ALGORITHMS AND SOFTWARE

UDC 004.424.5 https://doi.org/10.15407/vidbir2019.47.073

ACCELERATION OF LARGE INTEGER ARRAYS SORTING USING RANGES OF VALUES AND FREQUENCIES OF ELEMENTS

A. V. Shportko[1], L. V. Shportko[2]

[1]Academician Stepan Demianchuk International University of Economics and Humanities, Rivne;

[2]Rivne College of Economics and Business, Rivne

E-mail: ITShportko@ukr.net

The speed of sorting of various types of large integer arrays by popular algorithms of different groups of methods is analyzed. As a result, the algorithm of rapid exchange sorting with partitioning and moving of the supporting element QuickSort has again been recognized as the most effective. An alternative algorithm for partitioning of the QuickSort method in relation to the half the range of values of the elements of each submachine, which makes it possible to use sorting by narrow ranges, is given. It is shown that subarrays with the length of up to 10 elements should be sorted by simple inserts. For longer subarrays, if the range of values does not exceed its size, it is advisable to use the sorting method by counting. Otherwise, it is necessary to perform the iteration of the QuickSort method with a split in half the range of values of its elements. It has been experimentally proved that such a complex use of methods accelerates, for example, sorting of arrays with 1 million elements and with the range of values no more than 1000 by 34%. The results obtained can be used to accelerate the sorting of large integer arrays with the known or defined ranges of values.

Keywords: *sorting of arrays, QuickSort sorting method, range of array elements values, sorting by counting.*

ПРИСКОРЕННЯ СОРТУВАННЯ ВЕЛИКИХ ЦІЛОЧИСЛОВИХ МАСИВІВ ЗІ ЗАСТОСУВАННЯМ ДІАПАЗОНІВ ЗНАЧЕНЬ ТА ЧАСТОТ ЕЛЕМЕНТІВ

О. В. Шпортько[1], Л. В. Шпортько[2]

[1]ПВНЗ “Міжнародний економіко-гуманітарний університет імені академіка Степана Дем’янчука”, Рівне;

[2]ДВНЗ “Рівненський коледж економіки та бізнесу”, Рівне

Проаналізовано швидкість внутрішнього сортування різнотипних великих цілочислових масивів популярними на сьогодні алгоритмами з різних груп методів. Найефективнішим серед проаналізованих вкотре визнано алгоритм методу швидкого обмінного сортування QuickSort, в якому розбиття виконується відносно випадковим чином обраного переміщуваного опорного елемента. Наведено альтернативний алгоритм розбиття для методу QuickSort відносно половини діапазону значень елементів кожного підмасиву, який дає змогу за вузьких діапазонів застосовувати метод сортування підрахунком. Показано, що цей метод має лінійну обчислювальну складність для розміру сортованого підмасиву та діапазону значень його елементів, але його застосування допустиме лише тоді, коли розмір вказаного діапазону значень не перевищує розміру масиву частот. Тому розширення можливостей використання методу сортування підрахунком і відповідно прискорення сортування загалом, можливе через збільшення розміру масиву частот.

Згідно з аналізом обчислювальної складності, показано, що підмасиви з довжиною до 10 елементів включно доцільно сортувати методом простих вставок. Для довших підмасивів, коли діапазон їх значень не перевищує кількості елементів та розміру масиву частот, доцільно використати метод сортування підрахунком, а інакше варто виконати ітерацію методу QuickSort з розбиттям навпіл діапазону значень елементів підмасиву.

Експериментально доведено, що таке комплексне використання зазначених методів прискорює, наприклад, сортування масивів з діапазонами значень <1000 розміром 1 млн. елементів на 34%. Отримані результати можна використати для прискорення сортування та обчислення порядкових статистик (наприклад, медіан) великих цілочислових масивів з відомими чи визначеними діапазонами значень.

Ключові слова: *сортування масивів, метод сортування QuickSort, діапазон значень елементів масиву, сортування підрахунком.*

Currently, most of the information stored on electromagnetic media is provided in digitized discrete form. At the same time, for the fast processing of data of the same type, arrays, placed in RAM are used. Sorting of these arrays allows us to perform binary search instead of the linear one and to accelerate the further processing of elements. That's why improving the methods and algorithms of sorting is the up-to-date task and it will remain the urgent task in the future.

Imagine the situation when it is needed to sort the array $A(a_0, a_1, \ldots, a_{N-1})$, where $N \geq 1000$. For arrays, the concepts *of record* and *key* in the context of this paper actually coincide, so in the future, under the *element* of the array, we will keep in mind both these two concepts at the same time.

