# Lab Day 1: Revision of Data Structures

***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*

//Find second largest and second smallest element in an array

#include<stdio.h>

#include<limits.h>

int i;

void secondLargest( int arr[] , int n){

    int first,second;

    first = second = INT\_MIN;

    if(n<2){

        printf("Invalid input");

        return;

    }

    for(i=0; i<n; i++){

        if(arr[i] > first){

            second = first;

            first = arr[i];

        }

        else if(arr[i] > second && arr[i]!=first)

            second = arr[i];

    }

    if(second==INT\_MIN)

        printf("There is no second largest element");

    else

        printf("Second Largest Element: %d\n",second);

}

void secondSmallest( int arr[] , int n){

    int first,second;

    first = second = INT\_MAX;

    if(n<2){

        printf("Invalid input");

        return;

    }

    for(i=0; i<n; i++){

        if(arr[i] < first){

            second = first;

            first = arr[i];

        }

        else if(arr[i] < second && arr[i]!=first)

            second = arr[i];

    }

    if(second==INT\_MAX)

        printf("There is no second largest element");

    else

        printf("Second Smallest Element: %d",second);

}

int main(){

    int n;

    printf("Enter size of array: ");

    scanf("%d",&n);

    int arr[n];

    printf("Enter %d elements in array: ",n);

    for(i=0; i<n; i++){

        scanf("%d",&arr[i]);

    }

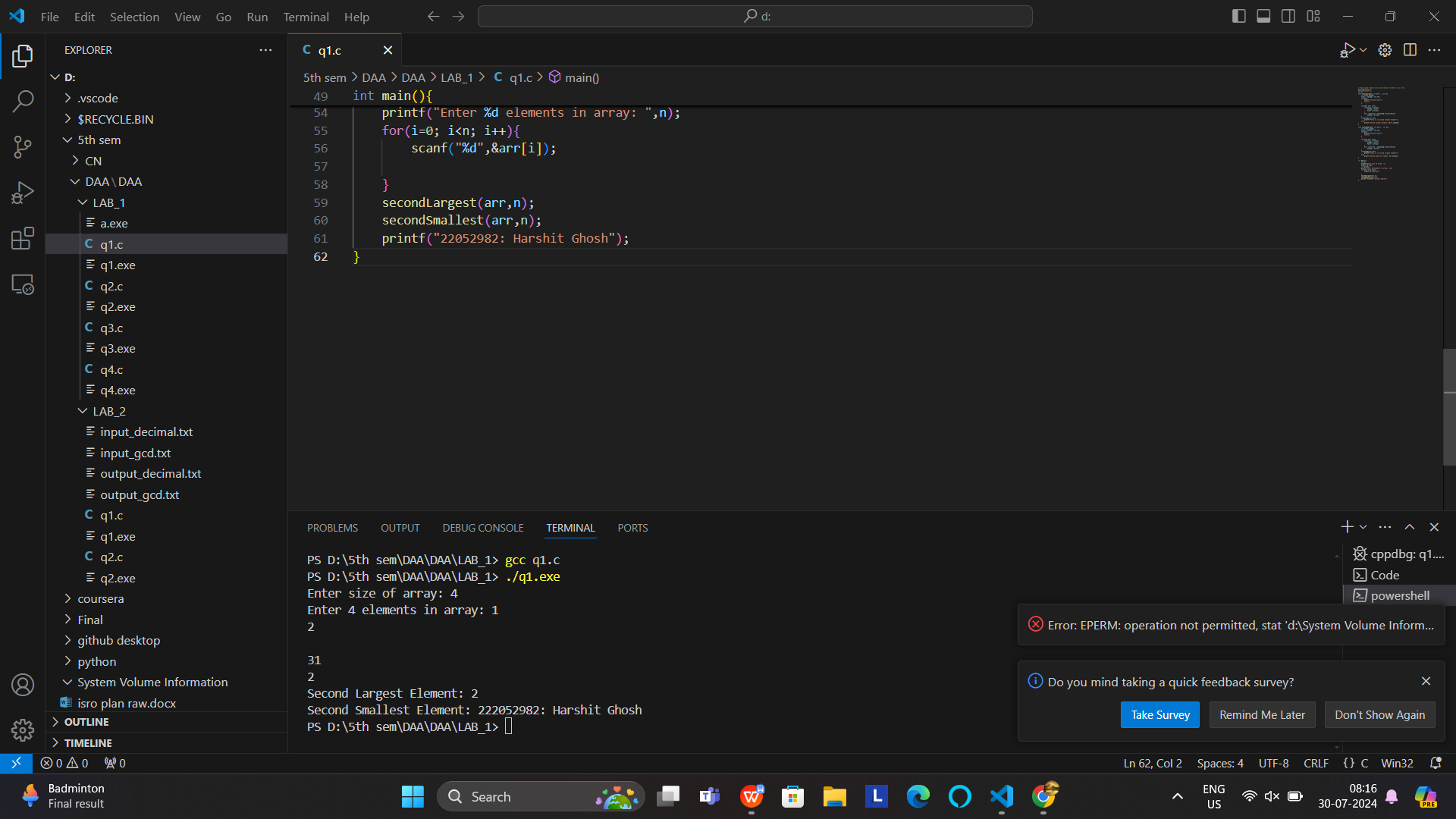
    secondLargest(arr,n);

    secondSmallest(arr,n);

    printf("22052982: Harshit Ghosh");

}

***INPUT/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].*

//PrefixSum array

#include<stdio.h>

int i;

void prefixSum(int arr[], int n){

    int sum = 0;

    int prefixSum[n];

    for(i=0; i<n; i++){

        sum += arr[i];

        prefixSum[i] = sum;

    }

    printf("Prefix Sum Array: ");

    for(i=0; i<n; i++)

        printf("%d ",prefixSum[i]);

}

int main(){

    int n;

    printf("Enter size of array: ");

    scanf("%d",&n);

    int arr[n];

    printf("Enter %d elements in array: ",n);

    for(i=0; i<n; i++){

        scanf("%d",&arr[i]);

