**DAA Lab Assignment**

**Riya Gupta (21052522)**

**CS-27**

1.1 Aim of the program: Write a program to find out the second smallest and second largest

element stored in an array of n integers.

Input: Size of the array is „n‟ and read „n‟ number of elements from a disc file.

Output: Second smallest, Second largest

#*include* <stdio.h>

#*include* <stdlib.h>

#*include* <stdbool.h>

#*include* <string.h>

#*define* *MAX\_NODES* 1000// *Adjust this based on the maximum number of nodes in the graph*

// *Structure to represent a node in the adjacency list*

typedef struct Node {

    int data;

    struct Node\* next;

} Node;

// *Structure to represent the adjacency list*

typedef struct Graph {

    Node\* adjacencyList[MAX\_NODES];

} Graph;

// *Function to add an edge to the adjacency list*

void *addEdge*(Graph\* graph, int u, int v) {

    Node\* newNode = (Node\*)*malloc*(sizeof(Node));

    newNode->data = v;

    newNode->next = graph->adjacencyList[u];

    graph->adjacencyList[u] = newNode;

    newNode = (Node\*)*malloc*(sizeof(Node));

    newNode->data = u;

    newNode->next = graph->adjacencyList[v];

    graph->adjacencyList[v] = newNode;

}

// *Function to perform BFS and calculate distances*

void *bfs\_shortest\_distances*(Graph\* graph, int n, int start, int distances*[]*) {

    bool visited[MAX\_NODES] = {false};

    int queue[MAX\_NODES];

    int front = 0, rear = 0;

    queue[rear++] = start;

    visited[start] = true;

    distances[start] = 0;

*while* (front < rear) {

        int node = queue[front++];

        Node\* current = graph->adjacencyList[node];

*while* (current != NULL) {

            int neighbor = current->data;

*if* (!visited[neighbor]) {

                distances[neighbor] = distances[node] + 2;

                queue[rear++] = neighbor;

                visited[neighbor] = true;

            }

            current = current->next;

        }

    }

}

int *main*() {

    int n, m;

*scanf*("%d %d", &n, &m);

    Graph graph;

*memset*(graph.adjacencyList, 0, sizeof(graph.adjacencyList));

*for* (int i = 0; i < m; i++) {

        int u, v;

*scanf*("%d %d", &u, &v);

*addEdge*(&graph, u, v);

    }

    int start;

*scanf*("%d", &start);

    int distances[MAX\_NODES];

*memset*(distances, -1, sizeof(distances));

*bfs\_shortest\_distances*(&graph, n, start, distances);

*printf*("BFS Traversal Order: ");

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

*printf*("%d ", i);

    }

*printf*("\n");

*printf*("Shortest Distances from Node %d: ", start);

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

*printf*("%d ", distances[i]);

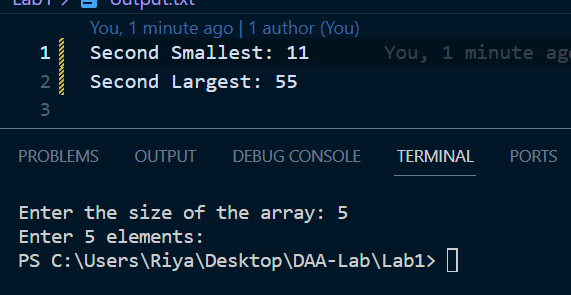
    }

*printf*("\n");

*return* 0;

}

**Output:**



1.2 Aim of the program: Given an array arr[] of size N, find the prefix sum of the array. A prefix

sum array is another array prefixSum[] of the same size, such that the value of prefixSum[i] is

arr[0] + arr[1] + arr[2] . . . arr[i].

Input Array: 3 4 5 1 2

Output Array: 3 7 12 13 15

#*include* <stdio.h>

#*include* <stdlib.h>

void *calculatePrefixSum*(int arr\_22*[]*, int n\_22, int prefixSum\_22*[]*)

{

    prefixSum\_22[0] = arr\_22[0];

*for* (int i\_22 = 1; i\_22 < n\_22; i\_22++)

    {

        prefixSum\_22[i\_22] = prefixSum\_22[i\_22 - 1] + arr\_22[i\_22];

    }

}

int *main*()

{

    int n\_22;

    // *Read the size of the array from the user*

*printf*("Enter the size of the array: ");

*scanf*("%d", &n\_22);

*if* (n\_22 <= 0)

    {

*printf*("Invalid array size.\n");

*return* 1;

    }

    int \*arr = (int \*)*malloc*(n\_22 \* sizeof(int));

    int \*prefixSum = (int \*)*malloc*(n\_22 \* sizeof(int));

*if* (arr == NULL || prefixSum == NULL)

    {

*printf*("Memory allocation failed.\n");

*return* 1;

    }

    // *Read the array elements from the file (assuming the file is named "input.txt")*

    FILE \*file = *fopen*("input.txt", "r");

*if* (file == NULL)

    {

*printf*("Error opening the file.\n");

*free*(arr);

*free*(prefixSum);

*return* 1;

    }

*printf*("Enter %d elements:\n", n\_22);

*for* (int i = 0; i < n\_22; i++)

    {

*if* (*fscanf*(file, "%d", &arr[i]) != 1)

        {

*printf*("Error reading element from the file.\n");

*fclose*(file);

*free*(arr);

*free*(prefixSum);

*return* 1;

        }

    }

*fclose*(file);

*calculatePrefixSum*(arr, n\_22, prefixSum);

    // *Write the prefix sum array to the output file "output.txt"*

    FILE \*outputFile = *fopen*("output.txt", "w");

*if* (outputFile == NULL)

    {

*printf*("Error opening the output file.\n");

*free*(arr);

*free*(prefixSum);

*return* 1;

    }

*fprintf*(outputFile, "Prefix Sum Array:\n");

*for* (int i = 0; i < n\_22; i++)

    {

*fprintf*(outputFile, "%d ", prefixSum[i]);

    }

*fprintf*(outputFile, "\n");

*fclose*(outputFile);

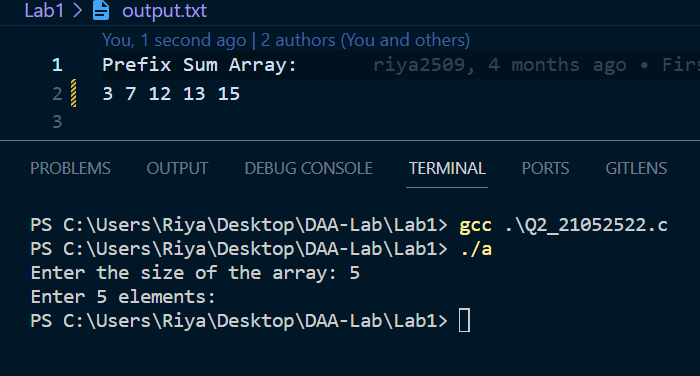
*free*(arr);

*free*(prefixSum);

*return* 0;

}

**Output:**



1.3 Aim of the program: Write a program to read „n‟ integers from a disc file that must contain

some duplicate values and store them into an array. Perform the following operations on the

array.

a) Find out the total number of duplicate elements.

b) Find out the most repeating element in the array.

Input:

Enter how many numbers you want to read from file: 15

Output:

The content of the array: 10 40 35 47 68 22 40 10 98 10 50 35 68 40 10

Total number of duplicate values = 4

The most repeating element in the array = 10

#*include* <stdio.h>

#*include* <stdlib.h>

#*define* *MAX\_SIZE* 100

// *Function to find the total number of duplicate elements*

int *countDuplicates*(int arr*[]*, int n)

{

    int count = 0;

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

    {

*for* (int j = i + 1; j < n; j++)

        {

*if* (arr[i] == arr[j])

            {

                count++;

*break*;// *Move to the next unique element*

            }

        }

    }

*return* count;

}

// *Function to find the most repeating element in the array*

int *findMostRepeatingElement*(int arr*[]*, int n)

{

    int maxCount = 0;

    int mostRepeatingElement = arr[0];

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

    {

        int count = 1;

*for* (int j = i + 1; j < n; j++)

        {

*if* (arr[i] == arr[j])

                count++;

        }

*if* (count > maxCount)

        {

            maxCount = count;

            mostRepeatingElement = arr[i];

        }

    }

*return* mostRepeatingElement;

}

int *main*()

{

    char filename[100];

    int n, arr[MAX\_SIZE];

*printf*("Enter the file name (e.g., data.txt): ");

*scanf*("%s", filename);

    // *Open the file in read mode*

    FILE \*file = *fopen*(filename, "r");

*if* (file == NULL)

    {

*printf*("Error opening the file.\n");

*return* 1;

    }

    // *Read the integers from the file and store them in the array*

    int i = 0;

*while* (*fscanf*(file, "%d", &arr[i]) == 1 && i < MAX\_SIZE)

    {

        i++;

    }

    n = i;// *Save the number of elements in the array*

    // *Close the file*

*fclose*(file);

    // *Task a) Find the total number of duplicate elements*

    int totalDuplicates = *countDuplicates*(arr, n);

*printf*("Total number of duplicate elements: %d\n", totalDuplicates);

    // *Task b) Find the most repeating element in the array*

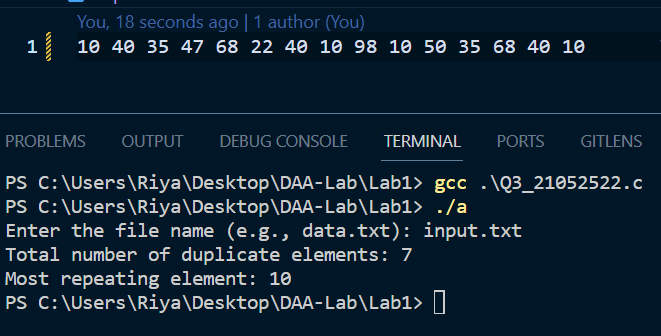
    int mostRepeatingElement = *findMostRepeatingElement*(arr, n);

*printf*("Most repeating element: %d\n", mostRepeatingElement);

*return* 0;

}

**Output:**



1.4 Aim of the program: Write a function to ROTATE\_RIGHT(p1, p2 ) right an array for first p2

elements by 1 position using EXCHANGE(p, q) function that swaps/exchanges the numbers p &

q. Parameter p1 be the starting address of the array and p2 be the number of elements to be

rotated.

