Bidirectional Bubble Sort Approach to Improving the Performance of Introsort in the Worst Case for Large Input Size

Oyelami Olufemi Moses olufemioyelami@gmail.com
College of Science and Technology/
Department of Computer and Information Sciences
Covenant University
Ota, Post code, Nigeria

Abstract

Quicksort has been described as the best practical choice for sorting. It is faster than many algorithms for sorting on most inputs and remarkably efficient on the average. However, it is not efficient in the worst case scenarios as it takes $O(n^2)$. Research efforts have been made to enhance this algorithm for the worst case scenarios by improving the way the algorithm chooses its pivot element for partitioning, but these approaches have the disadvantage of increasing the algorithm's average computing time. Introsort was, however, developed to overcome this limitation. This paper presents an approach that uses Bidirectional Bubble Sort to improve the performance of Introsort. Instead of using Insertion Sort as the last step of the sorting algorithm for small lists, the approach uses Bidirectional Bubble Sort. The results of the implementation and experimentation of this algorithm compared with Introsort shows its better performance in the worst case scenario as the size of the list increases.

Keywords: Quicksort, Introsort, Bidirectional Bubble Sort, Worst Case, Improved Introsort.

1. INTRODUCTION

Among the sorting algorithms that are not difficult to implement is Quicksort. "The algorithm works well for a variety of input data and consumes fewer resources than any other sorting method in many situations" [1]. The algorithm is also an in-place sorting algorithm. "It is the fastest known generic algorithm in practice" [2]. Its worst-case running time is, however, $O(n^2)$ on an input array of **n** numbers. "In spite of this slow worst-case running time, Quicksort is often the best practical choice for sorting because it is remarkably efficient on the average" [3, 4]. Research effort has, however, been made to improve the algorithm to eliminate its drawback in the worst case scenario. Introspective sorting, otherwise called Introsort, is a modified and improved Quicksort that is self-aware. Through its self-awareness it is able to solve the problem of inefficiency of Quicksort for the worst case scenario. This paper presents an approach to further enhance the performance of Introsort in the worst case scenario. The approach uses Bidirectional Bubble Sort in place of Insertion Sort employed by Introsort for small lists.

2. SORTING ALGORITHMS

Given a list of input elements or objects, sorting arranges the elements either in ascending order or descending order and produces a sorted list as the output. The elements to be sorted need to be stored in a data structure for manipulation. Among the various data structures usually used for sorting are: arrays, linked list, heap, etc. Sorting can either be internal or external. "Internal sorting is the type of sorting that requires all the elements to be sorted to be in the main memory throughout the sorting process while an external sorting allows part of the elements to be sorted to be outside the main memory during the sorting process" [5]. Examples of internal sorting algorithms are: Insertion Sort, Selection Sort, Bubble Sort, Shellsort, Quicksort, etc.

2.1 Quicksort

Quicksort uses divide and conquer approach to divide a list of elements to be sorted into two sub-lists. It chooses an element and then splits the whole list into two halves consisting of the elements smaller and the elements greater than the selected one. The same procedure is then applied to each half.

2.1.1 Improvements on Quicksort

There have been several successful attempts at improving quicksort:

2.1.1.1 Median-of-Three Rule

Unlike quicksort that picks the first element as the pivot value, the Median-of-Three version selects the median of the first, middle and the last elements in each sub-list. This approach increases the performance of quicksort when the list of elements to be sorted is already or partially sorted [1, 6].

2.1.1.2 Small Sub-lists

Quicksort is not efficient when the size of the elements to be sorted is less or equal to 20. The sub-list approach uses an efficient sort like Insertion Sort in this situation. Alternatively, small sub-lists may be ignored, and upon termination of Quicksort, the list will just be slightly unsorted. Insertion sort can then be applied [1].

2.1.1.3 Improved Median-of-Three Sort for the Average Case

"The proposed approach for improving Median-of-Three Sort for average case scenarios first of all divides the elements to be sorted into sub-sequences just like Shell Sort does, but by first of all comparing the first element with the last. If the last is less than the first, the two swap positions, otherwise, they maintain their positions. Later, the second element is compared with the second to the last, if the second to the last element is smaller than the second, they are swapped. Otherwise, they maintain their positions. This process continues until the last two consecutive middle elements are compared, or until it remains only one element in the middle"[7,8,9].

