1



# BansilalRamnathAgarwalCharitableTrust’s

Vishwakarma Institute of Technology,Pune-37

*(Anautonomous Institute of Savitribai PhulePune University)*

**Department of Computer Engineering Lab Manual**

|  |  |  |  |
| --- | --- | --- | --- |
| **Course Code** | **Course Name** | **Lab Scheme (Hrs./Week)** | **Credits** |
| **CI3002** | **Design and Analysis of Algorithms** | **2** | **4** |

**Course Outcomes:**

1. To formulate computational problems in abstract and mathematically precise manner
2. To design efficient algorithms for computational problems using appropriate algorithmic paradigm
3. To analyze asymptotic complexity of the algorithm for a complex computational problem using suitable mathematical techniques.
4. To establish NP-completeness of some decision problems, grasp the significance of the notion of NP-completeness and its relationship with intractability of the decision problems.
5. To understand significance of randomness, approximability in computation and design randomized algorithms for simple computational problems and design efficient approximation algorithms for standard NP-optimization problems.
6. To incorporate appropriate data structures, algorithmic paradigms to craft innovative scientific solutions for complex computing problems.

Class:- TY Branch: -CSE(AI)

Year: 2025-26 Prepared By: -

Required Hardware:

Required Software:

**Index**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SN** | **Title of Experiment** | **CO**  **Mapping** | **PO**  **Mapping** | **Page Number** |
| 1 | Assignment Based on simple Coding Problems on numbers, graph and matrices. (Implement bubble sort using cpp or java. Determine Time and space complexity) | CO1, CO4 | PO1, PO2, PO3 |  |
| 2 | Assignment Based on Divide and Conquer Strategy. (Implement Recursive and Non-Recursive Binary Search Algorithm using cpp or java. Determine Time and space complexity) | CO1,CO2,CO3 | PO1,PO2 ,PO3, PO4, PO5, PO11 |  |
| 3 | Assignment Based on Dynamic programming strategy. (Implement 0-1 Knapsack problem using cpp or java) | CO1,CO2,CO3 | PO1,PO2 ,PO3, PO4, PO5, PO11 |  |
| 4 | Assignment Based on Backtracking. (Implement N- Queen problem) | CO1,CO2,CO3 | PO1,PO2 ,PO3, PO4, PO5, PO11 |  |
| 5 | Assignment Based on Greedy strategy. (Implement Huffman encoding algorithm) | CO2, CO3 | PO2, PO3, PO4 |  |
| 6 | Assignment Based on Dynamic programming strategy to implement traveling salesman problem. | CO2, CO3, CO6 | PO2, PO3, PO4,PO5 |  |

**Experiment No. 1**

**Title:**

Write a program to implement Bubble Sort to sort an array of integers in ascending order. Find out Time and space complexity.

**CO-PO mapping**

|  |  |  |  |
| --- | --- | --- | --- |
| Title of Experiment | CO Mapping | CO Statements | PO Mapping |
| Assignment Based on Sorting strategy. (Implement Bubble Sort to sort an array in ascending order and analyze time & space complexity) | CO1, CO4 | CO2: To design efficient algorithms for computational problems using appropriate algorithmic paradigm. CO3: To analyze asymptotic complexity of the algorithm for a complex computational problem using suitable mathematical techniques. | PO1, PO2, PO3 |

**Objective:**

* To understand the mechanism of comparison-based sorting.
* To implement Bubble Sort in C++ or Java.
* To analyze time and space complexity of Bubble Sort.

**Software Requirements:**

* Operating System: Windows/Linux
* Language: C++ or Java
* Compiler: g++/javac

**Hardware Requirements:**

* Processor: 2 GHz or above
* RAM: 4 GB or more
* Disk Space: Minimum 500 MB

**Theory:**

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order**.**

**Algorithm:**

1. Check if the first element in the input array is greater than the next element in the array.
2. If it is greater, swap the two elements; otherwise move the pointer forward in the array.
3. Repeat Step 2 until we reach the end of the array.
4. Check if the elements are sorted; if not, repeat the same process (Step 1 to Step 3) from the last element of the array to the first.
5. The final output achieved is the sorted array.

**Pseudocode of bubble sort:**

Start

Repeat for i = 0 to n-1

a. Repeat for j = 0 to n-i-1

- If arr[j] > arr[j+1], swap them

End

**Time Complexity:**

| **Best Case** | **O(n)** |
| --- | --- |
| Average Case | O(n²) |
| Worst Case | O(n²) |

**Space Complexity:**

It sorts data directly within array without additional memory apart from few variables (counter and temp). The memory usage does not grow with the size of input. Regardless of whether you are sorting 10 elements or 10,000, fixed amount of memory is used for variables.

Hence **Space Complexity of bubble sort is O(1).**

**Conclusion:**

Bubble sort is easy to understand and implement. However, it is inefficient on large lists and is rarely used in practice for performance-critical applications.

**Source Code, with description and with Output Need to be Uploaded to the VOLP**

import java.util.Scanner;

public class SortingDemo {

    public static void bubbleSort(int[] arr) {

        int n = arr.length;

        for (int i = 0; i < n - 1; i++) {

            for (int j = 0; j < n - i - 1; j++) {

                if (arr[j] > arr[j + 1]) {

                    int temp = arr[j];

                    arr[j] = arr[j + 1];

                    arr[j + 1] = temp;

                }

            }

        }

    }

    public static void quickSort(int[] arr, int low, int high) {

        if (low < high) {

            int pi = partition(arr, low, high);

            quickSort(arr, low, pi - 1);

            quickSort(arr, pi + 1, high);

        }

    }

    public static int partition(int[] arr, int low, int high) {

        int pivot = arr[high];

        int i = low - 1;

        for (int j = low; j < high; j++) {

            if (arr[j] < pivot) {

                i++;

                int temp = arr[i];

                arr[i] = arr[j];

                arr[j] = temp;

            }

        }

        int temp = arr[i + 1];

        arr[i + 1] = arr[high];

        arr[high] = temp;

        return i + 1;

    }

    public static void printArray(int[] arr) {

        for (int num : arr) {

            System.out.print(num + " ");

        }

        System.out.println();

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter no. of elements:");

        int n = sc.nextInt();

        int[] arr = new int[n];

        System.out.println("Enter elements:");

        for (int i = 0; i < n; i++) {

            arr[i] = sc.nextInt();

        }

        int[] arrBubble = arr.clone();

        int[] arrQuick = arr.clone();

        System.out.println("Bubble Sort:");

        bubbleSort(arrBubble);

        printArray(arrBubble);

        System.out.println("Quick Sort:");

        quickSort(arrQuick, 0, arrQuick.length - 1);