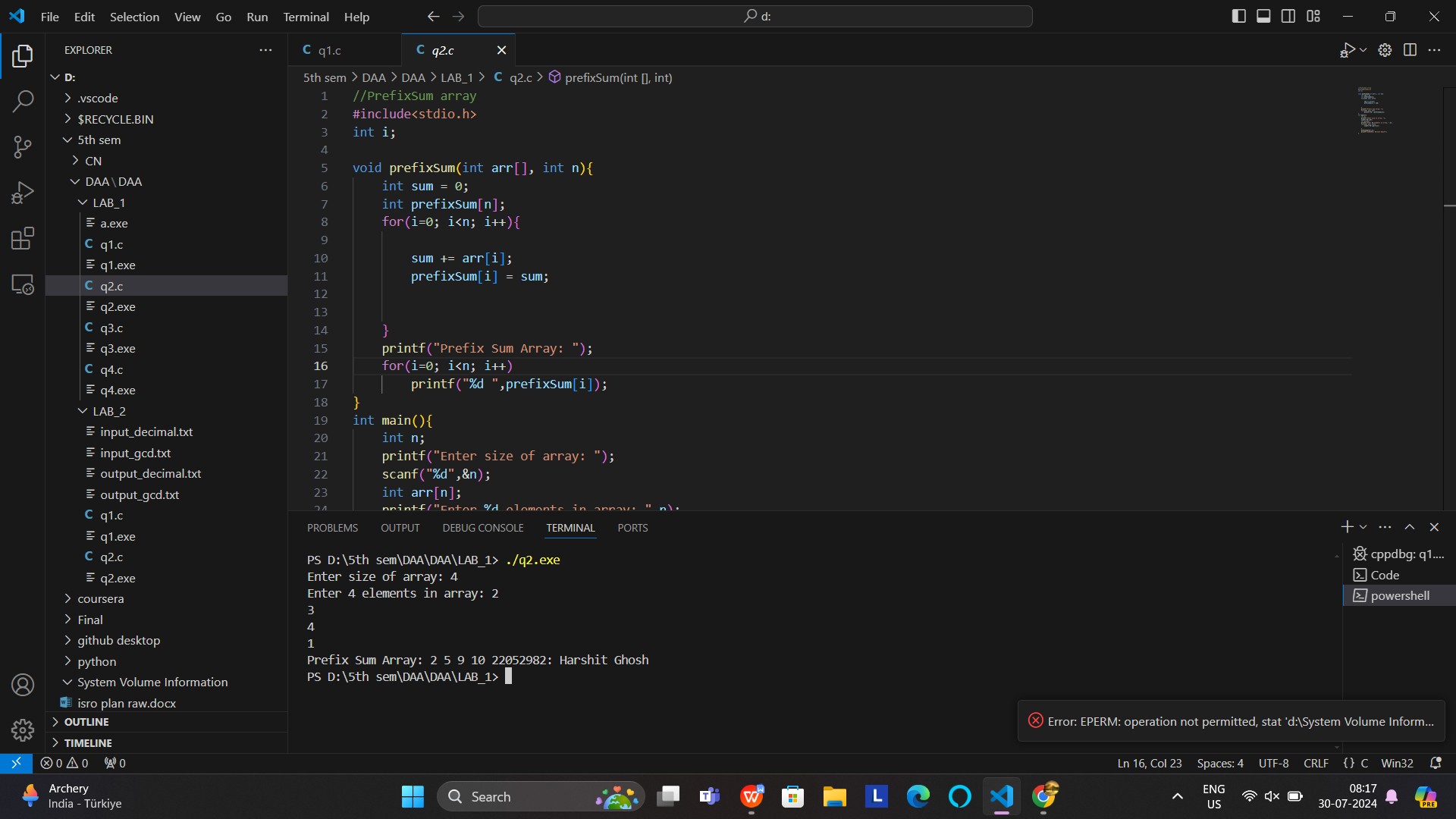
    }

    prefixSum(arr,n);

    printf("22052982: Harshit Ghosh");

}

***INPUT/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.*

//Find number of elements that have duplicates and element with max repetition

#include<stdio.h>

#include<stdlib.h>

int i;

#define MAX\_SIZE 1000

int countDuplicates(int arr[], int n, int count[]){

    int duplicates = 0;

    for(i=0; i<n; i++){

        count[arr[i]]++;

        if(count[arr[i]] == 2)

            duplicates++;

    }

    return duplicates;

}

int elementFrequency(int count[] , int size){

    int maxCount = 0 , maxElement = 0;

    for(i=0; i<size; i++){

        if(count[i] > maxCount){

        maxCount = count[i];

        maxElement = i;

        }

    }

    return maxElement;

}

int main(){

    int n;

    int count[MAX\_SIZE] = {0};

    printf("Enter size of array: ");

    scanf("%d",&n);

    int arr[n];

    printf("Enter %d elements in array: ",n);

    for(i=0; i<n; i++){

        scanf("%d",&arr[i]);

