Benchmarking Sorting Algorithms: Selection, Merge and Heap Sort in C

Daniel Marin
Department of Computer Science

Texas Tech University
San Jose, Costa Rica
danimari@ttu.edu

***Abstract*—The purpose of this paper is to provide an analysis on the efficiency of various sorting algorithms previously studied: Heap Sort, Merge Sort and Selection Sort. The analysis looks to quantify and verify if the theoritical differences in efficiency match real life; by creating an implementation in real life and contrasting the performance. Each algorithm was tested on arrays of randomly chosen integers of sizes: 50.000, 100.000, 150.000, and 200.000. The execution time of each run was measured, and analyzed to identify and look for discrepencies between the theoretical time complexities of each algorithm and their practical performance.**

***Index Terms*—Heap Sort, Merge Sort, Selection Sort, Time Complexities, Practical Implementation**

I. Introduction

THE SORTING Algorithms are fundamental in the area of computer science, they are the backbone of many of the algorithms, applications and systems we humans use on a daily basis. They play a critical role in making systems faster and more efficient. Thus, the efficiency of a sorting algorithm is a critical factor in determining the performance of various computational tasks that rely on them. The objective of this project is to implement and compare the efficiency of some of those sorting algorithms: Selection Sort, Merge Sort, and Heap Sort.

The primary goals of this paper is to:

- Verify that the theoretical differences in efficiency match the implemented performances
- Analyze the execution times of these algorithms in arrays of sizes: 50.000, 100.000, 150.000, and 200.000
- Draw meaningful conclusions about their efficiency and performance characteristics

Each algorithm that will be analyzed has a different method into sorting a list, with their respective time complexity. The time complexities, which are a measure of the algorithm's execution efficiency, depend on the size of the input data. Selection Sort has a time complexity of $\Theta(n^2)$, but a fairly simple implementation making it suitable for small datasets and simple systems. Merge and Heap Sort have a time complexity of $\Theta(nlogn)$ making them more efficient when dealing with large datasets, but require a complex implementation.

The environment of the system in which the project is being developed is the following: program will be developed in C, algorithms will sort arrays of the sizes stated above, and the development of tables and graphs will be done in LaTeX.

With the tests and results we look to observe the scalability of each algorithm as the size of the input increases, and compare those results by analyzing the shape of the execution times curves to identify the trends and patterns. Specificially, we look to visualize that the time complexies of each algorithm match their execution times. So, we expect the curve of $\Theta(n^2)$ to be exhibited in the graphs related to the selection sort algorithm, while the curve of $\Theta(nlogn)$ to be present in the graphs of the other two algorithms. The results of this paper will provide the valuable insights into the reality of these algorithms in application.

II. Methodology

This section looks to outline the procedures followed to conduct the experiments and collect the necessary data for evaluating the performance of the implemented sorting algorithms. This section provides information about the programming environment, data generation process, execution of experiments, and the algorithms in it. For this paper, three algorithms where used: Selection Sort, Merge Sort and Heap Sort[1].

A. Selection Sort

Selection sort is a simple algorithm that repeatedly finds the minimum element from the unsorted part of the array and puts it at the beginning of that part. This process continues until the entire array has been sorted. The implementation is as follows:

```
void selection_sort(int A[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int minInd = i;
        for (int j = i + 1; j < n; j++)
            if (A[j] < A[minInd])
                minInd = j;
        swap(&A[minInd], &A[i]);
    }
}
```

[1]The program expected Merge Sort implementation to use `mergesort` and Heap Sort's to use `heapsort` as the names of the functions, but do to an issues with the `stdlib.h` library the names where changed to `merge_sort` and `heap_sort`

B. Merge Sort

Merge Sort is a divide-and-conquer algorithm that divides the array into 'equal' halves, recursively sorts the two halves, and then merges them to produce a single sorted array. The main issue this algorithm poses is it's space complexity of $\Theta(n)$ making it unsuitable for big data sets.

The merge sort is implemented using three functions:

- `merge`: this function merges two sorted subarrays into a single sorted array. The implementation is as follows:

```
void merge(int A[], int beg, int mid, int end) {
    int i = beg, j = mid + 1, index = 0;
    int size = end - beg + 1;
    int* temp = (int*)malloc(size * sizeof(int))
    ;
    if (!temp) {
        perror("malloc failed");
        exit(EXIT_FAILURE);
    }
    while (i <= mid && j <= end) {
        if (A[i] <= A[j]) {
            temp[index++] = A[i++];
        } else {
            temp[index++] = A[j++];
        }
    }
    while (i <= mid) {
        temp[index++] = A[i++];
    }
    while (j <= end) {
        temp[index++] = A[j++];
    }
    for (i = 0; i < size; i++) {
        A[beg + i] = temp[i];
    }
    free(temp);
}
```

- `merge_sort_rec`: this function is in charge of the recursive behavior of the Merge Sort algorithm of dividing the array into halves. The implementation is as follows:

```
void merge_sort_rec(int A[], int beg, int end) {
    if (beg < end) {
        int mid = (beg + end) / 2;
        merge_sort_rec(A, beg, mid);
        merge_sort_rec(A, mid + 1, end);
        merge(A, beg, mid, end);
    }
}
```

- `merge_sort`: this is the main function of the algorithm that calls the recursive version. The implementation is as follows:

```
void merge_sort(int A[], int n) {
    merge_sort_rec(A, 0, n - 1);
}
```

The behavior of this algorithm is encapsulated and implemented with the use of the aforementioned functions.

C. Heap Sort

Heap Sort is a comparison-based sorting algorithm that usesa binary heap data structure to sort the elemenents of the array. In practice Heap Sort, builds a max heap from the array. Then, it repeatedly extracts the maximum element form the heap (the root) and rebuilds the heap, will not considering the extracted element.

The functions used to implement the behavior of this algorithm are the following:

- `heapify`: this function is used to build a max heap from the array. The implementation is as follows:

```
void heapify(int A[], int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    if (left < n && A[left] > A[largest])
        largest = left;
    if (right < n && A[right] > A[largest])
        largest = right;

    // Swap and continue heapifying if root is
    not largest
    if (largest != i) {
        swap(&A[i], &A[largest]);
        heapify(A, n, largest);
    }
}
```

- `swap`: this function is in charge of swapping elements in the array. The implementation is as follows:

```
        void swap(int *a, int *b) {
            int temp = *a;
            *a = *b;
            *b = temp;
            }
```

- `heap_sort`: this is the main function that builds the max heap, repeatedly extracts the maximum element and rebuilds the heap until the array is sorted. The implementation is as follows:

```
void heap_sort(int A[], int n) {
    // Heapify Original Array
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(A, n, i);
    // Heap Sort
    for (int i = n - 1; i >= 0; i--) {
        swap(&A[0], &A[i]);
        heapify(A, i, 0);
    }
}
```

D. Environmental Setup

The experiments where conducted using a C programming environment developed in VSCode on a computer system. The programming environment included the use of C libraries that provided basic functionality for the program to be meaningful. The experiments where performed on an operating system with no other significant processes running in the background to ensure consistent results.

The system specifications for the computer system utilized in the development of this computer are as follows:

- Operating System: [Windows 11 Home: 64 Bit]

- Processor: [Intel Core i5-10400F, 2.90 GHz (6 cores)]
- RAM: [32 GB]
- Storage: [SSD, 1TB]

E. Test Formats

The tests performed where as follows: an array of size ranging from 50.000 to 200.000 in 50.000 increments was filled with random numbers using the rand() function from the stdlib.h library, then the sorting algorithm was called using the following function that was in charge of measuring the execution time of the algorithm taking place:

```
double measureSortTime(SortFunction func, int arr[],
     int n) {
    clock_t start, end;
    start = clock();
    func(arr, n);
    end = clock();
    check_unsorted(arr, n);
    return ((double)(end - start)) / CLOCKS_PER_SEC;
}
```

Each sorting algorithm was measured 3 times per size, until all of the desired results where measured.

F. Limitations

The experiments are designed to provide meaningful insights into the performance and where developed to be run in any computer with access to a C compiler, we must take into consideration that there are some limitations out of our control such as: system resource constraints, potential biases introduced by the standard C libraries, and the overall capabilities of the hardware of the computer system. These limitations will influence the end results of this paper, still the behavior of the algorithms should still provide the insight we are looking to demonstrate.

III. Results

In this section we present the results of the experiments conducted to analyze the performance of the three sorting algorithms, while handling arrays of varying sizes (50.000 to 200.000 in intervals of 50.000).

TABLE I
TIME OF EXECUTION OF ALGORITHMS

		Time (s)			
		Execution Time			
Algorithm	Size (k)	1	2	3	Mean
Heap	50000	0.0085	0.0087	0.0088	0.0087
	100000	0.0184	0.0180	0.0182	0.0182
	150000	0.0280	0.0286	0.0283	0.0283
	200000	0.0393	0.0393	0.0388	0.0391
Quick	50000	0.0048	0.0048	0.0047	0.0048
	100000	0.0103	0.0103	0.0102	0.0103
	150000	0.0157	0.0157	0.0157	0.0157
	200000	0.0210	0.0209	0.0211	0.0210
Radix	50000	0.0030	0.0030	0.0032	0.0031
	100000	0.0060	0.0061	0.0061	0.0061
	150000	0.0091	0.0091	0.0091	0.0091
	200000	0.0120	0.0121	0.0122	0.0121