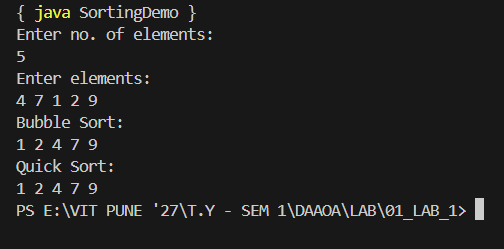
        printArray(arrQuick);

        sc.close();

    }

}

**OUTPUT:**

****

**Time and Space Complexity Analysis:**

**Bubble Sort:**

* Best Case Time Complexity: **O(n)** (when the array is already sorted)
* Average Case Time Complexity: **O(n²)**
* Worst Case Time Complexity: **O(n²)** (when the array is sorted in reverse order)
* Space Complexity: **O(1)** (in-place sorting with constant extra space)

**Quick Sort:**

* Best Case Time Complexity: **O(n log n)** (when partitioning divides array into nearly equal halves)
* Average Case Time Complexity: **O(n log n)**
* Worst Case Time Complexity: **O(n²)** (when partitioning is highly unbalanced, e.g., already sorted or reverse sorted with poor pivot choice)
* Space Complexity: **O(log n)** in best/average case (due to recursion stack)
* Space Complexity: **O(n)** in worst case (due to skewed recursion tree)

FUNCTION bubbleSort(A, n) // No new space; Space: +1 for n

FOR i FROM 0 TO n-2 // Time: +n-1 (≈+n)

FOR j FROM 0 TO n-i-2 // Time: +(n-i-1) per i; Total: \*n\*n

IF A[j] > A[j+1] // Time: +1

temp ← A[j] // Space: +1, Time: +1

A[j] ← A[j+1] // Time: +1

A[j+1] ← temp // Time: +1

ENDIF

ENDFOR

ENDFOR

ENDFUNCTION

// Space complexity for bubbleSort: O(1), as swaps use a single temp variable only[web:16].

FUNCTION quickSort(A, low, high) // Space: stack frames O(log n) avg., O(n) worst

IF low < high // Time: +1

pi ← partition(A, low, high) // Time: +n on avg.

quickSort(A, low, pi - 1) // Recursive call; Time: depends on split

quickSort(A, pi + 1, high) // Recursive call; Time: depends on split

ENDIF

ENDFUNCTION

FUNCTION partition(A, low, high) // Space: +1 for pivot, +1 for i, +1 for temp

pivot ← A[high] // Time: +1

i ← low - 1 // Time: +1

FOR j FROM low TO high - 1 // Time: +(high-low)

IF A[j] < pivot // Time: +1 per check

i ← i + 1 // Time: +1

temp ← A[i] // Space: +1, Time: +1

A[i] ← A[j] // Time: +1

A[j] ← temp // Time: +1

ENDIF

ENDFOR

temp ← A[i+1] // Space: +1, Time: +1

A[i+1] ← A[high] // Time: +1

A[high] ← temp // Time: +1

RETURN i+1 // Time: +1

ENDFUNCTION

FUNCTION printArray(A, n) // Space: +1 for temp (num)

FOR each num IN A // Time: +n

PRINT num // Time: +1 per iteration

ENDFOR

PRINT newline // Time: +1

ENDFUNCTION

FUNCTION main

DECLARE n // Space: +1

INPUT n // Time: +1

DECLARE array A of size n // Space: +n

FOR i FROM 0 TO n-1 // Time: +n

INPUT A[i] // Time: +1 per iteration

ENDFOR

DECLARE arrBubble as clone of A // Space: +n

DECLARE arrQuick as clone of A // Space: +n

PRINT "Bubble Sort:"

bubbleSort(arrBubble, n) // Time: +n^2, Space: +1

printArray(arrBubble, n) // Time: +n

PRINT "Quick Sort:"

quickSort(arrQuick, 0, n-1) // Avg Time: +n\*log(n), Space: O(log n)

printArray(arrQuick, n) // Time: +n

ENDFUNCTION

**Experiment No. 2**

**Title:**

Assignment Based on Divide and Conquer Strategy. (Implement Recursive and Non-Recursive Binary Search Algorithm using C++ or java. Determine Time and space complexity).

|  |  |  |  |
| --- | --- | --- | --- |
| Title of Experiment | CO Mapping | CO Statements | PO Mapping |
| Implement Recursive and Non-Recursive Binary Search Algorithm using C++ or java. Determine Time and space complexity | CO1, CO2, CO3 | Co1: To formulate computational problems in abstract and mathematically precise manner.  CO2: To design efficient algorithms for computational problems using appropriate algorithmic paradigm.  CO3:To analyze asymptotic complexity of the algorithm for a complex computational problem using suitable mathematical techniques | PO1, PO2, PO3, PO4, PO5, PO11 |

**Theory:**

**1. Introduction**

In this lab exercise, you will learn how to implement program to manage a integer numbers. The program will store this information in sorted order, and it will allow you to search number using binary search (both recursive and non-recursive methods).

**2. Theory**

**a. Binary Search**

Binary search is an efficient algorithm used to search for an element in a sorted list or array. It works by repeatedly dividing the search interval in half.

**Algorithm:**

1. Compare the target value with the middle element.
2. If the target matches the middle element, the search is successful.
3. If the target is less than the middle element, continue the search on the left half of the list.
4. If the target is greater than the middle element, continue the search on the right half of the list.
5. Repeat the process until the element is found or the search interval becomes empty.

**Time Complexity:**

**Best-case time complexity:**

* The best-case scenario occurs when the target element is the middle element of the array.
* In this case, the algorithm will find the element on the first iteration itself.
* Thus, the time complexity for the best case is O(1).

**Worst-case and Average-case time complexity**:

* In the worst-case scenario, the algorithm will continue splitting the array in half until the subarray is reduced to a single element.
* The number of iterations is proportional to the logarithm of the number of elements in the array because the array is halved with each iteration.
* Therefore, the time complexity for the worst and average cases is O(log n), where n is the number of elements in the array.

**Space Complexity of Binary Search**

The space complexity depends on the approach we use:

**Iterative Binary Search:**

* The iterative version of binary search does not use additional memory for recursion calls.
* The space complexity is O(1) because only a few variables (low, high, mid) are used to store the indices.

**Recursive Binary Search:**

* In the recursive version, each recursive call adds a new frame to the call stack.
* Since there are log n recursive calls (in the worst case), the space complexity is O(log n) due to the recursion stack.

**3. Lab Exercise**

**a. Program Requirements**

Your program should fulfill the following requirements:

1. Create an array of integers.
2. Implement a function to insert numbers into array.
3. Implement a function to search for number using binary search (both recursive and non-recursive methods).