The analysis of the last researches and publications. As of today all the general methods of practical internal sorting (sorting data located in RAM) the principle of functioning are divided into five groups [1]:

– *sorting by insertions*, when a duty element is pasted in the already sorted subsequence;

– *sorting by exchange*, when that elements are moved at the large distances;

– *sorting by choice*, during which among the non-sorted part the maximum item is selected each time and is written in the next position, for example;

– *merge sort*, the main idea of which is to combine sequentially each time the further sorted sequences of the elements;

– *sorting by a distribution*, in which the elements are not directly compared with each other but divided into different groups (for example, by the highest bit), after that sorting is performed within each group separately and the results are consistently combined with each other.

For sorting of large arrays the most effective algorithms of such methods are the following [1, p. 148]:

– *pyramidal sorting HeapSort* [2], which initially arranges elements of the array by the principle $a_m \geq a_{2m+1}; a_m \geq a_{2m+2}$, thus moving the largest value of the array into an element a_0 and forming a pyramid of subordination. Then it sequentially removes the value from this element (chooses the largest value), adjusts the placement of the remaining elements of the pyramid, so that the principle of ordering is preserved, and records the deleted value at the end of the unassembled part. This method belongs to the "Sort by choice" group;

– *sorting by insertions ShellSort* [3], according to which the remote elements of the array are sorted by inclusions in the separate subarrays, and the displacement between the elements for the formation of submachine sequentially decreases to unity. This method belongs to the "Sorting by insertions" group;

– *a quick exchange sorting with partition of QuickSort* [4], according to which at each iteration a reference element is selected and values not larger than this element are placed to the left of it. Elements that are larger or equal are placed to the right. After these operations the elements of the left and right parts are sorted in subsequent iterations of the algorithm separately. This method belongs to the "Sort by exchange" group.

The algorithms of these methods have the average calculative complexity of the order $O(N \log_2 N)$ [5], although the speed of their work depends on the placement of

the array elements. In addition, the efficiency of the sorting algorithms also significantly depends on the amount of used additional memory and implementation details. Let's see, for example, how to select a reference element for the QuickSort method. If the value of the reference element is located approximately in the middle of the values of the fragment of the array for sorting, then the recursion depth will be around $[\log_2 N]$ and the total computational difficulty will actually approach to $O(N \log_2 N)$, since for the next division, it is necessary to compare each element of the array with the supporting element of its fragment. But if at each iteration the reference element is minimal/maximal for its fragment, then in its right/left part after the move of all its other elements will fall and the length of the fragment for the next iteration will be reduced to only one element. In this case the depth of the iterations will be N–1, and therefore the total computational complexity of the algorithm will be closer to $O(N^2)$. The option of selecting such a supporting element is not with such a little probability as it is specified in [6], especially when the values of this supporting element are taken from the first or the last element of the fragment, and the array itself is already pre-sorted. The importance of choosing the correct supporting element was emphasized by the author of QuickSort C. A. R. Hoare in [4]. In this work he proposed two options for forming the value of the reference element: either set it to be equal to the median subset of the elements (for example, from the first, the last and the middle element of the fragment), or to select it randomly among the elements of the fragment. If for the first variant of the formation it is still possible to pick up the elements of the array so that the computational complexity of the sort is approaching $O(N^2)$, then for the second option it is almost unrealistic [7]. So, in practice it is expedient to select the reference element in a random manner.

On the other hand, in the QuickSort implementation, it is not expedient to break short subarrays (in our implementations up to ten elements inclusive) iteratively, but it is better to use "straightforward" sorting methods, for example, simple inserts, as it is specified in [1]. Also, modern partitioning schemes of this method, in contrast to the classical scheme of Hoare [7], always move the supporting element every time to the edge of the subarray, that avoids additional indexes comparisons [8, p. 92–93]. We applied all these modifications and schemes in the process of practical implementation of the QuickSort method.

So, **the purpose of this paper** is to study the speed of sorting large integer arrays of different types by the algorithms of above-mentioned methods, selecting among them the fastest average algorithm and its improvement by using a range of values and frequencies of the array elements.

Analysis of the speed of sorting large integer arrays by the algorithms of different methods. To verify the efficiency of the above-mentioned algorithms of sorting methods, we implement the programs presented in [8, p. 89–101] in the Microsoft Visual Studio program in C # programming language [9]. The choice of this programming language is due to its popularity today for the development of applications. For the QuickSort method, we have developed two variants of algorithms with the choice of the leading element at random: with the implementation of the partition of the Hoare scheme (*QuickSort 1*) and with moving of the reference element (*QuickSort 2*). Testing was performed on a computer with an Intel Pentium 4 processor with the clock speed of 3 GHz and 4 Gb of RAM. Test results are shown in Table. 1. The same table shows the sorting time of arrays by the standard Array.Sort() method.

