PRESIDENCY UNIVERSITY

GAIN MORE KNOWLEDGE
REACH GREATER HEIGHTS

**Department of Computer Science and Engineering**

**Academic Year 2025 – 2026**

**Even Semester**

# Lab Manual

---

## CSE2263 ANALYSIS OF ALGORITHMS

Prepared by,Dr. S. Aarif Ahamed-Asst. Prof-SCSE

**TABLE OF CONTENTS**

## Lab Sheet 1: Measuring Running Time of an Algorithm

**Aim:**

To measure the execution time of different algorithms using the clock() function in Turbo C, and to study the effect of algorithm complexity on running time for:

- Simple loop (O(n))
- Nested loop ($O(n^2)$)
- Recursive function (Factorial)

**Procedure:**

- Open Turbo C and create a new C program.
- Include the required header files.
- Write the program to record the start time using clock().
- Execute the given loop or recursive function.
- Record the end time after execution.
- Calculate the execution time using CLOCKS_PER_SEC.
- Compile and run the program.
- Observe the time taken displayed on the screen.

**Program 1: Simple Loop**

```
#include <stdio.h>

#include <time.h>

int main() {

   int i, n = 1000000;

   clock_t start, end;

   double cpu_time;

   start = clock();

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

   end = clock();

   cpu_time = ((double)(end - start)) / CLOCKS_PER_SEC;

   printf("Time taken: %f seconds\n", cpu_time);

   return 0;

}
```

**Program 2: Measuring Running Time of Nested Loops ($O(n^2)$)**

```
#include <stdio.h>

#include <time.h>
```

```
int main() {
    int i, j;
    int n = 2000;
    clock_t start, end;
    double cpu_time;
    start = clock();
    for(i = 0; i < n; i++) {
        for(j = 0; j < n; j++) {
            // Empty loop body
        }
    }
    end = clock();
    cpu_time = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Time taken for O(n^2) loop: %f seconds\n", cpu_time);
    return 0;
}
```

**Program 3: Measuring Running Time of Recursive Function (Factorial)**

```
#include <stdio.h>
#include <time.h>
long factorial(int n) {
    if(n == 0)
        return 1;
    return n * factorial(n - 1);
}
int main() {
    int n = 20;
    clock_t start, end;
    double cpu_time;
    start = clock();
    factorial(n);
```

```
    end = clock();
    cpu_time = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Time taken for recursive factorial: %f seconds\n", cpu_time);
    return 0;
}
```

**Result:**

The execution time of different algorithms was measured successfully using Turbo C. It was observed that the running time increases as the algorithm complexity increases from O(n) to $O(n^2)$.

## Lab Sheet 2: Compare Running Time of Algorithms

**Aim:**

To compare the running time of two algorithms for calculating the sum of first n natural numbers using:

- Loop-based method
- Formula-based method

**Procedure:**

- Open Turbo C and create a new C program.
- Include the required header files such as stdio.h and time.h.
- Write a program to calculate the sum of first n natural numbers using a loop-based approach.
- Use the clock() function to record the start time before executing the loop.
- Record the end time after the loop execution and calculate the total execution time using CLOCKS_PER_SEC.
- Write another program to calculate the sum using the formula-based approach.
- Measure the execution time of the formula-based method in the same manner.
- Compile and run both programs separately.
- Observe and compare the execution time obtained for both approaches.

**Program 1: Loop-Based Approach (O(n))**

```
#include <stdio.h>
#include <time.h>
int main() {
    int n = 1000000;
    long sum = 0;
    clock_t start, end;
```

```
    start = clock();
    for(int i = 1; i <= n; i++)
        sum += i;
    end = clock();
    printf("Loop Method Time: %f seconds\n",
           (double)(end-start)/CLOCKS_PER_SEC);
    return 0;
}
```

**Program 2: Formula-Based Approach (O(1))**

```
#include <stdio.h>
#include <time.h>
int main() {
    int n = 1000000;
    long sum;
    clock_t start, end;
    start = clock();
    sum = n * (n + 1) / 2;
    end = clock();
    printf("Formula Method Time: %f seconds\n",
           (double)(end-start)/CLOCKS_PER_SEC);
    return 0;
}
```

**Result:**

The formula-based approach executed faster than the loop-based approach, proving that algorithms with lower time complexity perform better.