**b. Step-by-Step Implementation**

Follow these steps to implement the program:

1. Create an empty array.
2. Implement a function to insert number into the array. Ensure that the array remains sorted.
3. Implement a recursive binary search function to search for a friend's mobile number.
4. Implement a non-recursive binary search function to achieve the same result.
5. Test the program with various scenarios.

**c. Testing the Program**

Test your program with various test cases to ensure it works correctly. Make sure to test:

* Inserting new element.
* Searching for existing element.
* Searching for non-existing element.

**Algorithm:**

**Data Structures:**

array to store numbers .

Functions:

**1. add\_Element(number):**

1. Create a new entry in the array.

2. Ensure the array remains sorted.

**2**. **recursive\_binary\_search(number):**

1. Initialize low = 0 and high = length of array - 1.

2. While low <= high:

a. Calculate the middle index: mid = (low + high) // 2.

b. If number == array[mid], return array[mid].

c. If name < array[mid], set high = mid - 1.

d. Otherwise, set low = mid + 1.

3. If the loop terminates without finding the name, return "Not found."

**3**. **non\_recursive\_binary\_search(name):**

1. Initialize low = 0 and high = length of array - 1.

2. While low <= high:

a. Calculate the middle index: mid = (low + high) // 2.

b. If array[mid] == number, return array[mid].

c. If name < array[mid], set high = mid - 1.

d. Otherwise, set low = mid + 1.

3. If the loop terminates without finding the name, return "Not found."

**4**. **main():**

1. Initialize an empty array.

2. Display a menu with the following options:

a. Insert number.

b. Search for a number (recursive).

c. Search for a number (non-recursive).

d. Exit.

3. Repeat the following until the user chooses to exit:

a. Prompt the user for their choice.

b. If the choice is 'a':

i. Prompt the user for a number.

c. If the choice is 'b':

i. Prompt the user for a number.

ii. Call recursive\_binary\_search(number) and display the result.

d. If the choice is 'c':

i. Prompt the user for number.

ii. Call non\_recursive\_binary\_search(number) and display the result.

e. If the choice is 'd', exit the program.

f. If the choice is invalid, display an error message.

4. End the program.

**4. Conclusion**

In this lab exercise, we learned how to create a program to manage and search number using binary search.

**Source Code, with description and with Output Need to be Uploaded to the VOLP**

import java.util.Scanner;

public class BinarySearchExample {

    public static int recursiveBinarySearch(int[] arr, int low, int high, int x) {

        if (high >= low) {

            int mid = low + (high - low) / 2;

            if (arr[mid] == x)

                return mid;

            if (arr[mid] > x)

                return recursiveBinarySearch(arr, low, mid - 1, x);

            return recursiveBinarySearch(arr, mid + 1, high, x);

        }

        return -1;

    }

    public static int iterativeBinarySearch(int[] arr, int x) {

        int low = 0, high = arr.length - 1;

        while (low <= high) {

            int mid = low + (high - low) / 2;

            if (arr[mid] == x)

                return mid;

            if (arr[mid] < x)

                low = mid + 1;

            else

                high = mid - 1;

        }

        return -1;

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        int n = sc.nextInt();

        int[] arr = new int[n];

        for (int i = 0; i < n; i++)

            arr[i] = sc.nextInt();

        int key = sc.nextInt();

        int resRec = recursiveBinarySearch(arr, 0, n - 1, key);

        System.out.println(resRec);

        int resIter = iterativeBinarySearch(arr, key);

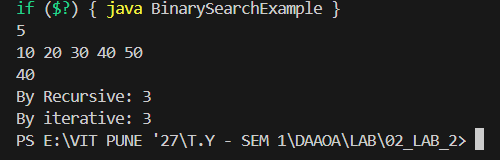
        System.out.println(resIter);

        sc.close();

    }

}

**OUTPUT:**

****

**Time and Space Complexity Analysis:**

**Recursive Binary Search:**

* Time Complexity:
  + Best Case: **O(1)** because if the middle element is the target, it takes just one comparison.
  + Average Case: **O(log n)** since the search space halves with each recursive call, reducing the problem size logarithmically.
  + Worst Case: **O(log n)** arises when the element is at one of the ends, requiring maximum recursive steps.
* Space Complexity: **O(log n)** due to the recursion call stack that holds at most one call per level down to log n levels.

**Iterative (Non-Recursive) Binary Search:**

* Time Complexity:
  + Best Case: **O(1)** when the middle element matches the target immediately.
  + Average Case: **O(log n)** because the halving of the search space continues iteratively until found.
  + Worst Case: **O(log n)** when the search narrows down to the edges of the array.
* Space Complexity: **O(1)** since only a few variables are used regardless of input size, with no recursion stack overhead.

FUNCTION recursiveBinarySearch(A, low, high, x) // Space: O(log n) for stack frames (recursion)

IF high >= low // Time: +1

mid ← low + (high - low) / 2 // Time: +1, Space: +1

IF A[mid] == x // Time: +1

RETURN mid // Time: +1

IF A[mid] > x // Time: +1

RETURN recursiveBinarySearch(A, low, mid-1, x) // Each call divides input; Time: \*log n

ELSE

RETURN recursiveBinarySearch(A, mid+1, high, x)// Each call divides input; Time: \*log n

RETURN -1 // Time: +1

ENDFUNCTION

// Recursive Binary Search: Time O(log n), Space O(log n) [web:26][web:27][web:28]

FUNCTION iterativeBinarySearch(A, x) // Space: O(1) extra variables

low ← 0 // Time: +1, Space: +1

high ← length(A) - 1 // Time: +1, Space: +1

WHILE low <= high // Time: \*log n (range halves every iteration)

mid ← low + (high - low) / 2 // Time: +1, Space: +1

IF A[mid] == x // Time: +1

RETURN mid // Time: +1

IF A[mid] < x // Time: +1

low ← mid + 1 // Time: +1

ELSE

high ← mid - 1 // Time: +1

RETURN -1 // Time: +1

ENDFUNCTION

// Iterative Binary Search: Time O(log n), Space O(1) [web:26][web:27][web:28]

FUNCTION main

DECLARE n // Space: +1

INPUT n // Time: +1

DECLARE array A of size n // Space: +n

FOR i = 0 TO n-1 // Time: +n

INPUT A[i] // Time: +1 per iteration

ENDFOR

INPUT key // Time: +1

resRec ← recursiveBinarySearch(A, 0, n-1, key) // Time: O(log n), Space: O(log n)

PRINT resRec // Time: +1

resIter ← iterativeBinarySearch(A, key) // Time: O(log n), Space: O(1)

PRINT resIter // Time: +1

ENDFUNCTION

**Complexity Analysis:**

|  |  |  |
| --- | --- | --- |
| **Type** | **Time Complexity** | **Space Complexity** |
| **Recursive** | O(log n) | O(log n) |
| **Iterative** | O(log n) | O(1) |

