STUDENT SESSION

# ALGORITHM FOR SORTING NON-NEGATIVE INTEGERS

Milan Savić*,
Miodrag Živković

Singidunum University,
Belgrade, Serbia

Abstract:

This paper presents an algorithm for sorting non-negative integers. The algorithm solves the sorting problem recursively. There are numerous sorting algorithms available today, however this paper suggests a different approach to solving this problem using a binary representation of integers. The paper graphically presents comparative tests that were executed with already existing sorting algorithms. The test results show in which situations it is best to use this algorithm. Possible further directions of development of this algorithm, as well as other algorithms that can use this approach to problem-solving, are also presented.

Keywords:

sorting algorithm, binary number, recursion.

## INTRODUCTION

Data sorting is one of the basic problems in programming. Tremendous amounts of data that require to be analyzed and processed are produced daily. Sorting can facilitate and speed up the process of data analysis and processing. A large number of applications use sorting algorithms to solve a particular problem in the fastest and easiest way. Sorted data are simpler to work with and enable users better data display as well as faster search. Dozens of algorithms that solve this problem have been written to date. Most of these algorithms work very quickly for small amounts of data, but if the data amount is large, the performing time of the algorithm can vary drastically. Therefore, one ought to choose the best possible algorithm that will provide the best and fastest solution to the problem. The sorting algorithm can be used to sort both simple (integers, decimal numbers, characters) and more complex data types (objects). Data sorting order can be ascending or descending.

Modern web, mobile and desktop applications use some of the existing sorting algorithms. Sorting problem is present in variety of domains in the modern computer science as well. Because of that, sorting algorithms are one of the basic algorithms in programming.

Correspondence:

Milan Savić

e-mail:

milan.savic.16@singimail.rs

DOI: 10.15308/Sinteza-2021-328-332

There is always the need to find new algorithms, even if they work for just a specific input data (such as positive integers), because even slightest improvement in the performances could lead to reducing the costs drastically.

This paper is organized as follows. Section 2 provides a short overview of already existing solutions. Section 3 proposes the new algorithm, while section 4 yields comparison with other approaches. Finally, section 5 concludes the paper.

## 2. EXISTING ALGORITHMS FOR SOLVING SORTING PROBLEMS

Several algorithms that solve the sorting problem have been written to date. Algorithms use various techniques to come up with solutions [1] [2]. Some algorithms come to the solution by an iterative procedure, others by recursive, as well as by using different types of data. Some of the basic algorithms are bubble sort, selection sort, insert sort, merge sort, quick sort.

Bubble sort is a simple sorting algorithm that works by comparing two adjacent elements of an array and changing their places if they are in the wrong order [3]. After each pass through the array, one element, the largest or the smallest, takes its position in the sorted array. The average time complexity of the algorithm is O($n^2$). Therefore, it belongs to the group of slower algorithms [4].

Selection sort is a more efficient algorithm than bubble sort, although the average time complexity is the same as with bubble sort O($n^2$) [5]. The idea of the algorithm is to find the smallest element after one pass through the array and place it in the proper position. After the first iteration, the smallest element of the array will be in the first place, after the second iteration the second smallest element of the array will be in the second place and so on. After the *i*-th iteration, the i smallest elements of the array will be sorted in ascending order.

Insert sort is practically a more efficient algorithm than the previously mentioned ones. Its average time complexity is O($n^2$). Sorting is done by adding a certain element of the array to the previously sorted part of the array [6]. Initially, only the first element belongs to the sorted part of the array and after that, the other elements are added to that part of the array at the positions so that the array remains sorted.

Merge sort was invented by John von Neumann in 1945. Unlike the aforementioned algorithms that use an iterative approach of passing through an array, merge sort uses a recursive method of passing through an array. The average time complexity of this algorithm is O($n$ log $n$ ) and it, therefore, belongs to the group of more efficient algorithms. This algorithm is much more effective than those previously mentioned. The idea of this algorithm is to divide the array which is to be sorted into two halves, i.e. two smaller arrays, which are further divided by a recursive call until an array of one element is obtained. After that, the smaller sorted parts are merged and sorted and thus a sorted initial array is obtained [7].

