Names:
Ng Wen Wen , Ng Yue Zhi

Requirements/Specification:
The program is supposed to showcase merge sort, quick sort and the comparisons. It will demonstrate the use of merge sort and quicksort algorithms on a number array. It will also demonstrate the use of merge sort and quicksort algorithms on multiple-key data sets to showcase the stability of the algorithms.
The input will be the user's choices of the options given and the output will showcase the before and after sorting of the algorithms.
Assumptions: The program already has data for sorting and does not require any data to be i

User Guide:
Compile: Enter “make” in the console to compile the program.
Run: Enter “./sort_program” to run the program
How to use: User has to select the given options to view the sorting and comparisons.
Option 1 showcases the merge sort algorithm. Option 2 showcases the quicksort algorithm. Option 3 will proceed to another 2 options to choose between 2 test cases, student dataset or music dataset (bigger dataset with multiple keys).

Design and analysis:
For merge sort, I approached it using the divide and conquer algorithm.
For quicksort, I approached it using the partitioning algorithm.
To showcase the stability of both algorithm, merge sort being stable and quicksort being unstable, I designed the program to include test cases that will showcase the stability, and compare both algorithms.
The time complexity:

Analysis of the efficiency of Merge Sort algorithm using the Big-oh notation

The time complexity of a Merge Sort is O(n lg n) in the worst, average and the best case. For best case it is when the array has already been sorted or close to being sorted. For the average case, it is when the array is randomly ordered. For the worst case, it is when the array is sorted in reverse order.
In the algorithm, it consists of functions merge and mergeSort. mergeSort function recursively divides the array into subarrays then merges them back together whereas the merge function merges the 2 subarrays into a single sorted array. For merge function the time complexity is O n and it is called O(log n) times recursively. mergeSort function has a time complexity of O(n log n).
With the same time complexity throughout the three cases. It shows that it operates consistently and reliably in a variety of situations. Additionally with both functions having a time complexity of O(n log n), it showcases that it scales competently with large datasets due to its logarithmic component.

Analysis of the efficiency of Quick Sort algorithm using the Big-oh notation

The time complexity of Quick Sort for any partitioning scenario where T(n) is the time complexity for sorting n elements is T(n) = T(k) + T(n-k-1) + θ(n), where T(k) is the time complexity to sort the left sub-array which is of size k and T(n-k-1) is the time complexity to sort the right sub-array which is of size n-k-1.
The best, average and worst case scenarios are dependent on the value of size k.
The best case scenario is when the array is divided into 2 equal halves by the pivot (partitioning is nearly equally balanced), so that k approximates to n/2, the average case scenario is when the array is divided into approximately equal halves (partitioning may be slightly unbalanced) so that k also approximates to n/2. In both cases, T(n) = 2T(n/2) + θ(n), or O(n lg n). In the worst case scenario, k = 0 or k = n-1 where the pivot is either the smallest or largest element. In this case, the time complexity is T(n)=T(0)+T(n−1)+θ(n) or O(n^2).
For best and average case, it is considered efficient when the pivot is well selected, it also provides fast performances. However, unlike merge sort, in the worst case, quicksort is inefficient and performs much slower.

Limitations:
The datasets are hard coded, testing with other datasets will not be available. The datasets hardcoded are to showcase the stability of the sorting algorithms.

Testing:
Tested merge sort algorithm and quicksort algorithm with integer array.

```
1: Merge Sort
2: Quick Sort
3: Compare Merge Sort(stable) and Quick Sort(unstable)
Enter selection number:1
=========================================================
Unsorted:
8 2 10 4 9 18

Sorted:
2 4 8 9 10 18
```

```
1: Merge Sort
2: Quick Sort
3: Compare Merge Sort(stable) and Quick Sort(unstable)
Enter selection number:2
=========================================================
Unsorted:
4 2 8 3 1 5 7 11 6

Sorted:
1 2 3 4 5 6 7 8 11
```