# Experiment Number: 03

**Title:** Assignment Based on Dynamic programming strategy. (Implement 0-1 Knapsack problem using cpp or java)

|  |  |  |  |
| --- | --- | --- | --- |
| **Title of Experimentation** | **CO**  **Mapping** | **CO-Statements** | **PO**  **Mapping** |
| Assignment Based on Dynamic programming strategy. (Implement 0-1 Knapsack problem using cpp or java) | CO1, CO2,C03 | To analyze asymptotic complexity of the algorithm for a complex computational problem using suitable mathematical techniques | PO1, PO2, PO3, PO4, PO5, PO11 |

**Theory:**

**0-1 Knapsack Problem** is a classic optimization problem:

* Given n items, each with a weight w[i] and a profit p[i], and a knapsack with capacity W.
* The goal is to maximize total profit by selecting items without exceeding capacity W.
* Each item can either be **taken (1)** or **not taken (0)** → hence “0-1 Knapsack.”

Dynamic Programming (DP) is used to solve this problem efficiently by avoiding recomputation.  
The key recurrence relation is:

dp[i][w]=max(dp[i−1][w],profit[i−1]+dp[i−1][w−weight[i−1]])

where:

* dp[i][w] = max profit using first i items with knapsack capacity w.

### **Input:**

* Number of items n
* Profit array p[n]
* Weight array w[n]
* Knapsack capacity W

### **Output:**

* Maximum profit achievable within capacity W.

### **Objective of Experiment:**

To implement and demonstrate how **Dynamic Programming** can be applied to solve the **0-1 Knapsack Problem** efficiently compared to recursive brute force.

### **Algorithm (Dynamic Programming – Bottom-Up):**

1. Initialize a DP table dp[n+1][W+1].
2. For each item i (1…n):
   * For each capacity w (1…W):
     + If weight[i-1] <= w, compute:

dp[i][w]=max⁡(dp[i−1][w], profit[i−1]+dp[i−1][w−weight[i−1]])dp[i][w] = \max(dp[i-1][w], \, profit[i-1] + dp[i-1][w - weight[i-1]])dp[i][w]=max(dp[i−1][w],profit[i−1]+dp[i−1][w−weight[i−1]])

* + - Else, inherit previous value:

dp[i][w]=dp[i−1][w]dp[i][w] = dp[i-1][w]dp[i][w]=dp[i−1][w]

1. Result is stored in dp[n][W].

### **Pseudo Code:**

function knapsack(profit[], weight[], n, W):

create dp[n+1][W+1]

for i = 0 to n:

for w = 0 to W:

if i == 0 or w == 0:

dp[i][w] = 0

else if weight[i-1] <= w:

dp[i][w] = max(profit[i-1] + dp[i-1][w - weight[i-1]],

dp[i-1][w])

else:

dp[i][w] = dp[i-1][w]

return dp[n][W]

### **Java Implementation:**

import java.util.Scanner;

public class KnapsackDP {

public static int knapsack(int[] profit, int[] weight, int n, int W) {

int[][] dp = new int[n + 1][W + 1];

for (int i = 0; i <= n; i++) {

for (int w = 0; w <= W; w++) {

if (i == 0 || w == 0) {

dp[i][w] = 0;

} else if (weight[i - 1] <= w) {

dp[i][w] = Math.max(profit[i - 1] + dp[i - 1][w - weight[i - 1]],

dp[i - 1][w]);

} else {

dp[i][w] = dp[i - 1][w];

}

}

}

return dp[n][W];

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter number of items: ");

int n = sc.nextInt();

int[] profit = new int[n];

int[] weight = new int[n];

System.out.println("Enter profits of items:");

for (int i = 0; i < n; i++) {

profit[i] = sc.nextInt();

}

System.out.println("Enter weights of items:");

for (int i = 0; i < n; i++) {

weight[i] = sc.nextInt();

}

System.out.print("Enter knapsack capacity: ");

int W = sc.nextInt();

int maxProfit = knapsack(profit, weight, n, W);

System.out.println("Maximum profit = " + maxProfit);

sc.close();

}

}

**Input:**

Enter number of items: 3

Enter profits of items:

60 100 120

Enter weights of items:

10 20 30

Enter knapsack capacity: 50

**Output:**

Maximum profit = 220

### **Flowchart (suggested structure):**

* **Start**
* Input: n, profits[], weights[], W
* Initialize DP table
* For each item i = 1 to n  
  → For each capacity w = 1 to W  
  → If item fits → choose max(include, exclude)  
  → Else inherit previous
* End loops
* Output dp[n][W]

**Source Code, with description and with Output Need to be Uploaded to the VOLP**

**Code:**

import java.util.Scanner;

public class Knapsack {

    public static int knapsack(int W, int wt[], int val[], int n) {

        int[][] K = new int[n + 1][W + 1];

        for (int i = 0; i <= n; i++) {

            for (int w = 0; w <= W; w++) {

                if (i == 0 || w == 0) {

                    K[i][w] = 0;

                } else if (wt[i - 1] <= w) {

                    K[i][w] = Math.max(val[i - 1] + K[i - 1][w - wt[i - 1]], K[i - 1][w]);

                } else {

                    K[i][w] = K[i - 1][w];

                }

            }

        }

        return K[n][W];

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter number of items:");

        int n = sc.nextInt();

        int[] val = new int[n];

        int[] wt = new int[n];

        System.out.println("Enter value and weight of each item:");

        for (int i = 0; i < n; i++) {

            val[i] = sc.nextInt();

            wt[i] = sc.nextInt();

        }

        System.out.println("Enter the capacity of the knapsack:");

        int W = sc.nextInt();

        int maxProfit = knapsack(W, wt, val, n);

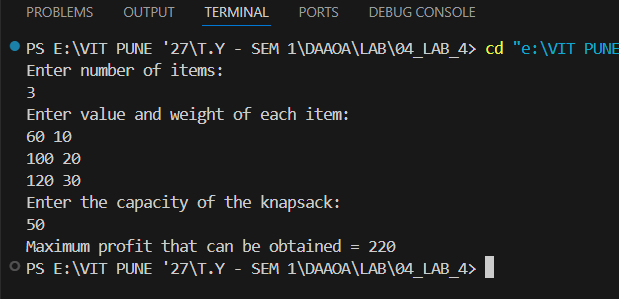
        System.out.println("Maximum profit that can be obtained = " + maxProfit);

        sc.close();

    }

}

**OUTPUT:**

****

**Time and Space Complexity Analysis:**

**Time Complexity**

* The algorithm fills an (n+1)×(W+1) table.
* Each cell computation takes constant time O(1)
* Total time complexity: **O(nW)** where n is item count and W is knapsack capacity.