## Lab Sheet 3: Implement Sorting Algorithms – Bubble Sort and Selection Sort

**Aim:**

To implement Bubble Sort and Selection Sort algorithms in C and to understand their working and time complexity.

**Procedure:**

• Open Turbo C and create a new C program.

• Include the required header file stdio.h.

• Read the number of elements and array values from the user.

• Implement the Bubble Sort algorithm to sort the array in ascending order.

• Display the sorted array obtained using Bubble Sort.

• Implement the Selection Sort algorithm on the same set of elements.

• Display the sorted array obtained using Selection Sort.

• Compile and run the program.

• Observe and compare the results of both sorting techniques.

**Program: Bubble Sort and Selection Sort**

```
#include <stdio.h>
#include <time.h>
// Function to perform Bubble Sort
void bubbleSort(int a[], int n) {
   int i, j, temp;
   for(i = 0; i < n - 1; i++) {
      for(j = 0; j < n - i - 1; j++) {
         if(a[j] > a[j + 1]) {
            temp = a[j];
            a[j] = a[j + 1];
            a[j + 1] = temp;
         }
      }
   }
}
// Function to perform Selection Sort
void selectionSort(int a[], int n) {
   int i, j, minIndex, temp;
```

```
    for(i = 0; i < n - 1; i++) {
        minIndex = i;
        for(j = i + 1; j < n; j++) {
            if(a[j] < a[minIndex])
                minIndex = j;
        }
        temp = a[i];
        a[i] = a[minIndex];
        a[minIndex] = temp;
    }
}
// Function to display array
void displayArray(int a[], int n) {
    for(int i = 0; i < n; i++)
        printf("%d ", a[i]);
    printf("\n");
}
int main() {
    int a[20], n, original[20];
    clock_t start, end;
    double time_taken;
    // Input array elements
    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter array elements:\n");
    for(int i = 0; i < n; i++) {
        scanf("%d", &a[i]);
        original[i] = a[i]; // store original array for fair comparison
    }
```

```
    // Bubble Sort
    start = clock();
    bubbleSort(a, n);
    end = clock();
    time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("\nSorted array using Bubble Sort:\n");
    displayArray(a, n);
    printf("Time taken by Bubble Sort: %f seconds\n", time_taken);
    // Restore original array for Selection Sort
    for(int i = 0; i < n; i++)
        a[i] = original[i];
    // Selection Sort
    start = clock();
    selectionSort(a, n);
    end = clock();
    time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("\nSorted array using Selection Sort:\n");
    displayArray(a, n);
    printf("Time taken by Selection Sort: %f seconds\n", time_taken);
    return 0;
}
```

**Result:**

Bubble Sort and Selection Sort algorithms were implemented successfully. Both algorithms sorted the given array correctly. It was observed that both algorithms have $O(n^2)$ time complexity and are suitable for small input sizes.

## Lab Sheet 4: Compare Searching Algorithms – Linear Search and Binary Search

**Aim:**

To implement Linear Search and Binary Search algorithms in C and to compare their performance for searching an element in an array.

**Procedure:**

• Open Turbo C and create a new C program.

• Include the required header file stdio.h.

• Read the number of elements and array values from the user.

• Read the search key to be found.

• Implement the Linear Search algorithm to search for the key in the array.

• Display whether the element is found and its position.

• Sort the array before applying Binary Search.

• Implement the Binary Search algorithm on the sorted array.

• Display the search result obtained using Binary Search.

• Compile and run the program.

• Observe and compare the results of both searching techniques.