    }

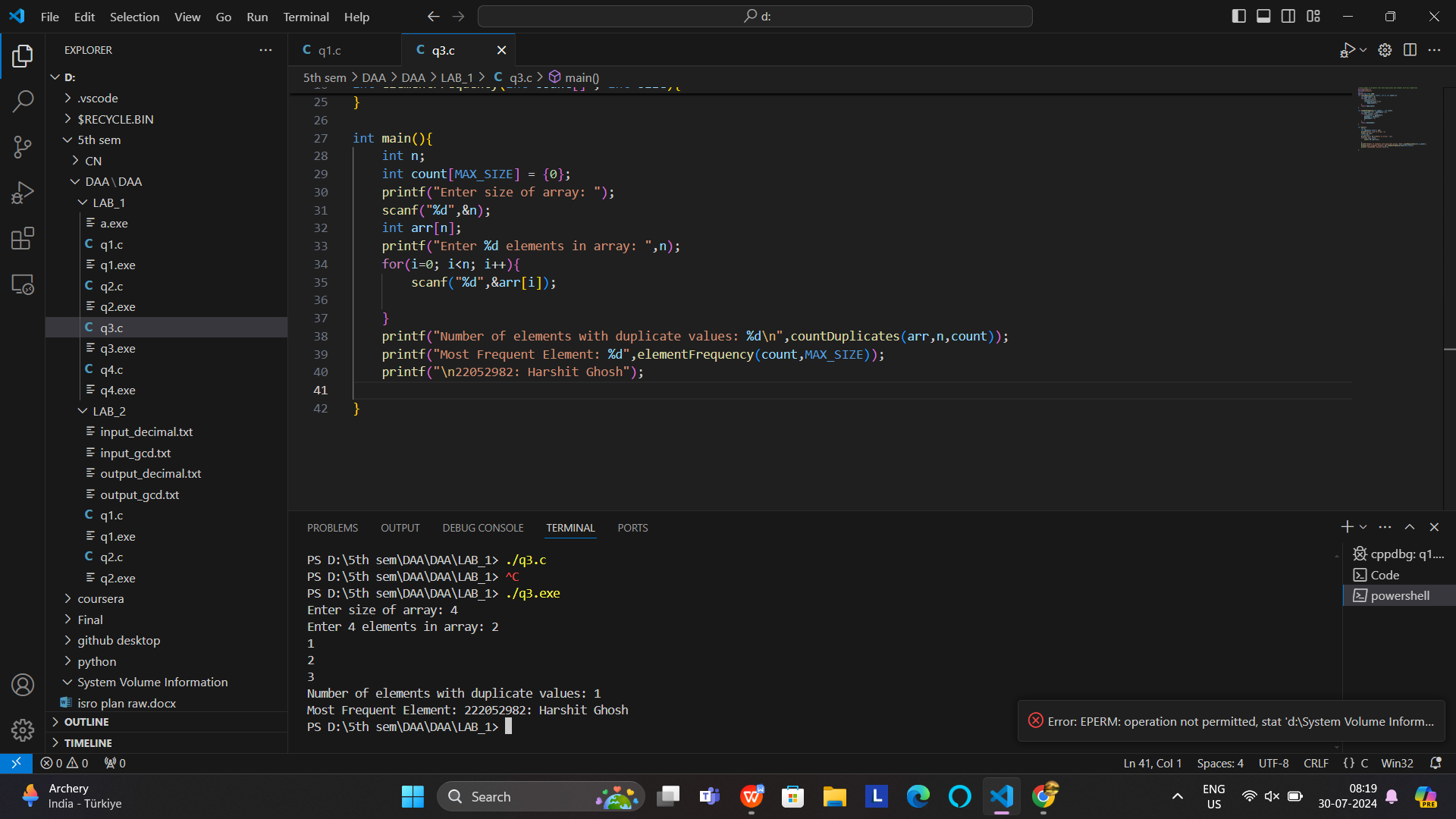
    printf("Number of elements with duplicate values: %d\n",countDuplicates(arr,n,count));

    printf("Most Frequent Element: %d",elementFrequency(count,MAX\_SIZE));

    printf("\n22052982: Harshit Ghosh");

}

***INPUT/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.*

#include <stdio.h>

void exchange(int \*p, int \*q) {

    int temp = \*p;

    \*p = \*q;

    \*q = temp;

}

void rotateRight(int \*p1, int p2) {

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

        exchange(&p1[i], &p1[i-1]);

    }

}

void printArray(int \*arr, int size) {

    for (int i = 0; i < size; i++) {

        printf("%d ", arr[i]);

    }

    printf("\n");

}

int main() {

    int size, rotations;

    // User input for array size

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

    scanf("%d", &size);

    int A[size];

    // User input for array elements

    printf("Enter %d elements of the array: ", size);

    for (int i = 0; i < size; i++) {

        scanf("%d", &A[i]);

    }

    // User input for the number of rotations

    printf("Enter the number of rotations: ");

    scanf("%d", &rotations);

    printf("Before rotate: ");

    printArray(A, size);

    // Perform the rotation

    rotateRight(A, rotations);

    printf("After rotate:  ");