**Space Complexity**

* Uses a 2D array Kof size (n+1)×(W+1)
* Requires space proportional to number of items and capacity.
* Total space complexity: **O(nW)**

**Pseudocode with Complexity Comments**

text

FUNCTION knapsack(W, wt, val, n)

DECLARE 2D array K of size (n+1) x (W+1) // Space: +(n+1)\*(W+1) = O(n\*W)

FOR i FROM 0 TO n // Time: +n+1

FOR w FROM 0 TO W // Time: +(W+1) per i; Total: \*n\*W

IF i == 0 OR w == 0 // Time: +1 per iteration

K[i][w] ← 0 // Time: +1

ELSE IF wt[i-1] <= w // Time: +1

K[i][w] ← MAX(val[i-1] + K[i-1][w - wt[i-1]], K[i-1][w]) // Time: +1 (max and addition)

ELSE

K[i][w] ← K[i-1][w] // Time: +1

ENDIF

ENDFOR

ENDFOR

RETURN K[n][W] // Time: +1 (return)

ENDFUNCTION

FUNCTION main

DECLARE scanner // Space: +1

PRINT "Enter number of items:" // Time: +1

INPUT n // Time: +1

DECLARE arrays val[n], wt[n] // Space: +n each = +2n total

PRINT "Enter value and weight of each item:" // Time: +1

FOR i FROM 0 TO n-1 // Time: +n

INPUT val[i], wt[i] // Time: +1 per read

ENDFOR

PRINT "Enter the capacity of the knapsack:" // Time: +1

INPUT W // Time: +1

maxProfit ← knapsack(W, wt, val, n) // Time: O(n\*W), Space: O(n\*W)

PRINT "Maximum profit that can be obtained = " + maxProfit // Time: +1

CLOSE scanner // Time: +1

ENDFUNCTION

**Complexity Explanation**

* **Time Complexity:** The nested loops iterate over each item (n) and capacity (W), so overall the time complexity is O(n×W))
* **Space Complexity:** The 2D DP table K requires O(n×W) space to store intermediate results.
* Input and output operations take linear time and constant extra space outside the storage arrays.
* All constant time operations (+1) occur within nested loops to build the solution table.

# Experiment Number: 04

**Title:** Assignment Based on Backtracking. (Implement N- Queen problem)

|  |  |  |  |
| --- | --- | --- | --- |
| **Title of Experimentation** | **CO**  **Mapping** | **CO-Statements** | **PO**  **Mapping** |
| Assignment Based on Backtracking. (Implement N- Queen problem) | CO1,CO2,CO3 | To establish NP-completeness of some decision problems, grasp the significance of the notion of NP-completeness and its relationship with intractability of the decision problems. |  |

### **Theory**

The **N-Queen problem** is a classical combinatorial problem:

* Place N queens on an N × N chessboard.
* Queens must be placed so that no two queens attack each other.
* A queen can attack another if they share the same **row, column, or diagonal**.

**Backtracking** is used:

* Place queens one by one in different rows.
* If placing a queen leads to a valid state, proceed to the next row.
* If a conflict occurs, backtrack and try the next column.

### **Input:**

* A single integer N → size of chessboard (and number of queens).

### **Output:**

* One or more valid configurations of N queens on the board.
* (Each solution shows positions where queens are placed safely.)

### **Objective of Experiment:**

To understand and implement **Backtracking** by solving the **N-Queen problem**, demonstrating how systematic trial and error with recursive backtracking helps solve constraint satisfaction problems.

### **Algorithm (Backtracking):**

1. Start with the first row.
2. Try placing a queen in each column of the current row.
3. If placing queen is **safe** (no other queen in same column/diagonal), place it.
4. Recurse to the next row.
5. If all queens are placed → print solution.
6. If no valid column exists in current row → backtrack (remove queen from previous row and try next possibility).

### **Pseudo Code:**

function solveNQueen(N):

create board[N][N] initialized to 0

if placeQueen(board, 0, N) == false:

print "No solution exists"

else:

print board

function placeQueen(board, row, N):

if row == N:

return true // all queens placed

for col = 0 to N-1:

if isSafe(board, row, col, N):

board[row][col] = 1

if placeQueen(board, row+1, N):

return true

board[row][col] = 0 // backtrack

return false

function isSafe(board, row, col, N):

check column above

check upper-left diagonal

check upper-right diagonal

if no conflicts → return true

else → return false

### **Java Implementation:**

import java.util.Scanner;

public class NQueen {

static int N;

// Function to print solution

static void printSolution(int board[][]) {

for (int i = 0; i < N; i++) {

for (int j = 0; j < N; j++) {

System.out.print((board[i][j] == 1 ? "Q " : ". "));

}

System.out.println();

}

System.out.println();

}

// Check if a queen can be placed at board[row][col]

static boolean isSafe(int board[][], int row, int col) {

// Check column

for (int i = 0; i < row; i++)

if (board[i][col] == 1)

return false;

// Check upper-left diagonal

for (int i = row, j = col; i >= 0 && j >= 0; i--, j--)

if (board[i][j] == 1)

return false;

// Check upper-right diagonal

for (int i = row, j = col; i >= 0 && j < N; i--, j++)

if (board[i][j] == 1)

return false;

return true;

}

// Recursive function to solve N-Queen problem

static boolean solveNQUtil(int board[][], int row) {

if (row == N) {

printSolution(board);

return true;

}

boolean res = false;

for (int col = 0; col < N; col++) {

if (isSafe(board, row, col)) {

board[row][col] = 1;

res = solveNQUtil(board, row + 1) || res;

board[row][col] = 0; // backtrack

}

}

return res;

}

static void solveNQ() {

int board[][] = new int[N][N];

if (!solveNQUtil(board, 0)) {

System.out.println("No solution exists");

}

}

public static void main(String args[]) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter value of N: ");

N = sc.nextInt();

solveNQ();

sc.close();

}

}

**Input:**

Enter value of N: 4

**Output:** (One possible solution)

. Q . .

. . . Q

Q . . .

. . Q .

. . Q .

Q . . .

. . . Q

. Q . .