Tested merge sort algorithm and quicksort algorithm with student dataset with 2 keys.

```
1: Merge Sort
2: Quick Sort
3: Compare Merge Sort(stable) and Quick Sort(unstable)
Enter selection number:3
=============================================================================================================
1: Student database (names and scores)
2: Music (name, release year, price of album)
Enter selection number:1
=============================================================================================================
Explanation:
In the student database, the students' name and scores are stored. The database is already sorted by name.
Now we are sorting based on scores.
In a stable sort, the original order (sorted by name), should maintain after sorting by scores.
Whereas in an unstable sort, the students with the same score may switch places in the array.
=============================================================================================================
Original data:
(Ben, 40) (Eric, 80) (Oscar, 80) (Rachel, 100)

MergeSort (stable):
(Ben, 40) (Eric, 80) (Oscar, 80) (Rachel, 100)

QuickSort (unstable):
(Ben, 40) (Oscar, 80) (Eric, 80) (Rachel, 100)
```

Tested merge sort algorithm and quicksort algorithm with music dataset with multiple keys.

```
1: Merge Sort
2: Quick Sort
3: Compare Merge Sort(stable) and Quick Sort(unstable)
Enter selection number:3
=============================================================================================
1: Student database (names and scores)
2: Music (name, release year, price of album)
Enter selection number:2
=============================================================================================
Explanation:
In the music database, the music's name, year of release and album price are stored.
Now we are by release year, by price than by name.
In a stable sort, each time as its sorted, it will maintain the relative order of the previously sorted order.
Whereas in an unstable sort, elements with the same value may swap positions in the array.
=============================================================================================
Original data:
(Music F, 2023, 25.5) (Music G, 2023, 25.5) (Music A, 2023, 25.5) (Music C, 2023, 18.5) (Music E, 2024, 18) (Music D, 2020, 30) (Music B, 2020, 30)

MergeSort (stable):
(Music B, 2020, 30) (Music D, 2020, 30) (Music C, 2023, 18.5) (Music A, 2023, 25.5) (Music F, 2023, 25.5) (Music G, 2023, 25.5) (Music E, 2024, 18)

QuickSort (unstable):
(Music B, 2020, 30) (Music D, 2020, 30) (Music C, 2023, 18.5) (Music A, 2023, 25.5) (Music G, 2023, 25.5) (Music F, 2023, 25.5) (Music E, 2024, 18)
```

No runtime error.

Listings:

SortingAlgorithms.cpp

```cpp
#include <iostream>
#include <string>
#include <algorithm>

//////////////////////////////////////////////////////////////////////////
///////

/* For regular merge sort */

void merge(int arr[], int left, int middle, int right)
{
    int i, j, k;
    int n1 = middle - left + 1;
    int n2 = right - middle;
```

```
int L[n1], R[n2];

for (i = 0; i < n1; i++)
{
    L[i] = arr[left + i];
}
for (j = 0; j < n2; j++)
{
    R[j] = arr[middle + 1 + j];
}

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++;
```

```
    }
}

void mergeSort(int arr[], int left, int right)
{
    if (left < right)
    {
        int middle = left + (right - left) / 2;

        mergeSort(arr, left, middle);
        mergeSort(arr, middle + 1, right);
        merge(arr, left, middle, right);
    }
}

/* For regular quick sort */

int partition(int arr[], int low, int high)
{
    int i = low;
    int j = high;
    int pivot = arr[low];
    while (i < j)
    {
        while (pivot >= arr[i]) i++;
        while (pivot < arr[j]) j--;
        if (i < j)
        {
            std::swap(arr[i], arr[j]);
        }
    }
    std::swap(arr[low], arr[j]);
    return j;
}

void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        int pivot = partition(arr, low, high);
        quickSort(arr, low, pivot - 1);
        quickSort(arr, pivot + 1, high);
    }
```

```
}

void printArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
    {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
}


////////////////////////////////////////////////////////////////////////
///////


struct Student
{
    std::string name;
    int score;
};


struct Music
{
    std::string name;
    int release;
    double price;
};

void printStudentsArray(Student arr[], int size)
{
    for (int i = 0; i < size; i++)
    {
        std::cout << "(" << arr[i].name << ", " << arr[i].score << ")
";
    }

    std::cout << std::endl;
}

void printMusicArray(Music arr[], int size)
{
```

```
    for (int i = 0; i < size; i++)
    {
        std::cout << "(" << arr[i].name << ", " << arr[i].release << ",
" << arr[i].price << ") ";
    }

    std::cout << std::endl;
}


void mergeStudentArray(Student arr[], int left, int middle, int right)
{
    int n1 = middle - left + 1;
    int n2 = right - middle;

    Student 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[middle + 1 + j];
    }

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2)
    {
        if (L[i].score <= R[j].score)
        {
            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++;
    }
}

void mergeMusicArray(Music arr[], int left, int middle, int right) {
    int n1 = middle - left + 1;
    int n2 = right - middle;

    Music 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[middle + 1 + j];
    }

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2)
    {
        if (L[i].release < R[j].release ||
            (L[i].release == R[j].release && L[i].price < R[j].price)
||
            (L[i].release == R[j].release && L[i].price == R[j].price
&& L[i].name <= R[j].name))
        {
            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++;
    }
}


void mergeSortStudentArray(Student arr[], int left, int right)
{
    if (left < right)
    {
        int middle = left + (right - left) / 2;
        mergeSortStudentArray(arr, left, middle);
        mergeSortStudentArray(arr, middle + 1, right);
        mergeStudentArray(arr, left, middle, right);
    }
}


void mergeSortMusicArray(Music arr[], int left, int right)
{
    if (left < right)
    {
        int middle = left + (right - left) / 2;
```

```
        mergeSortMusicArray(arr, left, middle);
        mergeSortMusicArray(arr, middle + 1, right);
        mergeMusicArray(arr, left, middle, right);
    }
}


int partitionStudentArray(Student arr[], int low, int high)
{
    int i = low;
    int j = high;
    int pivot = arr[low].score;

    while (i < j)
    {
        while (arr[i].score <= pivot && i < high)
        {
            i++;
        }
        while (arr[j].score > pivot)
        {
            j--;
        }
        if (i < j)
        {
            std::swap(arr[i], arr[j]);
        }
    }

    std::swap(arr[low], arr[j]);
    return j;
}


int partitionMusicArray(Music arr[], int low, int high)
{
    Music pivot = arr[low];
    int i = low, j = high;

    while (i < j)
    {
        while (arr[i].release < pivot.release ||
```

```
                (arr[i].release == pivot.release && arr[i].price <=
pivot.price) ||
                (arr[i].release == pivot.release && arr[i].price ==
pivot.price && arr[i].name <= pivot.name))
        {
            i++;
        }
        while (arr[j].release > pivot.release ||
                (arr[j].release == pivot.release && arr[j].price >
pivot.price) ||
                (arr[j].release == pivot.release && arr[j].price ==
pivot.price && arr[j].name > pivot.name))
        {
            j--;
        }
        if (i < j)
        {

            std::swap(arr[i], arr[j]);
        }
    }
    std::swap(arr[low], arr[j]);
    return j;
}


void quickSortStudentArray(Student arr[], int low, int high)
{
    if (low < high)
    {
        int pivot = partitionStudentArray(arr, low, high);
        quickSortStudentArray(arr, low, pivot - 1);
        quickSortStudentArray(arr, pivot + 1, high);
    }
}

void quickSortMusicArray(Music arr[], int low, int high) {
    if (low < high)
    {
        int pivot = partitionMusicArray(arr, low, high);
        quickSortMusicArray(arr, low, pivot - 1);
        quickSortMusicArray(arr, pivot + 1, high);
    }
```

```
}

int main() {
