Choosing best sorting algorithms for one-dimensional and two-dimensional arrays*

Konstantin Negrutsak

Slovenská technická univerzita v Bratislave

Fakulta informatiky a informačných technológií

xnegrutsak@stuba.sk

25. october 2021

1 Introduction

For a long time sorting arrays of data was very important because work with a sorted array becomes a lot easier and time-efficient. However, sorting arrays itself is a must be optimized as well. Every year companies are working with a bigger amount of data than before. And mistakes in choosing correct sorting algorithm can lead to a significant performance drops, which is unacceptable in a big projects. I want to do a research on this theme in order to clear up what algorithms are the best in different scenarios.

Firstly, I will find out what are the major differences between algorithms and what are the parameters that we count when choosing an algorithm to sort our data. (You can find it in chapter 2)

Secondly, I will define which algorithms are being used to sort data in one-dimensional arrays and in two-dimensional arrays of different length. (You can find it in chapters 3 and 4)

After all this work I will sum up everything that I did in my coursework and I will do a conclusion about what algorithms are the best choices for these two types of arrays. (You will be able to find it in chapter 5)

*Semestrálny projekt v predmete Metódy inžinierskej práce, ak. rok 2021/22, vedenie: Konstantin Negrutsak

2 Algorithm differences

Two main parameters we should focus on when it comes to choosing an algorithm are:

- BIG-O Complexity
- Memory consuming

In my coursework I will cover BIG-O Complexity and Memory consuming parameters.

2.1 BIG-O Complexity

To begin with, lets talk about BIG-O Complexity. This parameter demonstrates the dependence of the number of operations performed by the algorithm on the amount of information incoming. The smaller the growth of the function, the more efficient the algorithm is. For example the best scenario possible is O(1), when algorithm performs only one operation in order to get the needed result while the amount of information can be of any size possible.

By default number of elements in an array is defined by N. In algorithms such as binary search complexity usually equals to O($\log_2 N$). This is a very good result compared to default search, which complexity equals to O(N) (It is an algorithm in which we check up every consecutive element beginning with the first one in order to find the element we are looking for)

There are other algorithms like Bubble sort or Exchange sort which complexity is O(N^2) or above. These algorithms can be pretty fast for short arrays, but when it comes to sorting a very long array they are inefficient.

The other thing I want to mention is that when we are counting a complexity of the algorithm we don't have to take constants into account. Because the major thing that complexity tells us is how many actions will algorithm perform on a very big array. And when we say big array we mean infinite array. In this case it doesn't matter whether complexity is O(N) or O(2N). Furthermore, we also don't have to take additional complexity into account. Basically it means that if we have an algorithm, which complexity is O(N+1) it will perform almost the same amount of actions as an algorithm with complexity O(N).

When we have an algorithm with complexity O(N^2+2^N) we count the function with the slower growth as an additional complexity. In this example function N^2 grows slower than 2^N and we don't have to count it. So the final complexity will be O(2^N).

2.2 Memory consuming

Another parameter which also should be counted when it comes to comparing algorithms is how much memory it consumes. For example when we sort array with a Bubble sort, algorithm operates only with indexes of elements. We don't create any new arrays, which means that we don't consume more memory than we already did. In this case memory usage is 1. But there are some algorithms like a Merge sort which create an additional list of elements, which takes as much memory as an original unsorted list. In this case memory usage equals N. The smaller the memory consumption, the more efficient the algorithm is.

3 One-dimensional arrays

Firstly, I want to compare the most popular sorting algorithms by their complexity and memory consumption. As an example I will take 5 sorting algorithms that are often used in sorting one-dimensional arrays.

3.1 Bubble sort

One of the easiest sorting algorithm is a Bubble sort. Teachers and professors usually use it as an introduction to the sorting algorithms and Bubble sort is easy to code. This algorithm compares first two elements of an array and if the second one is less that the first one - it swaps them. Then it goes to the next pair of elements and does exactly the same. When it reaches the end of an array, it goes back to the beginning and compares elements again until the point when there will be no swaps on the iteration of the cycle.

Its complexity is O(N^2) and memory consumption equals 1. This algorithm is very inefficient when it comes to sorting a big arrays, but when you need to sort small array it is very useful because coding it is very easy.

3.2 Selection sort

Selection sort is another algorithm which is easy to implement. This algorithm finds the minimum in the entire list and swaps it with the first element. Then it finds the minimum in the remaining list(without first element) and swaps it with the second element. It continues going through the list like this until it finishes last swap.

Its complexity is O(N^2) and memory consumption equals 1. This algorithm is as inefficient as a Bubble sort when it comes to sorting big arrays, but this

algorithm can be coded very fast without any effort and if you need to sort a small array, Selection sort is one of the best available algorithms.

3.3 Insertion sort

Insertion sort is a very easy algorithm. It begins with a second element of an array and compares it to an elements to the left. Algorithm inserts taken element into the place, where an element to the left is smaller than a taken element. Then it shifts all the elements between the position where we took an element and the position where we placed that element to the right by 1. Then it goes to the third element and does exactly the same until it reaches the end of an array.

Its complexity is O(N^2) and memory consumption equals 1. While complexity and memory consumption is the same as in the Bubble sort and Selection sort, this algorithm is not as easy to code as previous algorithms that I mentioned, which puts this one in a disadvantage.

3.4 Merge sort

Firstly, I will explain how merge function works. It takes an array which consists of two sorted parts and it takes an index of the element that divides these two parts. Then it merges these two parts by comparing two elements on the left of these parts and taking them from that part to a new array. After doing it N times it results into a sorted array.

Merge sort itself divides our original array in halves, then it divides these halves in halves and so on until we will get arrays, consisting of one element. Then it consecutively applies merge function to these arrays. And because the result of the merge function is a sorted array, we can apply merge function to our sub-arrays until we will get the sorted version of the original array.

Its complexity is O($N * \log_2 N$) and memory consumption is N. While memory consumption is bigger than in algorithms that I described before, Merge sort works significantly faster when it comes to sorting big arrays.

I also want to mention that there are a lot of variations of a Merge sort such as Parallel Merge sort or Bitonic Merge sort and so on, which complexity differs from one to another. Information about these variations you can find in the article ”Performance Analysis of Merge Sort Algorithms” [1]

4 Two-dimensional arrays

5 Conclusion

Literatúra

[1] Joella Lobo and Sonia Kuwelkar. Performance analysis of merge sort algorithms. In *2020 International Conference on Electronics and Sustainable Communication Systems (ICESC)*, pages 110–115, 2020.