### **Flowchart :**

* **Start**
* Input N
* Initialize empty board[N][N]
* Call recursive function placeQueen(row)
  + If row == N → print solution
  + Else try placing queen in each column:
    - If safe → place queen → recurse → if fails → backtrack
* Repeat until all solutions are found
* **Stop**

**Source Code, with description and with Output Need to be Uploaded to the VOLP**

-

**Code:**

import java.util.Scanner;

public class NQueen {

    public static boolean isSafe(int[][] board, int row, int col, int N) {

        for (int i = 0; i < col; i++)

            if (board[row][i] == 1)

                return false;

        for (int i = row, j = col; i >= 0 && j >= 0; i--, j--)

            if (board[i][j] == 1)

                return false;

        for (int i = row, j = col; i < N && j >= 0; i++, j--)

            if (board[i][j] == 1)

                return false;

        return true;

    }

    public static boolean solveNQueen(int[][] board, int col, int N) {

        if (col >= N)

            return true;

        for (int i = 0; i < N; i++) {

            if (isSafe(board, i, col, N)) {

                board[i][col] = 1;

                if (solveNQueen(board, col + 1, N))

                    return true;

                board[i][col] = 0;

            }

        }

        return false;

    }

    public static void printBoard(int[][] board, int N) {

        for (int i = 0; i < N; i++) {

            for (int j = 0; j < N; j++) {

                System.out.print(board[i][j] + " ");

            }

            System.out.println();

        }

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the size of the board (N):");

        int N = sc.nextInt();

        int[][] board = new int[N][N];

        if (solveNQueen(board, 0, N)) {

            System.out.println("Solution exists:");

            printBoard(board, N);

        } else {

            System.out.println("Solution does not exist");

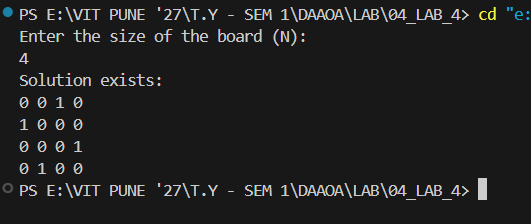
        }

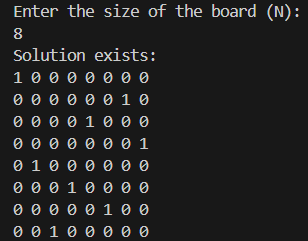
        sc.close();

    }

}

**OUTPUT:**

****

****

**Time and Space Complexity Analysis:**

## Time Complexity Analysis

* In worst case, the algorithm tries to place queens in all rows for each of the N columns.
* Possible configurations to explore: N^N
* Pruning reduces number of checks, but worst case remains O(N!) due to permutations of placing queens per row without conflicts.
* Hence time complexity is approximately **O(N!)**

## Space Complexity Analysis

* Uses an N×N times N×N board to store queen positions.
* Recursion stack depth can go up to N (one for each column).
* Total space complexity is **O(N^2)** for the board plus recursion call stack, which is **O(N)**
* Overall space complexity is **O(N^2)**

**Pseudocode with Complexity Comments**

text

FUNCTION knapsack(W, wt, val, n)

DECLARE 2D array K of size (n+1) x (W+1) // Space: +(n+1)\*(W+1) = O(n\*W)

FOR i FROM 0 TO n // Time: +n+1

FOR w FROM 0 TO W // Time: +(W+1) per i; Total: \*n\*W

IF i == 0 OR w == 0 // Time: +1 per iteration

K[i][w] ← 0 // Time: +1

ELSE IF wt[i-1] <= w // Time: +1

K[i][w] ← MAX(val[i-1] + K[i-1][w - wt[i-1]], K[i-1][w]) // Time: +1 (max and addition)

ELSE

K[i][w] ← K[i-1][w] // Time: +1

ENDIF

ENDFOR

ENDFOR

RETURN K[n][W] // Time: +1 (return)

ENDFUNCTION

FUNCTION main

DECLARE scanner // Space: +1

PRINT "Enter number of items:" // Time: +1

INPUT n // Time: +1

DECLARE arrays val[n], wt[n] // Space: +n each = +2n total

PRINT "Enter value and weight of each item:" // Time: +1

FOR i FROM 0 TO n-1 // Time: +n

INPUT val[i], wt[i] // Time: +1 per read

ENDFOR

PRINT "Enter the capacity of the knapsack:" // Time: +1

INPUT W // Time: +1

maxProfit ← knapsack(W, wt, val, n) // Time: O(n\*W), Space: O(n\*W)

PRINT "Maximum profit that can be obtained = " + maxProfit // Time: +1

CLOSE scanner // Time: +1

ENDFUNCTION

**Complexity Explanation**

* **Time Complexity:** The nested loops iterate over each item (n) and capacity (W), so overall the time complexity is O(n×W)
* **Space Complexity:** The 2D DP table K requires O(n×W) space to store intermediate results.
* Input and output operations take linear time and constant extra space outside the storage arrays.
* All constant time operations (+1) occur within nested loops to build the solution table.

# Experiment Number: 05

**Title:** Assignment Based on Greedy strategy. (Implement Huffman encoding algorithm)

|  |  |  |  |
| --- | --- | --- | --- |
| **Title of Experimentation** | **CO**  **Mapping** | **CO-Statements** | **PO**  **Mapping** |
| Assignment Based on Greedy strategy. (Implement Huffman encoding algorithm) | CO2, CO3 | CO2: To design efficient algorithms for computational problems using appropriate algorithmic paradigm  CO3: To analyze asymptotic complexity of the algorithm for a complex computational problem using suitable mathematical techniques | PO2, PO3, PO4 |

**Theory:**

* Greedy Strategy is a paradigm where local optimal choices are made at each step to find a global optimum.
* Huffman Encoding is a greedy algorithm that assigns variable-length binary codes to characters:
* Shorter codes → frequent characters.
* Longer codes → rare characters.
* Applications: Data compression in ZIP, JPEG, MP3, etc.
* **Time Complexity:**

1.Building priority queue: O(n)

2.Extract & merge steps: O(n log n)

Total: **O(n log n)**

**Input:**

Characters: {a, b, c, d, e, f}

Frequencies: {5, 9, 12, 13, 16, 45}

Huffman Codes (one possible solution):

a : 1100

b : 1101

c : 100

d : 101

e : 111

f : 0

Encoded String (for "face"):

f a c e → 0 1100 100 111 = 01100100111

**Output:**

Huffman Codes: {'f': '0', 'c': '100', 'd': '101', 'a': '1100', 'b': '1101', 'e': '111'}

Encoded: 01100100111

Decoded: face

**Objective of Experiment:**

* To implement Huffman Encoding using the Greedy strategy.
* To compress and decompress text data.
* To analyse space efficiency compared to fixed-length encoding

**Flow Chart/Pseudo Code/Algorithm:**

Algorithm:

1.Create a priority queue (min-heap) containing all characters with their frequencies.

2.While more than one node exists in the heap:

* Extract two nodes with the smallest frequency.
* Create a new internal node with these two as children.
* Insert the new node back into the heap.

3.The remaining node is the root of the Huffman tree.

4.Traverse the tree:

* Assign 0 for the left edge, 1 for the right edge.
* Generate codes for each character.