2.1.1.4 Introsort

"Introspective sort otherwise referred to as Introsort is a comparison sorting algorithm invented by David Musser in 1997" [10]. It starts with Quicksort, but switches to Heapsort if the depth of the recursion is too deep to eliminate the worst-case, and uses Insertion Sort for small cases because of its good locality of reference. "Introsort brings the introspective element into play by monitoring the recursion depth the algorithm reaches as a function of the length of the array of data. Since the recursion depths for the best and worst cases runtime are known, a reasonable value in-between can be calculated dynamically. This value acts as a threshold and once it is exceeded, Introsort detects that the Quicksort algorithm it uses degenerates to quadratic behaviour. The reaction to this is changing the sorting algorithm for the current sub-array of data" [11]. "The algorithm is the best of both average and worst cases worlds, with a worst-case and average case O(n log n) runtimes and practical performance comparable to Quicksort on typical data sets. The algorithm is presented in FIGURE 1 below. The test **p - f ≥ b - p** is to ensure that the recursive call is on a subsequence of length no more than half of the input sequence so that the stack depth is O(log N) rather than O(N)" [10].

Algorithm Introsort(A, f, b)
Inputs: **A**, a random access data structure containing the sequence of data to be sorted in positions A[f], …, A[b-1];
f, the first position of the sequence
b, the first position beyond the end of the sequence
size, the number of data to be sorted equivalent to b-f
Output: **A** is permuted so that A[f]≤A[f+1] ≤ …≤A[b-1]
Introsort_Loop(A, f, b, 2*FLOOR_LG(b-f)
Insertion_Sort (A, f, b)

```
Algorithm Introsort_Loop (A, f, b, depth_limit)
        Inputs: A, f, b as in Introsort;
                depth_limit, a nonnegative integer
        Output: A is permuted so that A[i] ≤ A[j] for all i, j: f≤i≤j<b and size_threshold < j-i
                while  b-f > size_threshold
                        do if depth_limit =0
                                then Heapsort(A, f, b)
                                        return
                                depth_limit=depth_limit -1
                                P=Partition(A, f, b, Median_of_3(A[f], A[f+(b-f)/2], A[b-1]))
                                Introsort_Loop(A,p,b,depth_limit)
                                        b = p
```

FIGURE 1: Introsort [10].

2.2. Bidirectional Bubble Sort

"Bidirectional Bubble Sort also known as Cocktail Sort or Shaker Sort is a variation of Bubble Sort that is both a stable sorting algorithm and a comparison sort. The algorithm differs from Bubble Sort in that it sorts in both directions each pass through the list. The average number of comparisons is slightly reduced by this approach" [12]. "This sorting algorithm is just slightly more difficult than Bubble Sort to implement. It solves the problem with so-called turtles in Bubble Sort" [13]. FIGURE 2 below illustrates Bidirectional Bubble Sort for sorting the list: 8 4 3 2 in ascending order [13].

