Vilnius Gedemino Technical University
Faculty of Economics

Comparison of two algorithms

Author:
Ivan Anton

Date:
November 29, 2023

Contents

1 Introduction

This report delves into the comparative analysis of two essential sorting algorithms: Selection Sort and Shell Sort. Sorting algorithms are pivotal in computer science, influencing the efficiency of data processing across diverse applications. Selection Sort, a straightforward method, repeatedly selects the minimum element, while Shell Sort, an advanced variation, employs a unique gap sequence for efficient sorting. Through meticulous benchmarking, we aim to uncover the strengths and weaknesses of these algorithms, providing valuable insights into their real-world applicability and efficiency under different scenarios.

2 What are sorting Algorithms?

Sorting algorithms are step-by-step procedures for arranging elements in a specific order. The primary objective of sorting is to organize data in a structured manner, making it easier to search, retrieve, and analyze. Sorting is a fundamental operation in computer science and plays a crucial role in various applications, including database management, data processing, and information retrieval.

There are numerous sorting algorithms, each with its own set of rules and procedures for rearranging elements. The choice of a sorting algorithm depends on factors such as the size of the dataset, the distribution of data, and the specific requirements of the application.

Common Sorting Algorithms

- **Bubble Sort:** Compares and swaps adjacent elements until the entire list is sorted.
- **Insertion Sort:** Builds the sorted list one element at a time by repeatedly taking elements from the unsorted part and inserting them into their correct position.
- **Selection Sort:** Divides the list into sorted and unsorted portions, repeatedly selects the smallest (or largest) element from the unsorted part and adds it to the sorted part.
- **Merge Sort:** Divides the list into smaller sublists, recursively sorts them, and then merges them to produce a sorted list.
- **Quick Sort:** Selects a 'pivot' element, partitions the other elements into two sublists according to whether they are less than or greater than the pivot, and then recursively sorts the sublists.

- **Heap Sort:** Builds a binary heap and repeatedly extracts the maximum element from it.
- **Shell Sort:** A variation of insertion sort that allows the exchange of items that are far apart.

3 Selection sort

Selection Sort is a simple sorting algorithm that works by repeatedly finding the minimum element from the unsorted part of the array and putting it at the beginning. The algorithm maintains two subarrays: the sorted subarray and the unsorted subarray.

The steps of the Selection Sort algorithm are as follows:

1. Find the minimum element in the unsorted subarray.
2. Swap the found minimum element with the first element.
3. Move the boundary between the sorted and unsorted subarrays one element to the right.

The process is repeated until the entire array is sorted.

Listing 1: Selection Sort in C++

```
#include <iostream>

void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; ++i) {
        int minIndex = i;
        for (int j = i + 1; j < n; ++j) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        // Swap the found minimum element with the first element
        int temp = arr[i];
        arr[i] = arr[minIndex];
        arr[minIndex] = temp;
    }
}
```

```
int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr) / sizeof(arr[0]);

    selectionSort(arr, n);

    std::cout << "Sorted array: ";
    for (int i = 0; i < n; i++) {
        std::cout << arr[i] << " ";
    }

    return 0;
}
```

4 Shell sort

Shell Sort is an in-place comparison-based sorting algorithm. It starts by sorting pairs of elements far apart from each other and progressively reduces the gap between elements to be compared. The final step of Shell Sort is a simple insertion sort, but by this point, the data is already partially sorted.

The steps of the Shell Sort algorithm are as follows:

1. Choose a gap sequence, which determines the gaps between elements to be compared.
2. Starting with the largest gap, perform an insertion sort on sublists with elements spaced by the chosen gap.
3. Reduce the gap and repeat step 2 until the gap becomes 1, at which point a final insertion sort is performed.

The process results in a partially sorted array, making the final insertion sort more efficient.

Listing 2: Shell Sort in C++

```
#include <iostream>

void shellSort(int arr[], int n) {
    for (int gap = n / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < n; ++i) {
            int temp = arr[i];
            int j;
```

```
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {
                arr[j] = arr[j - gap];
            }
            arr[j] = temp;
        }
    }
}

int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr) / sizeof(arr[0]);

    shellSort(arr, n);

    std::cout << "Sorted array: ";
    for (int i = 0; i < n; i++) {
        std::cout << arr[i] << " ";
    }

    return 0;
}
```

5 Comparison Selection and Shell algorithms

In this section, we present a comparison of the Selection Sort and Shell Sort algorithms based on the execution time and the number of swaps performed for different data sets.

