatching {
    static final int NO_OF_CHARS = 256;
    // Utility function to get the maximum of two integers
    static int max(int a, int b) {
        return (a > b) ? a : b;
    }
    // Function to fill the bad character heuristic table
    static void badCharHeuristic(char[] pattern, int size, int[] badChar) {
        // Initialize all occurrences of characters as -1
        for (int i = 0; i < NO_OF_CHARS; i++) {
            badChar[i] = -1;
        }
        // Fill the actual value of last occurrence of a character
        for (int i = 0; i < size; i++) {
            badChar[(int) pattern[i]] = i;
        }
    }
    // Function to perform the Horspool's algorithm for string matching
    static void search(char[] text, char[] pattern) {
        int m = pattern.length;
        int n = text.length;
        int[] badChar = new int[NO_OF_CHARS];
        // Fill the bad character heuristic table
        badCharHeuristic(pattern, m, badChar);
        int s = 0;  // s is the shift of the pattern with respect to text
        while (s <= (n - m)) {
            int j = m - 1;
            // Decrease j while characters of pattern and text are matching
            // at this shift s
            while (j >= 0 && pattern[j] == text[s + j]) {
                j--;
            }
            // If the pattern is present at the current shift, print the index
            if (j < 0) {
                System.out.println("Pattern occurs at shift = " + s);
                // Shift the pattern so that the next character in text aligns
                // with the last occurrence of it in pattern.
                // The condition s+m < n is necessary for the case when the
                // pattern occurs at the end of the text
                s += (s + m < n) ? m - badChar[text[s + m]] : 1;
            } else {
                // Shift the pattern so that the bad character in text
                // aligns with the last occurrence of it in pattern. The
                // max function is used to make sure that we get a positive
                // shift.
                s += max(1, j - badChar[text[s + j]]);
            }
        }
    }
    public static void main(String[] args) {
        char[] text = "ABAAABCD".toCharArray();
        char[] pattern = "ABC".toCharArray();
        search(text, pattern);
    }
}
```

Output:

```
sunil@sunil:~/daaLabManual/program08$ java HorspoolStringMatching.java
Pattern occurs at shift = 4
```

Viva Questions:

What is the Horspool algorithm used for?
The Horspool algorithm is used for substring search, efficiently finding occurrences of a pattern within a text.

What is the main advantage of the Horspool algorithm over the naive approach?
Horspool reduces the number of comparisons by using a shift table based on the characters in the pattern, allowing for skips in the text.

What is the time complexity of the Horspool algorithm in the average case?
The average case time complexity is O(n), where n is the length of the text.

How is the shift table constructed in the Horspool algorithm?
The shift table is built using the pattern, mapping each character to the number of positions to shift when a mismatch occurs.

9. Knapsack Problem

Implement 0/1 **Knapsack problem** using **dynamic programming**.

```java
class Knapsack {
    // Function to return the maximum of two integers
    static int max(int a, int b) {
        return (a > b) ? a : b;
    }
    // Function to solve the 0/1 Knapsack problem
    static int knapSack(int W, int wt[], int val[], int n) {
        // Base case: If no items left or capacity becomes 0, return 0
        if (n == 0 || W == 0)
            return 0;
        // If the weight of the nth item is more than the current capacity W,
        // skip this item
        if (wt[n - 1] > W)
            return knapSack(W, wt, val, n - 1);
        // Otherwise, return the maximum of two cases:
        // 1. nth item included
        // 2. nth item not included
        else
            return max(
                val[n - 1] + knapSack(W - wt[n - 1], wt, val, n - 1),
                knapSack(W, wt, val, n - 1)
            );
    }
    public static void main(String args[]) {
        int profit[] = {60, 100, 120};  // Profit values of the items
        int weight[] = {10, 20, 30};    // Weight values of the items
        int W = 50;                     // Maximum capacity of the knapsack
        int n = profit.length;          // Number of items
        // Print the result of the knapsack problem
        System.out.println("Maximum profit: " + knapSack(W, weight, profit, n));
    }
}
```

Output:

