roteiro-11/include/time.h

```c
#ifndef TEMPO_H
#define TEMPO_H

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

typedef long double Time;

Time getCpuTime();
Time formatTime(long int sec, long int usec);
void printElapsedTime(Time start, Time end);

#endif
```

roteiro-11/include/sort-module.h

```c
#ifndef SORT_MODULE_H
#define SORT_MODULE_H

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

#include "./time.h"

typedef int Key;

typedef struct {
    long long comps, swaps;
    Time start, end;
} Statistics;

Statistics* createStatistics();
void destroyStatistics(Statistics** s);
void getElements(Key* arr, int quantity);
int compare(Key a, Key b, int reverse, Statistics* statistics);
void swap(Key* a, Key* b, Statistics* statistics);
void print(Key* arr, int n);

#endif
```

roteiro-11/include/sort.h

```c
#ifndef SORT_H
#define SORT_H

#include "./sort-module.h"

// Sorting essential functions
void shellSort(Key* arr, int n, int reverse, Statistics* statistics);
void quickSort(Key* arr, int n, int reverse, Statistics* statistics);
void mergeSort(Key* arr, int n, int reverse, Statistics* statistics);
void heapSort(Key* arr, int n, int reverse, Statistics* statistics);
void getOptions(void (**sortFunction)(Key*, int, int, Statistics*), int* order, int* quantity);

#endif
```

roteiro-11/src/time.c

```c
#include "../include/time.h"

#include <sys/resource.h>

// Returns CPU time in that moment
Time getCpuTime() {
    struct rusage usage;
    getrusage(RUSAGE_SELF, &usage);
    return formatTime(usage.ru_utime.tv_sec, usage.ru_utime.tv_usec);
}

// Join time in seconds and microseconds into a long double variable
Time formatTime(long int sec, long int usec) {
    Time totalTime = sec + ((Time)usec / 1000000.0L);
    return totalTime;
}

// Prints the difference between start to end
void printElapsedTime(Time start, Time end) {
    Time elapsedTime = end - start;
    printf("%Lf", elapsedTime);
}

```

roteiro-11/src/sort-module.c

```c
#include "./sort-module.h"

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Allocate a new statistics pointer
Statistics* createStatistics() {
    Statistics* new = (Statistics*)malloc(sizeof(Statistics));
    new->comps = 0;
    new->swaps = 0;
    new->start = 0;
    new->end = 0;
    return new;
}

// Destroy statistics allocated pointer
void destroyStatistics(Statistics** s) {
    if (*s == NULL) return;
    free(*s);
    *s = NULL;
}

// Get elements from user input
void getElements(Key* arr, int quantity) {
    for (int i = 0; i < quantity; i++) {
        scanf("%d", &arr[i]);
    }
}

// Compare elements (a < b)
int compare(Key a, Key b, int reverse, Statistics* statistics)
{
    if (statistics != NULL) statistics->comps++;
    return (!reverse ? (a < b) : (a > b));
}

// Swap elements
void swap(Key* a, Key* b, Statistics* statistics) {
    if (statistics != NULL) statistics->swaps++;
    Key aux = *a;
    *a = *b;
    *b = aux;
}
```

```c

// Print all elements in the array
void print(Key* arr, int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