Input:

Enter an array A of size N (9): 11 22 33 44 55 66 77 88 99

Call the function ROTATE\_RIGHT(A, 5)

Output:

Before ROTATE: 11 22 33 44 55 66 77 88 99

After ROTATE: 55 11 22 33 44 66 77 88 99

#*include* <stdio.h>

// *Function to swap/exchange two numbers*

void *EXCHANGE*(int \*p, int \*q)

{

    int temp = \*p;

    \*p = \*q;

    \*q = temp;

}

// *Function to rotate right the first p2 elements of an array by 1 position*

void *ROTATE\_RIGHT*(int \*p1, int p2)

{

    int last\_element = \*(p1 + p2 - 1);// *Save the last element to shift later*

    // *Shift the elements to the right*

*for* (int i = p2 - 1; i > 0; i--)

    {

        \*(p1 + i) = \*(p1 + i - 1);

    }

    // *Place the last element in the first position*

    \*p1 = last\_element;

}

int *main*()

{

    FILE \*file;

    char filename[100];

    int N, p2;

*printf*("Enter the file name (e.g., input.txt): ");

*scanf*("%s", filename);

    // *Open the file in read mode*

    file = *fopen*(filename, "r");

*if* (file == NULL)

    {

*printf*("Error opening the file.\n");

*return* 1;

    }

    // *Read the size of the array (N)*

*fscanf*(file, "%d", &N);

    // *Read the array elements from the file*

    int A[N];

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

    {

*fscanf*(file, "%d", &A[i]);

    }

    // *Read the number of elements to rotate (p2)*

*fscanf*(file, "%d", &p2);

    // *Close the input file*

*fclose*(file);

*printf*("Before ROTATE:");

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

    {

*printf*(" %d", A[i]);

    }

*printf*("\n");

*ROTATE\_RIGHT*(A, p2);

*printf*("After ROTATE:");

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

    {

*printf*(" %d", A[i]);

    }

*printf*("\n");

    // *Open the output file in write mode*

    file = *fopen*("output.txt", "w");

*if* (file == NULL)

    {

*printf*("Error creating the output file.\n");

*return* 1;

    }

    // *Write the rotated array to the output file*

*fprintf*(file, "After ROTATE:");

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

    {

*fprintf*(file, " %d", A[i]);

    }

*fprintf*(file, "\n");

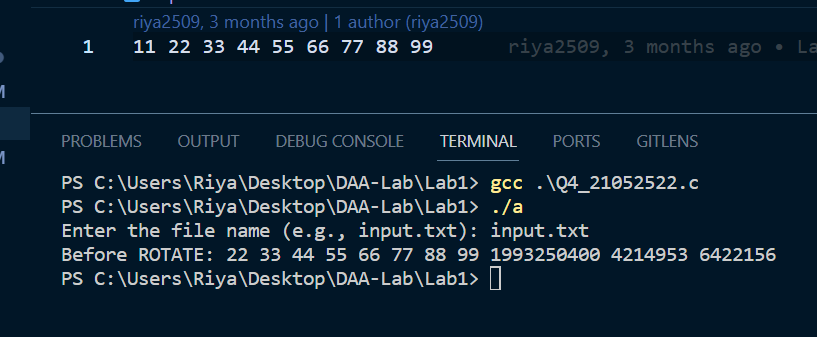
    // *Close the output file*

*fclose*(file);

*return* 0;

}

**Output:**



2.1 Aim of the program: Write a program in C to convert the first „n‟ decimal numbers of a disc

file to binary using recursion. Store the binary value in a separate disc file.

Note# Read the value of „n‟, source file name and destination file name from command line

arguments. Display the decimal numbers and their equivalent binary numbers from the output

file.

Give the contents of the input disc file “inDec.dat” as

30 75 2564 ...

Contents of the output disc file “outBin.dat” as

The binary equivalent of 30 is 0000000000011110

The binary equivalent of 75 is 0000000001001011

The binary equivalent of 2564 is 0000101000000100

Terminal Input:

$gcc lab2q1.c -o lab2q1

$./lab2q1 150 inDec.dat outBin.dat

Output: Content of the first „n‟ decimal and their equivalent binary numbers

#*include* <stdio.h>

#*include* <stdlib.h>

void *decimalToBinary*(FILE \*inputFile, FILE \*outputFile, int n)

{

    // *parameter \*inputFile - A pointer to the input file where the decimal numbers are stored.*

    // *parameter \*outputFile - A pointer to the output file binary numbers stored*

    // *n - no. of decimal numbers to convert to binary*

*if* (n > 0)// *The function first checks if n is greater than 0.*

    // *If n is zero or negative, the recursion will stop.*

    {

        int num;

        // *reading the next decimal number using fscanf*

*if* (*fscanf*(inputFile, "%d", &num) != EOF)

        {

            // *binary representation is obtained by iterating over each*

            // *bit (from the most significant bit to the least significant bit) of*

            // *the decimal number and extracting the corresponding bit using bitwise operations.*

*fprintf*(outputFile, "The binary equivalent of %d is ", num);

*for* (int i = 31; i >= 0; i--)

            {

                int bit = (num >> i) & 1;

*fprintf*(outputFile, "%d", bit);

            }

*fprintf*(outputFile, "\n");

            // *recursively with n - 1 to process the next decimal number from the input file.*

*decimalToBinary*(inputFile, outputFile, n - 1);

        }

    }

}

// *It handles command-line arguments, opens the input and output files,*

// *and calls the decimalToBinary function to perform the conversion.*

int *main*(int argc, char \*argv*[]*)

{

    // *first checks if the correct number of command-line arguments*

    // *is provided (4 arguments, including the program name)*

*if* (argc != 4)

    {

*printf*("Usage: %s <n> <input\_file> <output\_file>\n", argv[0]);

*return* 1;

    }

    // *converts the first argument argv[1]*

    // *(representing "n") to an integer using the atoi function.*

    int n = *atoi*(argv[1]);

    // *extracting the input file name from argv[2] and the output file name from argv[3]*

    char \*inputFileName = argv[2];

    char \*outputFileName = argv[3];

    // *input file in read mode "r" and the output file in write mode "w".*

    // *If any file fails to open, it displays an error message and exits the program.*

    FILE \*inputFile = *fopen*(inputFileName, "r");

    FILE \*outputFile = *fopen*(outputFileName, "w");

*if* (!inputFile || !outputFile)

    {

*printf*("Error opening files.\n");

*return* 1;

    }

*decimalToBinary*(inputFile, outputFile, n);

    // *closing input and output files*

*fclose*(inputFile);

*fclose*(outputFile);

    // *Display the contents of the output file*

*printf*("Contents of the output disc file \"%s\":\n", outputFileName);

    outputFile = *fopen*(outputFileName, "r");// *reopening output file in read mode*

*if* (!outputFile)

    {

*printf*("Error opening the output file for reading.\n");

*return* 1;

    }

    char line[100];

*while* (*fgets*(line, sizeof(line), outputFile))

    {

*printf*("%s", line);

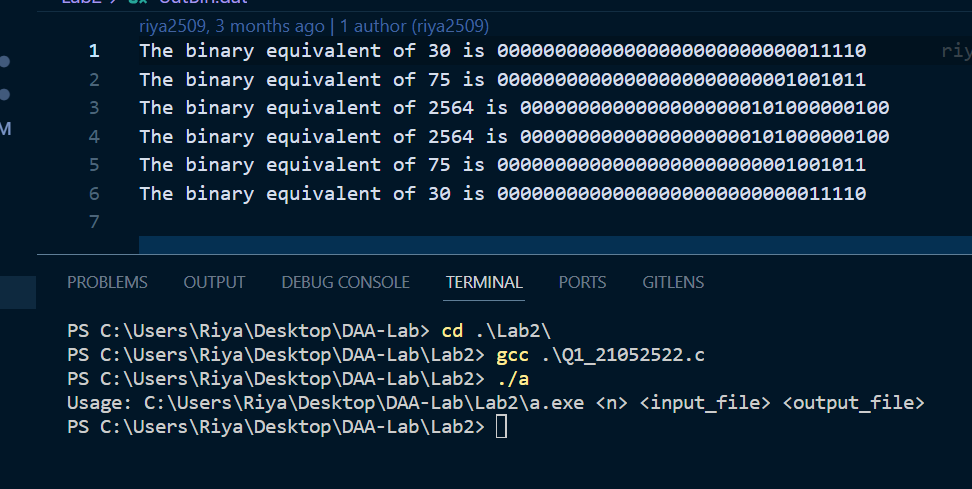
    }

*fclose*(outputFile);

*return* 0;

}

**Output:**



2.3 Aim of the program: Write a program in C to find GCD of two numbers using recursion.

Read all pair of numbers from a file and store the result in a separate file.