Quick sort was developed by Tony Hoare in 1959. The average time complexity of this algorithm is O($n$ log $n$ ). Although in the worst case the algorithm is no better than slow, square, sorting methods, in practical applications it has proven to be amongst the most efficient ones. The idea is to choose one element, called a pivot, based on which the other elements will be sorted. At the end of one pass through the array, the exact position of the pivot element in the sorted array is found and the array is divided into two parts, into elements smaller and larger than the pivot element, which are further sorted by a recursive call [8].

## 3. IMPLEMENTATION OF THE PROPOSED ALGORITHM

Unlike the previously mentioned algorithms, the proposed algorithm performs sorting based on the binary presentation of numbers. The paper presents an algorithm that sorts non-negative integers in ascending order. The algorithm solves the problem using recursion.

The idea of the algorithm is to regard each number as a binary number and to perform sorting based on the value of a certain bit.

The idea of the algorithm will be presented below.

Let unsorted array integers 9,6,6,1,8,9,9,7,4,3 be given. Each of these numbers can be represented in binary notation (Figure 1).

| | 9 | 6 | 6 | 1 | 8 | 9 | 9 | 7 | 4 | 3 |
|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 2 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |

Figure 1 - Unsorted array

First, sorting is performed based on the value of the highest bit using two pointers. One pointer points to the first element of the array (lowPointer), while the other points to the last (highPointer). If the value of the highest bit pointed to by the lowPointer is equal to 0 (zero), the pointer moves to the next element of the array (lowPointer++). If the value of the highest bit pointed to by the highPointer is equal to 1 (one), the pointer moves to the previous element of the array (highPointer--). If the value of the highest bit pointed to by the lowPointer is equal to 1, and the value of the highest bit pointed to by the highPointer is equal to 0, then the replacement of the value of the elements pointed to by these two pointers is performed. The lowPointer and highPointer continue to move through the array until the highPointer points to an element with a smaller index than the lowPointer.

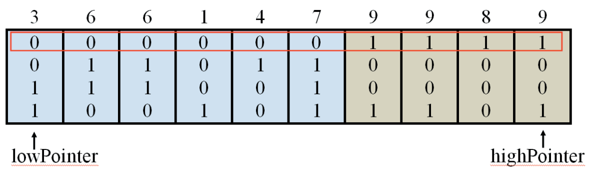

Figure 2 - The arrangement of elements after one pass through the array

After one pass through the array, the arrangement of elements as in Figure 2 is obtained. The grouping of elements based on the value of the bit at position 3 can clearly be seen. Elements starting with 0 are grouped to the left, while elements starting with 1 are grouped to the right.

The next step is sorting the left and then the right part of the array. This sorting is performed based on the value of the bit at position number 2. The same method of sorting using two pointers is used. After the elements are grouped based on the value of the bit at position 2, the sorting is moved to the lower bit positions until the bits at the lowest position are reached (Figure 3).