```

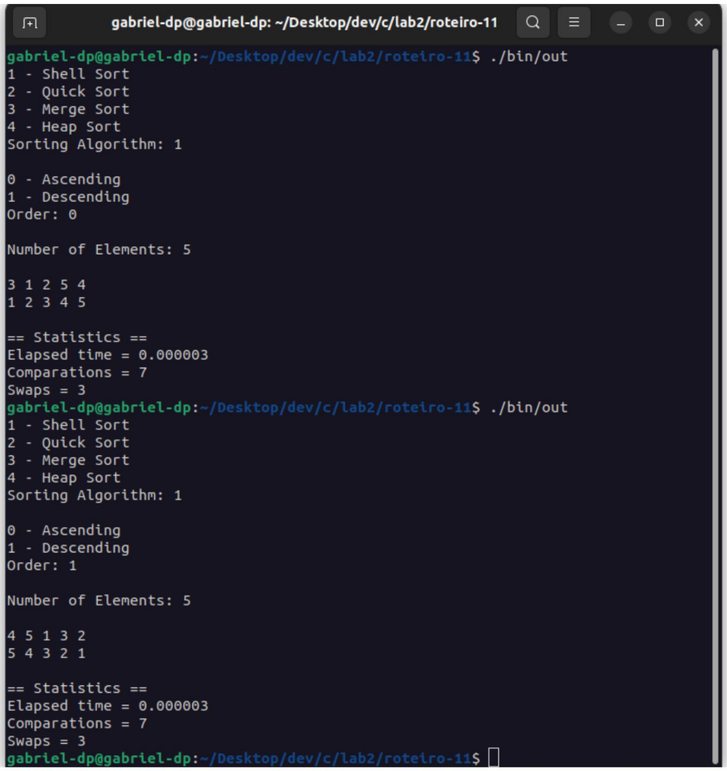
roteiro-11/src/sort.c

```c
#include "../include/sort.h"

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

#pragma region SHELL_SORT
void shellSort(Key* arr, int n, int reverse, Statistics* statistics) {
    Key aux;
    int h = 1;

    do {
        h = 3 * h + 1;
    } while (h < n);

    do {
        h /= 3;
        for (int i = h; i < n; i++) {
            aux = arr[i];

            int j = i;
            while (j >= h && !compare(arr[j - h], aux, reverse, statistics)) {
                arr[j] = arr[j - h];
                statistics->swaps++;
                j -= h;
            }

            arr[j] = aux;
        }
    } while (h > 1);
}
#pragma endregion

#pragma region QUICK_SORT
Key partition(Key* arr, int low, int high, int reverse, Statistics* statistics)
{
    Key pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j < high; j++) {
        if (!compare(pivot, arr[j], reverse, statistics)) {
            i++;
            swap(&arr[i], &arr[j], statistics);
        }
    }

    swap(&arr[i + 1], &arr[high], statistics);
    return (i + 1);
}

void quickSortRecursion(Key* arr, int low, int high, int reverse, Statistics*
statistics) {
    if (low < high) {
        int pi = partition(arr, low, high, reverse, statistics);
        quickSortRecursion(arr, low, pi - 1, reverse, statistics);
        quickSortRecursion(arr, pi + 1, high, reverse, statistics);
    }
```