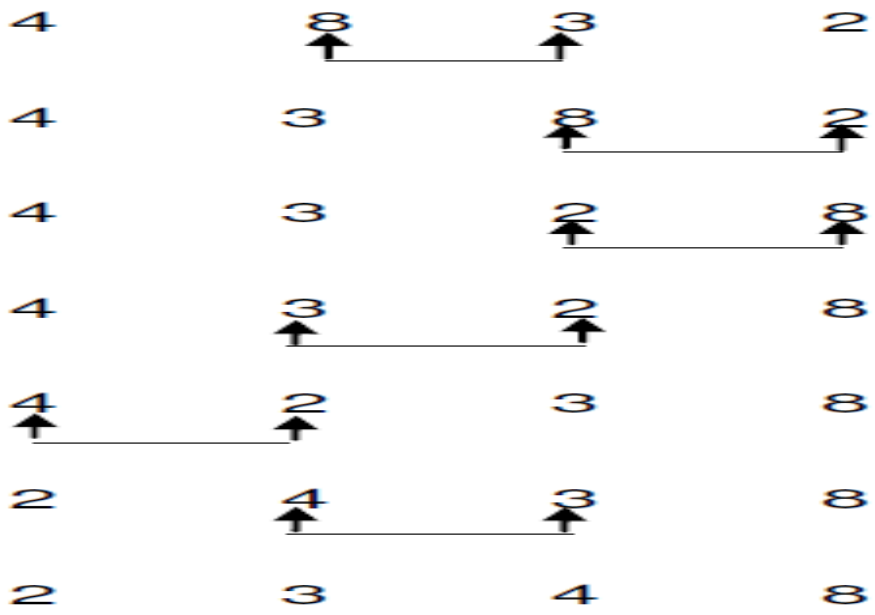

FIGURE 2: Bidirectional Bubble Sort.

2.3. Improved Introsort

The proposed algorithm (Improved Introsort) is shown in FIGURE 3 below. Call to **Insertion_Sort(A, f, b)** in FIGURE 1 has been replaced by a call to **Bidirectional Bubble Sort(A, size)**.

```
Algorithm Introsort(A, f, b)
        Inputs: A, a random access data structure containing the sequence of data to be
                sorted in positions A[f], …, A[b-1];
                f, the first position of the sequence
                b, the first position beyond the end of the sequence
                size, the number of data to be sorted equivalent to b-f
        Output: A is permuted so that A[f]≤A[f+1] ≤ …≤A[b-1]
    Introsort_Loop(A, f, b, 2*FLOOR_LG(b-f)
    Bidirectional Bubble Sort (A, size)
Algorithm Introsort_Loop (A, f, b, depth_limit)
    Inputs: A, f, b as in Introsort;
        depth_limit, a nonnegative integer
    Output: A is permuted so that A[i] ≤ A[j] for all i, j: f≤i≤j<b and size_threshold < j-i
        while  b-f > size_threshold
            do if depth_limit =0
                then Heapsort(A, f, b)
                    return
                depth_limit=depth_limit -1
                P=Partition(A, f, b, Median_of_3(A[f], A[f+(b-f)/2], A[b-1]))
                Introsort_Loop(A,p,b,depth_limit)
                    b = p
```

FIGURE 3: Improved Introsort.

3. PERFORMANCE ANALYSIS OF ALGORITHMS

"The most important attribute of a program/algorithm is correctness. An algorithm that does not give a correct output is useless. Correct algorithms may also be of little use. This often happens when the algorithm/program takes too much time than expected by the user to run or when it uses too much memory space than is available on the computer"[14]. "Performance of a program or an algorithm is the amount of time or computer memory needed to run the program/algorithm. Two methods are normally employed in analyzing an algorithm:

i.) Analytical method

ii.) Experimental method

In analytical method, the factors the time and space requirements of a program depend on are identified and their contributions are determined. But, since some of these factors are not known at the time the program is written, an accurate analysis of the time and space requirements cannot be made. Experimental method deals with actually performing experiment and measuring the space and time used by the program. Two manageable approaches to estimating run time are" [14]:

i.) Identify one or more key operations and determine the number of times they are performed.

ii.) Determine the total number of steps executed by the program.

3.1. Worst Case, Best Case and Average Case Analysis of Sorting Algorithms

"The worst-case occurs in a sorting algorithm when the elements to be sorted are in reverse order. The best-case occurs when the elements are already sorted. The average case may occur when part of the elements are already sorted. The average case has data randomly distributed in the list" [15]. "The average case may not be easy to determine in that it may not be apparent what constitutes an 'average' input. Concentration is always on finding only the worst-case running time for any input of size **n** due to the following reasons" [3]:

i.) The worst-case running time of an algorithm is an upper bound on the running time for any input. Knowing it gives us a guarantee that the algorithm will never take any longer. "We need not make some educated guess about the running time and hope that it never gets much worse".

ii.) For some algorithms, the worst-case occurs fairly often. For example, in searching a database for a particular piece of information, the searching algorithm's worst-case will often occur when the information is not present in the database. In some searching applications, searches for non-existent information may be frequent.

iii.) The average case is often roughly as bad as the worst case.