**Program: Linear Search and Binary Search**

```
#include <stdio.h>
#include <time.h>
// Linear Search Function
int linearSearch(int a[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (a[i] == key)
            return i;
    }
    return -1;
}
// Binary Search Function
int binarySearch(int a[], int n, int key) {
    int low = 0, high = n - 1, mid;
    while (low <= high) {
        mid = (low + high) / 2;
        if (a[mid] == key)
            return mid;
        else if (a[mid] < key)
```

```
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
}
// Function to sort array (for binary search)
void sortArray(int a[], int n) {
    int temp;
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (a[i] > a[j]) {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
}
int main() {
    int a[20], n, key, pos;
    clock_t start, end;
    double time_taken;
    // Input array
    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter array elements:\n");
    for (int i = 0; i < n; i++)
        scanf("%d", &a[i]);
    printf("Enter element to search: ");
```

```
    scanf("%d", &key);
    // Linear Search
    start = clock();
    pos = linearSearch(a, n, key);
    end = clock();
    time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
    if (pos != -1)
        printf("Linear Search: Element found at position %d\n", pos + 1);
    else
        printf("Linear Search: Element not found\n");
    printf("Time taken by Linear Search: %f seconds\n", time_taken);
    // Sort array for Binary Search
    sortArray(a, n);
    // Binary Search
    start = clock();
    pos = binarySearch(a, n, key);
    end = clock();
    time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
    if (pos != -1)
        printf("Binary Search: Element found at position %d\n", pos + 1);
    else
        printf("Binary Search: Element not found\n");
    printf("Time taken by Binary Search: %f seconds\n", time_taken);
    return 0;
}
```

**Result:**

Linear Search and Binary Search algorithms were implemented successfully. Linear Search works on both sorted and unsorted arrays but takes more time for large inputs, whereas Binary Search is faster with O(log n) time complexity but requires the array to be sorted.

**Lab Sheet 5: Compare Sorting Algorithms – Insertion Sort and Merge Sort**

**Aim:**

To implement Insertion Sort and Merge Sort algorithms in C and to compare their performance on the same set of input data.

**Procedure:**

• Open Turbo C and create a new C program.

• Include the required header file stdio.h.

• Read the number of elements and array values from the user.

• Implement the Insertion Sort algorithm and display the sorted array.

• Implement the Merge Sort algorithm using divide and conquer technique.

• Display the sorted array obtained using Merge Sort.

• Compile and run the program.

• Observe the execution and compare the performance of both algorithms.

**Program: Insertion Sort and Merge Sort**

```
#include <stdio.h>
#include <time.h>
void insertionSort(int a[], int n) {
   int i, key, j;
   for(i = 1; i < n; i++) {
      key = a[i];
      j = i - 1;
      while(j >= 0 && a[j] > key) {
         a[j + 1] = a[j];
         j--;
      }
      a[j + 1] = key;
   }
}
void merge(int a[], int low, int mid, int high) {
   int i = low, j = mid + 1, k = 0;
```

```
    int temp[100];
    while(i <= mid && j <= high) {
        if(a[i] < a[j])
            temp[k++] = a[i++];
        else
            temp[k++] = a[j++];
    }
    while(i <= mid)
        temp[k++] = a[i++];
    while(j <= high)
        temp[k++] = a[j++];

    for(i = low, k = 0; i <= high; i++, k++)
        a[i] = temp[k];
}
void mergeSort(int a[], int low, int high) {
    int mid;
    if(low < high) {
        mid = (low + high) / 2;
        mergeSort(a, low, mid);
        mergeSort(a, mid + 1, high);
        merge(a, low, mid, high);
    }
}
int main() {
    int a[100], b[100], n, i;
    clock_t start, end;
    double time_insert, time_merge;
    printf("Enter number of elements: ");
    scanf("%d", &n);
```

```
    printf("Enter array elements:\n");
    for(i = 0; i < n; i++) {
        scanf("%d", &a[i]);
        b[i] = a[i];    // Copy same input for Merge Sort
    }
    start = clock();
    insertionSort(a, n);
    end = clock();
    time_insert = (double)(end - start) / CLOCKS_PER_SEC;
    start = clock();
    mergeSort(b, 0, n - 1);
    end = clock();
    time_merge = (double)(end - start) / CLOCKS_PER_SEC;
    printf("\nSorted array using Insertion Sort:\n");
    for(i = 0; i < n; i++)
        printf("%d ", a[i]);
    printf("\n\nSorted array using Merge Sort:\n");
    for(i = 0; i < n; i++)
        printf("%d ", b[i]);
    printf("\n\nInsertion Sort Time: %f seconds", time_insert);
    printf("\nMerge Sort Time: %f seconds\n", time_merge);
    return 0;
}
```

**Result:**

Insertion Sort and Merge Sort algorithms were implemented successfully. Insertion Sort is efficient for small input sizes, while Merge Sort performs better for large datasets with O(n log n) time complexity.