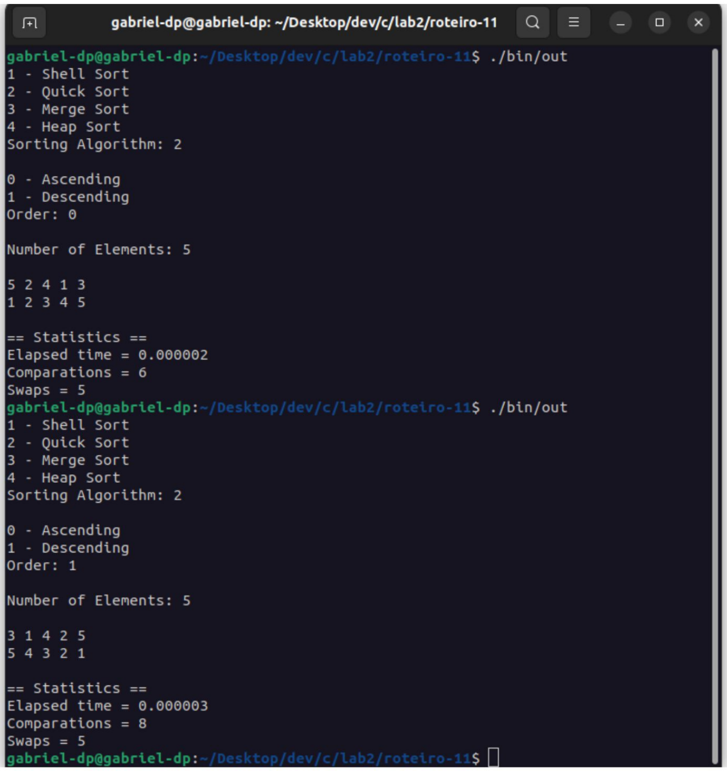
```c
}

void quickSort(Key* arr, int n, int reverse, Statistics* statistics) {
    quickSortRecursion(arr, 0, n - 1, reverse, statistics);
}
#pragma endregion

#pragma region MERGE_SORT
void merge(Key arr[], int l, int m, int r, int reverse, Statistics* statistics)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    int L[n1], R[n2];

    for (i = 0; i < n1; i++) {
        L[i] = arr[l + i];
        statistics->swaps++;
    }
    for (j = 0; j < n2; j++) {
        R[j] = arr[m + 1 + j];
        statistics->swaps++;
    }

    i = 0;
    j = 0;
    k = l;

    while (i < n1 && j < n2) {
        if (compare(L[i], R[j], reverse, statistics)) {
            arr[k] = L[i];
            statistics->swaps++;
            i++;
        } else {
            arr[k] = R[j];
            statistics->swaps++;
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        statistics->swaps++;
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        statistics->swaps++;
        j++;
        k++;
    }
}

void mergeSortRecursion(Key arr[], int l, int r, int reverse, Statistics*
statistics) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSortRecursion(arr, l, m, reverse, statistics);
        mergeSortRecursion(arr, m + 1, r, reverse, statistics);
        merge(arr, l, m, r, reverse, statistics);
    }
}

void mergeSort(Key* arr, int n, int reverse, Statistics* statistics) {
    mergeSortRecursion(arr, 0, n - 1, reverse, statistics);
}
#pragma endregion

#pragma region HEAP_SORT
void heapify(Key arr[], int n, int i, int reverse, Statistics* statistics) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && !compare(arr[left], arr[largest], reverse, statistics))
        largest = left;

    if (right < n && !compare(arr[right], arr[largest], reverse, statistics))
        largest = right;

    if (largest != i) {
        swap(&arr[i], &arr[largest], statistics);
        heapify(arr, n, largest, reverse, statistics);
    }
}

void heapSort(Key* arr, int n, int reverse, Statistics* statistics) {
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i, reverse, statistics);

    for (int i = n - 1; i >= 0; i--) {
        swap(&arr[0], &arr[i], statistics);
        heapify(arr, i, 0, reverse, statistics);
    }
}
#pragma endregion

// Function to exit the program if there is an error
void throwError(char* message) {
    printf("%s - Aborting...\n", message);
    exit(1);
}

// User select the sorting options
void getOptions(void (**sortFunction)(Key*, int, int, Statistics*), int* order,
int* quantity) {
    int option;

    // User selects the algorithm
    printf("1 - Shell Sort\n");
    printf("2 - Quick Sort\n");
    printf("3 - Merge Sort\n");
    printf("4 - Heap Sort\n");
    printf("Sorting Algorithm: ");
    scanf("%d", &option);
    switch (option) {
        case 1:
            *sortFunction = &shellSort;
            break;
        case 2:
            *sortFunction = &quickSort;
            break;
        case 3:
            *sortFunction = &mergeSort;
            break;
        case 4:
            *sortFunction = &heapSort;
            break;
        default:
            throwError("Invalid algorithm option");
    }

    printf("\n");

    // User selects the order
    printf("0 - Ascending\n");
    printf("1 - Descending\n");
    printf("Order: ");
    scanf("%d", &option);
    if (option != 0 && option != 1)
        throwError("Invalid order option");

    *order = option;

    printf("\n");

    // User enters the quantity
    printf("Number of Elements: ");
    scanf("%d", quantity);
    if (quantity < 0)
        throwError("Invalid number");

    printf("\n");
}