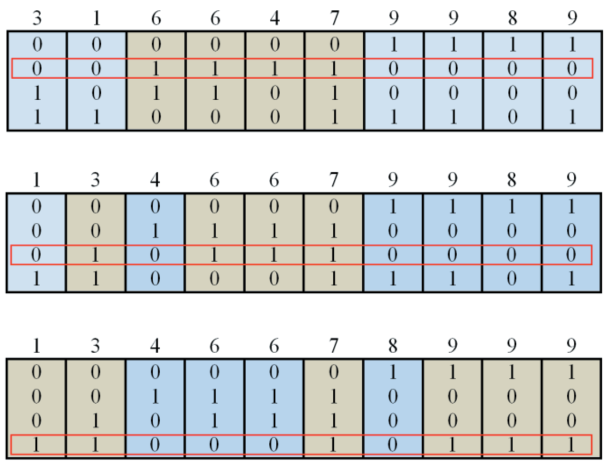

Figure 3 - Other steps for sorting an array

The pseudocode of this algorithm is given below (Listing 1). The *bitSort* algorithm has 4 parameters: the array to be sorted (*arr*), the low pointer (low), the high pointer (high), the bit position based on which the sorting is performed (*bitPosition*). The algorithm uses the *getBit* helper function for returning the bit value at a certain position (*bitPosition*) of a certain number (*number*).

```
getBit(number, bitPosition){
  return (number>>bitPosition)&0b1;
}

bitSort(arr[], low, high, bitPosition){
  if(low==high or bitPosition<0){
    return;
  }

  left = low;
  right = high;

  while(low <= high){
    if(getBit(arr[low],bitPosition)==0){
      low++;
    }

    while(low<=high and
          getBit(arr[high],bitPosition)==1){
      high--;
    }

    if(low<=high){
      swap arr[low] and arr[high];
      low++;
      high--;
    }
  }

  if(high>=0){
    bitSort(arr,left,high,bitPosition-1);
  }

  if(low<=right){
    bitSort(arr,low,right,bitPosition-1);
  }
}
```

Listing 1 - Pseudocode of the proposed algorithm

The value for the *bitPosition* parameter can be predetermined depending on the length of the binary record of the largest number in the array.

The time complexity is O(*kn*) and it can therefore be classified into a group of faster-sorting algorithms. The constant *k* represents the length of the binary record of the largest element of the array, while *n* represents the length of the array.

The advantage of this algorithm is that the recursion depth is limited by the number passed through the *bitPosition* parameter during the first function call.

## 4. COMPARISON WITH EXISTING ALGORITHMS

In this chapter, comparative tests of MergeSort, QuickSort and the proposed BitSort algorithms will be presented. All examples were tested in the Java programming language on a PC (Intel i5-3337U 1.8GHz).

Figure 4 shows how the performance time of the algorithm changes depending on the change in the range of values of the elements of the array. The example uses an array of 1,000,000 elements.

While the range of values of the elements of the array is [0, 1000000), [0, 100000), [0,10000), [0, 1000). The performance time shown is relative and depends on the performance platform.

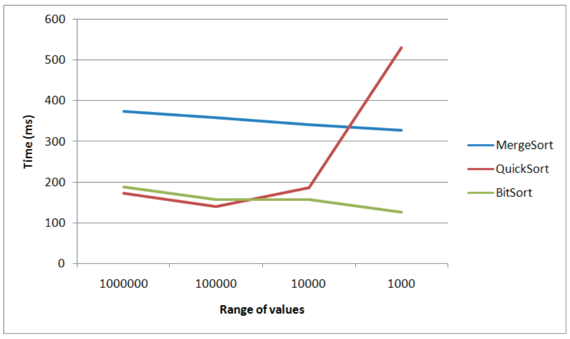

Figure 4 - Performance time depending on the range of values

The graph shows that the performance time of the BitSort algorithm is approximately the performance time of the QuickSort algorithm for array elements whose range is [0, 1000000) and [0, 100000), while it is much faster for a smaller range of values.

Figure 5 shows the performance time of array sorting algorithms depending on the number of array elements for array elements whose range is [0, 100000). The performance time shown is relative and depends on the performance platform.

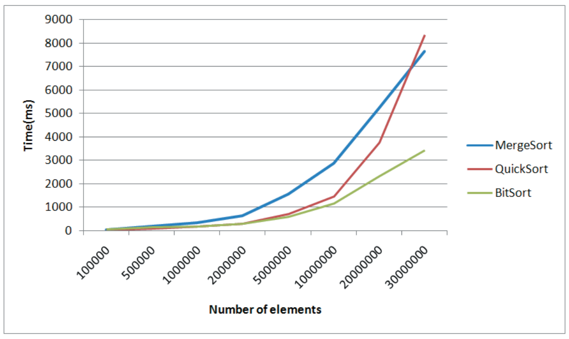

Figure 5 - Performance time depending on the number of array elements

For small amounts of data, all three algorithms perform relatively quickly. As the amount of data increases, it can be seen that the BitSort algorithm is faster than the MergeSort and QuickSort algorithms.

Figure 6 shows the number of checked array elements depending on the range of values of the elements of the array for a constant array length of 1,000,000 elements.

