PRG 1. Write a program to Demonstrate operation count

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

void main() {
  int i, a[20], n, sum = 0, count = 0;
  clrscr();

  count += 1;
  printf("\nEnter the size of the array: ");
  scanf("%d", &n);
  count += 1;

  printf("\nEnter the array elements: ");
  for (i = 0; i < n; i++) {
    count += 1;
    scanf("%d", &a[i]);
  }
  count += 1;
  for (i = 0; i < n; i++) {
    count += 1;
    sum += a[i];
    count += 1;
  }
  count += 1;
  printf("\nSum of the array elements is %d and count value is %d", sum, count);
  getch();
}
```

PRG 2. Write a recursive program to find GCD

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

int findgcd(int a, int b) {
  if (b == 0)
    return a;
  else
    return findgcd(b, a % b);
}

void main() {
  int n1, n2, gcd;
  clock_t start, end;
  double time_taken;
  clrscr();

  start = clock();
  printf("\nRECURSION: FIND GCD OF TWO NUMBER");
  printf("\nEnter the 1st number: ");
  scanf("%d", &n1);

  printf("\nEnter the 2st number: ");
  scanf("%d", &n2);
  gcd = findgcd(n1, n2);

  printf("\nThe gcd of %d and %d is %d", n1, n2, gcd);
  end = clock();
```

```
  time_taken = (double)(end - start) / CLOCKS_PER_SEC;
  printf("\nTime taken: %f seconds", time_taken);

  getch();
}
```

Output for GCD program:

PRG 3. Write a program to implement recursive binary search.

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

int binarySearch(int nums[], int low, int high, int target) {
  int mid;

  if (low > high)
    return -1;
  mid = (low + high) / 2;
  if (nums[mid] == target)
    return mid;
  else if (target > nums[mid])
    return binarySearch(nums, mid + 1, high, target);
  return binarySearch(nums, low, mid - 1, target);
}

void main() {
  int n, arr[10], ind, target, i;

  clock_t start, end;
  double time_taken;

  clrscr();

  printf("\nEnter the array size (max 10): ");
  scanf("%d", &n);

  printf("\nEnter the sorted array elements: ");
  for (i = 0; i < n; i++)
    scanf("%d", &arr[i]);

  printf("\nEnter the element to search: ");
  scanf("%d", &target);

  start = clock();
  ind = binarySearch(arr, 0, n, target);
  end = clock();

  if (ind == -1)
    printf("\nTarget %d is not found in the array.", target);
  else
    printf("\nTarget %d is found at position %d", target, ind);

  time_taken = (double)(end - start) / CLOCKS_PER_SEC;
  printf("\nTime taken: %f seconds", time_taken);

  getch();
```

```
}
```

Output for recursive binary search:

PRG 4. Program to implement iterative binary search

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

int binarySearch(int arr[], int n, int target) {
  int low = 0, high = n - 1, mid;

  while (low <= high) {
    mid = (low + high) / 2;

    if (arr[mid] == target)
      return mid;
    else if (target > arr[mid])
      low = mid + 1;
    else
      high = mid - 1;
  }
  return -1;
}

void main() {
  int n, arr[10], ind, target, i;
  clock_t start, end;
  double time_taken;

  clrscr();

  printf("\nEnter the array size: ");
  scanf("%d", &n);

  printf("\nEnter the array elements (sorted): ");
  for (i = 0; i < n; i++)
    scanf("%d", &arr[i]);

  printf("\nEnter the search element: ");
  scanf("%d", &target);

  start = clock();
  ind = binarySearch(arr, n, target);
  end = clock();

  if (ind == -1)
```

```
    printf("\nTarget %d not found in the array", target);
  else
    printf("\nTarget %d is at index %d", target, ind);

  time_taken = (double)(end - start) / CLOCKS_PER_SEC;
  printf("\nTime taken: %f seconds", time_taken);

  getch();
}
```

Output for iterative binary search:

PRG 5. Program to implement String Pattern search using Brute force method.

```
#include <conio.h>
#include <stdio.h>
#include <string.h>

void main() {
 char t[20], p[20];
 int i, j, k, flag = 0, m, n;
 clrscr();

 printf("\nEnter the text: ");
 gets(t);

 printf("\nEnter the pattern: ");
 gets(p);

 n = strlen(t);
 m = strlen(p);

 for (i = 0; i < n - m; i++) {
  j = 0;
  while (j < m && p[j] == t[i + j])
   j += 1;
  if (j == m) {
   flag = 1;
   k = i + 1;
  } else
   flag = 0;
 }

 if (flag == 1)
```

```
    printf("\nPattern found at %d position", k);
  else
    printf("\nPattern not found in the string");

  getch();
}
```

Output for string matching pattern:

PRG 6: Write a program for Quick sort

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

int partition(int arr[], int low, int high) {
 int pivot = arr[low], temp;
 int i = low, j = high;

 while (i < j) {
  while (arr[i] <= pivot && i <= high - 1) {
   i++;
  }

  while (arr[j] > pivot && j >= low + 1) {
   j--;
  }
  if (i < j) {
   temp = arr[i];
   arr[i] = arr[j];
   arr[j] = temp;
  }
 }
 temp = arr[low];
 arr[low] = arr[j];
 arr[j] = temp;
 return j;
}

void quickSort(int arr[], int low, int high) {
 int pIndex;
```

```
  if (low < high) {
    pIndex = partition(arr, low, high);
    quickSort(arr, low, pIndex - 1);
    quickSort(arr, pIndex + 1, high);
  }
}

void main() {
  int n, i, arr[10];
  clock_t start, end;
  double time_taken;
  clrscr();

  printf("Array Size (max 10): ");
  scanf("%d", &n);

  printf("Array Elements: \n");
  for (i = 0; i < n; i++) {
    scanf("%d", &arr[i]);
  }

  printf("Before Sorting Array: \n");
  for (i = 0; i < n; i++) {
    printf("%d ", arr[i]);
  }
  printf("\n");

  start = clock();
  quickSort(arr, 0, n - 1);
  end = clock();
```

```
  printf("After Sorting Array: \n");
  for (i = 0; i < n; i++) {
    printf("%d ", arr[i]);
  }
  printf("\n");

  time_taken = (double)(end - start) / CLOCKS_PER_SEC;
  printf("Time Taken: %f seconds\n", time_taken);

  getch();
}
```

Output for Quick sort:

PRG 7: Write a program for Merge Sort

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

void merge(int arr[], int low, int mid, int high) {
 int temp[10], i;
 int left = low;      // starting index of left half of arr
 int right = mid + 1; // starting index of right half of arr
 int k = 0;           // index for temporary array

 // storing elements in the temporary array in a sorted manner
 while (left <= mid && right <= high) {
  if (arr[left] <= arr[right]) {
   temp[k++] = arr[left++];
  } else {
   temp[k++] = arr[right++];
  }
 }

 // if elements on the left half are still left
 while (left <= mid) {
  temp[k++] = arr[left++];
 }
 // if elements on the right half are still left
 while (right <= high) {
  temp[k++] = arr[right++];
 }

 // transferring all elements from temporary to arr
 for (i = low; i <= high; i++) {
```

```
    arr[i] = temp[i - low];
  }
}

void mergeSort(int arr[], int low, int high) {
  int mid;
  if (low >= high)
    return;

  mid = (low + high) / 2;

  mergeSort(arr, low, mid);      // left half
  mergeSort(arr, mid + 1, high); // right half
  merge(arr, low, mid, high);    // merging sorted halves
}

void main() {
  int n, arr[10], i;
  clock_t start, end;
  double time_taken;

  clrscr();
  printf("Array Size (max 10): ");
  scanf("%d", &n);

  printf("\nEnter Array Elements: ");
  for (i = 0; i < n; i++) {
    scanf("%d", &arr[i]);
  }

  printf("\nBefore Sorting Array: ");
```

```
 for (i = 0; i < n; i++) {
  printf("%d ", arr[i]);
 }

 // calculate the computation time.
 start = clock();
 mergeSort(arr, 0, n - 1);
 end = clock();

 printf("\nAfter Sorting Array: ");
 for (i = 0; i < n; i++) {
  printf("%d ", arr[i]);
 }

 time_taken = (double)(end - start) / CLOCKS_PER_SEC;
 printf("\nTime Taken: %f seconds", time_taken);

 getch();
}
```

Output for Merge sort:

PRG 8: Write a program to find the maximum and minimum numbers in an array using the divide and conquer technique.