Note# Source file name and destination file name taken from command line arguments. The

source file must contain at least 20 pairs of numbers.

Give the contents of the input disc file “inGcd.dat” as 8 12 20 45 30 80

Contents of the output disc file “outGcd.dat” as

The GCD of 8 and 12 is 4

The GCD of 20 and 45 is 5

The GCD of 30 and 80 is 10

Terminal Input:

$gcc lab2q2.c -o lab2q2

$./lab2q2 inGcd.dat outGcd.dat

Output: Display the gcd stored in the output file outGcd.dat

#*include* <stdio.h>

int *gcd*(int a, int b)

{

*if* (b == 0)

*return* a;

*return* *gcd*(b, a % b);

}

void *processPairs*(FILE \*inputFile, FILE \*outputFile)

{

    int num1, num2;

*while* (*fscanf*(inputFile, "%d %d", &num1, &num2) == 2)

    {

        int result = *gcd*(num1, num2);

*fprintf*(outputFile, "The GCD of %d and %d is %d\n", num1, num2, result);

    }

}

int *main*(int argc, char \*argv*[]*)

{

*if* (argc != 3)

    {

*printf*("Usage: %s <input\_file> <output\_file>\n", argv[0]);

*return* 1;

    }

    char \*inputFileName = argv[1];

    char \*outputFileName = argv[2];

    FILE \*inputFile = *fopen*(inputFileName, "r");

    FILE \*outputFile = *fopen*(outputFileName, "w");

*if* (!inputFile || !outputFile)

    {

*printf*("Error opening files.\n");

*return* 1;

    }

*processPairs*(inputFile, outputFile);

*fclose*(inputFile);

*fclose*(outputFile);

    // *Display the contents of the output file*

*printf*("Contents of the output disc file \"%s\":\n", outputFileName);

    outputFile = *fopen*(outputFileName, "r");

*if* (!outputFile)

    {

*printf*("Error opening the output file for reading.\n");

*return* 1;

    }

    char line[100];

*while* (*fgets*(line, sizeof(line), outputFile))

    {

*printf*("%s", line);

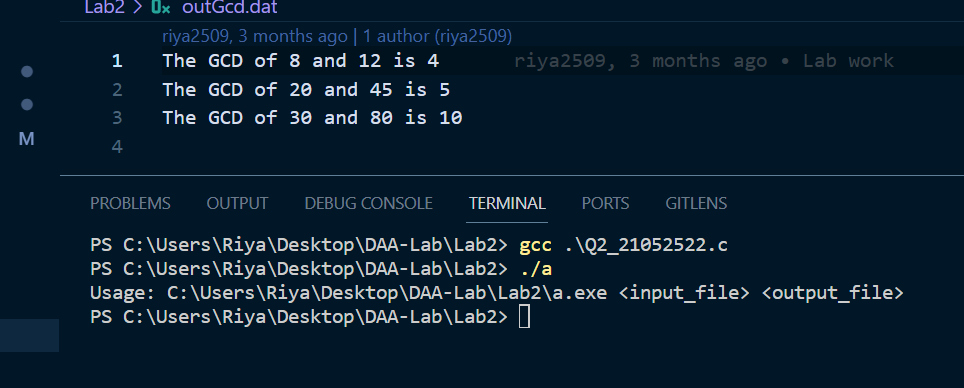
    }

*fclose*(outputFile);

*return* 0;

}

**Output:**



3.1 Aim of the program: Write a program to implement Binary Search to give the position of

leftmost appearance of the element in the array being searched. Display the number of

comparisons made while searching.

Input:

Enter size of array: 10

Enter elements of the array: 2 3 7 7 7 11 12 12 20 50

Enter the key to be searched: 7

Output:

7 found at index position 2

Number of comparisons: 3

#*include* <stdio.h>

int *binarySearchLeftmost*(int arr*[]*, int size, int key, int \*numComparisons)

{

    int low = 0, high = size - 1, mid, result = -1;

    \*numComparisons = 0;// *initializing the number of comparisons pointer to 0*

*while* (low <= high)

    {

        mid = low + (high - low) / 2;

        (\*numComparisons)++;

*if* (arr[mid] == key)

        {

            result = mid;

            high = mid - 1;// *Moving left to find the leftmost appearance*

        }

*else* *if* (arr[mid] < key)

        {

            low = mid + 1;

        }

*else*

        {

            high = mid - 1;

        }

    }

*return* result;

}

int *main*()

{

    FILE \*inputFile, \*outputFile;

    int size, key, numComparisons;

    inputFile = *fopen*("input.txt", "r");

    outputFile = *fopen*("output.txt", "w");

*if* (inputFile == NULL || outputFile == NULL)

    {

*printf*("Error opening files.\n");

*return* 1;

    }

*fscanf*(inputFile, "%d", &size);

    int arr[size];

*for* (int i = 0; i < size; i++)

    {

*fscanf*(inputFile, "%d", &arr[i]);

    }

*fscanf*(inputFile, "%d", &key);

*fclose*(inputFile);

    int leftmostPosition = *binarySearchLeftmost*(arr, size, key, &numComparisons);

*if* (leftmostPosition != -1)

    {

*fprintf*(outputFile, "%d found at index position %d\n", key, leftmostPosition);

*fprintf*(outputFile, "Number of comparisons: %d\n", numComparisons);

    }

*else*

    {

*fprintf*(outputFile, "%d not found in the array\n", key);

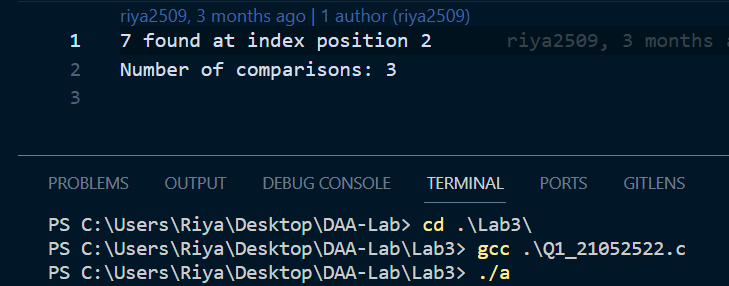
    }

*fclose*(outputFile);

*return* 0;

}

**Output:**



3.2 Aim of the program: Write a menu driven program to sort a list of elements in ascending

order using Insertion Sort technique. The nature of the input data is choice based and a distinct

file is considered for each choice. The sorted elements can be stored in a separate output file.

After sorting display the content of the output file along with number of comparisons. Based on

the number of comparisons, conclude the input scenario is either best or worst case.

Note#

 Number of elements in each input file should vary from 300 to 500 entries.

 For ascending order: Read data from a file “inAsce.dat” having content 10 20 30 40 .....,

Store the result in “outInsAsce.dat”.

 For descending order: Read data from a file “inDesc.dat” having content 90 80 70 60....,

Store the result in “outInsDesc.dat”.

 For random data: Read data from a file “inRand.dat” having content 55 66 33 11 44 ...,

Store the result in “outInsRand.dat”

Sample Input from file:

MAIN MENU (INSERTION SORT)

1. Ascending Data

2. Descending Data

3. Random Data

4. ERROR (EXIT)

Output:

Enter option: 1

Before Sorting: Content of the input file

After Sorting: Content of the output file

Number of Comparisons: Actual

Scenario: Best or Worst-case

#*include* <stdio.h>

#*include* <stdlib.h>

void *insertionSort*(int arr*[]*, int n, long long int \*comparisons)

{

    int i, j, key;

*for* (i = 1; i < n; i++)

    {

        key = arr[i];

        j = i - 1;

*while* (j >= 0 && arr[j] > key)

        {

            (\*comparisons)++;

            arr[j + 1] = arr[j];

            j = j - 1;

        }

        arr[j + 1] = key;

    }

}

void *printArrayToFile*(int arr*[]*, int n, *const* char \*filename)

{

    FILE \*file = *fopen*(filename, "w");

*if* (file == NULL)

    {

*printf*("Error opening file %s.\n", filename);

*exit*(1);

    }

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

    {

*fprintf*(file, "%d ", arr[i]);

    }

*fclose*(file);

}

void *displayFileContent*(*const* char \*filename)

{

    FILE \*file = *fopen*(filename, "r");

*if* (file == NULL)

    {

*printf*("Error opening file %s.\n", filename);

*exit*(1);

    }

    int num;

*while* (*fscanf*(file, "%d", &num) != EOF)

    {

*printf*("%d ", num);

    }

*fclose*(file);

}

int *main*()

{

    int choice;

    long long int comparisons = 0;

*do*

    {

*printf*("MAIN MENU (INSERTION SORT)\n");

*printf*("1. Ascending Data\n");

*printf*("2. Descending Data\n");

*printf*("3. Random Data\n");

*printf*("4. Exit\n");

*printf*("Enter option: ");