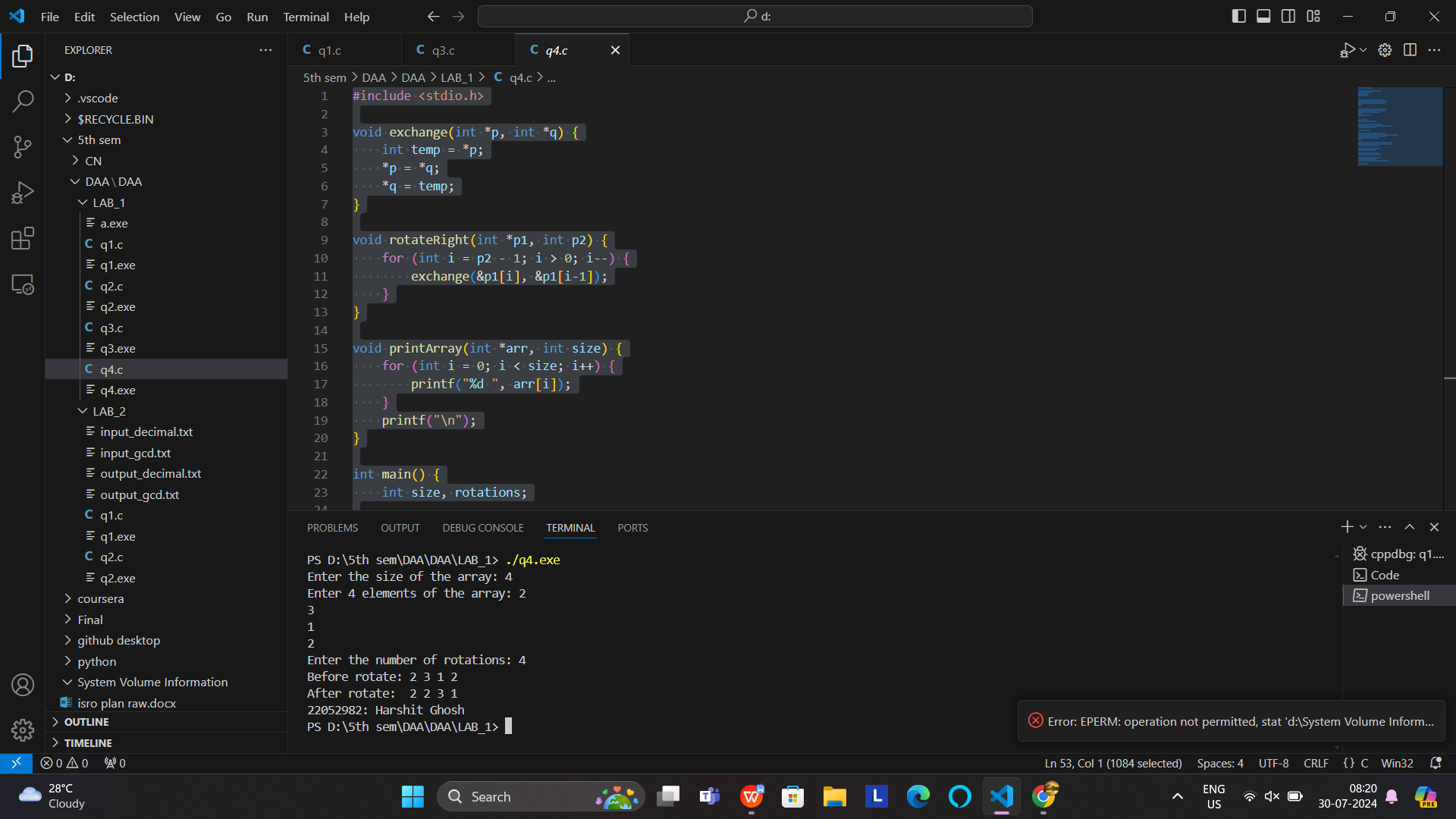
    printArray(A, size);

    printf("22052982: Harshit Ghosh\n");

    return 0;

}

***INPUT/OUTPUT***



# Lab Day 2: Fundamentals of Algorithmic Problem Solving

***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.*

#include<stdio.h>

#define LENGTH 100

int i;

long int decimal\_binary(int dec\_num){

    if(dec\_num==0)

        return 0;

    else

        return(dec\_num % 2 + 10 \* decimal\_binary(dec\_num / 2));

}

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

    printf("\n22052982: Harshit Ghosh\n");

    char \*inputFile = argv[1];

    int arr[LENGTH];

    int n;

    FILE \*fp = fopen(inputFile,"r");

    if(fp==NULL){

        printf("Cannot open file");

        return 0;

    }

    printf("Enter array length: ");

    scanf("%d",&n);

    for(i=0; i<n; i++){

        fscanf(fp,"%d",&arr[i]);

    }

    fclose(fp);

    long int bin\_arr[LENGTH];

    for(i=0; i<n; i++){

        bin\_arr[i] = decimal\_binary(arr[i]);

    }

    char \*outputFile = argv[2];

    FILE \*fp1 = fopen(outputFile,"w");

    if(fp1==NULL){

        printf("Cannot open file to write");

        return 0;

    }

    for(i=0; i<n; i++){

        fprintf(fp1,"The binary equivalent of %d is %ld\n",arr[i],bin\_arr[i]);

    }

    fclose(fp1);

    FILE \*fp2 = fopen(outputFile,"r");

    char c = fgetc(fp2);

    while(c!=EOF){

        printf("%c",c);

        c=fgetc(fp2);