Table 1. The duration of sorting in ascending order of variants of integer array with 1 million elements by the algorithms of different methods, ms

Algorithm of sorting method	Variant of array			
	Generated randomly	Sorted in ascending order	Sorted in descending order	From equal elements
QuickSort 1	**427**	207	193	242
QuickSort 2	**383**	**141**	**156**	**250**
HeapSort	**921**	633	625	109
ShellSort	**1133**	164	313	164
Array.Sort	**219**	31	94	109

We see that among the explored algorithms, the sorting of large integer arrays is performed on the average quickly by the QuickSort algorithm with the movement of the supporting element. It is 19.7% faster than the QuickSort algorithm with the Hoare scheme and yields to it only in the arrays of identical elements, and is on average faster by 183% of the HeapSort algorithm and by 69.6% of the ShellSort algorithm. In our opinion, the shortfall of QuickSort from the standard sorting method Array.Sort is due to the use of the references in it instead of the index of elements, which reduces the number of checks for out of range. It is also interesting that the pyramid sorting for random-generated arrays turned out to be faster than ShellSort by an average of 23%, as predicted in [1, p. 159]. Therefore, in this paper we will explore the possibilities of improving the QuickSort algorithm.

Implementation of the QuickSort method by partitioning the range of array elements values. The idea of this method is to split the subarray elements relative to the half range of its values and in the subsequent iterative sorting of each of the two received parts. An opinion on the use of a breakdown of the value *close* to the middle of the range was expressed by John McCarthy [1, p. 128]. We recommend to execute split *self* in relation to this middle value.

Let $minA \leftarrow \min(a_0, a_1, ..., a_{N-1})$, $\max A \leftarrow \max(a_0, a_1, ..., a_{N-1})$ are respectively, the minimum and maximum values of elements of the array. We also set the value of the constant $Q \leftarrow 11$, where Q is the minimum length of the subarray, for which the QuickSort iterative call is already used, rather than the method of sorting by simple inserts [1]. Then, the QuickSort sorting algorithm that uses the partition of values is sequentially written as follows:

1. Execution of the initial initializing, setting $i \leftarrow 0$, $j \leftarrow N-1$ (sorting limits) and $minElement \leftarrow minA$, $maxElement \leftarrow maxA$ (range of values of elements).
2. If $minElement = maxElement$, we complete an iteration (as all elements of subarray are equal).
3. Calculation of $d \leftarrow maxElement - minElement + 1$, $l \leftarrow j - i + 1$, where d is a range of values of elements of subarray, l is an amount of elements of subarray. If $l < Q$, then we sort a subarray by the method of simple insertions and complete the iteration.
4. Calculation of $element \leftarrow \lfloor (minElement + maxElement)/2 \rfloor$. This is the middle of the range of values of elements. Like splitting of Hoare, we need to rearrange the elements of subarray $a_i, a_{i+1}, ..., a_{j-1}, a_j$ so that the elements belong to the ranges of values $a_m \in [minElement; element]$, $m = \overline{i, k}$; $a_m \in [element + 1; maxElement]$, $m = \overline{k+1, j}$. We need to break up elements in relation to the middle of the range of values.

5. Assigning $j \leftarrow k$, $maxElement \leftarrow element$ and iteratively sort the "left" subarray, passing to step 2.
6. Restoring the previous values of the variables and assigning them $i \leftarrow k+1$, $\min Element \leftarrow element+1$ and iteratively sorting the "right" subarray, passing to step 2.

Finding the maximum and minimum values of the array of elements is not included in this algorithm, since these values can be known to the user in advance and so it is not expedient to spend time on their calculations. For example, in our experiments for an array of millions of elements, the duration of computing these values on the average took 11 ms. In contrast to the classic QuickSort algorithm, the presented algorithm does not remove the reference element from subsequent consideration at each step of the iteration, but the range of values of the elements is always split equally, so its computational complexity would not exceed $O(N \log_2(maxA - minA + 1))$. So, **the QuickSort method with a breakdown of the range of values should be used when this range does not exceed the number of elements of the array.**

Adaptation of sorting method by counting to subarrays of QuickSort method. The sorting method by counting allows us to arrange integer elements using the additional array of frequencies [1, p. 80]. To sort the array in ascending order, this method can be briefly paraphrased as follows:
1. Calculate the frequencies of the values of the elements of the array;
2. Consistently selecting the values of the elements in ascending order, repeate in each resultant array each value as many times as it is encountered in the inputting array.