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

int max, min;
int a[100];

void maxmin(int i, int j) {
 int max1, min1, mid;
 if (i == j)
  max = min = a[i];
 else {
  if (i == j - 1) {
   if (a[i] < a[j]) {
    max = a[j];
    min = a[i];
   } else {
    max = a[i];
    min = a[j];
   }
  } else {
   mid = (i + j) / 2;
   maxmin(i, mid);
   max1 = max;
   min1 = min;
   maxmin(mid + 1, j);
   if (max < max1)
    max = max1;
   if (min > min1)
    min = min1;
```

```
  }
 }
}

void main() {
 int i, num;
 clock_t start, end;
 double time_taken;
 clrscr();

 start = clock();
 printf("\nArray size: ");
 scanf("%d", &num);

 printf("\nEnter the numbers: ");
 for (i = 1; i <= num; i++)
  scanf("%d", &a[i]);

 max = a[1];
 min = a[1];

 maxmin(1, num);

 printf("\nminimum element in the array: %d", min);
 printf("\nMaximum element in the array: %d", max);

 end = clock();

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

 printf("\nTime taken: %f seconds", time_taken);
```

```
  getch();
}
```

Output for MaxMin problem:

PRG 9: Write a program for Minimum Spanning Trees using Prim's algorithm

```
/*
    a, b    -> Stores the nodes of the selected edge.
    v, u    -> Temporary variables to store selected nodes.
    n       -> Number of nodes in the graph.
    ne      -> Counter for the number of edges added to MST (starts at 1).
    min     -> Stores the smallest edge weight found in each iteration.
    mincost -> Total cost of the Minimum Spanning Tree (MST).
    c[10][10]-> Adjacency matrix to store graph edges.
    vis[10] -> Array to track visited nodes (1 = visited, 0 = not visited).
*/
#include <conio.h>
#include <stdio.h>
int i, j, a, b, v, u, n, ne = 1;
int min, mincost = 0, c[10][10], vis[10] = {0};

void main() {
 clrscr();

 printf("Enter number of nodes: \n");
 scanf("%d", &n);

 printf("Enter the adjacency matrix: \n");
 for (i = 1; i <= n; i++) {
  for (j = 1; j <= n; j++) {
   scanf("%d", &c[i][j]);
   if (c[i][j] == 0) {
    c[i][j] = 999; // Treat 0 as no edge
   }
```

```
  }
 }
 vis[1] = 1; // since we are storing matrix as 1 based index.
 printf("\n");

 while (ne < n) {
  min = 999;
  for (i = 1; i <= n; i++) {
   if (vis[i]) {
    for (j = 1; j <= n; j++) {
     if (c[i][j] < min && !vis[j]) {
      min = c[i][j];
      a = u = i;
      b = v = j;
     }
    }
   }
  }
  if (!vis[u] || !vis[v]) {
   printf("\nEdge %d: (%d %d) cost:%d\n", ne++, a, b, min);
   mincost += min;
   vis[b] = 1;
  }
  c[a][b] = c[b][a] = 999;
 } // end of while

 printf("\nMinimum cost = %d\n", mincost);
 getch();
}
```

PRG 10: Write a program for Minimum Spanning Trees using Kruskal's algorithm

```
/*
 a, b    -> Stores the nodes of the selected edge.
 v, u    -> Temporary variables to store selected nodes.
 n       -> Number of vertices in the graph.
 ne      -> Counter for the number of edges added to MST (starts at 0).
 min     -> Stores the smallest edge weight found in each iteration.
 mincost  -> Total cost of the Minimum Spanning Tree (MST).
 cost[10][10] -> Cost matrix representing the graph.
 parent[10]   -> Array to track parent nodes for cycle detection (used in
 Kruskal's algorithm).
*/

#include <conio.h>
#include <stdio.h>

int i, j, k, a, b, v, u, n, ne = 0;
int min, mincost = 0, cost[10][10], parent[10];

int find(int i) {
 while (parent[i]) {
   i = parent[i];
 }
 return i;
}

void uni(int i, int j) {
 if (i != j) {
   parent[j] = i;
 }
}

void main() {
 clrscr();

 printf("Enter number of vertices: \n");
 scanf("%d", &n);

 printf("Enter the cost matrix: \n");
 for (i = 1; i <= n; i++) {
   for (j = 1; j <= n; j++) {
     scanf("%d", &cost[i][j]);
```