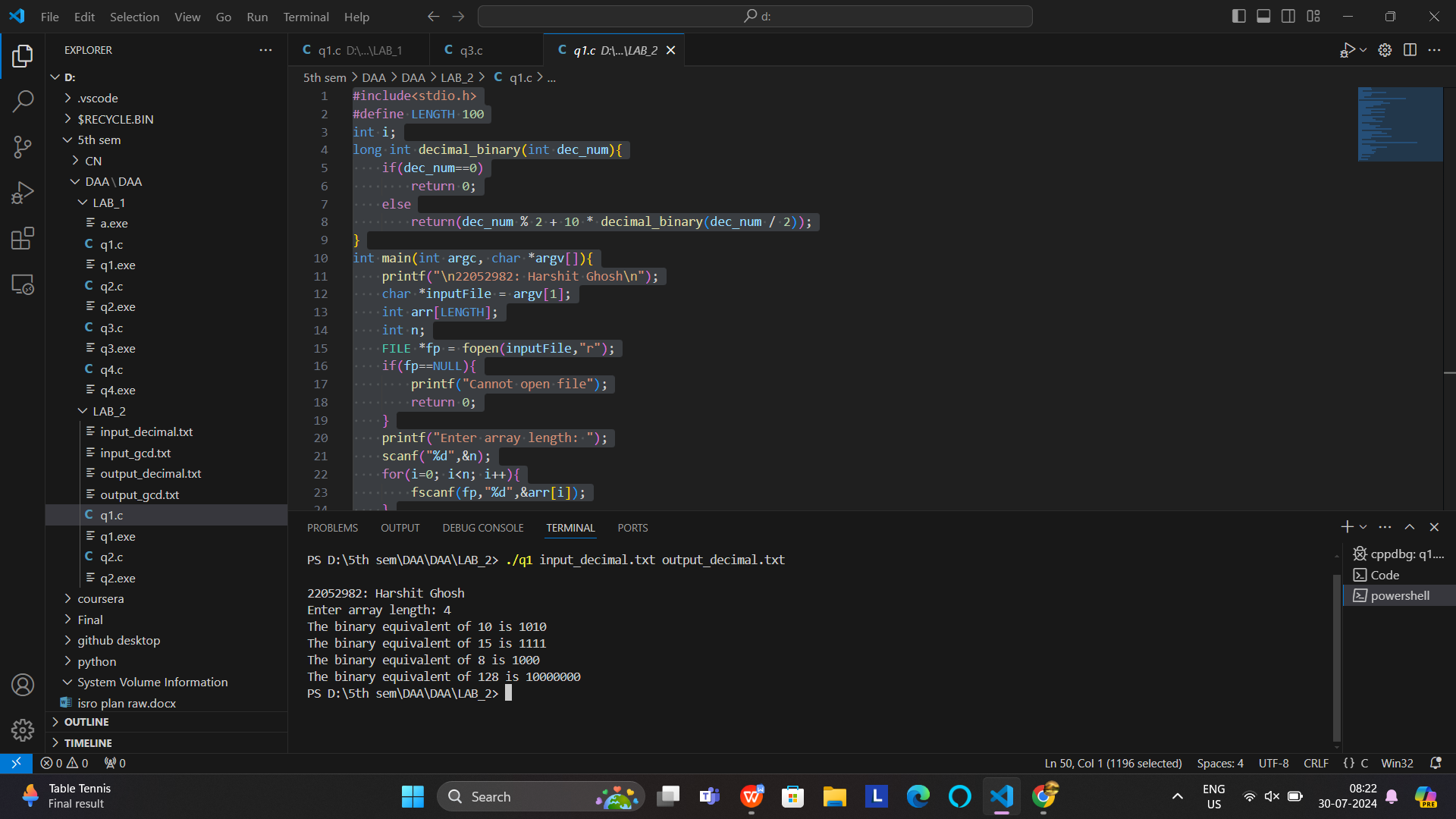
    }

    fclose(fp2);

    return 0;

}

***INPUT/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.*

#include<stdio.h>

#include<stdlib.h>

int gcd(int a, int b){

    if(b == 0)

        return a;

    return gcd(b, a % b);

}

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

    FILE \*infile = fopen(argv[1], "r");

    FILE \*outfile = fopen(argv[2], "w");

    if(infile == NULL || outfile == NULL){

        printf("Cannot Open File");

        return 0;

    }

    int n1, n2;

    while(fscanf(infile, "%d %d", &n1, &n2) == 2){

        int res = gcd(n1, n2);

        fprintf(outfile, "The GCD of %d and %d is %d\n", n1, n2, res);

    }

    fclose(infile);

    fclose(outfile);

    outfile = fopen(argv[2], "r");

    char c = getc(outfile);

    while(c != EOF){

        printf("%c", c);

        c = getc(outfile);

    }

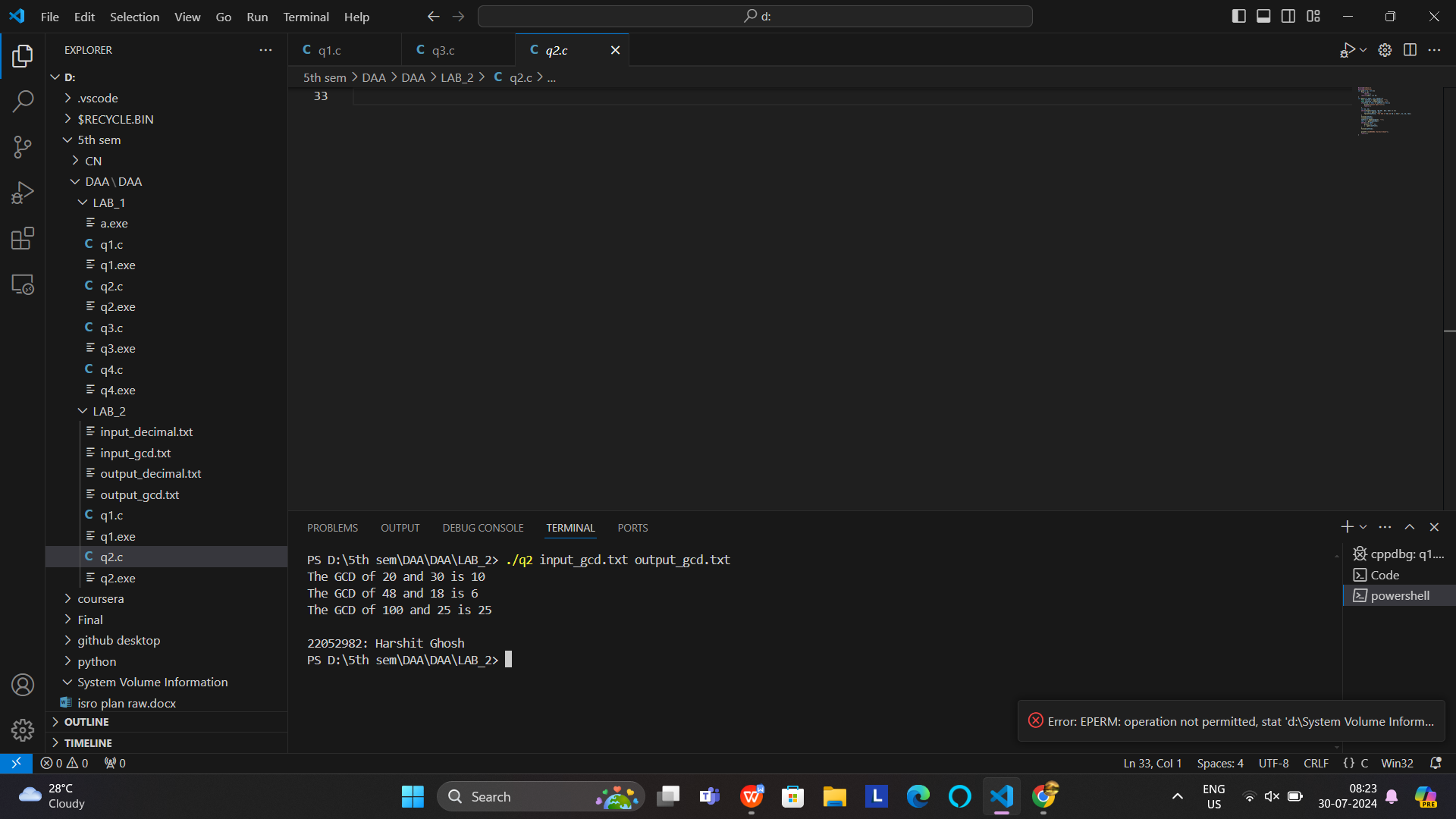
    fclose(outfile);

    printf("\n22052982: Harshit Ghosh");

    return 0;