```
sunil@sunil:~/daaLabManual/program09$ java Knapsack.java
Maximum profit: 220
```

Viva Questions:

What is the 0/1 Knapsack problem?
The 0/1 Knapsack problem involves selecting items with given weights and values to maximize total value without exceeding a weight limit, where each item can either be included or excluded.

How does dynamic programming solve the 0/1 Knapsack problem?
Dynamic programming builds a table that stores the maximum value that can be achieved for every weight from 0 to the maximum weight, using previously computed results to avoid redundant calculations.

What is the time complexity of the dynamic programming solution to the 0/1 Knapsack problem?
The space complexity is O(n * W) for the full table, but it can be optimized to O(W) using a one-dimensional array to store results for the current and previous items.

What are the space complexity considerations for the dynamic programming approach?
The shift table is built using the pattern, mapping each character to the number of positions to shift when a mismatch occurs.

Can you explain the significance of the decision to include or exclude an item?
The decision to include or exclude an item affects the remaining capacity and the total value, and is determined by comparing the value of including the item versus the maximum value without it.

10. Prim’s Algorithm

Find **Minimum Cost Spanning Tree** of a given undirected graph using **Prim’s algorithm**.

```java
import java.util.Scanner;

public class Prims {
    public static void main(String[] args) {
        int w[][] = new int[10][10];  // Weight matrix for the graph
        int n, i, j, s, k = 0, min, sum = 0, u = 0, v = 0, flag = 0;
        int sol[] = new int[10];  // Array to keep track of selected vertices
        Scanner sc = new Scanner(System.in);
        // Input number of vertices
        System.out.println("Enter the number of vertices:");
        n = sc.nextInt();
        // Initialize solution array
        for (i = 1; i <= n; i++)
            sol[i] = 0;
        // Input the weighted graph
        System.out.println("Enter the weighted adjacency matrix:");
        for (i = 1; i <= n; i++) {
            for (j = 1; j <= n; j++) {
                w[i][j] = sc.nextInt();
            }
        }
        // Input the source vertex
        System.out.println("Enter the source vertex:");
        s = sc.nextInt();
        sol[s] = 1;  // Mark the source vertex as selected
        k = 1;
        // Prim's Algorithm: Loop until all vertices are included
        while (k <= n - 1) {
            // Set min to a large value to find the smallest weight
            min = 99;
            // Find the minimum edge (u, v) where u is already in the MST and v
            // is not
            for (i = 1; i <= n; i++) {
                for (j = 1; j <= n; j++) {
                    if (sol[i] == 1 && sol[j] == 0) {  // i is in MST, j is not
                        if (i != j && min > w[i][j]) {
                        // Ensure not self-loop and check for minimum weight
```

```
                            min = w[i][j];
                            u = i;
                            v = j;
                        }
                    }
                }
            }
            // Add vertex v to the MST
            sol[v] = 1;
            sum += min;  // Add the weight of the edge to the total sum
            k++;
            // Output the selected edge and its weight
            System.out.println(u + " -> " + v + " = " + min);
        }
        // Check if all vertices are included in the MST
        for (i = 1; i <= n; i++) {
            if (sol[i] == 0) {
                flag = 1;  // Not all vertices are included
                break;
            }
        }
        // Output the result
        if (flag == 1)
            System.out.println("No spanning tree exists.");
        else
            System.out.println("The cost of the minimum spanning tree is: " +
                                                                        sum);
        sc.close();
    }
}
```

Output:

```
sunil@sunil:~/daaLabManual/program10$ java Prims.java
Enter the number of vertices: 5
Enter the weighted adjacency matrix:
0 10 70 80 60
10 0 20 30 90
70 20 0 40 100
80 30 40 0 50
60 90 100 50 0
```

```
Enter the source vertex: 1
1 -> 2 = 10
2 -> 3 = 20
2 -> 4 = 30
4 -> 5 = 50
The cost of the minimum spanning tree is: 110
```

Viva Questions:

What is Prim's algorithm used for?

Prim's algorithm is used to find the minimum spanning tree of a weighted undirected graph.

How does Prim's algorithm start?

The algorithm starts with an arbitrary vertex and grows the minimum spanning tree by repeatedly adding the smallest edge connecting a vertex in the tree to a vertex outside the tree.

What data structure is commonly used to implement Prim's algorithm?

A priority queue (or min-heap) is commonly used to efficiently retrieve the minimum edge during the execution of the algorithm.

What is the time complexity of Prim's algorithm using an adjacency matrix and a priority queue?

The time complexity is O(E log V), where E is the number of edges and V is the number of vertices.

How does Prim's algorithm ensure that no cycles are formed?

Prim's algorithm only adds edges that connect a vertex in the tree to a vertex outside the tree, ensuring that no cycles can be formed in the minimum spanning tree.

11. Kruskal’s Algorithm

Find **Minimum Cost Spanning Tree** of a given undirected graph using **Kruskal’s algorithm**.