```

roteiro-11/src/main.c

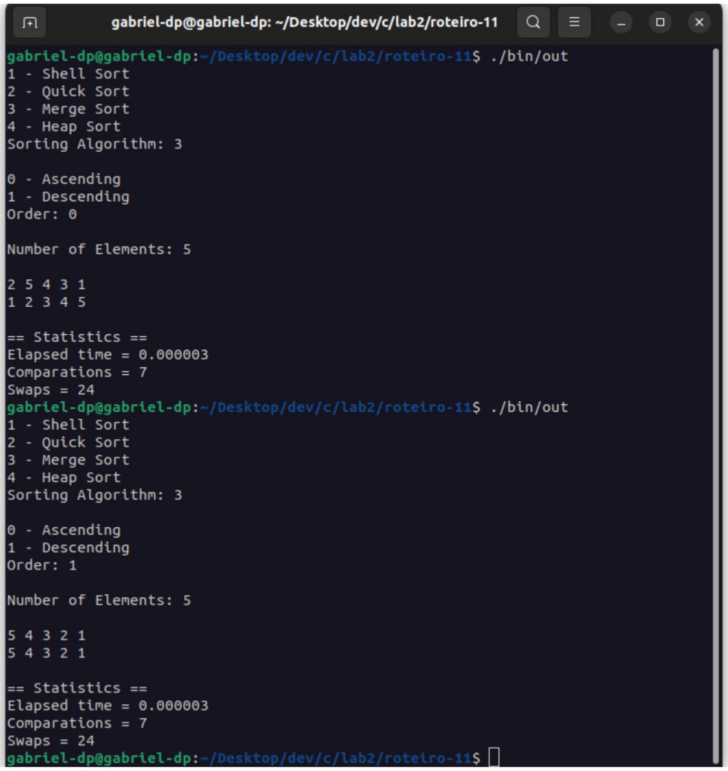
```c
#include <stdio.h>
#include <stdlib.h>

#include "../include/sort-module.h"
#include "../include/sort.h"
#include "../include/time.h"

int main() {
    void (*sortFunction)(Key *, int, int, Statistics *);
    int order, quantity;
    getOptions(&sortFunction, &order, &quantity);

    Key *arr = (Key *)malloc(quantity * sizeof(Key));
    getElements(arr, quantity);

    Statistics *s = createStatistics();
    s->start = getCpuTime();
    (*sortFunction)(arr, quantity, order, s);
    s->end = getCpuTime();

    print(arr, quantity);

    printf("\n== Statistics ==\n");
    printf("Elapsed time = ");
    printElapsedTime(s->start, s->end);
    printf("\nComparations = %Ld\n", s->comps);
    printf("Swaps = %Ld\n", s->swaps);

    destroyStatistics(&s);
    free(arr);

    return 0;
}
```



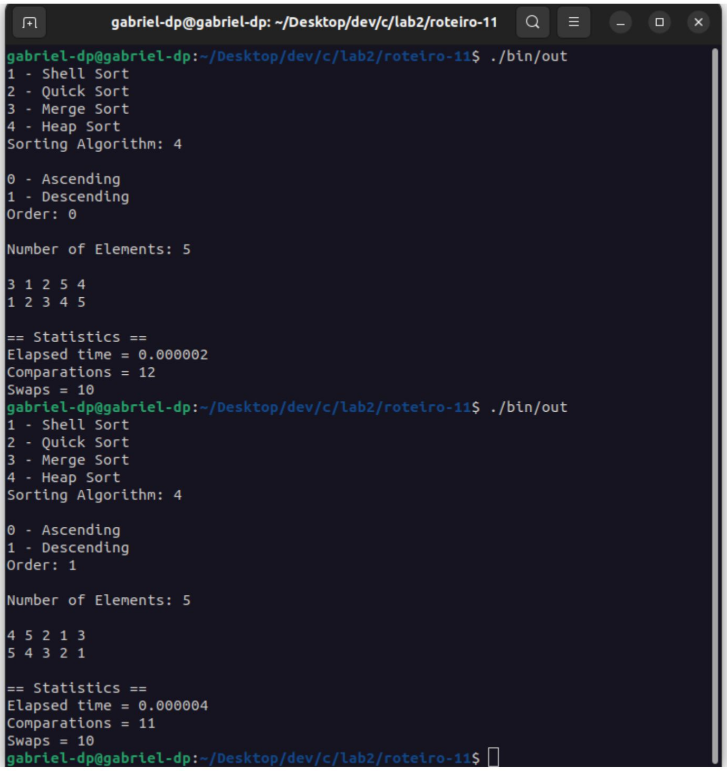
```
gabriel-dp@gabriel-dp:~/Desktop/dev/c/lab2/roteiro-11$ ./bin/out
1 - Shell Sort
2 - Quick Sort
3 - Merge Sort
4 - Heap Sort
Sorting Algorithm: 1

0 - Ascending
1 - Descending
Order: 0

Number of Elements: 5

3 1 2 5 4
1 2 3 4 5

== Statistics ==
Elapsed time = 0.000003
Comparations = 7
Swaps = 3
gabriel-dp@gabriel-dp:~/Desktop/dev/c/lab2/roteiro-11$ ./bin/out
1 - Shell Sort
2 - Quick Sort
3 - Merge Sort
4 - Heap Sort
Sorting Algorithm: 1

0 - Ascending
1 - Descending
Order: 1

Number of Elements: 5

4 5 1 3 2
5 4 3 2 1

== Statistics ==
Elapsed time = 0.000003
Comparations = 7
Swaps = 3
gabriel-dp@gabriel-dp:~/Desktop/dev/c/lab2/roteiro-11$
```