3.2. Analysis of Improved Introsort

"Generally, the running time of a sorting algorithm is proportional to the number of comparisons that the algorithm uses, to the number of times items are moved or exchanged, or both" [1]. Because both Introsort and Improved Introsort are comparison-based algorithms, the approach employed in their analysis was to compare the number of comparison operations carried out by each algorithm, as well as the number of swappings or exchanges carried out. The same sets of data were used for the same sizes of input to the algorithms. Also, the two algorithms were tested for data in reverse order resulting in the worst case scenario since concentration is always on finding the worst-case as justified earlier by Thomas, et al. [3]. The two algorithms were also tested for the best-case scenario when the data are already sorted.

4. RESULTS AND DISCUSSION

The results obtained from the experiments carried out when the programs were tested on a system running Windows 7 Ultimate using Bloodshed Dev-C++ 4.9.9.2 are presented in tables 1 and 2.

	Introsort				**Improved Introsort**			
Size	**Comparison**	**Swapping**	**Swapping-based Assignment Operations**	**Total Operations**	**Comparison**	**Swapping**	**Swapping-based Assignment Operations**	**Total Operations**
900	6,364	450	1,350	7,714	8,159	450	1,350	9,509
5,000	50,512	2,500	7,500	58,012	60,507	2,500	7,500	68,007
10,000	111,024	14,999	44,997	156,021	131,019	5,000	15,000	146,019
20,000	242,048	29,999	89,997	332,045	282,043	10,000	30,000	312,043
80,000	1,128,192	119,999	359,997	1,488,189	1,288,187	40,000	120,000	1,408,187

TABLE 1: Performance of Introsort and Improved Introsort in the Worst Case Scenario.

	Introsort		Improved Introsort	
Size	**Comparison**	**Swapping**	**Comparison**	**Swapping**
900	6,363	0	6,362	0
5,000	50,511	0	50,510	0
10,000	111,023	0	111,022	0
20,000	242,047	0	242,046	0
80,000	1,128,191	0	1,128,190	0

TABLE 2: Performance of Introsort and Improved Introsort in the Best Case Scenario.

TABLE 1 shows the performances of Introsort and Improved Introsort in the worst case scenario. For small input sizes, Introsort has a reduced number of comparisons while both sorting methods have the same number of swappings. This means that Introsort is more efficient when the input size is small. However, as the size of the input grows, Improved Introsort gets more efficient than Introsort by having a reduced number of swappings (although the comparisons are still higher than for Introsort). The reduced number of swappings could be attributed to the stability of Bidirectional Bubble Sort. This has resulted in its better performance as the input size grows. From the results presented in TABLE 1, one might be tempted to conclude that Introsort also performs better when the number of comparisons and swappings are added together as the input size increases, but because each swapping takes three assignment statements, the column labeled "**Swapping-based Assignment Operations**" is instead added to the column for comparison which gives the total number of operations. TABLE 2 shows the performance of the two algorithms in the best case scenario. In this scenario, Improved Introsort outperforms Introsort for all sizes of input. However, the difference in performance is marginal. The simulation results also show that Improved Introsort is especially efficient when the items to be sorted are in reverse order and as the size of the list to be sorted increases. In comparison with other enhancements of quicksort like Median-of-Three, Small Sub-list, Improved Median-of-Three and Introsort, Improved Introsort enhances the performance of quicksort in the worst case significantly better than Introsort and marginally better than Introsort in the best case scenario. Median-of-Three improves the performance of quicksort when the list to be sorted is partially sorted. Small Sub-list enhances the performance of quicksort when the list to be sorted is small and Improved Median-of-Three enhances the performance of quicksort for special types of average case scenario.

5. CONCLUSION

Quicksort has been identified to be a very good sorting technique that is very efficient on all classes of sorting problems. However, it is inefficient in the worst case situation. Introsort solves the problem of the inefficiency of Quicksort for the worst case scenario through the concept of introspection. This makes it a practical choice for all classes of sorting problems. In an attempt to beat the performance of Introsort for the worst case scenario, this paper presented an improved Introsort. The algorithm is efficient on inputs of large size. The different sorting methods have features that make them suitable for different classes of sorting problems and since it has been observed that "there is no known "best" way to sort; there are many best methods, depending on what is to be sorted, on what machine and for what purpose" [16], the algorithm presented in this paper is therefore recommended for sorting data when the size of the list to be sorted is large, especially from 10,000 upward.

6. FUTURE WORK

It is the intention of the author to implement the algorithm as a generic algorithm and measure its performance in the framework of C++ Standard Template Library.

7. ACKNOWLEDGMENT

The author would like to thank the writer of **A guide to Introsort (see:** http://debugjung.tistory.com/entry/intro-sortintrospective-sort-????.) whose implementation of Introsort in Java served as a guide in implementing Introsort in C++.

8. REFERENCES

[1] S. Robert. Algorithms in C. USA: Addison-Wesley, 1998; pp. 1- 4, 267, 303.

[2] A.W. Mark. Data Structures and Algorithm Analysis in C++. USA: Pearson Education. Inc., 2006, pp. 279.

[3] H.C. Thomas, E.L. Charles, L.R. Ronald and S. Clifford. Introduction to Algorithm. USA: The Massachusetts Institute of Technology, 2001, pp. 25-26, 145.

[4] A.D. Vladmir. Methods in Algorithmic Analysis. USA: CRC Press, 2010.

[5] P.B. Shola. Data Structures With Implementation in C and Pascal. Nigeria: Reflect Publishers, 2003, pp. 134.

[6] R.C. Singleton.(1969). "Algorithm 347 (An Efficient Algorithm for Sorting With Minimal Storage)". Communications of the ACM, vol. 12, pp. 187-195, 1969.

[7] M.O. Oyelami and I.O. Akinyemi. (2011, April). "Improving the Performance of Quicksort for Average Case Through a Modified Diminishing Increment Sorting." Journal of Computing, 3(1), pp. 193-197. Available: http://www.scribd.com/doc/54847050/Improving-the-Performance-of-Quicksort-for-Average-Case-Through-a-Modified-Diminishing-Increment-Sorting

[8] M. O. Oyelami (2008). "A Modified Diminishing Increment Sort for Overcoming the Search for Best Sequence of Increment for Shellsort." Journal of Applied Sciences Research. [On-line], 4, pp. 760-766. Available: http://www.aensiweb.com/jasr/jasr/2008/760-766.pdf [Nov. 12, 2013]

[9] M.O. Oyelami, A.A. Azeta and C.K Ayo. "Improved Shellsort for the Worst-Case, the Best-Case and a Subset of the Average-Case Scenarios." Journal of Computer Science & Its Application. vol. 14, pp. 73 – 84, Dec. 2007.

[10] D. Musser (1997). "Introspective Sorting and Selection Algorithms." Software: Practice and Experience (Wiley). [On-line]. 27(8), pp. 983-993. Available: http://www-home.fh-konstanz.de/~bittel/prog2/Praktikum/musser97introspective.pdf [January 15, 2012].

[11] "A guide to Introsort." Internet: http://debugjung.tistory.com/entry/intro- sortintrospective-sort-????, [February 16, 2012].

[12] E.K. Donald. The Art of Computer Programming. Volume 3, Sorting and Searching. USA: Addison-Wesley, 1998; pp. 110.

[13] O.M. Oyelami. "Improving the performance of bubble sort using a modified diminishing increment sorting." Scientific Research and Essay, vol. 4, pp. 740 -744, 2009.

[14] S. Sartaj. Data Structures, Algorithms and Applications in Java. USA: McGrawHill, 2000, pp. 65 – 67.

[15] F. William and T. William. Data Structures With C++ Using STL. USA: Prentice Hall, 2002, pp. 131.

[16] E.K. Donald. The Art of Computer Programming. Volume I, Fundamental Algorithms. USA: Addison-Wesley, 1997.