}

***INPUT/OUTPUT***



# Lab Day 3: Divide and Conquer Method

**3.1 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>

#ifdef \_WIN32

#include <windows.h>

#else

#include <time.h>

#endif

// Function prototypes

void mergesort(int a[], int low, int high, int \*comparison);

void merge(int a[], int low, int mid, int high, int \*comparison);

void process\_file(const char \*input\_filename, const char \*output\_filename);

#ifdef \_WIN32

// Function to get time in nanoseconds on Windows

long long get\_time\_in\_nanoseconds() {

    LARGE\_INTEGER frequency, start, end;

    QueryPerformanceFrequency(&frequency);

    QueryPerformanceCounter(&start);

    // The time measurement will be done in the function where it's used

    return start.QuadPart;

}

long long get\_elapsed\_time(long long start) {

    LARGE\_INTEGER end;

    QueryPerformanceCounter(&end);

    return (end.QuadPart - start) \* 1000000000LL / get\_time\_in\_nanoseconds();

}

#else

// Function to get time in nanoseconds on Linux

long long get\_time\_in\_nanoseconds() {

    struct timespec ts;

    clock\_gettime(CLOCK\_MONOTONIC, &ts);

    return ts.tv\_sec \* 1e9 + ts.tv\_nsec;

}

#endif

int main() {

    int choice;

    while (1) {

        printf("22052982: HARSHIT GHOSH");

        printf("MAIN MENU:\n1. Ascending Data\n2. Descending Data\n3. Random Data\n4. EXIT\nEnter option: ");

        if (scanf("%d", &choice) != 1) {

            fprintf(stderr, "Invalid input. Please enter an integer.\n");

            while (getchar() != '\n'); // Clear invalid input

            continue;

        }

        switch (choice) {

            case 1: process\_file("inAsce.txt", "outMergeAsce.txt"); break;

            case 2: process\_file("inDesc.txt", "outMergeDesc.txt"); break;

            case 3: process\_file("inRand.txt", "outMergeRand.txt"); break;

            case 4: printf("Exiting...\n"); exit(0);

            default: printf("Invalid option. Please try again.\n");

        }

    }

    return 0;

}

void merge(int a[], int low, int mid, int high, int \*comparison) {

    int n1 = mid - low + 1, n2 = high - mid;

    int \*L = (int \*)malloc(n1 \* sizeof(int));

    int \*R = (int \*)malloc(n2 \* sizeof(int));

    if (!L || !R) { fprintf(stderr, "Memory allocation failed\n"); exit(EXIT\_FAILURE); }

    for (int i = 0; i < n1; i++) L[i] = a[low + i];

    for (int i = 0; i < n2; i++) R[i] = a[mid + i + 1];

    int i = 0, j = 0, k = low;

    while (i < n1 && j < n2) {

        a[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];

        (\*comparison)++;

    }

    while (i < n1) a[k++] = L[i++];

    while (j < n2) a[k++] = R[j++];

    free(L);

    free(R);

}

void mergesort(int a[], int low, int high, int \*comparison) {

    if (low < high) {

        int mid = (low + high) / 2;

        mergesort(a, low, mid, comparison);

        mergesort(a, mid + 1, high, comparison);

        merge(a, low, mid, high, comparison);

    }

}

void process\_file(const char \*input\_filename, const char \*output\_filename) {

    FILE \*input = fopen(input\_filename, "r");

    FILE \*output = fopen(output\_filename, "w");

    if (!input || !output) { perror("Error opening file"); return; }

    int n;

    if (fscanf(input, "%d", &n) != 1) { fprintf(stderr, "Error reading array size.\n"); fclose(input); fclose(output); return; }

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

    if (!arr) { fprintf(stderr, "Memory allocation failed\n"); fclose(input); fclose(output); return; }

    for (int i = 0; i < n; i++) {

        if (fscanf(input, "%d", &arr[i]) != 1) { fprintf(stderr, "Error reading array element.\n"); free(arr); fclose(input); fclose(output); return; }

    }

    fprintf(output, "Before sorting:\n");

    for (int i = 0; i < n; i++) fprintf(output, "%d ", arr[i]);

    fprintf(output, "\n");

    int comparison = 0;

    long long start\_time = get\_time\_in\_nanoseconds();

    mergesort(arr, 0, n - 1, &comparison);

    long long end\_time = get\_time\_in\_nanoseconds();

    long long time\_taken = end\_time - start\_time;

    fprintf(output, "After sorting:\n");

    for (int i = 0; i < n; i++) fprintf(output, "%d ", arr[i]);

    fprintf(output, "\nNumber of Comparisons: %d\nExecution Time: %lld nanoseconds\n", comparison, time\_taken);