```java
import java.util.Scanner;

public class Kruskal {
    int parent[] = new int[10];  // Array to store the parent of each node
    // Method to find the root of the set containing element m
    int find(int m) {
        while (parent[m] != 0)
            m = parent[m];
        return m;
    }
    // Method to perform the union of two sets
    void union(int i, int j) {
        parent[j] = i;  // Make the root of set j point to the root of set i
    }
    // Kruskal's algorithm to find the Minimum Spanning Tree
    void kruskal(int a[][], int n) {
        int u = 0, v = 0, min, sum = 0, k = 0;
        // Repeat until n-1 edges are found
        while (k < n - 1) {
            min = 99;  // Set to a large number to find the minimum edge
            // Find the minimum edge in the graph
            for (int i = 1; i <= n; i++) {
                for (int j = 1; j <= n; j++) {
                    if (a[i][j] != 0 && a[i][j] < min && i != j) {
                        min = a[i][j];
                        u = i;
                        v = j;
                    }
                }
            }
            // Find the roots of the sets that u and v belong to
            int i = find(u);
            int j = find(v);
            // If u and v belong to different sets, add the edge to the MST
            if (i != j) {
                union(i, j);  // Merge the sets
                System.out.println("(" + u + "," + v + ") = " + a[u][v]);
                sum += a[u][v];  // Add the weight of the edge to the total cost
                k++;  // Increase the edge count in the MST
            }
            // Remove the selected edge from consideration by setting its weight
            // to a large value
            a[u][v] = a[v][u] = 99;
        }
        // Output the total cost of the Minimum Spanning Tree
        System.out.println("The cost of the minimum spanning tree = " + sum);
    }
    public static void main(String[] args) {
        int a[][] = new int[10][10];
        Scanner sc = new Scanner(System.in);
        // Input number of vertices
        System.out.println("Enter the number of vertices of the graph:");
        int n = sc.nextInt();
        // Input the weighted adjacency matrix
        System.out.println("Enter the weighted adjacency matrix (use 0 for no
                                                                connection):");
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                a[i][j] = sc.nextInt();
            }
        }
        // Create an object of Kruskal class and run the algorithm
        Kruskal k = new Kruskal();
        k.kruskal(a, n);
        sc.close();
    }
}
```

Output:

```
sunil@sunil:~/daaLabManual/program11$ java Kruskal.java
Enter the number of vertices of the graph: 5
Enter the weighted adjacency matrix (use 0 for no connection):
0 1 5 0 0
1 0 4 2 0
5 4 0 3 0
0 2 3 0 6
0 0 0 6 0
```

```
(1,2) = 1
(2,4) = 2
(3,4) = 3
(4,5) = 6
The cost of the minimum spanning tree = 12
```

Viva Questions:

What is Kruskal's algorithm used for?
Kruskal's algorithm is used to find the minimum spanning tree of a weighted undirected graph.

How does Kruskal's algorithm begin?
The algorithm starts by sorting all the edges of the graph in non-decreasing order of their weights.

What data structure is commonly used to detect cycles in Kruskal's algorithm?
A disjoint-set data structure (or union-find) is used to detect cycles when adding edges to the minimum spanning tree.

What is the time complexity of Kruskal's algorithm?
The time complexity is O(E log E) or O(E log V), where E is the number of edges and V is the number of vertices.

How does Kruskal's algorithm ensure that the minimum spanning tree is formed?
Kruskal's algorithm adds edges one by one, ensuring that no cycles are formed, until V-1 edges are included in the tree, which guarantees a minimum spanning tree.

12. Dijkstra’s Algorithm

From a given vertex in a **weighted connected graph**, find shortest paths to other vertices using **Dijkstra's algorithm**.