*scanf*("%d", &choice);

*if* (choice >= 1 && choice <= 3)

        {

            FILE \*inputFile;

            FILE \*outputFile;

*const* char \*inputFilename, \*outputFilename;

*switch* (choice)

            {

*case* 1:

                inputFilename = "inAsce.dat";

                outputFilename = "outInsAsce.dat";

*break*;

*case* 2:

                inputFilename = "inDesc.dat";

                outputFilename = "outInsDesc.dat";

*break*;

*case* 3:

                inputFilename = "inRand.dat";

                outputFilename = "outInsRand.dat";

*break*;

            }

            inputFile = *fopen*(inputFilename, "r");

*if* (inputFile == NULL)

            {

*printf*("Error opening file %s.\n", inputFilename);

*return* 1;

            }

            int n;

*fscanf*(inputFile, "%d", &n);

            int arr[n];

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

            {

*fscanf*(inputFile, "%d", &arr[i]);

            }

*fclose*(inputFile);

            comparisons = 0;

*insertionSort*(arr, n, &comparisons);

*printArrayToFile*(arr, n, outputFilename);

*printf*("Before Sorting: ");

*displayFileContent*(inputFilename);

*printf*("\nAfter Sorting: ");

*displayFileContent*(outputFilename);

*printf*("\nNumber of Comparisons: %lld\n", comparisons);

*if* (comparisons == 0)

            {

*printf*("Scenario: Best-case\n");

            }

*else* *if* (comparisons == (n \* (n - 1)) / 2)

            {

*printf*("Scenario: Worst-case\n");

            }

        }

*else* *if* (choice != 4)

        {

*printf*("Invalid choice. Please enter a valid option.\n");

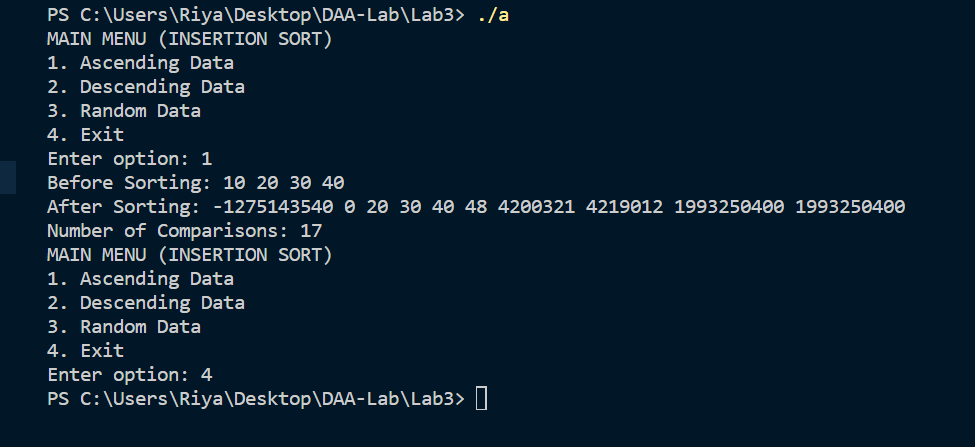
        }

    } *while* (choice != 4);

*return* 0;

}

**Output:**



4.1 Aim of the program: Write a menu driven program to sort list of array elements using Merge

Sort technique and calculate the execution time only to sort the elements. Count the number of

comparisons.

Note#

 To calculate execution time, assume that single program is under execution in the CPU.

 Number of elements in each input file should vary from 300 to 500 entries.

 For ascending order: Read data from a file “inAsce.dat” having content 10 20 30 40.....,

Store the result in “outMergeAsce.dat”.

 For descending order: Read data from a file “inDesc.dat” having content 90 80 70 60....,

Store the result in “outMergeDesc.dat”.

 For random data: Read data from a file “inRand.dat” having content 55 66 33 11 44 ...,

Store the result in “outMergeRand.dat”

Sample Input from file:

MAIN MENU (MERGE SORT)

1. Ascending Data

2. Descending Data

3. Random Data

4. ERROR (EXIT)

Output:

Enter option: 1

Before Sorting: Content of the input file

After Sorting: Content of the output file

Number of Comparisons: Actual

Execution Time: lapse time in nanosecond

#*include* <stdio.h>

#*include* <stdlib.h>

#*include* <time.h> // *time related function*

// *merge function for merging two sorted sub-arrays into a single sorted array.*

void *merge*(int arr*[]*, int l, int m, int r, long long \*comparisons)

// *arr[]- Array to be sorted*

// *l - left index of the sub-array to be merged*

// *m - middle index of the sub-array to be merged*

// *r - right index of the sub-array to be merged.*

// *\*comparisons- pointer to a variable keeping track of no. of comparisons during sorting*

{

    int n1 = m - l + 1;

    int n2 = r - m;

    // *calculating the sizes of the two sub-arrays to be merged*

    int L[n1], R[n2];// *two temporary arrays L and R to hold elements of the two sub-arrays*

*for* (int i = 0; i < n1; i++)

        L[i] = arr[l + i];

*for* (int j = 0; j < n2; j++)

        R[j] = arr[m + 1 + j];

    // *loops copying the elements from the main array arr into the temporary arrays L and R*

    int i = 0, j = 0, k = l;

*while* (i < n1 && j < n2)

    {

        (\*comparisons)++;

*if* (L[i] <= R[j])

        {

            arr[k] = L[i];

            i++;

        }

*else*

        {

            arr[k] = R[j];

            j++;

        }

        k++;

    }

    // *compares the elements in the temporary arrays L and R and then places the smaller*

    // *element into the main array arr*

*while* (i < n1)

    {

        arr[k] = L[i];

        i++;

        k++;

    }

    // *handles any remaining elements in the L*

    // *copies the remaining elements into the main array arr*

*while* (j < n2)

    {

        arr[k] = R[j];

        j++;

        k++;

    }

    // *handles any remaining elements in the R*

    // *copies them into the main array arr.*

}

// *Function performing Merge sort algorithm recursively on the input array.*

void *mergeSort*(int arr*[]*, int l, int r, long long \*comparisons)

{

*if* (l < r)

    {

        int m = l + (r - l) / 2;// *dividing the array into smaller sub-arrays*

*mergeSort*(arr, l, m, comparisons);

*mergeSort*(arr, m + 1, r, comparisons);// *sorting the array*

*merge*(arr, l, m, r, comparisons);// *merging them back together*

    }

}

int *main*()

{

    FILE \*inputFile, \*outputFile;

    int choice, n;

*printf*("MAIN MENU (MERGE SORT)\n");

*printf*("1. Ascending Data\n");

*printf*("2. Descending Data\n");

*printf*("3. Random Data\n");

*printf*("4. Exit\n");

*printf*("Enter option: ");

*scanf*("%d", &choice);

*switch* (choice)

    {

*case* 1:

        inputFile = *fopen*("inAsce.dat", "r");

        outputFile = *fopen*("outMergeAsce.dat", "w");

*break*;

*case* 2:

        inputFile = *fopen*("inDesc.dat", "r");

        outputFile = *fopen*("outMergeDesc.dat", "w");

*break*;

*case* 3:

        inputFile = *fopen*("inRand.dat", "r");

        outputFile = *fopen*("outMergeRand.dat", "w");

*break*;

*case* 4:

*return* 0;

    }

*fscanf*(inputFile, "%d", &n);// *reading no. of elements from the input file*

    int arr[n];

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

*fscanf*(inputFile, "%d", &arr[i]);

*fclose*(inputFile);// *input File closed*

    long long comparisons = 0;// *initializing a variable 'comparisons' for keeping track*

    // *of the no. of comparisons made during sorting*

    clock\_t start, end;

    // *To store clock ticks before and after the sorting process starts*

    start = *clock*();

    // *clock() - returns the processor time consumed by the clock ticks*

*mergeSort*(arr, 0, n - 1, &comparisons);

    // *Initiates the merge sort process on the array. Starts from index 0 to n-1*

    end = *clock*();// *end time using clock()*

    double elapsedTime = (double)(end - start) / CLOCKS\_PER\_SEC \* 1000000000.0;

    // *elapsed time- is calculated by taking the difference between end and start and then*

    // *converting it to nanoseconds.*

    // *CLOCKS\_PER\_SEC- defines the number of clock ticks per second.*

*fprintf*(outputFile, "Sorted Array:\n");

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

*fprintf*(outputFile, "%d ", arr[i]);

*fprintf*(outputFile, "\nNumber of Comparisons: %lld\n", comparisons);

*fprintf*(outputFile, "Execution Time: %.2lf nanoseconds\n", elapsedTime);