5.Encode the input string using generated codes.

6.Decode the encoded string using the Huffman tree.

**Flowchart:**

(You can insert a flowchart here showing recursive splitting and combining steps)

**Source Code, with description and with Output Need to be Uploaded to the VOLP**

**Code:**

import java.util.\*;

class Node implements Comparable<Node> {

    char ch;

    int freq;

    Node left, right;

    Node(char ch, int freq) {

        this.ch = ch;

        this.freq = freq;

    }

    Node(int freq, Node left, Node right) {

        this.ch = '\0';

        this.freq = freq;

        this.left = left;

        this.right = right;

    }

    public int compareTo(Node other) {

        return this.freq - other.freq;

    }

}

public class HuffmanCoding {

    public static void generateCodes(Node root, String code, Map<Character, String> codes) {

        if (root == null)

            return;

        if (root.left == null && root.right == null) {

            codes.put(root.ch, code);

        }

        generateCodes(root.left, code + "0", codes);

        generateCodes(root.right, code + "1", codes);

    }

    public static Map<Character, String> buildHuffmanTree(char[] chars, int[] freq) {

        PriorityQueue<Node> pq = new PriorityQueue<>();

        for (int i = 0; i < chars.length; i++) {

            pq.add(new Node(chars[i], freq[i]));

        }

        while (pq.size() > 1) {

            Node left = pq.poll();

            Node right = pq.poll();

            Node merged = new Node(left.freq + right.freq, left, right);

            pq.add(merged);

        }

        Node root = pq.poll();

        Map<Character, String> codes = new HashMap<>();

        generateCodes(root, "", codes);

        return codes;

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the number of characters:");

        int n = sc.nextInt();

        char[] chars = new char[n];

        int[] freq = new int[n];

        System.out.println("Enter characters:");

        for (int i = 0; i < n; i++) {

            chars[i] = sc.next().charAt(0);

        }

        System.out.println("Enter their frequencies:");

        for (int i = 0; i < n; i++) {

            freq[i] = sc.nextInt();

        }

        Map<Character, String> huffmanCodes = buildHuffmanTree(chars, freq);

        System.out.println("Huffman Codes:");

        for (char c : chars) {

            System.out.println(c + ": " + huffmanCodes.get(c));

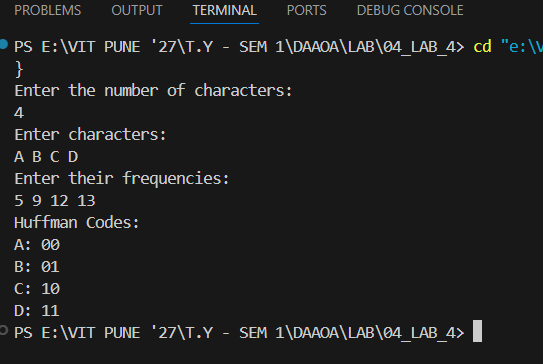
        }

        sc.close();

    }

}

**OUTPUT:**

****

**Time and Space Complexity Analysis:**

## Time Complexity

* Building the priority queue takes O(n) for n characters.
* Each merge operation occurs n−1 times, and each operation involves extracting two min nodes and adding one new node, each costing O(logn) because of the priority queue.
* Total time complexity: **O(nlogn)**.

## Space Complexity

* The tree stores all nodes including merged internal nodes; a total of 2n−1 nodes for n characters.
* Codes are stored for each character.
* Priority queue space: O(n)
* Overall space complexity: **O(n)**

## Pseudocode with Complexity Comments

text

CLASS Node

VARIABLE ch, freq

VARIABLE left, right

FUNCTION Node(ch, freq) // Constructor for leaf node

this.ch ← ch // Time: +1

this.freq ← freq // Time: +1

ENDFUNCTION

FUNCTION Node(freq, left, right) // Constructor for internal node

this.ch ← '\0' // Time: +1

this.freq ← freq // Time: +1

this.left ← left // Time: +1

this.right ← right // Time: +1

ENDFUNCTION

FUNCTION compareTo(other) // For priority queue ordering

RETURN this.freq - other.freq // Time: +1

ENDFUNCTION

ENDCLASS

FUNCTION generateCodes(root, code, codes) // Generate Huffman codes by tree traversal

IF root == null // Time: +1 per call

RETURN

IF root.left == null AND root.right == null // Leaf node // Time: +1 per leaf

codes[root.ch] ← code // Store code for character // Time: +1

generateCodes(root.left, code + "0", codes) // Traverse left subtree // Time: O(n) overall

generateCodes(root.right, code + "1", codes) // Traverse right subtree // Time: O(n) overall

ENDFUNCTION

FUNCTION buildHuffmanTree(chars, freq) // chars, freq: arrays of size n

DECLARE priorityQueue pq // Space: +n for storing nodes

FOR i = 0 TO n-1 // Time: +n

pq.add(new Node(chars[i], freq[i])) // Time: O(log n) per insertion

WHILE pq.size() > 1 // Time: O(n log n) for building tree

left ← pq.poll() // Extract min node // Time: O(log n)

right ← pq.poll() // Extract next min // Time: O(log n)

merged ← new Node(left.freq + right.freq, left, right) // Create new node // Time: +1

pq.add(merged) // Insert merged node // Time: O(log n)

root ← pq.poll() // Root of Huffman Tree // Time: +1

DECLARE codes map // Space: +n for character codes

generateCodes(root, "", codes) // Time: O(n) to generate codes

RETURN codes // Map char → code // Time: +1

ENDFUNCTION

FUNCTION main

DECLARE scanner // Space: +1

PRINT "Enter the number of characters:" // Time: +1

INPUT n // Time: +1

DECLARE chars[n], freq[n] // Space: +2n

PRINT "Enter characters:" // Time: +1

FOR i = 0 TO n-1 // Time: +n

INPUT chars[i] // Time: +1 per input

PRINT "Enter their frequencies:" // Time: +1

FOR i = 0 TO n-1 // Time: +n

INPUT freq[i] // Time: +1 per input

huffmanCodes ← buildHuffmanTree(chars, freq) // Time: O(n log n), Space: O(n)

PRINT "Huffman Codes:" // Time: +1

FOR each c IN chars // Time: +n

PRINT c + ": " + huffmanCodes[c] // Time: +1 per output

CLOSE scanner // Time: +1

ENDFUNCTION

# Experiment Number: 06

**Title:** Assignment Based on Dynamic programming strategy to implement traveling salesman problem

|  |  |  |  |
| --- | --- | --- | --- |
| **Title of Experimentation** | **CO**  **Mapping** | **CO-Statements** | **PO**  **Mapping** |
| Assignment Based on Dynamic programming strategy to implement traveling salesman problem | CO2, CO3, CO6 | CO2: To design efficient algorithms for computational problems using appropriate algorithmic paradigm  CO3: To analyze asymptotic complexity of the algorithm for a complex computational problem using suitable mathematical techniques.  CO3: To analyze asymptotic complexity of the algorithm for a complex computational problem using suitable mathematical techniques | PO2, PO3, PO4, PO5 |