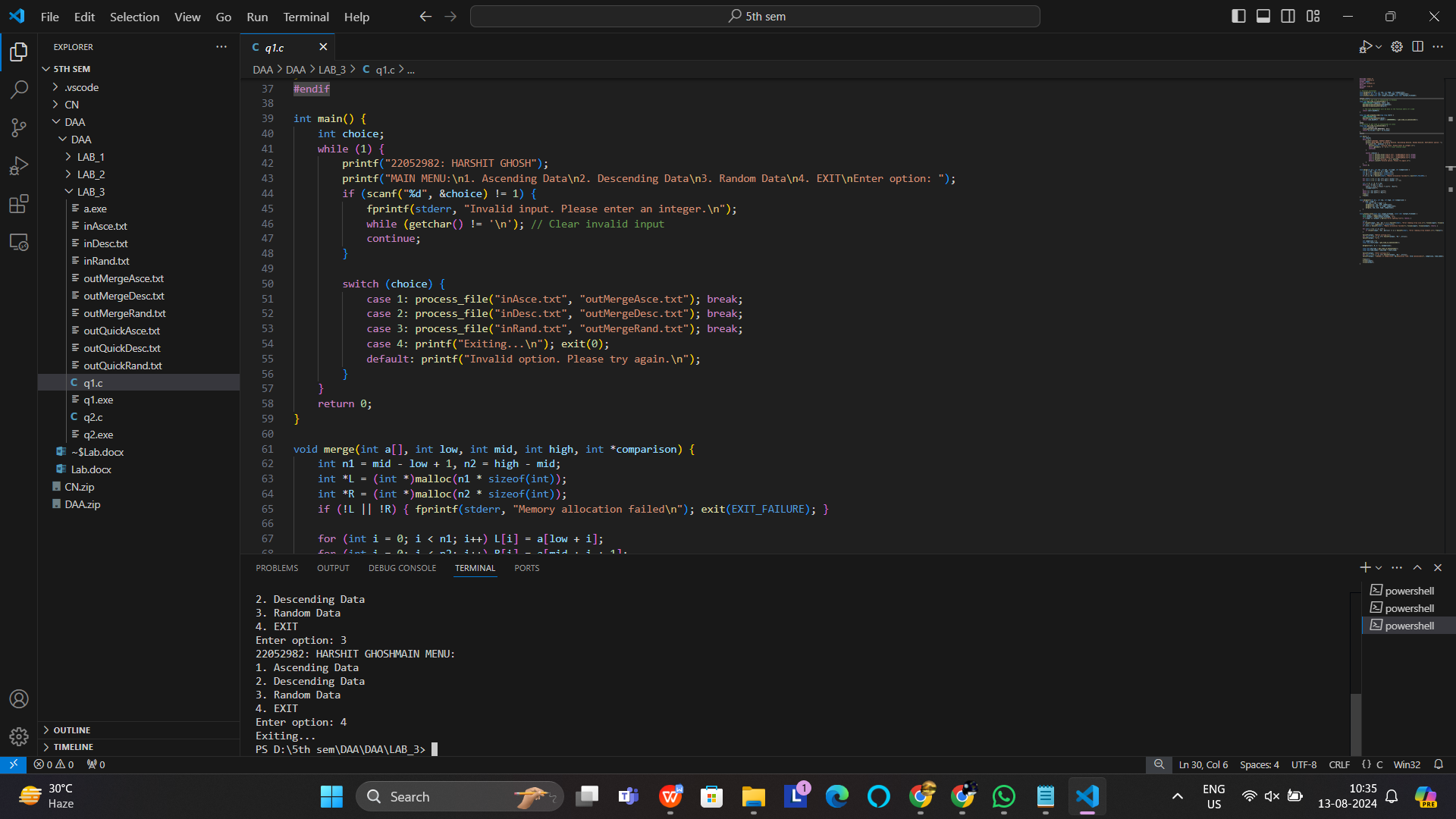
    free(arr);

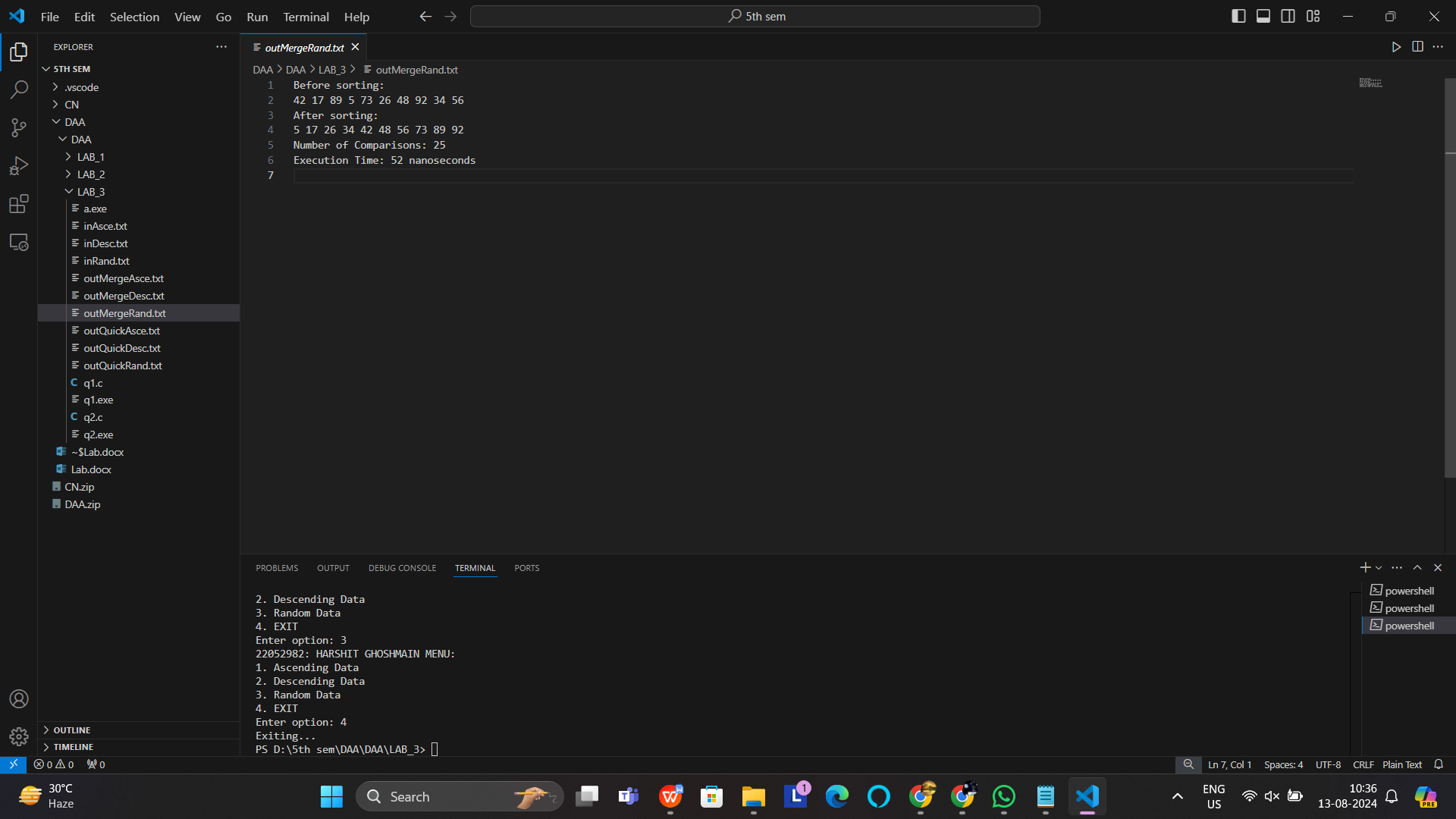
    fclose(input);