# Lab Sheet 6: Quick Sort and Analysis of Pivot Selection

**Aim:**
To implement the Quick Sort algorithm in C, analyze the effect of different pivot selection strategies on performance, and display the sorted array along with the execution time.

**Procedure:**

1. Open Turbo C and create a new C program.
2. Include the required header files (stdio.h, time.h).
3. Initialize an array with sample data.
4. Implement Quick Sort using a chosen pivot strategy (first element, last element, or middle element).
5. Record the start time using clock().
6. Sort the array using Quick Sort.
7. Record the end time and calculate the CPU time.
8. Display the sorted array and the time taken.
9. Repeat the steps using different pivot strategies to observe the effect on execution time.

**Program: Quick Sort with First Element as Pivot**

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

void swap(int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}

int partition(int arr[], int low, int high) {
   int pivot = arr[low]; // pivot as first element
   int i = low + 1;
   int j = high;
   while (1) {
      while (i <= high && arr[i] <= pivot) i++;
      while (arr[j] > pivot) j--;
      if (i < j) swap(&arr[i], &arr[j]);
      else break;
   }
   swap(&arr[low], &arr[j]);
   return j;
}

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);
   }
}

int main() {
   int arr[] = {24, 17, 12, 22, 19, 15, 20};
   int n = sizeof(arr)/sizeof(arr[0]);
   clock_t start, end;
```

```
    double cpu_time;

    start = clock();
    quickSort(arr, 0, n-1);
    end = clock();
    cpu_time = ((double)(end - start)) / CLOCKS_PER_SEC;

    printf("Quick Sorted array: ");
    for(int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\nTime taken by Quick Sort: %f seconds\n", cpu_time);

    return 0;
}
```

**Result:**

The Quick Sort algorithm successfully sorted the array: 12 15 17 19 20 22 24. The execution time was recorded, showing efficient performance, and pivot choice affects sorting speed.

## Lab Sheet 7: Implementing Factorial and Coin Change Problem using Dynamic Programming

**Aim:**

To implement factorial calculation and the coin change problem using Dynamic Programming in C, display the results, and measure the execution time.

**Procedure:**

- Open Turbo C and create a new C program.
- Include required header files (stdio.h, time.h).
- Implement factorial calculation using bottom-up dynamic programming.
- Implement the coin change problem using dynamic programming to find the number of ways to make change for a given amount.
- Record the start time using clock().
- Execute the programs.
- Record the end time and calculate the CPU time.
- Display the results and the execution time.

**Program 1: Factorial using Dynamic Programming**

```
#include <stdio.h>

#include <time.h>

int main() {

    int n = 10;

    long fact[20];
```

```
    fact[0] = 1;
    clock_t start, end;
    double cpu_time;
    start = clock();
    for(int i = 1; i <= n; i++)
        fact[i] = i * fact[i-1];
    end = clock();
    printf("Factorial of %d: %ld\n", n, fact[n]);
    cpu_time = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Time taken: %f seconds\n", cpu_time);
    return 0;
}
```

**Program 2: Coin Change Problem using Dynamic Programming**

```
#include <stdio.h>
#include <time.h>
int main() {
    int coins[] = {1, 2, 5};
    int m = sizeof(coins)/sizeof(coins[0]);
    int n = 10; // Amount
    int ways[n+1];
    clock_t start, end;
    double cpu_time;
    for(int i=0;i<=n;i++) ways[i]=0;
    ways[0] = 1;
    start = clock();
    for(int i=0;i<m;i++)
        for(int j=coins[i]; j<=n; j++)
            ways[j] += ways[j - coins[i]];
    end = clock();
```

```
    printf("Number of ways to make %d: %d\n", n, ways[n]);
    cpu_time = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Time taken: %f seconds\n", cpu_time);
    return 0;
}
```

**Result:**

The programs executed successfully. The factorial and coin change calculations produced correct results, and the execution time was measured, demonstrating the efficiency of the dynamic programming approach.

## Lab Sheet 8: 0/1 Knapsack Problem using Dynamic Programming

**Aim:**

To implement the 0/1 Knapsack problem using the Dynamic Programming technique in C and to compute the maximum profit for a given capacity.

**Procedure:**

- Open Turbo C and create a new C program.
- Include the required header files stdio.h and time.h.
- Read the number of items, weights, values, and knapsack capacity.
- Construct the dynamic programming table to store intermediate results.
- Compute the maximum achievable profit using the DP approach.
- Record the start and end time using the clock() function.
- Display the maximum profit obtained and the execution time.
- Compile and run the program.
- Observe the output and efficiency of the algorithm.

**Program: 0/1 Knapsack using Dynamic Programming**

