#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/time.h>  // For gettimeofday()

#define MAX\_SIZE 2000

#define ALGO\_COUNT 7 // Updated to include Quick Sort

// Struct to store sorting algorithm name and time

typedef struct {

    char name[20];

    double time;

} AlgorithmTime;

// Function declarations for sorting algorithms

void bubbleSort(int arr[], int n);

void selectionSort(int arr[], int n);

void insertionSort(int arr[], int n);

void mergeSort(int arr[], int left, int right);

void heapSort(int arr[], int n);

void radixSort(int arr[], int n);

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

// Utility functions

void merge(int arr[], int left, int mid, int right);

void heapify(int arr[], int n, int i);

int getMax(int arr[], int n);

void countSort(int arr[], int n, int exp);

int partition(int arr[], int low, int high);

void copyArray(int source[], int dest[], int n);

// Timer utility using gettimeofday

double calculateTime(void (\*sortFunc)(int[], int), int arr[], int n);

double calculateTimeMergeQuick(void (\*sortFunc)(int[], int, int), int arr[], int left, int right);

// Function to compare algorithm times

int compare(const void \*a, const void \*b) {

    double timeA = ((AlgorithmTime \*)a)->time;

    double timeB = ((AlgorithmTime \*)b)->time;

    if (timeA < timeB) return -1;

    if (timeA > timeB) return 1;

    return 0;

}

int main() {

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

    if (file == NULL) {

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

        return 1;

    }

    char line[MAX\_SIZE];

    int arr[MAX\_SIZE], tempArr[MAX\_SIZE];

    int n, arrayCount = 0;

    // Variables to store total time for each algorithm

    AlgorithmTime algoTimes[ALGO\_COUNT] = {

        {"Bubble Sort", 0},

        {"Selection Sort", 0},

        {"Insertion Sort", 0},

        {"Merge Sort", 0},

        {"Heap Sort", 0},

        {"Radix Sort", 0},

        {"Quick Sort", 0}

    };

    // Read each line (each array)

    while (fgets(line, sizeof(line), file)) {

        n = 0; // Reset array size

        char \*token = strtok(line, " ");

        while (token != NULL) {

            arr[n++] = atoi(token); // Convert string to int

            token = strtok(NULL, " ");

        }

        // Increment the number of arrays

        arrayCount++;

        // Bubble Sort

        copyArray(arr, tempArr, n);

        algoTimes[0].time += calculateTime(bubbleSort, tempArr, n);

        // Selection Sort

        copyArray(arr, tempArr, n);

        algoTimes[1].time += calculateTime(selectionSort, tempArr, n);

        // Insertion Sort

        copyArray(arr, tempArr, n);

        algoTimes[2].time += calculateTime(insertionSort, tempArr, n);

        // Merge Sort

        copyArray(arr, tempArr, n);

        algoTimes[3].time += calculateTimeMergeQuick(mergeSort, tempArr, 0, n - 1);

        // Heap Sort

        copyArray(arr, tempArr, n);

        algoTimes[4].time += calculateTime(heapSort, tempArr, n);

        // Radix Sort

        copyArray(arr, tempArr, n);

        algoTimes[5].time += calculateTime(radixSort, tempArr, n);

        // Quick Sort

        copyArray(arr, tempArr, n);

        algoTimes[6].time += calculateTimeMergeQuick(quickSort, tempArr, 0, n - 1);

        if (arrayCount >= MAX\_SIZE) {

            break;

        }

    }

    // Calculate average times

    for (int i = 0; i < ALGO\_COUNT; i++) {

        algoTimes[i].time /= arrayCount;

    }

    // Sort the algorithms by time to determine best, worst, and median

    qsort(algoTimes, ALGO\_COUNT, sizeof(AlgorithmTime), compare);

    // Output the times in a clean format

    printf("\n--- Sorting Algorithm Time Comparison ---\n\n");

    printf("%-20s %-10s\n", "Algorithm", "Avg Time (s)");

    printf("----------------------------------------\n");

    for (int i = 0; i < ALGO\_COUNT; i++) {

        printf("%-20s %.6f\n", algoTimes[i].name, algoTimes[i].time);

    }

    // Display best, worst, and median algorithms

    printf("\n--- Summary ---\n");

    printf("Best Sorting Algorithm: %s (%.6f seconds)\n", algoTimes[0].name, algoTimes[0].time);

    printf("Worst Sorting Algorithm: %s (%.6f seconds)\n", algoTimes[ALGO\_COUNT-1].name, algoTimes[ALGO\_COUNT-1].time);

    printf("Median Sorting Algorithm: %s (%.6f seconds)\n", algoTimes[ALGO\_COUNT/2].name, algoTimes[ALGO\_COUNT/2].time);

    fclose(file);

    return 0;