```
      if (cost[i][j] == 0)
        cost[i][j] = 999;
    }
  }

  for (i = 1; i <= n; i++) {
    parent[i] = 0;
  }

  printf("Edges of spanning tree are: \n");
  while (ne < n - 1) {
    min = 999;
    for (i = 1; i <= n; i++) {
      for (j = 1; j <= n; j++) {
        if (cost[i][j] < min) {
          min = cost[i][j];
          a = u = i;
          b = v = j;
        }
      }
    }

    u = find(u);
    v = find(v);
    if (u != v) {
      printf("%d. edge(%d,%d) = %d\n", ++ne, a, b, min);
      mincost += min;
      uni(u, v);
    }
    cost[a][b] = cost[b][a] = 999;
  }

  printf("Minimum Cost = %d\n", mincost);
  getch();
}
```

PRG 11: Write a program to implement Knapsack Algorithm

```
/*
 weight[MAX_ITEMS]  -> Array to store the weights of items.
 profit[MAX_ITEMS]  -> Array to store the profits of items.
 ratio[MAX_ITEMS]   -> Stores profit-to-weight ratio for each item.
 totalValue       -> Stores the total profit obtained.
 temp            -> Temporary variable (not used in this version).
 capacity        -> Maximum weight the knapsack can hold.
 n              -> Number of items.
*/

#include <conio.h>
#include <stdio.h>

#define MAX_ITEMS 50

void main() {
 float weight[MAX_ITEMS], profit[MAX_ITEMS], ratio[MAX_ITEMS], totalValue = 0;
 float temp, capacity;
 int n, i, j;
 clrscr();

 printf("Enter the number of items (up to %d): ", MAX_ITEMS);
 scanf("%d", &n);

 if (n <= 0 || n > MAX_ITEMS) {
  printf("Invalid number of items.\n");
  return;
 }

 printf("Enter weight and profit for each item:\n");
 for (i = 0; i < n; i++) {
  printf("Item %d: ", i + 1);
  scanf("%f %f", &weight[i], &profit[i]);
  if (weight[i] <= 0 || profit[i] < 0) {
   printf("Invalid input. Weight must be positive and profit must be "
       "non-negative.\n");
   return;
  }
  ratio[i] = profit[i] / weight[i];
 }
```

```
  printf("Enter the capacity of Knapsack: ");
  scanf("%f", &capacity);

  if (capacity <= 0) {
    printf("Invalid capacity. Capacity must be positive.\n");
    return;
  }

  for (i = 0; i < n; i++) {
    if (weight[i] <= capacity) {
      totalValue += profit[i];
      capacity -= weight[i];
    } else {
      totalValue += (ratio[i] * capacity);
      break;
    }
  }

  printf("The maximum profit is %f\n", totalValue);
  getch();
}
```

Output for Knapsack problem:

PRG 12: Write a program for floyd-Warshall algorithm

/*
This program computes the transitive closure of a directed graph using Warshall's algorithm.
It reads a cost (adjacency) matrix of a graph and outputs the path matrix showing reachability.

n - Number of nodes (vertices) in the graph

a[][] - Cost/adjacency matrix input by the user (1 = edge exists, 0 = no edge)

p[][] - Path matrix that stores whether a path exists between each pair of nodes

i, j, k - Loop counters for iterating through the matrix

*/

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

void main() {
  int i, j, k, n;
  int a[10][10], p[10][10];  // Matrices: 'a' = input, 'p' = result (path matrix)
  clrscr();

  // Read number of nodes
  printf("Enter the number of nodes: ");
  scanf("%d", &n);

  // Read the adjacency matrix
  printf("Enter the cost matrix:\n");
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      scanf("%d", &a[i][j]);  // 1 = edge exists, 0 = no edge
    }
  }

  // Initialize the path matrix with the input matrix
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      p[i][j] = a[i][j];
    }
  }
```

```
    // Warshall's algorithm: compute transitive closure
    for (k = 0; k < n; k++) {            // For each intermediate node
        for (i = 0; i < n; i++) {        // For each source node
            for (j = 0; j < n; j++) {    // For each destination node
                // If a path exists from i to j through k, mark it as reachable
                if (p[i][j] == 1 || (p[i][k] == 1 && p[k][j] == 1)) {
                    p[i][j] = 1;
                }
            }
        }
    }

    // Print the transitive closure (path matrix)
    printf("The path matrix shown below:\n");
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            printf("%d\t", p[i][j]);
        }
        printf("\n");
    }
   getch();
}
```

