A Paradigmatic Comparison of Sorting Algorithms on Integer Arrays

Adigun A. A
Department of Computer Science and Engineering
LadokeAkintola University of Technology (LAUTECH)
Ogbomosho, Nigeria

Asani, E. O., Adegun, A.A. & Mfoniso, E.A.
Department of Computer Science
Landmark University, Omuaran, Nigeria.
asani.emmanuel@lmu.edu.ng, adegun.adekanmi@lmu.edu.ng

ABSTRACT

Sorting algorithms are used very often by computer applications owing to their relative importance in ordering data in ways that can be easily manipulated. It is one area of computing that has received much attention, and this is due to its wide application as a preliminary to many other computer operations. This study provides a comparative complexity study of three popular sorting algorithms (Radix sort, Quick sort and Bubble sort). These algorithms have been in use for a long time and continue to elicit research interest from researchers. However, the question of "which and when to use?" remains. Each of these algorithm solves the sorting problem in dynamic ways. The aim of the study is to analyse the working complexities of these three sorting algorithms and compare them to determine their performance over different test cases and programming paradigms.

Keywords – Bubble sort, quick sort, radix sort, time complexity and performance metrics.

1. INTRODUCTION

In computing, sorting is the arrangement of the elements of a list in a given order. Sorting is often done either in numerical or lexicographical order. Sorting is fundamental to computer science because data can be handled more efficiently when sorted than in a randomized way [1]. A particular sorting algorithm is considered to be efficient if it uses computer resources such as memory, CPU time and disk usage in an effective manner. The effectiveness here implies that, given a particular size of inputs, how fast does it take (CPU time) to sort the inputs in a particular order (time efficiency) or how much of memory space does the algorithm consume to sort the inputs (space efficiency). In order to find the efficiency of each of these sorting algorithms, we carry out a complexity analysis of each of algorithm first, then we compare.

2. WORKING PROCEDURE OF THE SORTING ALGORITHMS

2.1 Quick Sort:
The Quick sort algorithm uses the divide and conquer strategy and works recursively until all the elements are sorted. This is based on the philosophy that small lists are easier to sort than long ones. It works by choosing an element as a pivot and then making a recursive call to the quicksort algorithm to sort the elements on both sides. After sorting the smaller partitions, they are all combined together to have a sorted form of the original array. So typically, the quick sort has three basic subroutines vis *divide, conquer* and *combine* as described below.

a. Divide: Given an array A [p….s] such that p <= s-1, the *divide* subroutine re-arranges the array into two non-empty sub arrays A[p….q] and A[q+1....s] such that each element of sub array A[p….q] is less than or equal to each element of sub array A[q+1….r]. The index of q is outputted as part of this partitioning procedure.
b. Conquer: The 2 sub arrays A[p….q] and A[q+1….r] are further partitioned by recursive calls till the resulting sub-arrays become too small to sort by comparison.
c. Combine: Since the sub arrays are sorted in place, we then combine them (the sorted sub-arrays to produce the sorted elements of array A[p…s].

For instance:
Consider a list of 11 values:
2 6 11 5 8 4 9 3 10 1 7
Fig 1 illustrates the quicksort process

If 7 is chosen as the pivot element, then all values less than 7 will be taken to the left of 7 and all values greater than 7 will be taken to the right of 7. This recursive process continues until a sorted list is achieved for the minutest division. Finally the algorithm combines.
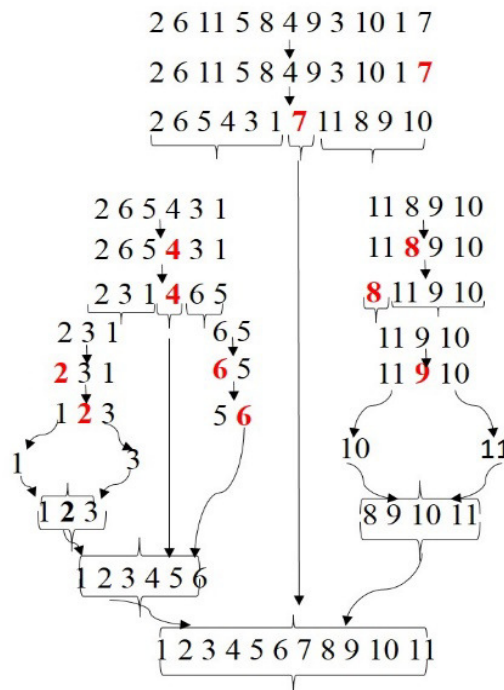