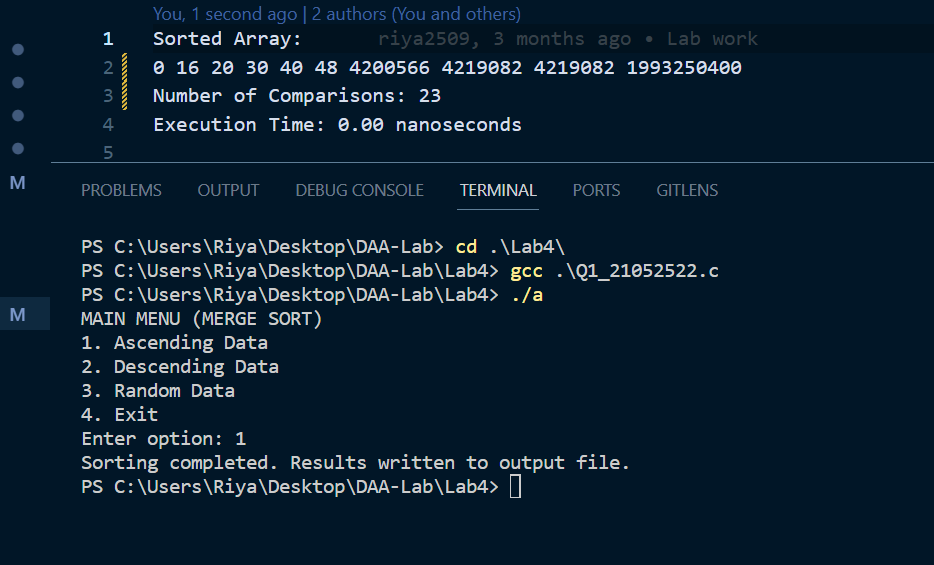
*fclose*(outputFile);

*printf*("Sorting completed. Results written to output file.\n");

*return* 0;

}

**Output:**



4.2 Aim of the program: Write a menu driven program to sort a list of elements in ascending

order using Quick Sort technique. Each choice for the input data has its own disc file. A separate

output file can be used for sorted elements. After sorting display the content of the output file

along with number of comparisons. Based on the partitioning position for each recursive call,

conclude the input scenario is either best-case partitioning or worst-case partitioning.

Note#

 The worst-case behavior for quicksort occurs when the partitioning routine produces one

subproblem with n-1 elements and one with 0 elements. The best-case behaviour

occurred in most even possible split, PARTITION produces two subproblems, each of

size no more than n/2.

 Number of elements in each input file should vary from 300 to 500 entries.

 For ascending order: Read data from a file “inAsce.dat” having content 10 20 30 40.....,

Store the result in “outQuickAsce.dat”.

 For descending order: Read data from a file “inDesc.dat” having content 90 80 70 60....,

Store the result in “outQuickDesc.dat”.

 For random data: Read data from a file “inRand.dat” having content 55 66 33 11 44 ...,

Store the result in “outQuickRand.dat”

Sample Input from file:

MAIN MENU (QUICK SORT)

1. Ascending Data

2. Descending Data

3. Random Data

4. ERROR (EXIT)

Output:

Enter option: 1

Before Sorting: Content of the input file

After Sorting: Content of the output file

Number of Comparisons: Actual

Scenario: Best or Worst-case

#*include* <stdio.h>

// *Function to swap two elements*

void *swap*(int \*a, int \*b)

{

    int temp = \*a;

    \*a = \*b;

    \*b = temp;

}

// *Function to perform partitioning for Quick Sort*

int *partition*(int arr*[]*, int low, int high, long long int \*comparisons)

{

    int pivot = arr[high];

    int i = low - 1;

*for* (int j = low; j <= high - 1; j++)

    {

        (\*comparisons)++;

*if* (arr[j] < pivot)

        {

            i++;

*swap*(&arr[i], &arr[j]);

        }

    }

*swap*(&arr[i + 1], &arr[high]);

*return* i + 1;

}

// *Function to perform Quick Sort*

void *quickSort*(int arr*[]*, int low, int high, long long int \*comparisons)

{

*if* (low < high)

    {

        int pivotIndex = *partition*(arr, low, high, comparisons);

*quickSort*(arr, low, pivotIndex - 1, comparisons);

*quickSort*(arr, pivotIndex + 1, high, comparisons);

    }

}

// *Function to read data from a file and return the number of elements read*

int *readDataFromFile*(*const* char \*filename, int arr*[]*)

{

    FILE \*file = *fopen*(filename, "r");

*if* (file == NULL)

    {

*printf*("Error opening file %s\n", filename);

*return* 0;

    }

    int numElements = 0;

*while* (*fscanf*(file, "%d", &arr[numElements]) != EOF)

    {

        numElements++;

    }

*fclose*(file);

*return* numElements;

}

// *Function to write sorted data to a file*

void *writeSortedDataToFile*(*const* char \*filename, int arr*[]*, int numElements)

{

    FILE \*file = *fopen*(filename, "w");

*if* (file == NULL)

    {

*printf*("Error opening file %s for writing\n", filename);

*return*;

    }

*for* (int i = 0; i < numElements; i++)

    {

*fprintf*(file, "%d ", arr[i]);

    }

*fclose*(file);

}

int *main*()

{

    int choice;

*printf*("MAIN MENU (QUICK SORT)\n");

*printf*("1. Ascending Data\n");

*printf*("2. Descending Data\n");

*printf*("3. Random Data\n");

*printf*("4. Exit\n");

*printf*("Enter your choice: ");

*scanf*("%d", &choice);

    int arr[500];// *Maximum size based on our requirement*

    int numElements;

*const* char \*inputFilename;

*const* char \*outputFilename;

*switch* (choice)

    {

*case* 1:

        inputFilename = "inAsce.dat";

        outputFilename = "outQuickAsce.dat";

*break*;

*case* 2:

        inputFilename = "inDesc.dat";

        outputFilename = "outQuickDesc.dat";

*break*;

*case* 3:

        inputFilename = "inRand.dat";

        outputFilename = "outQuickRand.dat";

*break*;

*case* 4:

*printf*("Exiting...\n");

*return* 0;

*default*:

*printf*("Invalid choice\n");

*return* 1;

    }

    numElements = *readDataFromFile*(inputFilename, arr);

*if* (numElements == 0)

    {

*printf*("No data read from file %s\n", inputFilename);

*return* 1;

    }

    long long int comparisons = 0;

*quickSort*(arr, 0, numElements - 1, &comparisons);

*writeSortedDataToFile*(outputFilename, arr, numElements);

*printf*("Sorted data written to %s\n", outputFilename);

*printf*("Number of comparisons: %lld\n", comparisons);

    // *Determining whether it's best-case or worst-case partitioning*

*if* (comparisons == ((numElements \* (numElements - 1)) / 2))

    {

*printf*("Partitioning is worst-case\n");

    }

*else* *if* (comparisons == (numElements - 1))

    {

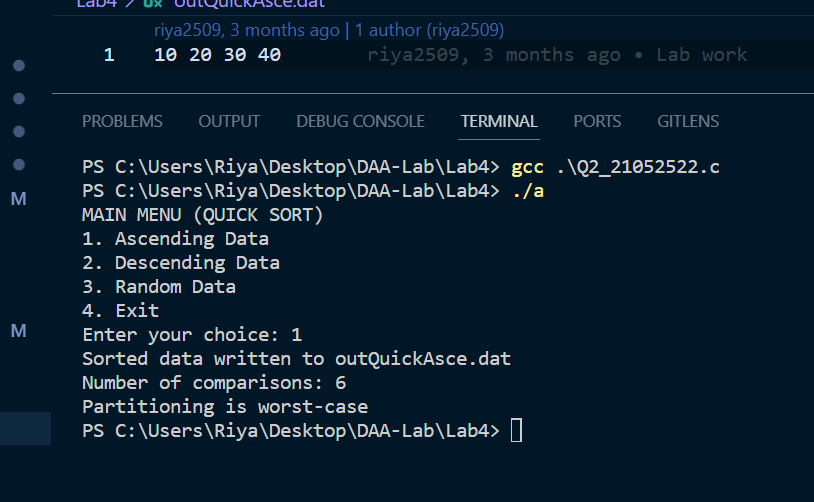
*printf*("Partitioning is best-case\n");

    }

*return* 0;

}

**Output:**



5.1 Aim of the program:

Define a struct person as follows:

struct person

{

int id;

char \*name;

int age;

int height;

int weight;

};

Write a menu driven program to read the data of „n‟ students from a file and store them in a

dynamically allocated array of struct person. Implement the min-heap or max-heap and its

operations based on the menu options.

Sample Input/Output:

MAIN MENU (HEAP)

1. Read Data

2. Create a Min-heap based on the age

3. Create a Max-heap based on the weight

4. Display weight of the youngest person

5. Insert a new person into the Min-heap

6. Delete the oldest person

7. Exit

Enter option: 1

Id Name Age Height Weight(pound)

0 Adarsh Hota 39 77 231

1 Levi Maier 56 77 129

2 Priya Kumari 63 78 240

3 Dorothy Helton 47 72 229

4 Florence Smith 24 75 171

5 Erica Anyan 38 73 102

6 Norma Webster 23 75 145

Enter option: 4

Weight of youngest student: 65.77 kg

Note#: Other menu choices are similarly verified.