```
import java.util.Scanner;

public class Dijkstra {
    int d[] = new int[10];  // Distance array
    int p[] = new int[10];  // Predecessor array
    int visited[] = new int[10];  // Visited array to track processed vertices
    // Method to perform Dijkstra's Algorithm
    public void dijk(int[][] a, int s, int n) {
        int u = -1, v, i, j, min;
        // Initialize distance and predecessor arrays
        for (i = 0; i < n; i++) {
            d[i] = 99;  // Initialize distances to a large number
            visited[i] = 0;  // Mark all vertices as unvisited
            p[i] = -1;  // No predecessor for any vertex initially
        }
        d[s] = 0;  // Distance of source vertex to itself is 0
        // Main loop of Dijkstra's algorithm
        for (i = 0; i < n; i++) {
            min = 99;
            u = -1;
            // Find the unvisited vertex with the smallest distance
            for (j = 0; j < n; j++) {
                if (visited[j] == 0 && d[j] < min) {
                    min = d[j];
                    u = j;
                }
            }
            // Mark the vertex as visited
            visited[u] = 1;
            // Update the distances of the adjacent vertices
            for (v = 0; v < n; v++) {
                if (a[u][v] != 0 && visited[v] == 0 && d[u] + a[u][v] < d[v]) {
                    d[v] = d[u] + a[u][v];  // Update the distance
                    p[v] = u;  // Set the predecessor
                }
            }
        }
    }
    // Method to print the shortest path from source to a vertex
    void path(int v, int s) {
        if (p[v] != -1) {
            path(p[v], s);  // Recursively find the path
        }
        if (v != s) {
            System.out.print("->" + v);  // Print the vertex in the path
        }
    }
    // Method to display the shortest paths and their distances
    void display(int s, int n) {
        for (int i = 0; i < n; i++) {
            if (i != s) {
                System.out.print(s);  // Print source vertex
                path(i, s);  // Print the path to vertex i
                System.out.print(" = " + d[i] + "\n");
            }
        }
    }
    public static void main(String[] args) {
        int a[][] = new int[10][10];  // Adjacency matrix
        int i, j, n, s;
        Scanner sc = new Scanner(System.in);
        // Input the number of vertices
        System.out.println("Enter the number of vertices:");
        n = sc.nextInt();
        // Input the weighted adjacency matrix
        System.out.println("Enter the weighted matrix (use 0 for no
                                                     connection):");
        for (i = 0; i < n; i++) {
            for (j = 0; j < n; j++) {
                a[i][j] = sc.nextInt();
            }
        }
        // Input the source vertex
        System.out.println("Enter the source vertex:");
        s = sc.nextInt();
        // Create a Dijkstra object and run the algorithm
        Dijkstra tr = new Dijkstra();
        tr.dijk(a, s, n);
        // Display the shortest paths
        System.out.println("The shortest paths from source vertex " + s + " to
                                                    the remaining vertices are:");
        tr.display(s, n);
        sc.close();
    }
}
```

Output:

```
sunil@sunil:~/daaLabManual/program12$ java Dijkstra.java
Enter the number of vertices: 5
Enter the weighted matrix (use 0 for no connection):
0 4 2 0 0
4 0 1 5 0
2 1 0 8 10
0 5 8 0 2
0 0 10 2 0
Enter the source vertex: 0
The shortest paths from source vertex 0 to the remaining vertices are:
0->2->1 = 3
0->2 = 2
0->2->1->3 = 8
0->2->1->3->4 = 10
```

Viva Questions:

What is Dijkstra's algorithm used for?
Dijkstra's algorithm is used to find the shortest path from a source vertex to all other vertices in a weighted graph with non-negative weights.
How does Dijkstra's algorithm initialize distances?
It initializes the distance of the source vertex to 0 and all other vertices to infinity.
What data structure is commonly used to implement Dijkstra's algorithm?
A priority queue (or min-heap) is commonly used to efficiently retrieve the vertex with the minimum distance.
Can Dijkstra's algorithm handle negative edge weights?
No, Dijkstra's algorithm cannot handle negative edge weights; it may produce incorrect results in such cases.

13. Traveling Salesman Problem

Write a program to solve the **Travelling Salesman Problem** using **dynamic programming approach.**