    Student students[] = {{"Ben", 40}, {"Eric", 80}, {"Oscar", 80},
{"Rachel", 100}};
    int totalStudents = sizeof(students) / sizeof(students[0]);

    Music musics[] = {
                {"Music F", 2023, 25.5}, {"Music G", 2023, 25.5},
{"Music A", 2023, 25.5},
                {"Music C", 2023, 18.5}, {"Music E", 2024, 18.0},
{"Music D", 2020, 30.0},
                {"Music B", 2020, 30.0}
                    };

    int choice;
    int secondChoice;
    std::cout << "1: Merge Sort" << std::endl;
    std::cout << "2: Quick Sort" << std::endl;
    std::cout << "3: Compare Merge Sort(stable) and Quick
Sort(unstable)" << std::endl;
    std::cout << "Enter selection number:" ;
    std::cin >> choice;

    if (choice == 1) {
        int arr[] = {8, 2, 10, 4, 9, 18};
        int n = sizeof(arr) / sizeof(arr[0]);
        std::cout <<
"=======================================================================
========================================" << std::endl;
        std::cout << "Unsorted:" << std::endl;
        printArray(arr, n);

        mergeSort(arr, 0, n - 1);

        std::cout << std::endl;
        std::cout << "Sorted:" << std::endl;
        printArray(arr, n);
    }

    else if (choice == 2)
    {
        int arr[] = {4, 2, 8, 3, 1, 5, 7, 11, 6};
```

```
        int size = sizeof(arr) / sizeof(int);
        std::cout <<
"=======================================================================
========================================" << std::endl;
        std::cout << "Unsorted:" << std::endl;
        printArray(arr, size);

        quickSort(arr, 0, size - 1);

        std::cout << std::endl;
        std::cout << "Sorted:" << std::endl;
        printArray(arr, size);
    }
    else if (choice == 3) {

        std::cout <<
"=======================================================================
========================================" << std::endl;
        std::cout << "1: Student database (names and scores)" <<
std::endl;
        std::cout << "2: Music (name, release year, price of album)" <<
std::endl;
        std::cout << "Enter selection number:";
        std::cin >> secondChoice;
        if (secondChoice == 1)
        {
            std::cout <<
"=======================================================================
========================================" << std::endl;
            std::cout << "Explanation: "<< std::endl;
            std::cout << "In the student database, the students' name
and scores are stored. The database is already sorted by name." <<
std::endl;
            std::cout << "Now we are sorting based on scores." <<
std::endl;
            std::cout << "In a stable sort, the original order (sorted
by name), should maintain after sorting by scores."<< std::endl;
            std::cout << "Whereas in an unstable sort, the students
with the same score may switch places in the array." << std::endl;
            std::cout <<
"=======================================================================
========================================" << std::endl;
            std::cout << "Original data:" << std::endl;
```

```
            printStudentsArray(students, totalStudents);

            Student mergeArray[totalStudents],
quickArray[totalStudents];

            std::copy(students, students + totalStudents, mergeArray);
            std::copy(students, students + totalStudents, quickArray);

            mergeSortStudentArray(mergeArray, 0, totalStudents - 1);
            std::cout << std::endl;
            std::cout << "MergeSort (stable):" << std::endl;
            printStudentsArray(mergeArray, totalStudents);

            quickSortStudentArray(quickArray, 0, totalStudents - 1);
            std::cout << std::endl;
            std::cout << "QuickSort (unstable):" << std::endl;;
            printStudentsArray(quickArray, totalStudents);
        }
        else if (secondChoice == 2) {
            std::cout <<
"=============================================================================
=========================================" << std::endl;
            std::cout << "Explanation: "<< std::endl;
            std::cout << "In the music database, the music's name, year
of release and album price are stored." << std::endl;
            std::cout << "Now we are by release year, by price than by
name." << std::endl;
            std::cout << "In a stable sort, each time as its sorted, it
will maintain the relative order of the previously sorted order."<<
std::endl;
            std::cout << "Whereas in an unstable sort, elements with
the same value may swap positions in the array." << std::endl;
            std::cout <<
"=============================================================================
=========================================" << std::endl;



            int totalMusic = sizeof(musics) / sizeof(musics[0]);

            std::cout << "Original data:" << std::endl;;
            printMusicArray(musics, totalMusic);
```

```
            Music mergeArray[totalMusic], quickArray[totalMusic];
            std::copy(musics, musics + totalMusic, mergeArray);
            std::copy(musics, musics + totalMusic, quickArray);

            mergeSortMusicArray(mergeArray, 0, totalMusic - 1);
            std::cout << std::endl;
            std::cout << "MergeSort (stable):" << std::endl;;
            printMusicArray(mergeArray, totalMusic);

            quickSortMusicArray(quickArray, 0, totalMusic - 1);
            std::cout << std::endl;
            std::cout << "QuickSort (unstable):" << std::endl;;
            printMusicArray(quickArray, totalMusic);
        }
    }
    else
    {
        std::cout << "Invalid selection.\n";
    }

    return 0;
}
```