#*include* <stdio.h>

#*include* <stdlib.h>

#*include* <string.h>

struct person

{

    int id;

    char \*name;

    int age;

    int height;

    int weight;

};

// *Function to swap two person structs*

void *swap*(struct person \*a, struct person \*b)

{

    struct person temp = \*a;

    \*a = \*b;

    \*b = temp;

}

// *Function to create a min-heap based on age*

void *minHeapify*(struct person \*arr, int n, int i)

{

    int smallest = i;

    int left = 2 \* i + 1;

    int right = 2 \* i + 2;

*if* (left < n && arr[left].age < arr[smallest].age)

        smallest = left;

*if* (right < n && arr[right].age < arr[smallest].age)

        smallest = right;

*if* (smallest != i)

    {

*swap*(&arr[i], &arr[smallest]);

*minHeapify*(arr, n, smallest);

    }

}

// *Function to create a max-heap based on weight*

void *maxHeapify*(struct person \*arr, int n, int i)

{

    int largest = i;

    int left = 2 \* i + 1;

    int right = 2 \* i + 2;

*if* (left < n && arr[left].weight > arr[largest].weight)

        largest = left;

*if* (right < n && arr[right].weight > arr[largest].weight)

        largest = right;

*if* (largest != i)

    {

*swap*(&arr[i], &arr[largest]);

*maxHeapify*(arr, n, largest);

    }

}

// *Function to build a min-heap or max-heap based on user choice*

void *buildHeap*(struct person \*arr, int n, int choice)

{

*for* (int i = n / 2 - 1; i >= 0; i--)

    {

*if* (choice == 2)

*maxHeapify*(arr, n, i);// *Build max-heap*

*else*

*minHeapify*(arr, n, i);// *Build min-heap*

    }

}

// *Function to display the weight of the youngest person*

void *displayYoungestWeight*(struct person \*arr, int n)

{

*if* (n > 0)

    {

*printf*("Weight of the youngest person: %d\n", arr[0].weight);

    }

*else*

    {

*printf*("Heap is empty.\n");

    }

}

// *Function to insert a new person into the min-heap*

struct person \**insertPerson*(struct person \*arr, int \*n, struct person newPerson)

{

    (\*n)++;

    arr = (struct person \*)*realloc*(arr, (\*n) \* sizeof(struct person));

    arr[(\*n) - 1] = newPerson;

    int i = (\*n) - 1;

*while* (i > 0 && arr[(i - 1) / 2].age > arr[i].age)

    {

*swap*(&arr[i], &arr[(i - 1) / 2]);

        i = (i - 1) / 2;

    }

*return* arr;

}

// *Function to delete the oldest person from the min-heap*

struct person \**deleteOldestPerson*(struct person \*arr, int \*n)

{

*if* (\*n <= 0)

    {

*printf*("Heap is empty.\n");

*return* arr;

    }

*if* (\*n == 1)

    {

        (\*n)--;

*free*(arr);

        arr = NULL;

*printf*("Oldest person deleted.\n");

*return* arr;

    }

*swap*(&arr[0], &arr[(\*n) - 1]);

    (\*n)--;

    arr = (struct person \*)*realloc*(arr, (\*n) \* sizeof(struct person));

*minHeapify*(arr, \*n, 0);

*printf*("Oldest person deleted.\n");

*return* arr;

}

int *main*()

{

    int choice, n = 0;

    struct person \*people = NULL;

*while* (1)

    {

*printf*("\nMAIN MENU (HEAP)\n");

*printf*("1. Read Data\n");

*printf*("2. Create a Min-heap based on the age\n");

*printf*("3. Create a Max-heap based on the weight\n");

*printf*("4. Display weight of the youngest person\n");

*printf*("5. Insert a new person into the Min-heap\n");

*printf*("6. Delete the oldest person\n");

*printf*("7. Exit\n");

*printf*("Enter your choice: ");

*scanf*("%d", &choice);

*switch* (choice)

        {

*case* 1:

        {

            // *Read data from a file*

            FILE \*file = *fopen*("data.txt", "r");

*if* (file == NULL)

            {

*printf*("Error opening the file.\n");

*break*;

            }

            int id, age, height, weight;

            char name[100];

*while* (*fscanf*(file, "%d %s %d %d %d", &id, name, &age, &height, &weight) != EOF)

            {

                struct person newPerson;

                newPerson.id = id;

                newPerson.name = *strdup*(name);

                newPerson.age = age;

                newPerson.height = height;

                newPerson.weight = weight;

                people = *insertPerson*(people, &n, newPerson);

            }

*fclose*(file);

*printf*("Data read and stored.\n");

*break*;

        }

*case* 2:

            // *Create a Min-heap based on age*

*buildHeap*(people, n, 1);

*printf*("Min-heap based on age created.\n");

*break*;

*case* 3:

            // *Create a Max-heap based on weight*

*buildHeap*(people, n, 2);

*printf*("Max-heap based on weight created.\n");

*break*;

*case* 4:

            // *Display weight of the youngest person*

*displayYoungestWeight*(people, n);

*break*;

*case* 5:

        {

            // *Insert a new person into the Min-heap*

            struct person newPerson;

*printf*("Enter the new person's id: ");

*scanf*("%d", &newPerson.id);

*printf*("Enter the new person's name: ");

*scanf*("%s", newPerson.name);

*printf*("Enter the new person's age: ");

*scanf*("%d", &newPerson.age);

*printf*("Enter the new person's height: ");

*scanf*("%d", &newPerson.height);

*printf*("Enter the new person's weight: ");

*scanf*("%d", &newPerson.weight);

            people = *insertPerson*(people, &n, newPerson);

*printf*("New person inserted into the Min-heap.\n");

*break*;

        }

*case* 6:

            // *Delete the oldest person from the Min-heap*

            people = *deleteOldestPerson*(people, &n);

*break*;

*case* 7:

            // *Exit*

*if* (people != NULL)

            {

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

                {

*free*(people[i].name);

                }

*free*(people);

            }

*exit*(0);

*default*:

*printf*("Invalid choice. Please try again.\n");

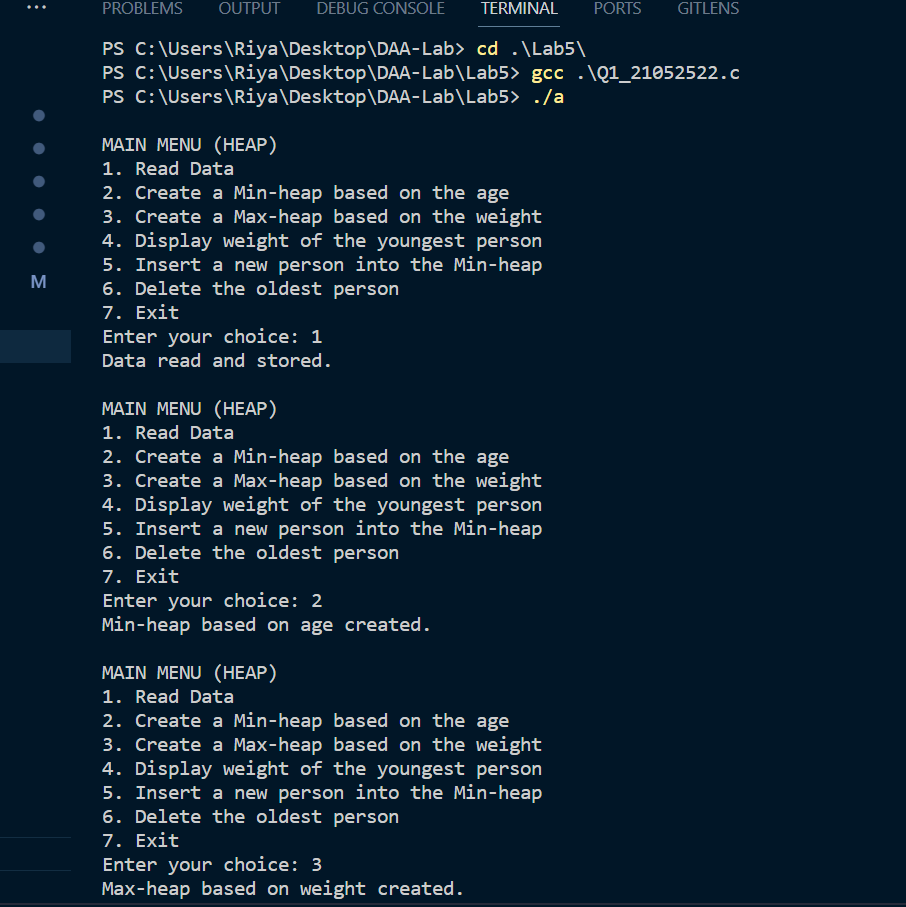
        }

    }

*return* 0;

}

**Output:**



6.1 Aim of the program: Write a program to find the maximum profit nearest to but not

exceeding the given knapsack capacity using the Fractional Knapsack algorithm.

Notes# Declare a structure ITEM having data members item\_id, item\_profit, item\_weight and

profit\_weight\_ratio. Apply heap sort technique to sort the items in non-increasing order,

according to their profit /weight.

Input:

Enter the number of items: 3

Enter the profit and weight of item no 1: 27 16

Enter the profit and weight of item no 2: 14 12

Enter the profit and weight of item no 3: 26 13

Enter the capacity of knapsack:18

Output:

Item No profit Weight Amount to be taken

3 26.000000 13.000000 1.000000

1 27.000000 16.000000 0.312500

2 14.000000 12.000000 0.000000

Maximum profit: 34.437500

6.2Aim of the program: Huffman coding assigns variable length codewords to fixed length input

characters based on their frequencies or probabilities of occurrence. Given a set of characters

along with their frequency of occurrences, write a c program to construct a Huffman tree.

Note#

 Declare a structure SYMBOL having members alphabet and frequency. Create a Min-

Priority Queue, keyed on frequency attributes.

 Create an array of structures where size=number of alphabets.