```
#include <stdio.h>
#include <time.h>
int max(int a, int b) {
    return (a > b) ? a : b;
}
```

```
int main() {
    int n, W, i, w;
    int wt[20], val[20];
    int dp[21][51];
    clock_t start, end;
    double cpu_time;
    printf("Enter number of items: ");
    scanf("%d", &n);
    printf("Enter weights of items:\n");
    for(i = 0; i < n; i++)
        scanf("%d", &wt[i]);
    printf("Enter values of items:\n");
    for(i = 0; i < n; i++)
        scanf("%d", &val[i]);
    printf("Enter knapsack capacity: ");
    scanf("%d", &W);
    start = clock();
    for(i = 0; i <= n; i++) {
        for(w = 0; w <= W; w++) {
            if(i == 0 || w == 0)
                dp[i][w] = 0;
            else if(wt[i-1] <= w)
                dp[i][w] = max(val[i-1] + dp[i-1][w - wt[i-1]], dp[i-1][w]);
            else
                dp[i][w] = dp[i-1][w];
        }
    }
    end = clock();
    cpu_time = (double)(end - start) / CLOCKS_PER_SEC;
```

```
    printf("Maximum profit: %d\n", dp[n][W]);
    printf("Execution time: %f seconds\n", cpu_time);
    return 0;
}
```

**Result:**

The 0/1 Knapsack problem was solved successfully using the dynamic programming approach. The maximum profit was obtained for the given capacity, and the execution time was measured, demonstrating the effectiveness of dynamic programming.

## Lab Sheet 9: Floyd–Warshall's Algorithm

**Aim:**

To implement Floyd–Warshall's Algorithm in C to find the shortest paths between all pairs of vertices in a weighted graph.

**Procedure:**

• Open Turbo C and create a new C program.

• Include the required header file stdio.h.

• Define the number of vertices and initialize the adjacency matrix.

• Represent infinite distance using a large constant value.

• Apply Floyd–Warshall's algorithm to compute shortest paths between all vertex pairs.

• Update the distance matrix iteratively using intermediate vertices.

• Display the final shortest path matrix.

• Compile and run the program.

• Observe the shortest distances between all pairs of vertices.

**Program: Floyd–Warshall's Algorithm**

```
#include <stdio.h>
#define INF 999
int main() {
    int n = 4;
    int dist[4][4] = {
```

```
        {0,   3,   INF, 7},
        {8,   0,   2,   INF},
        {5,   INF, 0,   1},
        {2,   INF, INF, 0}
    };
    int i, j, k;
    for(k = 0; k < n; k++) {
        for(i = 0; i < n; i++) {
            for(j = 0; j < n; j++) {
                if(dist[i][j] > dist[i][k] + dist[k][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }
    printf("All-Pairs Shortest Path Matrix:\n");
    for(i = 0; i < n; i++) {
        for(j = 0; j < n; j++)
            printf("%4d", dist[i][j]);
        printf("\n");
    }
    return 0;
}
```

**Result:**

Floyd–Warshall's algorithm was implemented successfully. The shortest paths between all pairs of vertices were computed correctly.

## Lab Sheet 10: Fractional Knapsack Problem using Greedy Technique

**Aim:**

To implement the Fractional Knapsack Problem using the Greedy technique in C.

**Procedure:**

• Open Turbo C and create a new C program.

• Include the required header files stdio.h.

• Read the number of items, their weights, and profits.

• Calculate the profit-to-weight ratio for each item.

• Sort the items in descending order of profit-to-weight ratio.

• Add items fully or fractionally to the knapsack until capacity is reached.

• Calculate the maximum achievable profit.

• Display the total profit obtained.

• Compile and execute the program.

**Program: Fractional Knapsack (Greedy Approach)**