```
import java.util.Arrays;

public class TSP {
    static int n = 4;  // Number of cities
    static int MAX = 1000000;  // A large value representing infinity
    // Distance matrix
    static int[][] dist = {
        {0, 0, 0, 0, 0},
        {0, 0, 10, 30, 20},
        {0, 10, 0, 60, 0},
        {0, 30, 60, 0, 50},
        {0, 20, 0, 50, 0}
    };
    // Memoization table to store subproblem results
    static int[][] memo = new int[n + 1][1 << (n + 1)];
    // Recursive function to calculate the minimum cost
    static int fun(int i, int mask) {
        if (mask == ((1 << i) | 3)) {  // Base case: only start and
                                        // current city are left
            return dist[1][i];  // Return distance from start to current
                                // city
        }
        if (memo[i][mask] != 0) {  // If this subproblem has already been
                                   // solved
            return memo[i][mask];  // Return the stored result
        }
        int res = MAX;  // Initialize result to a large value
        // Try visiting every other city and find the minimum cost
        for (int j = 1; j <= n; j++) {
            if ((mask & (1 << j)) != 0 && j != i && j != 1) {  // Check if
                             // city j is in the mask and is different from i
                res = Math.min(res, fun(j, mask & (~(1 << i))) + dist[j][i]);
            }
        }
        return memo[i][mask] = res;  // Store the result and return it
    }
    public static void main(String[] args) {
        int ans = MAX;  // Initialize the answer to a large value
        // Try every possible starting city and find the minimum cost
        for (int i = 1; i <= n; i++) {
            ans = Math.min(ans, fun(i, (1 << (n + 1)) - 1) + dist[i][1]);
        }
        // Output the result
        System.out.println("The cost of the most efficient tour = " + ans);
    }
}
```

Output:

```
sunil@sunil:~/daaLabManual/program13$ java TSP.java
The cost of the most efficient tour = 90
```

Viva Questions:

What is the Traveling Salesman Problem?
The TSP seeks to find the shortest possible route that visits each city exactly once and returns to the starting city.

How does the dynamic programming approach solve TSP?
It breaks the problem into smaller subproblems, solving for the shortest paths from the starting city to all other cities and using memoization to store intermediate results.

What is the time complexity of the dynamic programming solution for TSP?
The time complexity is O(n^2 * 2^n), where n is the number of cities.

What is the state representation in the dynamic programming approach for TSP?
The state is usually represented by the current city and a bitmask that indicates which cities have been visited.

Why is TSP considered NP-hard?
TSP is NP-hard because there is no known polynomial-time solution for it, and it requires checking all possible permutations of cities to ensure the shortest path.

14. N-Queens Problem

Implement **N-Queens Problem** using **Backtracking**.

```java
public class NQueenProblem {
    final int N = 5;
    // Function to print the solution
    void printSolution(int board[][]) {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                if (board[i][j] == 1)
                    System.out.print("Q ");
                else
                    System.out.print(". ");
            }
            System.out.println();
        }
    }
    // Function to check if a queen can be placed on board[row][col]
    boolean isSafe(int board[][], int row, int col) {
        int i, j;
        // Check this row on the left side
        for (i = 0; i < col; i++)
            if (board[row][i] == 1)
                return false;
        // Check upper diagonal on the left side
        for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
            if (board[i][j] == 1)
                return false;
        // Check lower diagonal on the left side
        for (i = row, j = col; j >= 0 && i < N; i++, j--)
            if (board[i][j] == 1)
                return false;
        return true;
    }
    // Recursive utility function to solve N Queen problem
    boolean solveNQUtil(int board[][], int col) {
        // If all queens are placed, return true
        if (col >= N)
            return true;
        // Try placing queen in all rows one by one
        for (int i = 0; i < N; i++) {
            if (isSafe(board, i, col)) {
                board[i][col] = 1; // Place this queen in board[i][col]
                // Recur to place rest of the queens
                if (solveNQUtil(board, col + 1))
                    return true;
                board[i][col] = 0; // BACKTRACK
            }
        }
        // If queen cannot be placed in any row, return false
        return false;
    }
    // This function solves the N Queen problem
    boolean solveNQ() {
        int board[][] = new int[N][N];  // Initialize an empty board
        if (!solveNQUtil(board, 0)) {
            System.out.println("Solution does not exist");
            return false;
        }
        printSolution(board);  // Print the solution if it exists
        return true;
    }
    public static void main(String[] args) {
        NQueenProblem Queen = new NQueenProblem();
        Queen.solveNQ();  // Call the function to solve the N-Queen problem
    }
}
```

Output:

```
sunil@sunil:~/daaLabManual/program14$ java NQueenProblem.java
Q . . . .
. . . Q .
. Q . . .
. . . . Q
. . Q . .
```

Viva Questions:

What is the N-Queens Problem?
The N-Queens Problem involves placing N queens on an N×N chessboard so that no two queens threaten each other.

How does the backtracking approach work for solving the N-Queens Problem?
It places a queen in a valid position and recursively attempts to place queens in subsequent rows, backtracking when a conflict occurs.

What conditions must be checked to ensure a queen's placement is safe?
You must check the same column, upper diagonal, and lower diagonal for existing queens.

What is the time complexity of the backtracking solution for the N-Queens Problem?
The time complexity is O(N!), as it may require exploring all possible arrangements of N queens.

How can the N-Queens Problem be visualized on a chessboard?
The solution can be visualized by marking the positions of the queens on the board, often represented as 'Q' for queens and '.' for empty squares.

15. Subset Sum Problem

Find a **subset** of a given set **S = {S1,S2,...,Sn}** of n positive integers whose **SUM** is equal to a given positive integer **d**.
import java.util.ArrayList;
import java.util.Scanner;

```java
public class SubsetSumDP {
    // Function to check if a subset with the given sum exists and to retrieve
    // the subset
    public static boolean subsetSum(int[] arr, int sum, ArrayList<Integer>
                                                                   subset) {
        int n = arr.length;
        boolean[][] dp = new boolean[n + 1][sum + 1];
        // Initialize dp array: dp[i][0] is true for all i
        for (int i = 0; i <= n; i++) {
            dp[i][0] = true;
        }
        // Fill dp array
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= sum; j++) {
                if (j >= arr[i - 1]) {
                    dp[i][j] = dp[i - 1][j] || dp[i - 1][j - arr[i - 1]];
                } else {
                    dp[i][j] = dp[i - 1][j];
                }
            }
        }
        // If the subset sum doesn't exist, return false
        if (!dp[n][sum]) {
            return false;
        }
        // Trace back the elements contributing to the subset sum
        int i = n, j = sum;
        while (i > 0 && j > 0) {
            if (dp[i][j] != dp[i - 1][j]) {
                subset.add(arr[i - 1]);
                j -= arr[i - 1];
            }
            i--;
        }
        return true;
    }
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        // Input the number of elements
        System.out.print("Enter the number of elements: ");
        int n = scanner.nextInt();
        // Input the array elements
        int[] arr = new int[n];
        System.out.println("Enter the elements:");
        for (int i = 0; i < n; i++) {
            arr[i] = scanner.nextInt();
        }
        // Input the target sum
        System.out.print("Enter the target sum: ");
        int sum = scanner.nextInt();
        // Initialize subset array and time the execution
        ArrayList<Integer> subset = new ArrayList<>();
        long startTime = System.nanoTime();
        boolean hasSubsetSum = subsetSum(arr, sum, subset);
        long endTime = System.nanoTime();
        // Output the result
        System.out.println("Subset sum exists: " + hasSubsetSum);
        if (hasSubsetSum) {
            System.out.println("Subset contributing to the sum: " + subset);
        }
        // Display the time complexity in milliseconds
        double timeElapsed = (endTime - startTime) / 1e6;
        System.out.println("Time Taken: " + timeElapsed + " ms");
        // Close the scanner
        scanner.close();
    }
}
```

Output:

```
sunil@sunil:~/daaLabManual/program15$ java SubsetSumDP.java
Enter the number of elements: 5
Enter the elements:
10 20 30 40 50
Enter the target sum: 40
```

```
Subset sum exists: true
Subset contributing to the sum: [30, 10]
Time Taken: 0.117341 ms
```

Viva Questions:

What is the Subset Sum Problem?
The Subset Sum Problem involves determining whether a subset of a given set of integers can sum to a specified target value.

What is the difference between the recursive and dynamic programming approaches to solving the Subset Sum Problem?
The recursive approach explores all subsets, leading to exponential time complexity, while the dynamic programming approach uses a table to store intermediate results, significantly improving efficiency.

What is the time complexity of the dynamic programming solution for the Subset Sum Problem?
The time complexity is O(n * sum), where n is the number of elements in the set and sum is the target sum.

How does the dynamic programming table work in solving the Subset Sum Problem?
The table stores boolean values indicating whether a specific sum can be formed using the first i elements of the set.

Can the Subset Sum Problem be solved in polynomial time?
No, the Subset Sum Problem is NP-complete, meaning no polynomial-time solution is known for all cases.