}

// Function to calculate the time taken by a sorting algorithm using gettimeofday()

double calculateTime(void (\*sortFunc)(int[], int), int arr[], int n) {

    struct timeval start, end;

    gettimeofday(&start, NULL);

    sortFunc(arr, n);

    gettimeofday(&end, NULL);

    return (double)(end.tv\_sec - start.tv\_sec) + (double)(end.tv\_usec - start.tv\_usec) / 1000000.0;

}

// Special function to calculate the time taken by mergeSort and quickSort

double calculateTimeMergeQuick(void (\*sortFunc)(int[], int, int), int arr[], int left, int right) {

    struct timeval start, end;

    gettimeofday(&start, NULL);

    sortFunc(arr, left, right);

    gettimeofday(&end, NULL);

    return (double)(end.tv\_sec - start.tv\_sec) + (double)(end.tv\_usec - start.tv\_usec) / 1000000.0;

}

// Bubble Sort

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

    for (int i = 0; i < n-1; i++) {

        for (int j = 0; j < n-i-1; j++) {

            if (arr[j] > arr[j+1]) {

                int temp = arr[j];

                arr[j] = arr[j+1];

                arr[j+1] = temp;

            }

        }

    }

}

// Selection Sort

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

    for (int i = 0; i < n-1; i++) {

        int minIdx = i;

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

            if (arr[j] < arr[minIdx]) {

                minIdx = j;

            }

        }

        int temp = arr[minIdx];

        arr[minIdx] = arr[i];

        arr[i] = temp;

    }

}

// Insertion Sort

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

    for (int i = 1; i < n; i++) {

        int key = arr[i];

        int j = i - 1;

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

            arr[j + 1] = arr[j];

            j = j - 1;

        }

        arr[j + 1] = key;

    }

}

// Merge Sort

void mergeSort(int arr[], int left, int right) {

    if (left < right) {

        int mid = left + (right - left) / 2;

        mergeSort(arr, left, mid);

        mergeSort(arr, mid + 1, right);

        merge(arr, left, mid, right);

    }

}

void merge(int arr[], int left, int mid, int right) {

    int n1 = mid - left + 1;

    int n2 = right - mid;

    int L[n1], R[n2];

    for (int i = 0; i < n1; i++)

        L[i] = arr[left + i];

    for (int j = 0; j < n2; j++)

        R[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {

        if (L[i] <= R[j]) {

            arr[k] = L[i];

            i++;

        } else {

            arr[k] = R[j];

            j++;

        }

        k++;

    }

    while (i < n1) {

        arr[k] = L[i];

        i++;

        k++;

    }

    while (j < n2) {

        arr[k] = R[j];

        j++;

        k++;

    }

}

// Heap Sort

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

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

        heapify(arr, n, i);

    for (int i = n - 1; i > 0; i--) {

        int temp = arr[0];

        arr[0] = arr[i];

        arr[i] = temp;

        heapify(arr, i, 0);

    }

}

void heapify(int arr[], int n, int i) {

    int largest = i;

    int left = 2 \* i + 1;

    int right = 2 \* i + 2;

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

        largest = left;

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

        largest = right;

    if (largest != i) {

        int temp = arr[i];

        arr[i] = arr[largest];

        arr[largest] = temp;

        heapify(arr, n, largest);

    }

}

// Radix Sort

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

    int max = getMax(arr, n);

    for (int exp = 1; max / exp > 0; exp \*= 10)

        countSort(arr, n, exp);

}

int getMax(int arr[], int n) {

    int max = arr[0];

    for (int i = 1; i < n; i++)

        if (arr[i] > max)

            max = arr[i];

    return max;

}

void countSort(int arr[], int n, int exp) {

    int output[n];

    int count[10] = {0};

    for (int i = 0; i < n; i++)

        count[(arr[i] / exp) % 10]++;

    for (int i = 1; i < 10; i++)

        count[i] += count[i - 1];

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

        output[count[(arr[i] / exp) % 10] - 1] = arr[i];

        count[(arr[i] / exp) % 10]--;

    }

    for (int i = 0; i < n; i++)

        arr[i] = output[i];

}

// Quick Sort

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

    if (low < high) {

        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);

        quickSort(arr, pi + 1, high);

    }

}

int partition(int arr[], int low, int high) {

    int pivot = arr[high];

    int i = (low - 1);

    for (int j = low; j < high; j++) {

        if (arr[j] < pivot) {

            i++;

            int temp = arr[i];

            arr[i] = arr[j];

            arr[j] = temp;

        }

    }

    int temp = arr[i + 1];

    arr[i + 1] = arr[high];

    arr[high] = temp;

    return i + 1;

}

// Utility to copy arrays

void copyArray(int source[], int dest[], int n) {

    for (int i = 0; i < n; i++) {

        dest[i] = source[i];

    }

}