```
#include <stdio.h>
int main() {
    int n, i, j;
    float weight[10], profit[10], ratio[10];
    float capacity, totalProfit = 0, temp;
    printf("Enter number of items: ");
    scanf("%d", &n);
    printf("Enter weights and profits:\n");
    for(i = 0; i < n; i++) {
        scanf("%f %f", &weight[i], &profit[i]);
        ratio[i] = profit[i] / weight[i];
    }
    printf("Enter knapsack capacity: ");
    scanf("%f", &capacity);
    for(i = 0; i < n; i++) {
        for(j = i + 1; j < n; j++) {
            if(ratio[i] < ratio[j]) {
                temp = ratio[i]; ratio[i] = ratio[j]; ratio[j] = temp;
                temp = weight[i]; weight[i] = weight[j]; weight[j] = temp;
```

```
            temp = profit[i]; profit[i] = profit[j]; profit[j] = temp;
        }
    }
  }
  for(i = 0; i < n; i++) {
    if(capacity > 0 && weight[i] <= capacity) {
      capacity -= weight[i];
      totalProfit += profit[i];
    } else {
      totalProfit += ratio[i] * capacity;
      break;
    }
  }
  printf("Maximum Profit: %.2f\n", totalProfit);
  return 0;
}
```

**Result:**

The Fractional Knapsack problem was implemented successfully using the greedy approach, and the maximum profit was calculated correctly.

## Lab Sheet 11: Minimum Spanning Tree using Prim's Algorithm

**Aim:**

To implement Prim's Algorithm in C to find the Minimum Spanning Tree (MST) of a given graph.

**Procedure:**

• Open Turbo C and create a new C program.

• Include the required header files stdio.h and limits.h.

• Read the number of vertices and the adjacency matrix of the graph.

• Initialize arrays to track visited vertices and minimum edge weights.

• Select the vertex with the minimum key value that is not yet included in the MST.

• Update the key values of adjacent vertices.

• Repeat until all vertices are included in the MST.

• Display the edges of the MST and the total minimum cost.

• Compile and execute the program.

**Program: Prim's Algorithm**

```
#include <stdio.h>
#include <limits.h>
#define V 5
int minKey(int key[], int mstSet[]) {
   int min = INT_MAX, minIndex, i;
   for(i = 0; i < V; i++) {
      if(mstSet[i] == 0 && key[i] < min) {
         min = key[i];
         minIndex = i;
      }
   }
   return minIndex;
}
void printMST(int parent[], int graph[V][V]) {
   int i, totalCost = 0;
   printf("Edge \tWeight\n");
   for(i = 1; i < V; i++) {
      printf("%d - %d \t%d\n", parent[i], i, graph[i][parent[i]]);
      totalCost += graph[i][parent[i]];
   }
   printf("Total cost of MST: %d\n", totalCost);
}
void primMST(int graph[V][V]) {
   int parent[V];
   int key[V];
```

```
   int mstSet[V];
   int i, count, u, v;
   for(i = 0; i < V; i++) {
      key[i] = INT_MAX;
      mstSet[i] = 0;
   }
   key[0] = 0;
   parent[0] = -1;
   for(count = 0; count < V - 1; count++) {
      u = minKey(key, mstSet);
      mstSet[u] = 1;
      for(v = 0; v < V; v++) {
         if(graph[u][v] && mstSet[v] == 0 && graph[u][v] < key[v]) {
            parent[v] = u;
            key[v] = graph[u][v];
         }
      }
   }
   printMST(parent, graph);
}
int main() {
   int graph[V][V] = {
      {0, 2, 0, 6, 0},
      {2, 0, 3, 8, 5},
      {0, 3, 0, 0, 7},
      {6, 8, 0, 0, 9},
      {0, 5, 7, 9, 0}
   };
   primMST(graph);
   return 0;
```

}

**Result:**

The Minimum Spanning Tree was generated successfully using Prim's Algorithm, and the minimum total cost was computed correctly.

## Lab Sheet 12: Minimum Spanning Tree using Kruskal's Algorithm

**Aim:**

To implement Kruskal's Algorithm in C to find the Minimum Spanning Tree (MST) of a given graph.

**Procedure:**

• Open Turbo C and create a new C program.

• Include the required header files stdio.h.

• Read the number of vertices and edges of the graph.

• Store all edges with their respective weights.

• Sort the edges in ascending order of weight.

• Select edges one by one and include them in the MST if they do not form a cycle.

• Continue until the MST contains (V − 1) edges.

• Display the edges in the MST and calculate the total minimum cost.

• Compile and execute the program.

**Program: Kruskal's Algorithm**