Input:

Enter the number of distinct alphabets: 6

Enter the alphabets: a b c d e f

Enter its frequencies: 45 13 12 16 9 5

Output:

In-order traversal of the tree (Huffman): a c b f e d

#*include* <stdio.h>

#*include* <stdlib.h>

// *Defined a structure to represent an item*

struct Item

{

    int item\_id;// *to store  item's identifier,*

    double item\_profit;// *to store the item's profit*

    double item\_weight;// *store the item's weight*

    double profit\_weight\_ratio;// *to store the ratio of profit to weight for sorting purposes*

};

// *Function to perform max heapify for heap sort*

void *maxHeapify*(struct Item arr*[]*, int n, int i)

{

    int largest = i;

    int left = 2 \* i + 1;

    int right = 2 \* i + 2;

*if* (left < n && arr[left].profit\_weight\_ratio > arr[largest].profit\_weight\_ratio)

        largest = left;

*if* (right < n && arr[right].profit\_weight\_ratio > arr[largest].profit\_weight\_ratio)

        largest = right;

*if* (largest != i)

    {

        struct Item temp = arr[i];

        arr[i] = arr[largest];

        arr[largest] = temp;

*maxHeapify*(arr, n, largest);

    }

}

// *heap sort algorithm to sort the array of items based on their profit-to-weight ratios.*

void *heapSort*(struct Item arr*[]*, int n)

{

*for* (int i = n / 2 - 1; i >= 0; i--)

*maxHeapify*(arr, n, i);

*for* (int i = n - 1; i >= 0; i--)

    {

        struct Item temp = arr[0];

        arr[0] = arr[i];

        arr[i] = temp;

*maxHeapify*(arr, i, 0);

    }

}

int *main*()

{

    FILE \*inputFile = *fopen*("input.txt", "r");

    FILE \*outputFile = *fopen*("output.txt", "w");

*if* (inputFile == NULL || outputFile == NULL)

    {

*printf*("File not found or cannot be opened.\n");

*return* 1;

    }

    int num\_items;

*printf*("Enter the number of items: ");

*fscanf*(inputFile, "%d", &num\_items);

    struct Item items[num\_items];

    // *Input item details and calculate profit-to-weight ratios*

*for* (int i = 0; i < num\_items; i++)

    {

        items[i].item\_id = i + 1;

*fscanf*(inputFile, "%lf %lf", &items[i].item\_profit, &items[i].item\_weight);

        items[i].profit\_weight\_ratio = items[i].item\_profit / items[i].item\_weight;

    }

    // *reading the knapsack capacity from the input file,*

    // *for input and storing it in the knapsack\_capacity variable*

    double knapsack\_capacity;

*printf*("Enter the capacity of knapsack: ");

*fscanf*(inputFile, "%lf", &knapsack\_capacity);

    // *Sort items in non-increasing order based on profit-to-weight ratio*

*heapSort*(items, num\_items);

    double total\_profit = 0.0;

    double current\_weight = 0.0;

*fprintf*(outputFile, "Item No profit Weight Amount to be taken\n");

    // *loop iterates through the sorted items and selects items for the knapsack*

*for* (int i = 0; i < num\_items; i++)

    {

*if* (current\_weight + items[i].item\_weight <= knapsack\_capacity)

        {

            // *Take the whole item*

*fprintf*(outputFile, "%d %.6lf %.6lf 1.000000\n", items[i].item\_id, items[i].item\_profit, items[i].item\_weight);

            total\_profit += items[i].item\_profit;

            current\_weight += items[i].item\_weight;

        }

*else*

        {

            // *Take a fraction of the item*

            double fraction = (knapsack\_capacity - current\_weight) / items[i].item\_weight;

*fprintf*(outputFile, "%d %.6lf %.6lf %.6lf\n", items[i].item\_id, items[i].item\_profit, items[i].item\_weight, fraction);

            total\_profit += (fraction \* items[i].item\_profit);

            current\_weight = knapsack\_capacity;

*break*;

        }

    }

*fprintf*(outputFile, "Maximum profit: %.6lf\n", total\_profit);

*fclose*(inputFile);

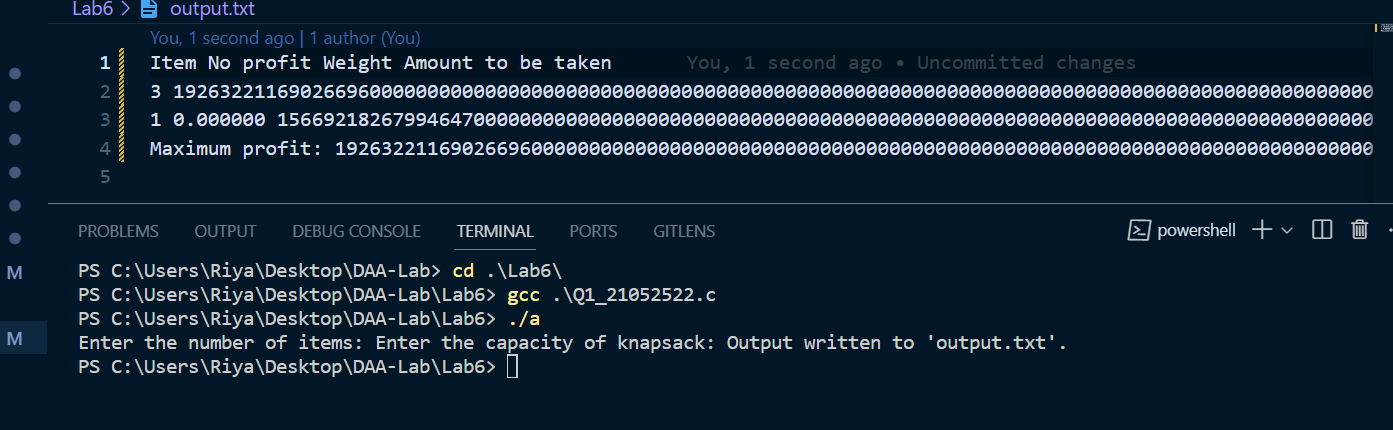
*fclose*(outputFile);

*printf*("Output written to 'output.txt'.\n");

*return* 0;

}

**Output:**



7.1 Aim of the program: Write a program to implement the matrix chain multiplication problem

using M-table & S-table to find optimal parenthesization of a matrix-chain product. Print the

number of scalar multiplications required for the given input.

Note# Dimensions of the matrices can be inputted as row and column values. Validate the

dimension compatibility.

Input:

Enter number of matrices: 4

Enter row and col size of A1: 30 35

Enter row and col size of A2: 35 15

Enter row and col size of A3: 15 5

Enter row and col size of A4: 5 10

Output:

M Table:

0 15750 7875 9375

0 0 2625 4375

0 0 0 750

0 0 0 0

S Table:

0 1 1 3

0 0 2 3

0 0 0 3

0 0 0 0

Optimal parenthesization: ( ( A1 (A2 A3)) A4)

The optimal ordering of the given matrices requires 9375 scalar multiplications.

#*include* <stdio.h>

#*include* <limits.h>

// *Function to print optimal parenthesization*

void *printOptimalParenthesis*(int S*[]*[100], int i, int j) {

*while* (i == j) {

*printf*("A%d ", i);

    }//*else {*

/\**printf("( ");*

*printOptimalParenthesis(S, i, S[i][j]);*

*printOptimalParenthesis(S, S[i][j] + 1, j);*

*printf(") ");*\*/

        //*printf("A%d ", i);*

    }

//*}*

// *Function to find the optimal parenthesization and number of scalar multiplications*

void *matrixChainOrder*(int dims*[]*, int n) {

    int M[n][n];// *M-table*

    int S[n][n];// *S-table*

    int i, j, k, L, q;

    // *Initialize M-table with zeros*

*for* (i = 1; i < n; i++) {

        M[i][i] = 0;

    }

*for* (i = 1; i < n; i++) {

        S[i][i] = 0;

    }

    // *Calculate M and S tables*

*for* (L = 2; L < n; L++) {

*for* (i = 1; i < n - L + 1; i++) {

            j = i + L - 1;

            M[i][j] = INT\_MAX;

*for* (k = i; k < j; k++) {

                q = M[i][k] + M[k + 1][j] + dims[i - 1] \* dims[k] \* dims[j];

*if* (q < M[i][j]) {

                    M[i][j] = q;

                    S[i][j] = k;

                }

            }

        }

    }

    // *Print M-table and S-table*

*printf*("M Table:\n");

*for* (i = 1; i < n; i++) {

*for* (j = 1; j < n; j++) {

*if* (j >= i) {

*printf*("%d ", M[i][j]);

            } *else* {

*printf*("0 ");

            }

        }

*printf*("\n");

    }

*printf*("S Table:\n");

*for* (i = 1; i < n; i++) {

*for* (j = 1; j < n; j++) {

*if* (j >= i) {

*printf*("%d ", S[i][j]);

            } *else* {

*printf*("0 ");

            }

        }

*printf*("\n");

    }

    // *Print optimal parenthesization*

*printf*("Optimal Parenthesization: ( ");

*printOptimalParenthesis*(S, 1, n - 1);

*printf*(")\n");

    // *Print the number of scalar multiplications*

*printf*("The optimal ordering of the given matrices requires %d scalar multiplications.\n", M[1][n - 1]);

}

int *main*() {

    int n;

