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**

**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**

# 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**

# 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**

# 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**

# 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**