```
#include <stdio.h>
#define MAX 20
struct edge {
   int u, v, w;
};
struct edge edges[MAX];
struct edge mst[MAX];
int parent[MAX];
int find(int i) {
   while(parent[i])
      i = parent[i];
   return i;
```

```
}
int unionSet(int i, int j) {
    if(i != j) {
        parent[j] = i;
        return 1;
    }
    return 0;
}
int main() {
    int n, e, i, j, count = 0;
    int totalCost = 0;
    struct edge temp;
    printf("Enter number of vertices: ");
    scanf("%d", &n);
    printf("Enter number of edges: ");
    scanf("%d", &e);
    printf("Enter edges (u v w):\n");
    for(i = 0; i < e; i++) {
        scanf("%d %d %d", &edges[i].u, &edges[i].v, &edges[i].w);
    }
    for(i = 0; i < e - 1; i++) {
        for(j = 0; j < e - i - 1; j++) {
            if(edges[j].w > edges[j + 1].w) {
                temp = edges[j];
                edges[j] = edges[j + 1];
                edges[j + 1] = temp;
            }
        }
    }
    for(i = 0; i < n; i++)
```

```
        parent[i] = 0;
    for(i = 0; i < e && count < n - 1; i++) {
        int u = find(edges[i].u);
        int v = find(edges[i].v);
        if(unionSet(u, v)) {
            mst[count++] = edges[i];
            totalCost += edges[i].w;
        }
    }
    printf("Edges in MST:\n");
    for(i = 0; i < count; i++) {
        printf("%d - %d : %d\n", mst[i].u, mst[i].v, mst[i].w);
    }
    printf("Total cost of MST: %d\n", totalCost);
    return 0;
}
```

**Result:**

The Minimum Spanning Tree was successfully constructed using Kruskal's Algorithm, and the minimum total cost was calculated correctly.

## Lab Sheet 13: Knapsack Problem using Branch and Bound Technique

**Aim:**

To implement the 0/1 Knapsack Problem using the Branch and Bound technique in C to obtain the optimal solution.

**Procedure:**

• Open Turbo C and create a new C program.

• Include the required header files stdio.h.

• Read the number of items, their weights, profits, and knapsack capacity.

• Sort items based on profit-to-weight ratio.

• Use the Branch and Bound approach to explore promising nodes.

• Calculate upper bounds to prune non-optimal branches.

• Determine the maximum achievable profit.

• Display the optimal profit obtained.

• Compile and execute the program.

**Program: 0/1 Knapsack using Branch and Bound**

```
#include <stdio.h>
struct Item {
   int weight;
   int profit;
};
int maxProfit = 0;
float bound(int idx, int weight, int profit, int n, int W, struct Item items[]) {
   float boundProfit = profit;
   int totalWeight = weight;
   int i;
   for(i = idx; i < n && totalWeight + items[i].weight <= W; i++) {
      totalWeight += items[i].weight;
      boundProfit += items[i].profit;
   }
   if(i < n)
      boundProfit += (W - totalWeight) *
      ((float)items[i].profit / items[i].weight);
   return boundProfit;
}
void knapsack(int idx, int weight, int profit, int n, int W, struct Item items[]) {
   float b;
   if(weight <= W && profit > maxProfit)
      maxProfit = profit;
   if(idx == n)
      return;
```

```
    b = bound(idx + 1, weight, profit, n, W, items);
    if(b > maxProfit)
        knapsack(idx + 1, weight, profit, n, W, items);
    if(weight + items[idx].weight <= W)
        knapsack(idx + 1, weight + items[idx].weight,
                profit + items[idx].profit, n, W, items);
}
int main() {
    int n, W, i;
    struct Item items[10];
    printf("Enter number of items: ");
    scanf("%d", &n);
    printf("Enter weights and profits:\n");
    for(i = 0; i < n; i++) {
        scanf("%d %d", &items[i].weight, &items[i].profit);
    }
    printf("Enter knapsack capacity: ");
    scanf("%d", &W);
    knapsack(0, 0, 0, n, W, items);
    printf("Maximum profit using Branch and Bound: %d\n", maxProfit);
    return 0;
}
```

**Result:**

The 0/1 Knapsack problem was solved successfully using the Branch and Bound technique, and the optimal profit was obtained.

## Lab Sheet 14: N-Queens Problem using Backtracking

**Aim:**

To implement the N-Queens Problem using the Backtracking technique in C and to find all possible solutions.

**Procedure:**

• Open Turbo C and create a new C program.

• Include the required header files stdio.h and math.h.

• Read the number of queens (N).

• Place queens row by row, ensuring no two queens attack each other.

• Use backtracking to remove queens when a conflict occurs.

• Continue until all valid arrangements are found.

• Display the positions of queens for each solution.

• Compile and execute the program.

**Program: N-Queens using Backtracking**