    fclose(output);

}

***INPUT/OUTPUT***





**3.2 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>

#include <stdlib.h>

#include <time.h>

#ifdef \_WIN32

#include <windows.h>

#else

#include <sys/time.h>

#endif

void swap(int \*x, int \*y);

int partition(int arr[], int low, int high, int \*comparison);

void quicksort(int arr[], int low, int high, int \*comparison);

void driver(FILE \*input, int arr[], FILE \*output);

void print\_array(FILE \*output, int arr[], int n);

#ifdef \_WIN32

double get\_time\_in\_seconds() {

    LARGE\_INTEGER frequency;

    LARGE\_INTEGER start;

    QueryPerformanceFrequency(&frequency);

    QueryPerformanceCounter(&start);

    return (double)start.QuadPart / frequency.QuadPart;

}

#else

double get\_time\_in\_seconds() {

    struct timeval time;

    gettimeofday(&time, NULL);

    return (double)time.tv\_sec + (double)time.tv\_usec \* .000001;

}

#endif

int main() {

    FILE \*inputfile, \*outputfile;

    int choice = 0;

    int arr[500];

    do {

        printf("22052982: HARSHIT GHOSH");

        printf("MAIN MENU:\n");

        printf("1. Ascending Data\n2. Descending Data\n3. Random Data\n4. EXIT\n");

        printf("Enter option: ");

        scanf("%d", &choice);

        switch (choice) {

            case 1:

                inputfile = fopen("inAsce.txt", "r");

                outputfile = fopen("outQuickAsce.txt", "w");

                if (inputfile && outputfile) {

                    driver(inputfile, arr, outputfile);

                    fclose(inputfile);

                    fclose(outputfile);

                } else {

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

                }

                break;

            case 2:

                inputfile = fopen("inDesc.txt", "r");

                outputfile = fopen("outQuickDesc.txt", "w");

                if (inputfile && outputfile) {

                    driver(inputfile, arr, outputfile);

                    fclose(inputfile);

                    fclose(outputfile);

                } else {

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

                }

                break;

            case 3:

                inputfile = fopen("inRand.txt", "r");

                outputfile = fopen("outQuickRand.txt", "w");

                if (inputfile && outputfile) {

                    driver(inputfile, arr, outputfile);

                    fclose(inputfile);

                    fclose(outputfile);

                } else {

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

                }

                break;

            case 4:

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

                exit(0);

                break;

            default:

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

                break;

        }

    } while (choice != 4);

    return 0;

}

void swap(int \*x, int \*y){

    int temp = \*x;

    \*x = \*y;

    \*y = temp;

}

int partition(int arr[], int low, int high, int \*comparison){

    int i = low, j = high;

    int pivot = arr[low]; // starting element as pivot element

    while (i < j) {

        while (arr[i] <= pivot && i <= high - 1) {

            i++;

            (\*comparison)++;

        }

        while (arr[j] > pivot && j >= low + 1) {

            j--;

            (\*comparison)++;

        }

        if (i < j) {

            swap(&arr[i], &arr[j]);

            (\*comparison)++;

        }

    }

    swap(&arr[low], &arr[j]);

    (\*comparison)++;

    return j;

}

void quicksort(int arr[], int low, int high, int \*comparison){

    if (low < high) {

        int partitionIndex = partition(arr, low, high, comparison);

        quicksort(arr, low, partitionIndex - 1, comparison);

        quicksort(arr, partitionIndex + 1, high, comparison);

    }

}

void print\_array(FILE \*output, int arr[], int n) {

    for (int i = 0; i < n; i++) {

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

    }

    fprintf(output, "\n");

}

void driver(FILE \*input, int arr[], FILE \*output) {

    int n;

    int comparison = 0; // Initialize comparison counter locally

    // Read size of array

    fscanf(input, "%d", &n);

    // Read array

    for (int i = 0; i < n; i++) {

        fscanf(input, "%d", &arr[i]);

    }

    // Printing array before sorting

    fprintf(output, "Before sorting:\n");

    print\_array(output, arr, n);

    // Sorting and calculating execution time

    double start = get\_time\_in\_seconds();

    quicksort(arr, 0, n - 1, &comparison);

    double end = get\_time\_in\_seconds();

    long time\_taken = (long)((end - start) \* 1e9);

    // Printing array after sorting

    fprintf(output, "After sorting:\n");

    print\_array(output, arr, n);

    fprintf(output, "Number of Comparisons: %d\n", comparison);

    fprintf(output, "Execution Time: %ld nanoseconds\n", time\_taken);

}

***INPUT/OUTPUT***