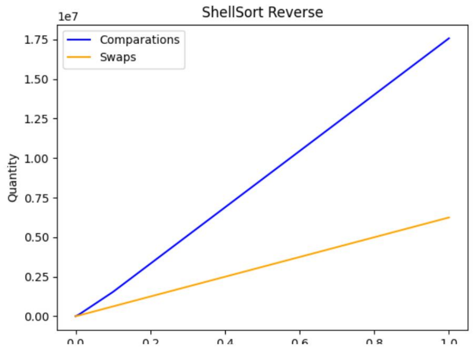
```
gabriel-dp@gabriel-dp:~/Desktop/dev/c/lab2/roteiro-11$ ./bin/out
1 - Shell Sort
2 - Quick Sort
3 - Merge Sort
4 - Heap Sort
Sorting Algorithm: 2

0 - Ascending
1 - Descending
Order: 0

Number of Elements: 5

5 2 4 1 3
1 2 3 4 5

== Statistics ==
Elapsed time = 0.000002
Comparations = 6
Swaps = 5
gabriel-dp@gabriel-dp:~/Desktop/dev/c/lab2/roteiro-11$ ./bin/out
1 - Shell Sort
2 - Quick Sort
3 - Merge Sort
4 - Heap Sort
Sorting Algorithm: 2

0 - Ascending
1 - Descending
Order: 1

Number of Elements: 5

3 1 4 2 5
5 4 3 2 1

== Statistics ==
Elapsed time = 0.000003
Comparations = 8
Swaps = 5
gabriel-dp@gabriel-dp:~/Desktop/dev/c/lab2/roteiro-11$
```

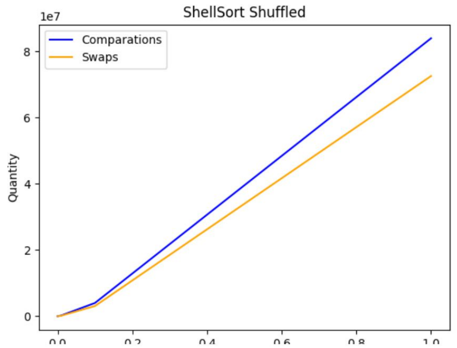


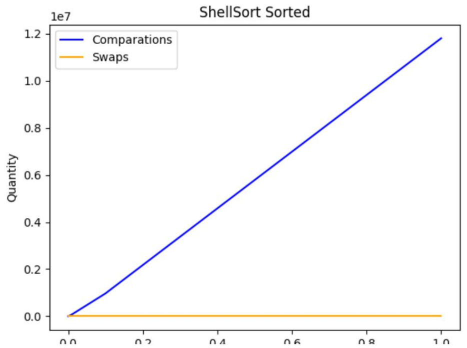
```
gabriel-dp@gabriel-dp:~/Desktop/dev/c/lab2/roteiro-11$ ./bin/out
1 - Shell Sort
2 - Quick Sort
3 - Merge Sort
4 - Heap Sort
Sorting Algorithm: 3

0 - Ascending
1 - Descending
Order: 0

Number of Elements: 5

2 5 4 3 1
1 2 3 4 5

== Statistics ==
Elapsed time = 0.000003
Comparations = 7
Swaps = 24
gabriel-dp@gabriel-dp:~/Desktop/dev/c/lab2/roteiro-11$ ./bin/out
1 - Shell Sort
2 - Quick Sort
3 - Merge Sort
4 - Heap Sort
Sorting Algorithm: 3

0 - Ascending
1 - Descending
Order: 1

Number of Elements: 5

5 4 3 2 1
5 4 3 2 1

== Statistics ==
Elapsed time = 0.000003
Comparations = 7
Swaps = 24
gabriel-dp@gabriel-dp:~/Desktop/dev/c/lab2/roteiro-11$
```

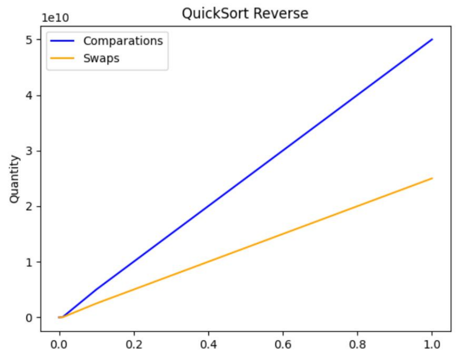


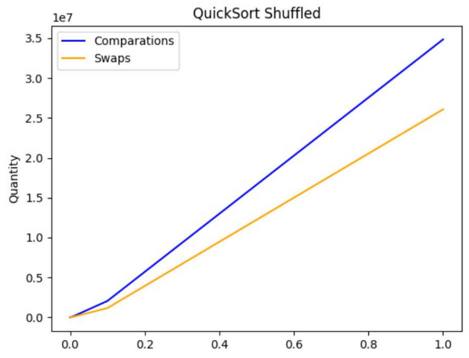
```
gabriel-dp@gabriel-dp:~/Desktop/dev/c/lab2/roteiro-11$ ./bin/out
1 - Shell Sort
2 - Quick Sort
3 - Merge Sort
4 - Heap Sort
Sorting Algorithm: 4

0 - Ascending
1 - Descending
Order: 0

Number of Elements: 5

3 1 2 5 4
1 2 3 4 5

== Statistics ==
Elapsed time = 0.000002
Comparations = 12
Swaps = 10
gabriel-dp@gabriel-dp:~/Desktop/dev/c/lab2/roteiro-11$ ./bin/out
1 - Shell Sort
2 - Quick Sort
3 - Merge Sort
4 - Heap Sort
Sorting Algorithm: 4

0 - Ascending
1 - Descending
Order: 1

Number of Elements: 5

4 5 2 1 3
5 4 3 2 1

== Statistics ==
Elapsed time = 0.000004
Comparations = 11
Swaps = 10
gabriel-dp@gabriel-dp:~/Desktop/dev/c/lab2/roteiro-11$
```

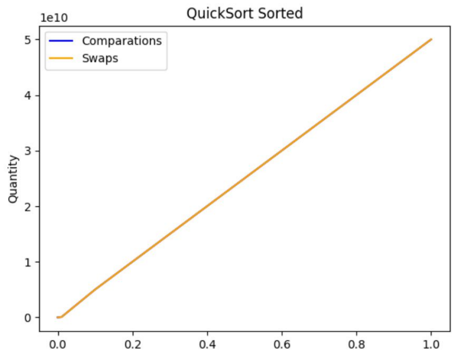

ShellSort Reverse
1e7
Comparations
Swaps
Quantity

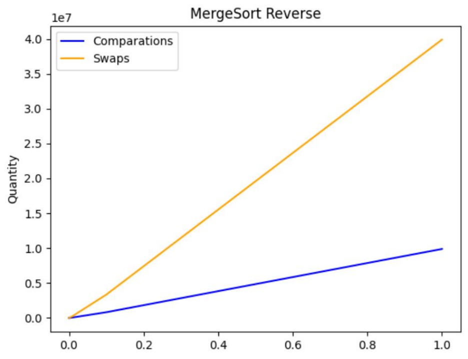

ShellSort Shuffled
1e7
Comparations
Swaps
Quantity

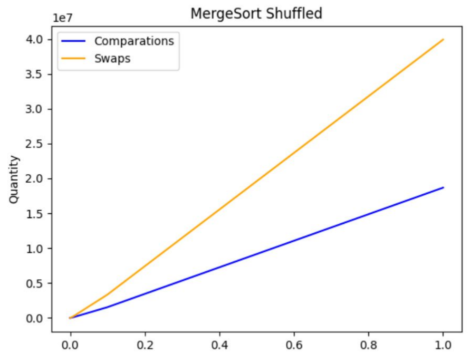

ShellSort Sorted
1e7
Comparations
Swaps
Quantity

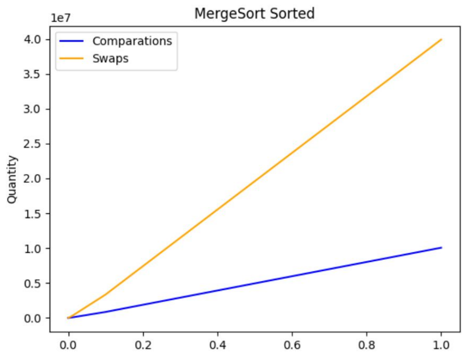

QuickSort Reverse
1e10
Comparations
Swaps
Quantity

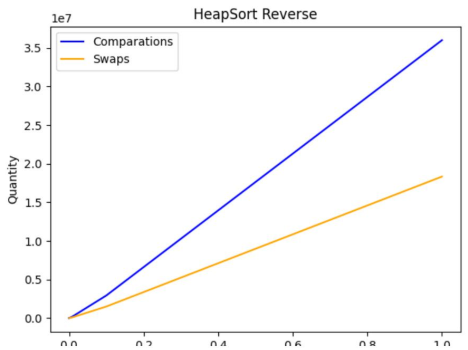

QuickSort Shuffled
1e7
Comparations
Swaps
Quantity

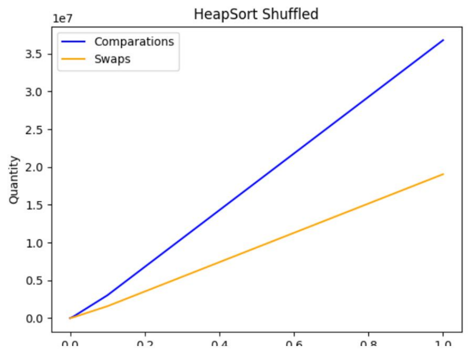

QuickSort Sorted
1e10
Comparations
Swaps
Quantity

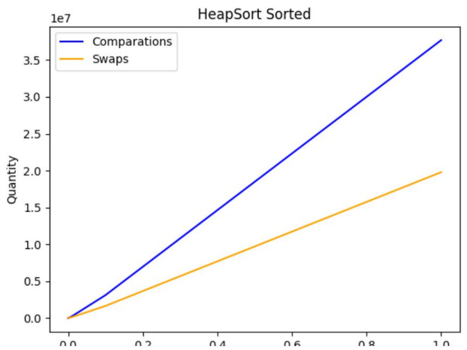

MergeSort Reverse
1e7
Comparations
Swaps
Quantity


MergeSort Shuffled
1e7
Comparations
Swaps
Quantity


MergeSort Sorted
1e7
Comparations
Swaps
Quantity


HeapSort Reverse
1e7
Comparations
Swaps
Quantity


HeapSort Shuffled
1e7
Comparations
Swaps
Quantity


HeapSort Sorted
1e7
Comparations
Swaps
Quantity