```
#include <stdio.h>
#include <math.h>
int x[10];
int n;
int place(int k, int i) {
   int j;
   for(j = 1; j < k; j++) {
      if(x[j] == i || abs(x[j] - i) == abs(j - k))
         return 0;
   }
   return 1;
}
void nQueen(int k) {
   int i;
   for(i = 1; i <= n; i++) {
      if(place(k, i)) {
         x[k] = i;
         if(k == n) {
            int j;
            printf("Solution: ");
```

```
            for(j = 1; j <= n; j++)
                printf("%d ", x[j]);
            printf("\n");
        } else {
            nQueen(k + 1);
        }
    }
  }
}
int main() {
  printf("Enter number of queens: ");
  scanf("%d", &n);
  nQueen(1);
  return 0;
}
```

**Result:**

The N-Queens problem was solved successfully using the backtracking technique, and all valid solutions were generated.

## Lab Sheet 15: Case Study – Knapsack Problem using Multiple Techniques

**Aim:**

To study and compare different techniques used to solve the Knapsack Problem, including Greedy, Dynamic Programming, and Branch and Bound approaches.

**Procedure:**

• Open Turbo C and create a new C program.

• Include the required header files stdio.h.

• Read the number of items, their weights, profits, and knapsack capacity.

• Solve the knapsack problem using:

Greedy approach (Fractional Knapsack)

Dynamic Programming approach (0/1 Knapsack)

Branch and Bound approach

• Record the maximum profit obtained using each technique.

• Compare the results and observe the differences.

• Compile and execute the program.

**Program: Knapsack using Multiple Techniques (Case Study)**

```
#include <stdio.h>
int max(int a, int b) {
    return (a > b) ? a : b;
}
void knapsackDP(int n, int W, int wt[], int val[]) {
    int K[20][20], i, w;
    for(i = 0; i <= n; i++) {
        for(w = 0; w <= W; w++) {
            if(i == 0 || w == 0)
                K[i][w] = 0;
            else if(wt[i - 1] <= w)
                K[i][w] = max(val[i - 1] + K[i - 1][w - wt[i - 1]], K[i - 1][w]);
            else
                K[i][w] = K[i - 1][w];
        }
    }
    printf("Maximum profit using Dynamic Programming: %d\n", K[n][W]);
}
void knapsackGreedy(int n, int W, int wt[], int val[]) {
    float ratio[10], totalProfit = 0;
    int i, capacity = W;
```

```
    for(i = 0; i < n; i++)
        ratio[i] = (float)val[i] / wt[i];
    for(i = 0; i < n && capacity > 0; i++) {
        if(wt[i] <= capacity) {
            capacity -= wt[i];
            totalProfit += val[i];
        } else {
            totalProfit += ratio[i] * capacity;
            break;
        }
    }
    printf("Maximum profit using Greedy approach: %.2f\n", totalProfit);
}
int main() {
    int n, W, i;
    int wt[10], val[10];
    printf("Enter number of items: ");
    scanf("%d", &n);
    printf("Enter weights and profits:\n");
    for(i = 0; i < n; i++)
        scanf("%d %d", &wt[i], &val[i]);
    printf("Enter knapsack capacity: ");
    scanf("%d", &W);
    knapsackGreedy(n, W, wt, val);
    knapsackDP(n, W, wt, val);
    return 0;
}
```

**Result:**

The Knapsack problem was solved using multiple techniques. The results showed that different approaches provide different solutions, and the Dynamic Programming method yields the optimal solution for the 0/1 Knapsack problem.