**Theory:**

* Dynamic Programming (DP): A problem-solving technique where a problem is divided into overlapping subproblems, and results of subproblems are reused (memorization).
* Traveling Salesman Problem (TSP):
* Problem: A salesman must visit every city exactly once and return to the starting city with minimum travel cost.
* Brute force approach → O(n!) complexity (checking all permutations).
* DP (Held-Karp Algorithm): Solves TSP in O(n² · 2ⁿ) time by storing results of subproblems using bit masking.
* Applications: Vehicle routing, logistics, circuit design, route optimization, DNA sequencing

**Input:**

Number of cities: 4

Cost Matrix:

0 10 15 20

10 0 35 25

15 35 0 30

20 25 30 0

**Output:**

Minimum travel cost: 80

Path: 0 → 1 → 3 → 2 → 0

**Objective of Experiment:**

* To apply Dynamic Programming strategy to solve the Traveling Salesman Problem.
* To compare brute force vs. DP in terms of computational complexity.
* To understand the importance of overlapping subproblems and optimal substructure in TSP.

**Flow Chart/Pseudo Code/Algorithm:**

Algorithm (Held-Karp DP Algorithm for TSP)

1. Let dp[mask][i] = minimum cost to visit the set of cities represented by mask ending at city i.
2. Initialize dp[1<<i][i] = cost[start][i].
3. For each subset of cities (represented as bitmask):
   * For each city i in subset:
     + For each city j in subset, j != i:
     + dp[mask][i] = min(dp[mask][i], dp[mask ^ (1<<i)][j] + cost[j][i])
4. Answer = min(dp[(1<<n)-1][i] + cost[i][start]) for all i.

**Flowchart:**

(You can insert a flowchart here showing recursive splitting and combining steps)

**Source Code, with description and with Output Need to be Uploaded to the VOLP**

**Code:**

import java.util.Scanner;

public class TravellingSalesman {

    static int tsp(int[][] dist, int mask, int pos, int n, int[][] memo) {

        if (mask == (1 << n) - 1) {

            return dist[pos][0];

        }

        if (memo[pos][mask] != -1) {

            return memo[pos][mask];

        }

        int ans = Integer.MAX\_VALUE;

        for (int city = 0; city < n; city++) {

            if ((mask & (1 << city)) == 0) {

                int newAns = dist[pos][city] + tsp(dist, mask | (1 << city), city, n, memo);

                ans = Math.min(ans, newAns);

            }

        }

        memo[pos][mask] = ans;

        return ans;

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter number of cities:");

        int n = sc.nextInt();

        int[][] dist = new int[n][n];

        System.out.println("Enter distance matrix:");

        for (int i = 0; i < n; i++) {

            for (int j = 0; j < n; j++) {

                dist[i][j] = sc.nextInt();

            }

        }

        int[][] memo = new int[n][1 << n];

        for (int i = 0; i < n; i++) {

            for (int j = 0; j < (1 << n); j++) {

                memo[i][j] = -1;

            }

        }

        int result = tsp(dist, 1, 0, n, memo);

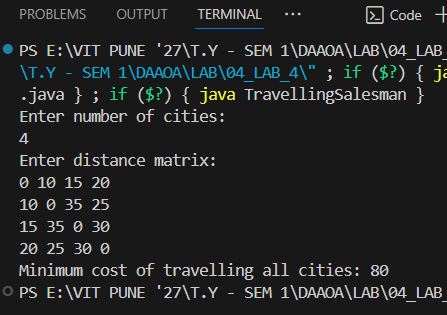
        System.out.println("Minimum cost of travelling all cities: " + result);

        sc.close();

    }

}

**OUTPUT:**

****

**Time and Space Complexity Analysis:**

## Time Complexity

* Number of subproblems: n×2^n (position × visited subset)
* For each subproblem, we check up to n cities.
* Overall time complexity: **O(n^2×2^n)**

## Space Complexity

* Memoization table size: n×2^n
* Extra space for recursion stack: O(n)
* Overall space complexity: **O(n×2^n)**

## Pseudocode with Complexity Comments

text

FUNCTION tsp(dist, mask, pos, n, memo)

IF mask == (1 << n) - 1 // Check if all cities visited; Time: +1

RETURN dist[pos][0] // Return distance back to start; Time: +1

IF memo[pos][mask] != -1 // Memoization check; Time: +1

RETURN memo[pos][mask]

ans ← INFINITY // Time: +1

FOR city FROM 0 TO n-1 // Time: +n (check all possible next cities)

IF (mask & (1 << city)) == 0 // If city not visited; Time: +1 per city

newAns ← dist[pos][city] + tsp(dist, mask | (1 << city), city, n, memo) // Recursion; Time: exponential by subproblem count

ans ← MIN(ans, newAns) // Time: +1 (compare and assign)

memo[pos][mask] ← ans // Store computed answer; Time: +1

RETURN ans // Return minimum cost found; Time: +1

ENDFUNCTION

FUNCTION main

DECLARE scanner // Space: +1

PRINT "Enter number of cities:" // Time: +1

INPUT n // Time: +1

DECLARE 2D array dist[n][n] // Space: +n^2

PRINT "Enter distance matrix:" // Time: +1

FOR i FROM 0 TO n-1 // Time: +n

FOR j FROM 0 TO n-1 // Time: +n per i; Total: \*n^2

INPUT dist[i][j] // Time: +1 per input

DECLARE 2D array memo[n][1 << n] // Space: +n \* 2^n

FOR i FROM 0 TO n-1 // Time: +n

FOR j FROM 0 TO (1 << n) - 1 // Time: +2^n per i; Total: \*n\*2^n

memo[i][j] ← -1 // Time: +1 per initialization

result ← tsp(dist, 1, 0, n, memo) // Recursive call; Time: O(n^2 \* 2^n)

PRINT "Minimum cost of travelling all cities: " + result // Time: +1

CLOSE scanner // Time: +1

ENDFUNCTION

## Complexity Explanation

* **Time Complexity:** There are n×2n possible state combinations of (current city, visited subset). Each state recursively calls up to n cities, so total time complexity is O(n^2 2^n)
* **Space Complexity:** The memoization table stores results for each (city, mask) state: O(n 2^n)space.
* Input distance matrix uses O(n^2) space, but this is dominated by memo space.
* The recursive recursion stack depth at most O(n), which is less significant compared to memo.