It is clear that application of this method to sorting the subarray is possible only when the range of values of its elements does not exceed the size of the array of frequencies. In addition, we will apply this method for sorting subarrays of the QuickSort method only if the same range of values does not exceed the number of its elements. The C # language code snippet for sorting of such subarrays is the following:

```
if (d <= l && d <= countFreq)
 {for (m = 0; m < d; m++) //reset frequencies of values
   freq[m]=0;
  for (m = i; m <= j; m++) //count up frequencies of values
   freq[a[m]-minElement]++;
  index = i;
  for (m = 0; m < d; m++) //sort possible values
   for (c = 0; c < freq[m]; c++) //repeat the frequency value
    a[index++] = minElement + m;
  continue; //complete sorting of subarray
 }
```

It is interesting that this *method does not compare the elements with each* other directly and thus it is similar to the methods of the "Sorting by distribution" group, but in the process of its work $2l + 2d$ comparisons of indexes and l appropriations are performed. The method of simple insertion performs on average $l(l-1)/4$ comparisons and the same number of assignments. Therefore, use of the method by counting instead of a simple insertion method is appropriate when $l(l-1)/2 \geq 3l + 2d$. If $l \geq d$, this inequality will always be true at $l \geq 11$ (this is what determines the value of the constant Q). Comparing the computational complexity of the sorting method by counting with the classic QuickSort method, we find that if $l \geq d$ and $l \geq 11$ it is expedient to use the first from these methods of sorting, and at less l – the second one.

Results of experiments. Let's match the data in Table 1 and Table 2, which present the results of testing the QuickSort algorithms with a breakdown of the range of values (*QuickSort 3*) and the same breakdown and implementation of the counting method (*QuickSort 4*), which use a frequency array of 100 elements.

Table 2. The duration of sorting of variants of integer arrays with 1 million elements in ascending order by algorithms of the QuickSort method with a breakdown of the range of values, ms

Algorithm of sorting	Variant of array			
	Generated randomly	Sorted in ascending order	Sorted in descending order	From equal elements
QuickSort 3	**429**	149	156	16
QuickSort 4	**429**	**133**	**133**	**16**

Let's also consider the length of sorting by the given modifications of the QuickSort algorithm of integer arrays with different ranges of elements values (Table 3). Note that the efficiency of the sorting method by counting also depends on the capacity of the array of frequencies: if, for example, this array is increased to 10^4 elements, the array sorting with a range of values 10^5 will last 117 milliseconds. And if we increase the number of elements to 10^5, the sorting process will take just 39 milliseconds.

Table 3. The duration of sorting of integer arrays with 1 million randomly generated elements in ascending order by different modifications of the QuickSort algorithm, with different ranges of element values, ms

The modification of QuickSort algorithm	Width of the range of values of elements						
	10^9	10^7	10^5	10^4	10^3	10^1	10^0
QuickSort 2 (with moving of supporting element)	383	383	365	343	313	273	269
QuickSort 3 (with splitting of the range of values)	430	430	364	281	211	86	16
QuickSort 4 (with splitting of the range of values and sorting a count)	430	430	234	**172**	**109**	**31**	**16**
Array.Sort	203	203	195	**180**	**164**	**132**	**109**

CONCLUSIONS

In order to sort integer arrays, in most cases it is sufficient to use the standard method of the programming language.

If you only need to get a fragment of a sorted array (for example, its median), you should use the QuickSort algorithm to move the support element.

In cases where the width of the range of values of the array of elements is known and this width does not exceed 10^5, in order to accelerate its sorting, it is expedient to use the QuickSort method with a breakdown of the range of values with the methods of sorting by counting and simple insertion. In this case, if the length of the subarray does not exceed 10 elements it should be sorted by simple inserts, otherwise, if the range of values does not exceed the size of the subarray, you have to use the sorting method by counting. If these two conditions are not fulfilled, it is advisable to perform the iteration of the QuickSort method with a split in half the range of values of the elements.

In the future, we plan to use methods of splitting the range of values and sorting by counting in order to improve the algorithms that use sorting, in particular, the median search algorithms and general *i*-th of index statistics.

1. Knuth, D. E. *The Art of Computer Programming. Sorting and Searching*, 2rd ed.; Massachusetts: Addison Wesley Longman, **1997**; 3; p 791.
2. Williams, J. W. Algorithm 232 – Heapsort. *Communications of the ACM* **1964**, 7 (6), 347–348.
3. Shell, D. L. A high-speed sorting procedure. *Communications of the ACM* **1959**, 2 (7), 30–32.
4. Hoare, C. A. R. Quicksort. *The Computer Journal* **1962**, 5(1), 10–16.
5. Cormen, T. H, Leiserson, C. E., Rivest, R. L., Stein, C. *Introduction to Algorithms*, 3rd ed.; Moscow: Williams, **2013**; pp 1328. (in Russian).
6. Quick sort http://algolist.manual.ru/sort/quick sort.php (accessed by Jun 05, 2019). (in Russian).
7. Quicksort – Wikipedia https://en.wikipedia.org/wiki/Quicksort (accessed by Jun 05, 2019).
8. Vlasyuk, A. P. *Practical Programming Work in the Environment of Turbo Pascal*; Rivne: NUWEE, 2005; Part.1; p 179. (in Ukrainian).
9. *C# Language Specification. Standard ECMA-334*, 5_{th} ed.; ECMA International, 2017.

Received 17.05.2019