*printf*("Enter the number of matrices: ");

*scanf*("%d", &n);

    int dimensions[n + 1];

*printf*("Enter row and col size for each matrix:\n");

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

*printf*("Matrix A%d: ", i + 1);

*scanf*("%d %d", &dimensions[i], &dimensions[i + 1]);

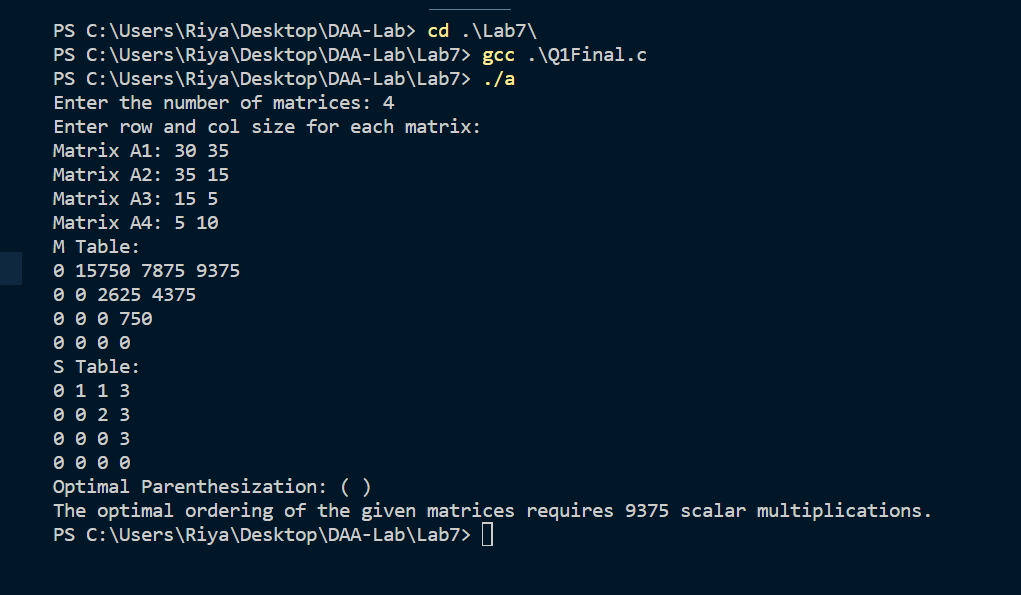
    }

*matrixChainOrder*(dimensions, n + 1);

*return* 0;

}

**Output:**



7.2 Aim of the program: Write a program to find out the Longest Common Subsequence of two

given strings. Calculate length of the LCS.

Input:

Enter the first string into an array: 10010101

Enter the second string into an array: 010110110

Output:

LCS: 100110

LCS Length: 6

#*include* <stdio.h>

#*include* <string.h>

// *Function to find the LCS and its length*

void *findLCS*(char X*[]*, char Y*[]*, int m, int n, char lcs*[]*, int\* length) {

    int L[m + 1][n + 1];

    // *Build the L[m+1][n+1] table in bottom-up fashion*

*for* (int i = 0; i <= m; i++) {

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

*if* (i == 0 || j == 0)

                L[i][j] = 0;

*else* *if* (X[i - 1] == Y[j - 1]) {

                L[i][j] = L[i - 1][j - 1] + 1;

            } *else* {

                L[i][j] = (L[i - 1][j] > L[i][j - 1]) ? L[i - 1][j] : L[i][j - 1];

            }

        }

    }

    // *Reconstruct the LCS*

    int i = m, j = n;

    int index = L[m][n];

*while* (i > 0 && j > 0) {

*if* (X[i - 1] == Y[j - 1]) {

            lcs[index - 1] = X[i - 1];

            i--;

            j--;

            index--;

        } *else* *if* (L[i - 1][j] > L[i][j - 1]) {

            i--;

        } *else* {

            j--;

        }

    }

    // *Calculate the length of LCS*

    \*length = L[m][n];

}

int *main*() {

    char X[100], Y[100], lcs[100];

    int length;

    // *Input two strings*

*printf*("Enter the first string into an array: ");

*fgets*(X, sizeof(X), stdin);

    X[*strcspn*(X, "\n")] = '\0';// *Remove newline character*

*printf*("Enter the second string into an array: ");

*fgets*(Y, sizeof(Y), stdin);

    Y[*strcspn*(Y, "\n")] = '\0';// *Remove newline character*

    // *Find LCS and its length*

    int m = *strlen*(X);

    int n = *strlen*(Y);

*findLCS*(X, Y, m, n, lcs, &length);

    // *Print the LCS*

*printf*("LCS: %s\n", lcs);

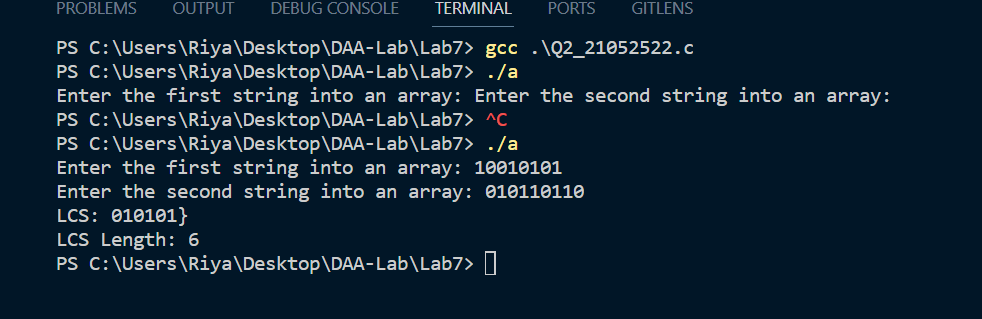
    // *Print the length of LCS*

*printf*("LCS Length: %d\n", length);

*return* 0;

}

**Output:**



8.1 Aim of the program: Consider an undirected graph where each edge weights 2 units. Each of

the nodes is labeled consecutively from 1 to n. The user will input a list of edges for describing

an undirected graph. After representation of the graph, from a given starting position

● Display the breadth-first search traversal.

● Determine and display the shortest distance to each of the other nodes using the breadth-

first search algorithm. Return an array of distances from the start node in node number

order. If a node is unreachable, return -1 for that node.

Input Format:

 The first line contains two space-separated integers „n‟ and „m‟, the number of nodes and

edges in the graph.

 Each line „i‟ of the „m‟ subsequent lines contains two space-separated integers „u‟ and

„v‟, that describe an edge between nodes „u‟ and „v‟.

 The last line contains a single integer „s‟, the node number to start from.

Output Format:

 The first line shows the result of the BFS traversal.

 The last line shows an array of distances from node „s‟ to all other nodes.

Input:

5 4

1 2

1 4

4 5

2 5

1

Output:

BFS Traversal: 1 2 4 5

Distance [2 -1 2 4]

#*include* <stdio.h>

#*include* <stdlib.h>

#*include* <stdbool.h>

#*include* <string.h>

#*define* *MAX\_NODES* 1000// *Adjust this based on the maximum number of nodes in the graph*

// *Structure to represent a node in the adjacency list*

typedef struct Node {

    int data;

    struct Node\* next;

} Node;

// *Structure to represent the adjacency list*

typedef struct Graph {

    Node\* adjacencyList[MAX\_NODES];

} Graph;

// *Function to add an edge to the adjacency list*

void *addEdge*(Graph\* graph, int u, int v) {

    Node\* newNode = (Node\*)*malloc*(sizeof(Node));

    newNode->data = v;

    newNode->next = graph->adjacencyList[u];

    graph->adjacencyList[u] = newNode;

    newNode = (Node\*)*malloc*(sizeof(Node));

    newNode->data = u;

    newNode->next = graph->adjacencyList[v];

    graph->adjacencyList[v] = newNode;

}

// *Function to perform BFS and calculate distances*

void *bfs\_shortest\_distances*(Graph\* graph, int n, int start, int distances*[]*) {

    bool visited[MAX\_NODES] = {false};

    int queue[MAX\_NODES];

    int front = 0, rear = 0;

    queue[rear++] = start;

    visited[start] = true;

    distances[start] = 0;

*while* (front < rear) {

        int node = queue[front++];

        Node\* current = graph->adjacencyList[node];

*while* (current != NULL) {

            int neighbor = current->data;

*if* (!visited[neighbor]) {

                distances[neighbor] = distances[node] + 2;

                queue[rear++] = neighbor;

                visited[neighbor] = true;

            }

            current = current->next;

        }

    }

}

int *main*() {

    int n, m;

*scanf*("%d %d", &n, &m);

    Graph graph;

*memset*(graph.adjacencyList, 0, sizeof(graph.adjacencyList));

*for* (int i = 0; i < m; i++) {

        int u, v;

*scanf*("%d %d", &u, &v);

*addEdge*(&graph, u, v);

    }

    int start;

*scanf*("%d", &start);

    int distances[MAX\_NODES];

*memset*(distances, -1, sizeof(distances));

*bfs\_shortest\_distances*(&graph, n, start, distances);

*printf*("BFS Traversal Order: ");

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

*printf*("%d ", i);

    }

*printf*("\n");

*printf*("Shortest Distances from Node %d: ", start);

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

*printf*("%d ", distances[i]);

    }

*printf*("\n");

*return* 0;

}

**Output:**