5.1 Table with Time and Swaps

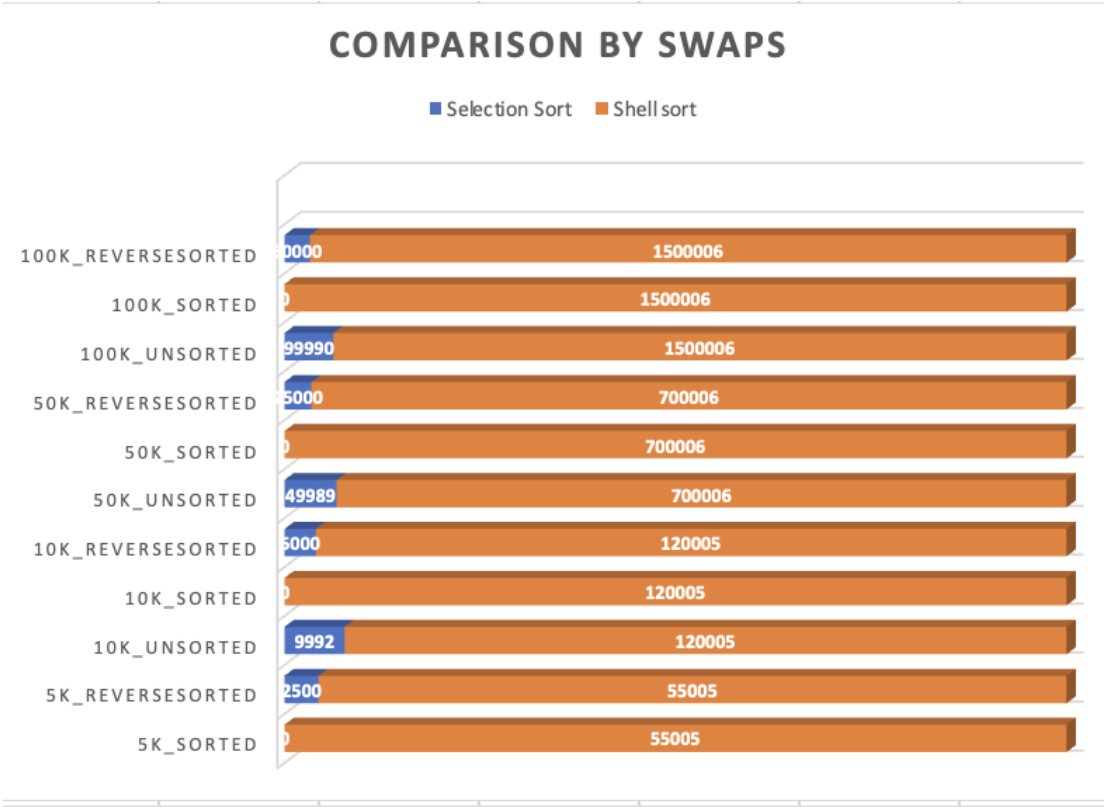
Table 1 presents the number of interation and execution time (in microseconds) for both algorithms across various data sets.

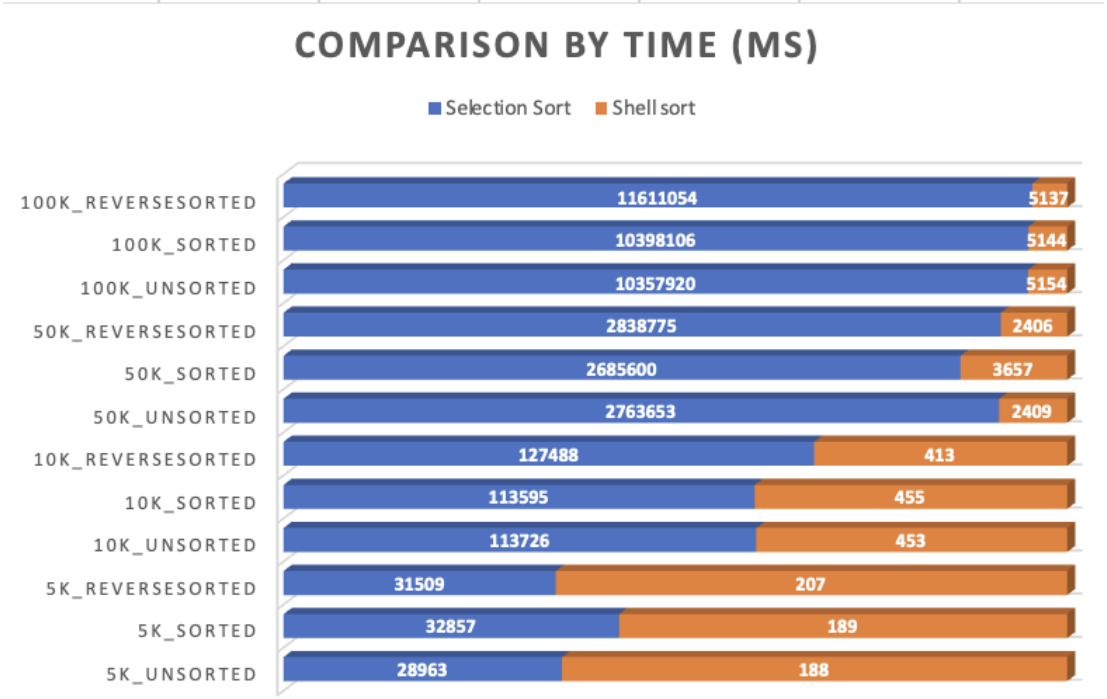
Table 1: Comparison of Selection Sort and Shell Sort

Data Set	Algorithm	Time (microseconds)	Swaps
5k_unsorted	Selection Sort	28963	4988
	Shell Sort	188	55005
5k_sorted	Selection Sort	32857	0
	Shell Sort	189	55005
5k_reverseSorted	Selection Sort	31509	2500
	Shell Sort	207	55005
10k_unsorted	Selection Sort	113726	9992
	Shell Sort	453	120005
10k_sorted	Selection Sort	113595	0
	Shell Sort	455	120005
10k_reverseSorted	Selection Sort	127488	5000
	Shell Sort	413	120005
50k_unsorted	Selection Sort	2763653	49989
	Shell Sort	2409	700006
50k_sorted	Selection Sort	2685600	0
	Shell Sort	3657	700006
50k_reverseSorted	Selection Sort	2838775	25000
	Shell Sort	2406	700006
100k_unsorted	Selection Sort	10357920	99990
	Shell Sort	5154	1500006
100k_sorted	Selection Sort	10398106	0
	Shell Sort	5144	1500006
100k_reverseSorted	Selection Sort	11611054	50000
	Shell Sort	5137	1500006

5.2 Graphs with Time and Swaps

Figure 1 and 2 visually represents the execution time for both algorithms.


COMPARISON BY SWAPS
Selection Sort
Shell sort
100K_REVERSESORTED
100K_SORTED
100K_UNSORTED
50K_REVERSESORTED
50K_SORTED
50K_UNSORTED
10K_REVERSESORTED
10K_SORTED
10K_UNSORTED
5K_REVERSESORTED
5K_SORTED
1500006
1500006
1500006
700006
700006
700006
120005
120005
120005
55005
55005


COMPARISON BY TIME (MS)
Selection Sort
Shell sort
100K_REVERSESORTED
100K_SORTED
100K_UNSORTED
50K_REVERSESORTED
50K_SORTED
50K_UNSORTED
10K_REVERSESORTED
10K_SORTED
10K_UNSORTED
5K_REVERSESORTED
5K_SORTED
5K_UNSORTED
11611054
5137
10398106
5144
10357920
5154
2838775
2406
2685600
3657
2763653
2409
127488
413
113595
455
113726
453
31509
207
32857
189
28963
188

5.3 Discuss about output value

Time Comparison

For both algorithms, Shell Sort consistently outperforms Selection Sort in terms of time. The time taken by Shell Sort remains relatively stable across different scenarios (unsorted, sorted, and reverse sorted), showcasing its efficiency. Selection Sort's time increases significantly with larger dataset sizes and varies noticeably between sorted, unsorted, and reverse sorted scenarios.

Swaps Comparison:

In terms of swaps, Selection Sort demonstrates a different pattern compared to time. Selection Sort requires more swaps as the dataset becomes more unsorted, and the number of swaps decreases when the data is already sorted. Shell Sort consistently performs a high number of swaps regardless of the initial state of the dataset. The number of swaps for both algorithms increases with larger dataset sizes, as expected.

6 Conclusion

If the primary concern is minimizing the number of swaps, Shell Sort is less favorable compared to Selection Sort, especially when the dataset is mostly sorted. However, if the emphasis is on overall efficiency, considering both time and swaps, Shell Sort is a more reliable choice. It consistently provides better time performance across different scenarios, and its additional swaps may be acceptable in practical applications. It's important to note that the efficiency of sorting algorithms can vary based on the specific characteristics of the dataset and the implementation details. The choice between Selection Sort and Shell Sort depends on the specific requirements and constraints of the given problem.