Output for Warshall's Algorithm:

PRG 13: Write a program to implement the N Queens problem using back tracking.

```
/*
It places N queens on an N×N chessboard such that no two queens attack each other,
and prints all possible solutions.

MAX_QUEENS - Maximum number of queens allowed (set to 30)
queens[]   - Array where index represents the column and value represents the row of a
queen.
count      - Counter to track the number of valid solutions found
*/

#include <stdio.h>
#include <conio.h>
#include <math.h>
#define MAX_QUEENS 30

int queens[MAX_QUEENS];
int count = 0;          // Number of valid solutions found

int isSafe(int row, int col) {
   for (int i = 1; i < col; i++) {
      // Check row conflict and diagonal conflicts
      if (queens[i] == row || abs(queens[i] - row) == abs(i - col))
         return 0;
   }
   return 1;
}

void printSolution(int n) {
   count++;
   printf("\n\nSolution #%d:\n", count);
   for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= n; j++) {
         if (queens[i] == j)
            printf("Q\t");
         else
            printf("*\t");
      }
      printf("\n");
   }
}
```

```
void solveNQueens(int n) {
   int col = 1;
   queens[col] = 0;

   while (col != 0) {
      queens[col]++;
      while (queens[col] <= n && !isSafe(queens[col], col))
         queens[col]++;

      if (queens[col] <= n) {
         if (col == n)
            printSolution(n);
         else {
            col++;
            queens[col] = 0;
         }
      } else
         col--;
   }
}

void main() {
   int n;
   clrscr();

  printf("Enter the number of queens (<= %d): ", MAX_QUEENS);
   scanf("%d", &n);

   if (n < 1 || n > MAX_QUEENS) {
      printf("Invalid input!\n");
      return 1;
   }

   solveNQueens(n);

   printf("\nTotal Solutions = %d\n", count);
   getch();
}
```

PRG 14: Program to solve Sum of subset problem.

```
/*
MAX_SIZE - Maximum number of elements allowed in the input.
s[]    - Array to store the input set (must be sorted in increasing order)
x[]    - Binary array to track which elements are included in the current subset
d      - Target sum.
m      - Current sum of selected elements in the subset
k      - Index of the current element
r      - Remaining sum of elements not yet considered
*/

#include <stdio.h>
#include <conio.h>
#define MAX_SIZE 10

int s[MAX_SIZE], x[MAX_SIZE],d;

// Recursive function to generate subsets whose sum is equal to 'd'
void sumofSub(int m, int k, int r) {
   int i;
   x[k] = 1;  // Include s[k] in the subset

   if (m + s[k] == d) {
      // Subset sum matches target, print it
      printf("Subset: ");
      for (i = 0; i <= k; i++) {
         if (x[i] == 1)
            printf("%d ", s[i]);
      }
      printf("\n");
   } else {
      if (m + s[k] + s[k + 1] <= d) {
         sumofSub(m + s[k], k + 1, r - s[k]);
      }

      // Explore alternative path with s[k] excluded
      if ((m + r - s[k] >= d) && (m + s[k + 1] <= d)) {
         x[k] = 0;  // Exclude s[k] from the subset
         sumofSub(m, k + 1, r - s[k]);
      }
```

```
    }
}

void main() {
    int n, sum = 0;  // n = number of elements, sum = total sum of set
    int i;
    clrscr();

    printf("Enter the size of the set (up to %d): ", MAX_SIZE);
    scanf("%d", &n);

    printf("Enter the set in increasing order: ");
    for (i = 0; i < n; i++) {
        scanf("%d", &s[i]);
        sum += s[i];  // Compute total sum
    }

    printf("Enter the value of d: ");
    scanf("%d", &d);

    // Check feasibility before starting
    if (sum < d || s[0] > d) {
        printf("No subset possible.\n");
    } else {
        sumofSub(0, 0, sum);  // Begin backtracking from index 0
    }
 getch();
}
```

Output for Sum of subset problem: