Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică
Departamentul Ingineria Software și Automatică

Laboratory Work No.2

Theme: Algorithm Analysis. Sorting

Realized by:

Popa Eugeniu FAF-202

Chisinau 2021

Contents

1. Algorithm analysis 3
1.1 The purpose of the work 3
1.2 Theoretical notes 3
2. Introduction 4
3. Implementation 5
3.1 Quick Sort 5
3.2 Merge Sort 6
3.3 Heap Sort 8
3.4 Bubble Sort 9
4. Analysis of the results. Empirical analysis of algorithms using time 10
5. Conclusions 14

1. Algorithm analysis

1.1 The purpose of the work:

Study empirical analysis of sorting algorithms. Quick Sort, Merge Sort, Heap Sort and Bubble Sort.

1.2 Theoretical notes:

Sorting is the rearrangement of a list into an order defined by a monotonically increasing or decreasing sequence of sort keys, where each sort key is a single-valued function of the corresponding element of the list. Sorting reorders a list into a sequence suitable for further processing or searching. Often the sorted output is intended for people to read; sorting makes it much easier to understand the data and to find a datum.

Sorting is used in many types of programs and on all kinds of data. It is such a common, resource-consuming operation that sorting algorithms and the creation of optimal implementations constitute an important branch of computer science.

Classification of sorting algorithms:

Computational complexity (worst, average and best behavior) in terms of the size of the list (n). For typical serial sorting algorithms good behavior is O (n log n) and bad behavior is O(n2). Ideal behavior for a serial sort is O(n), but this is not possible in the average case. Optimal parallel sorting is O (log n). Comparison-based sorting algorithms, need at least O (n log n) comparisons for most inputs.

- Memory usage: In particular, some sorting algorithms are "in-place". Strictly, an in-place sort needs only O(1) memory beyond the items being sorted; sometimes O(log(n)) additional memory is considered "in place".
- Recursion: Some algorithms are either recursive or non-recursive, while others may be both (e.g., merge sort).
- Stability: stable sorting algorithms maintain the relative order of records with equal keys (i.e., values).
- Whether or not they are a comparison sort. A comparison sort examines the data only by comparing two elements with a comparison operator.
- General method: insertion, exchange, selection, merging, etc.
- Exchange sorts include bubble sort and quicksort.
- Selection sorts include heap sort. Also, whether the algorithm is serial or parallel.
- Adaptability: Whether or not the presort of the input affects the running time. Algorithms that take this into account are known to be adaptive.

Criteria for sorting:

The piece of data actually used to determine the sorted order is called the key. Sorting algorithms are usually judged by their efficiency. In this case, efficiency refers to the algorithmic efficiency as the size of the input grows large and is generally based on the number of elements to sort. Most of the algorithms in use have an algorithmic efficiency of either O (n^2) or O(n*log(n)).

2. Introduction

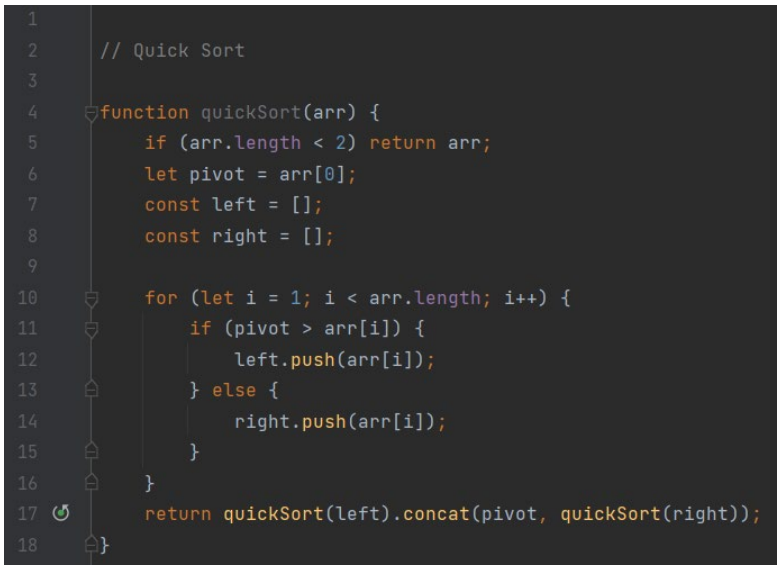
In this laboratory work, I am going to study the sorting algorithms, and to make an empirical analysis of them. In order to better understand the topic, we have to explain what is sorting.
Searching and Sorting are the tasks that are frequently encountered in various Computer Applications. Since they reflect fundamental tasks that must be tackled quite frequently, researchers have attempted in past to develop algorithms efficient in terms of optimum memory requirement and minimum time requirement i.e., Time or Space Complexities. Together with searching, sorting is probably the most used algorithm in Computing, and one in which, statistically, computers spend around half of the time performing.
Sorting algorithms are always attractive because of the amount of time computers spend on the process of sorting has always been a matter of research attention. For this reason, the development of fast, efficient and inexpensive algorithms for sorting and ordering lists and arrays is a fundamental field of computing.
By optimizing sorting, computing as a whole will be faster. When we look to develop or use a sorting algorithm on large problems, we came across previous research literature where it was mentioned clearly to concentrate on how long the algorithm might take to run.
We discovered that, the time for most sorting algorithms depends on the amount of data or size of the problem and in order to analyze an algorithm, we required to find a relationship showing how the time needed for the algorithm depends on the amount of data. We found that, for an algorithm, when we double the amount of data, the time needed is also doubled.
The analysis of another algorithm told us that when we double the amount of data, the time is increased by a factor of four. The latter algorithm would have the time needed increase much more rapidly than the first.
We also found that, the analysis of efficiency depends considerably on the nature of the data. For example, if the original data set is almost ordered already, a sorting algorithm may behave rather differently than if the data set originally contains random data or is ordered in the reverse direction. The purpose of this investigation is to magnify analysis of sorting algorithms considering all possible factors and make a concise note of it.
My work may be useful for some applications that seek to determine which sorting algorithm is the fastest to sort the lists of different lengths, and, to, therefore determine which algorithm should be used depending on the list length.
For example, Shell Sort should be used for sorting of small (less than 1000 items) arrays. It has the advantage of being an in-place and non-recursive algorithm, and is faster than Quicksort up to a certain point. For larger arrays, the best choice is Quicksort, which uses recursion to sort lists, leading to higher system usage but significantly faster results. We have attempted to review the rich body of sorting literature in accord with their utility and performance so as to make a critical analysis of them in order to discover tuning factors. These factors are intended to help the reader to avoid wasted efforts in order to produce correct complexity values. Most of the part of this paper concentrates on the study of algorithms for problems in the standard format where both instances and outputs are finite objects, and the key metrics are resource usage (typically time and memory).
Several of the suggestions enunciated here may be somewhat controversial, but I have, at least elaborated them.
Indeed, although there is much common agreement on what makes good experimental analysis of sorting algorithms, certain aspects have been the subject of debate, such as the relevance of running time comparisons.

3. Implementation

3.1 Quick Sort

The quick sort is an in-place, divide-and conquer, massively recursive sort. As a normal person would say, it's essentially a faster in place version of the merge sort. The quick sort algorithm is simple in theory, but very difficult to put into code. This recursive algorithm consists of making decisions based on the pivot element. It then splits the array into two parts - one with elements larger than the pivot and the other with elements smaller than the pivot. If there are one or less elements in the array to be sorted, then returns immediately. The efficiency of the algorithm is majorly impacted by which element is chosen as the pivot point. **The worst-case efficiency of the quick sort, O (n^2),** occurs when the list is sorted and the leftmost element is chosen. Randomly choosing a pivot point rather than using the left-most element is recommended if the data to be sorted isn't random. **As long as the pivot point is chosen randomly, the quick sort has an algorithmic complexity of O (n log n).** It is extremely fast. It is very complex algorithm, massively recursive. The running time of quick sort depends on whether partition is balanced or unbalanced, which in turn depends on which elements of an array to be sorted are used for partitioning. A very good partition splits an array up into two equal sized arrays. A bad partition, on other hand, splits an array up into two arrays of very different sizes. The worst partition puts only one element in one array and all other elements in the other array. If the partitioning is balanced, the Quick sort runs asymptotically as fast as merge sort. On the other hand, if partitioning is unbalanced, the Quick sort runs asymptotically as slow as insertion sort. The best thing that could happen in Quick sort would be that each partitioning stage divides the array exactly in half. The quick sort is by far the fastest of the common sorting algorithms. It's possible to write a special-purpose sorting algorithm that can beat the quick sort for some data sets, but for general-case sorting there isn't anything faster. We conclude with an observation. It has been brought to our notice that the empirical data obtained reveals the speed of each algorithm, from slowest to fastest, for a sufficiently large list, ranks as 1) Quicksort, 2) Selection sort, 3) Insertion sort, 4) Bubble sort. There is a large difference in the time taken to sort very large lists between the fastest two and the slowest three. This is due to the efficiency Quicksort has over the others when the list sorted is sufficiently large. Also, the results show that for a very small list size, only selection sort and insertion sort are faster than Quick, and by a very small amount.

Quick Sort code implementation:



```
// Quick Sort

function quickSort(arr) {
    if (arr.length < 2) return arr;
    let pivot = arr[0];
    const left = [];
    const right = [];

    for (let i = 1; i < arr.length; i++) {
        if (pivot > arr[i]) {
            left.push(arr[i]);
        } else {
            right.push(arr[i]);
        }
    }
    return quickSort(left).concat(pivot, quickSort(right));
}
```

3.2 Merge Sort

Merge sort is a sorting technique based on **divide and conquer technique. With worst-case time complexity being O(n log n),** it is one of the most respected algorithms.

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

To understand merge sort, we take an unsorted array as the following –

We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.

This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

We further divide these arrays and we achieve atomic value which can no more be divided.

Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.

In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.

After the final merging, the list should look like this –

Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

Step 1 − if it is only one element in the list it is already sorted, return.

Step 2 − divide the list recursively into two halves until it can no more be divided.

Step 3 − merge the smaller lists into new list in sorted order.

Merge Sort code implementation:

```
19
20  // Merge Sort
21
22  function merge(left, right) {
23      let sortedArr = []
24      while (left.length && right.length) {
25          if (left[0] < right[0]) {
26              sortedArr.push(left.shift())
27          } else {
28              sortedArr.push(right.shift())
29          }
30      }
31      return [...sortedArr, ...left, ...right]
32  }
33
34  function mergeSort(arr) {
35      if (arr.length <= 1) return arr
36      let mid = Math.floor( x: arr.length / 2)
37      let left = mergeSort(arr.slice(0, mid))
38      let right = mergeSort(arr.slice(mid))
39      return merge(left, right)
40  }
41
```

3.3 Heap Sort

Heap Sort is another example of an efficient sorting algorithm. Its main advantage is that it has a **great worst-case runtime of O(n*log n)** regardless of the input data. As the name suggests, Heap Sort relies heavily on the heap data structure. Without a doubt, **Heap Sort is one of the simplest sorting algorithms to implement** and coupled with the fact that it's a fairly efficient algorithm compared to other simple implementations.

Heap Sort works by "removing" elements from the heap part of the array one-by-one and adding them to the sorted part of the array. It is an in-place algorithm, meaning that it requires a constant amount of additional memory, i.e. the memory needed doesn't depend on the size of the initial array itself, other than the memory needed to store that array.

For example, no copies of the original array are necessary, and there is no recursion and recursive call stacks. The simplest implementation of Heap Sort usually uses a second array to store the sorted values.

Heap Sort is **unstable**, meaning that it does not maintain the relative order of elements with equal values. This isn't an issue with primitive types (like integers and characters...) but it can be a problem when we sort complex types, like objects.

For example, imagine we have a custom class Person with the age and name fields, and several objects of that class in an array, including a person called "Mike" aged 19 and "David", also aged 19 - appearing in that order.

If we decided to sort that array of people by age, there would be no guarantee that "Mike" would appear before "David" in the sorted array, even though they appeared in that order in the initial array. It can happen, but it's not guaranteed.

Heap Sort code implementation:

```
50
51  function maxHeap(array, index, length) {
52      let left = 2 * index;
53      let right = 2 * index + 1;
54      let maximum;
55      if (right < length) {
56          if (array[left] >= array[right]) {
57              maximum = left;
58          } else {
59              maximum = right;
60          }
61      } else if (left < length) {
62          maximum = left;
63      } else {
64          return;
65      }
66      if (array[index] < array[maximum]) {
67          swap(array, index, maximum);
68          maxHeap(array, maximum, length);
69      }
70  }
```

```
function heapSort(array) {
    let length = array.length;
    for (let i = Math.floor( x: length / 2) - 1; i >= 0; i--) {
        maxHeap(array, i, length);
    }
    for (let i = length - 1; i >= 0; i--) {
        swap(array, index_A: 0, i);
        maxHeap(array, index: 0, i);
    }
    return array;
}
```

3.4 Bubble Sort

The bubble sort is the oldest and simplest sort in use. Unfortunately, it's the slowest one. The bubble sort works by comparing each item in the list with the item next to it, and swapping them if required. The algorithm repeats this process until it makes a pass all the way through the list without swapping any items (in other words, all items are in the correct order). This causes larger values to "bubble" to the end of the list while smaller values "sink" towards the beginning of the list.

The bubble sort is generally considered to be **the most inefficient sorting algorithm** in common usage. While the insertion and selection sorts also have **O(n^2)** complexities, they are significantly more efficient than the bubble sort. A fair number of algorithm purists (which means they've probably never written software for a living) claim that the bubble sort should never be used for any reason.

Realistically, there isn't a noticeable performance difference between the various sorts for 100 items or less, and the simplicity of the bubble sort makes it attractive. The bubble sort shouldn't be used for repetitive sorts or sorts of more than a couple hundred items. Clearly, bubble sort does not require extra memory.

Bubble Sort code implementation:

```
function bubbleSort(arr) {
    let len = arr.length;
    for (let i = 0; i < len; i++) {
        for (let j = 0; j < len; j++) {
            if (arr[j] > arr[j + 1]) {
                let tmp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = tmp;
            }
        }
    }
    return arr;
}
```

4. Analysis of the results. Empirical analysis of the algorithms using time

Code snippet for analyzing the time comparison of the algorithms:

Written form:

```
// Time comparison of the algorithms in written form

const { performance } = require('perf_hooks')
const { plot } = require('nodeplotlib')

const generateArray = (min, max, n) => {
    const arr = []
    while (arr.length < n) {
        const rand = Math.floor( x: Math.random() * (max - min) + min)
        arr.push(rand)
    }
    return arr
}

const getTime = (arr, cb) => {
    const start = performance.now();
    cb(arr);
    const end = performance.now();
    return end - start;
}

let n = [1, 5, 10, 50, 100, 500, 1000, 5000, 10000, 50000,
    100000, 200000, 300000, 400000, 500000, 600000, 700000,
    800000, 900000, 1000000]

```

```
function run(nums) {
    var quicksort = []
    var mergesort = []
    var heapsort = []
    var bubblesort = []

    for (let i = 0; i < n.length; i++) {
        const arr = generateArray( min: 1,  max: 100000, n[i])
        console.log(`For n = ${n[i]}`)

        const qSort = getTime(arr, quickSort)
        console.log(`Quick Sort`)
        console.log(`Execution time: ${qSort}`)
        quicksort.push(qSort)

        const mSort = getTime(arr, mergeSort)
        console.log(`Merge Sort`)
        console.log(`Execution time: ${mSort}`)
        mergesort.push(mSort)

        const hSort = getTime(arr, heapSort)
        console.log(`Heap Sort`)
        console.log(`Execution time: ${hSort}`)
        heapsort.push(hSort)

        const bSort = getTime(arr, bubbleSort)
        console.log(`Bubble Sort`)
        console.log(`Execution time: ${bSort}`)
        bubblesort.push(bSort)
    }
```

Graphical form:

```
    // Time comparison of the algorithms in graphical form

    const qs = {x: n, y: quicksort, type: 'line'}
    const ms = {x: n, y: mergesort, type: 'line'}
    const hs = {x: n, y: heapsort, type: 'line'}
    const bs = {x: n, y: bubblesort, type: 'line'}

    plot( data: [
        qs, ms, hs, bs
    ])
}

run(n)
```

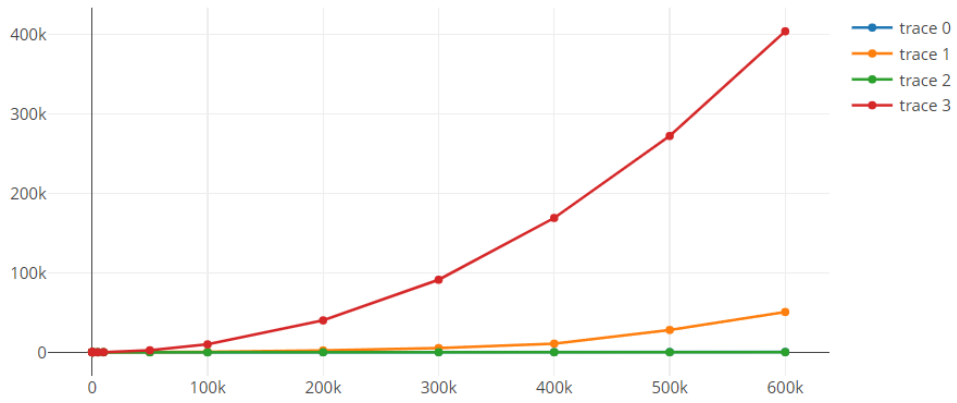
Below you can see the results of the empirical analysis in written form.

```
For n = 1                          For n = 100
Quick Sort                         Quick Sort
Execution time: 0.03839999437332153 Execution time: 0.08069999516010284
Merge Sort                         Merge Sort
Execution time: 0.043699994683265686 Execution time: 0.12699998915195465
Heap Sort                          Heap Sort
Execution time: 0.094200000166893  Execution time: 0.1413000077009201
Bubble Sort                        Bubble Sort
Execution time: 0.04720000922679901 Execution time: 0.4016000032424927
For n = 5                          For n = 500
Quick Sort                         Quick Sort
Execution time: 0.01590000092983246 Execution time: 0.6044000089168549
Merge Sort                         Merge Sort
Execution time: 0.07590000331401825 Execution time: 1.2985999882221222
Heap Sort                          Heap Sort
Execution time: 0.02229999005794525 Execution time: 2.088799998164177
Bubble Sort                        Bubble Sort
Execution time: 0.007500007748603821 Execution time: 2.9478000104427338
For n = 10                         For n = 1000
Quick Sort                         Quick Sort
Execution time: 0.025099992752075195 Execution time: 1.4653000086545944
Merge Sort                         Merge Sort
Execution time: 0.0414000004529953 Execution time: 2.7566999942064285
Heap Sort                          Heap Sort
Execution time: 0.015199989080429077 Execution time: 0.3724000006914139
Bubble Sort                        Bubble Sort
Execution time: 0.008500009775161743 Execution time: 1.2324999868869781
For n = 50                         For n = 5000
Quick Sort                         Quick Sort
Execution time: 0.03890000283718109 Execution time: 2.287200003862381
Merge Sort                         Merge Sort
Execution time: 0.058600008487701416 Execution time: 5.397799998521805
Heap Sort                          Heap Sort
Execution time: 0.06450000405311584 Execution time: 1.5703999996185303
Bubble Sort                        Bubble Sort
Execution time: 0.08609999716281891 Execution time: 25.15870000422001
```

```
For n = 10000
Quick Sort
Execution time: 4.850100010633469
Merge Sort
Execution time: 8.859900012612343
Heap Sort
Execution time: 2.4306000024080276
Bubble Sort
Execution time: 101.4716000109911
For n = 50000
Quick Sort
Execution time: 26.53839999437332
Merge Sort
Execution time: 139.809199988842
Heap Sort
Execution time: 7.105800002813339
Bubble Sort
Execution time: 2472.162000000477
For n = 100000
Quick Sort
Execution time: 59.575800001621246
Merge Sort
Execution time: 558.669599995017
Heap Sort
Execution time: 16.113299995660782
Bubble Sort
Execution time: 9910.638099998236
For n = 200000
Quick Sort
Execution time: 115.98270000517368
Merge Sort
Execution time: 2360.061700001359
Heap Sort
Execution time: 34.36410000920296
Bubble Sort
Execution time: 40306.792999997735
```

```
For n = 300000
Quick Sort
Execution time: 211.14259999990463
Merge Sort
Execution time: 5281.7874999940395
Heap Sort
Execution time: 52.33840000629425
Bubble Sort
Execution time: 91196.42729999125
For n = 400000
Quick Sort
Execution time: 269.84539999067783
Merge Sort
Execution time: 10793.054700002074
Heap Sort
Execution time: 73.54850000143051
Bubble Sort
Execution time: 168886.97910000384
For n = 500000
Quick Sort
Execution time: 305.811700001359
Merge Sort
Execution time: 27998.74400000274
Heap Sort
Execution time: 101.18369999527931
Bubble Sort
Execution time: 271966.64450000226
For n = 600000
Quick Sort
Execution time: 383.4836999922991
Merge Sort
Execution time: 50643.706299990416
Heap Sort
Execution time: 131.31440000236034
Bubble Sort
Execution time: 403582.40450000763
```

Below you can see the results of the empirical analysis in graphical form.



5. Conclusion

In the following laboratory work I have performed an empirical analysis of sorting algorithms, including Quick Sort, Merge Sort, Heap Sort and Bubble Sort. For empiric analysis, I have tested every algorithm on different sets of data on average case (when the array consists of random values).

I have tested all 4 implementations for n = 1, 5, 10, 50, 100, 100, 500, 1000, 5000, 10000, 50000, 100000, 200000, 300000, 400000, 500000, 600000.

On very small numbers ($n \in [1, 50]$), Bubble Sort was surprisingly the fastest, while Heap Sort was the slowest one. Bubble Sort is very easy to implement, while Heap Sort is a complex algorithm.

After n = 100, we can see that Bubble Sort is very slow compared to the other algorithms.

However, we can see that the first three algorithms showed satisfying results, being quite optimized for small and big range of numbers for data to be sorted.

If we make an in-depth analysis, Heap Sort shows the best time results, because the heap can easily be implemented in-place in the original array, so it needs no additional memory.

For $n \in [1000, 600\ 000]$, the fastest sorting algorithm is Heap Sort. The second one is Quick Sort, showing very good results too. The third place takes Merge Sort. Even if it performed quite bad, it can still be considered a good algorithm, due to its stability, but in this case, it is just a basic form of Merge Sort, which needs a lot of additional memory. Unfortunately, Bubble Sort shows terrible results, so we can say that it is not a useful option.

Taking in consideration all the analysis that I did during this laboratory work, I can say that if I will have to sort an array in a distant future, I will know what to choose.