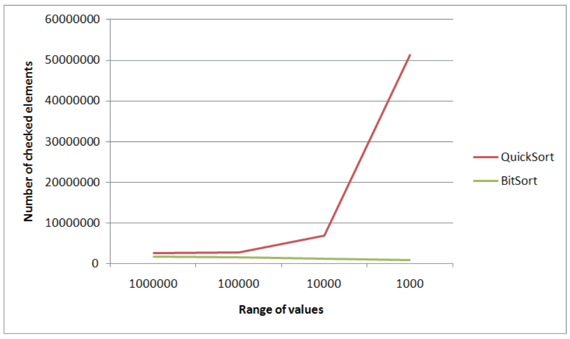

Figure 6 - Number of checked elements depending on the range of value

The graph shows that the BitSort algorithm uses a smaller number of array element approaches during sorting.

Figure 7 shows the number of checked array elements depending on the number of array elements for a constant range of element values [0, 100000).

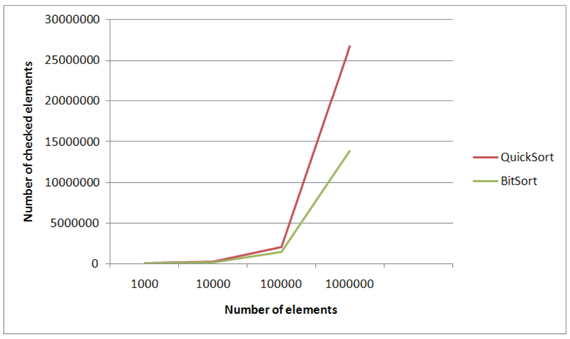

Figure 7 - Number of checked elements depending on the number of array elements

The graph shows the BitSort algorithm checking a smaller number of array elements than the QuickSort algorithm.

## 5. CONCLUSION

This paper presents a sorting algorithm that sorts non-negative integers based on a binary representation of an array element. Based on the time complexity of the algorithm, as well as comparative tests with already existing algorithms, it is shown that the algorithm belongs to the group of faster-performing algorithms. The proposed algorithm performs relatively quickly for large amounts of data and can therefore be used in today's world where huge amounts of data are produced daily. The advantage of the algorithm is that the number of recursive function calls is known in advance.

Since the algorithm sorts only non-negative integers, it is possible to continue with the further development of this algorithm. The next steps in the development of this algorithm are sorting negative integers, as well as other data types. Since the algorithm uses binary data presentation, it is possible to lower the algorithm to a level as close as possible to the hardware and thus further reduce the performance time. One can use this way of developing an algorithm, using binary data presentation, on other algorithms and thus attempt to improve the existing ones or write new ones.

## REFERENCES

[1] J. Anderson and S. M., "Sequential coding algorithms: A survey and cost analysis," *IEEE Transactions on communications*, vol. 32, no. 2, pp. 169-176, 1984.

[2] P. Vitanyi, "Analysis of sorting algorithms by kolmogorov complexity (a survey)," *Entropy, Search, Complexity*, pp. 209-232, 2007.

[3] O. Astrachan, "Bubble sort: an archaeological algorithmic analysis," *ACM Sigcse Bulletin*, vol. 35, no. 1, pp. 1-5, 2003.

[4] W. Min, "Analysis on bubble sort algorithm optimization," in *2010 International forum on information technology and applications,* 2010.

[5] R. Edjlal, A. Edjlal and T. Moradi, "A sort implementation comparing with bubble sort and selection sort," in *2011 3rd International Conference on Computer Research and Development*, 2011.

[6] I. Beck and S. Krogdahl, " A select and insert sorting algorithm," *BIT Numerical Mathematics*, vol. 28, no. 4, pp. 725-735, 1988.

[7] C. Bron, "Merge sort algorithm [m1]," *Communications of the ACM*, vol. 15, no. 5, pp. 357-358, 1972.

[8] R. Sedgewick, "Implementing quicksort programs," *Communications of the ACM*, vol. 21, no. 10, pp. 847-857, 1978.