Fig 1: *Quick sort process*

Algorithm: Given an array A of [1-N] elements[2]

The implementation is below [2]

QUICK (A,p,r)
```
1.      If p>=r then return
//      q is the pivot
2.      q = PARTITION(A,p,r)
//      after 'partition'
//      A[left..q-1] ≤ A[ q] ≤ A[q+1..right]
3.      QUICK(A,p,q-1)
4.      QUICK(A,q+1,r)
5.      Exit
```

The subroutine PARTITION( ) is used to scan the array and then send all elements less than the pivot to the left and all elements greater than the pivot is moved to the right. Quick sort depends largely on this routine to partition the unsorted array into two sub lists at every stage of the sorting process with reference to the pivot element.

It is defined as follows:
PARTITION (A, p, r)

1.   X = A[r]
2.   I = p -1
3.   For j = p to r-1 do
   i.   If A[j] <= x then
   ii.   i = i+1
   iii.   Exchange A[i] and A[j]
   iv.   END IF
4.   Exchange A[i+1] and A[r]
5.   Return i+1
6.   END FOR LOOP
7.   Exit

Quicksort Analysis

To analyse quick sort algorithm, we shall assume that the pivot chosen divides the array into two sizes of k and n-k. The running time of quicksort depends on whether the partitioning is balanced or unbalanced,and this in turn depends on which elements are used for partitioning. If the partitioning is balanced, the algorithm runs asymptotically as fast as merge sort. If the partitioning is unbalanced, however, it can run asymptotically as slowly as insertion sort. [3]

Best Case

The best case of quicksort occurs when the pivot chosen divides the array into two equal parts in every step. Thus we have k = n/2 and n-k = n/2 for the original array of size n.[4][3]
The total runtime can be written as follows:
$T(n) = 2T(n/2) + \alpha n$ ……………eqn.1

Substituting n/2 for n in eqn.1
$= T(n/2) = 2(2T(n/4) + \alpha n/2) + \alpha n$
$= 2^2 T(n/4) + 2\alpha n$
$= 2^2(2T(n/8) + \alpha n/4) + 2\alpha n$
$= 2^2 T(n/8) + 3\alpha n$
$= 2^k T(n/2^k) + 3\alpha k$ (continuing to the kth step)………..eqn.2

Observing the above equation, it can be noted that the recursion will continue until $n = 2^k$. This is because when $n = 2^k$, $n/2^k = 0$. Therefore the recursion will continue until $k = \log n$.

Hence, we substitute log n for k in eqn.2
$T(n) = nT(1) + \alpha n \log n$
$= O(n \log n)$

Worst Case Analysis

The worst case scenario occurs when the pivot chosen is the least element in the array. Thus we have k = 1 and n-k = n – 1.

$T(n) = T(1) + T(n-1) + \alpha n$
$= T(n-2) + T(1) + \alpha(n-1) + T(1) + \alpha n$
$= T(n-2) + 2T + \alpha n - \alpha + \alpha n$
$= T(n-2) + 2T + \alpha(n-1+n)$
$= T(n-3) + T(1) + \alpha(n-2) + 2T + \alpha(n-1+n)$
$= T(n-3) + 3T + (\alpha n - 2\alpha + \alpha n - \alpha - \alpha n)$
$= T(n-3) + 3T + \alpha(n - 2 + n -1 - n)$
$= T(n-4) + T(1) + \alpha(n-3) + 3T + \alpha(n - 2 + n - 1 - n)$
$= T(n-4) + 4T + (\alpha n - 3\alpha + \alpha n - 2\alpha + \alpha n - \alpha - \alpha n)$
$= T(n-4) + 4T + \alpha(n - 3 + n - 2 + n - 1 - n)$
$= T(n-i) + IT(1) + \alpha(n - i + 1 \ ….. + n - 2 + n - 1 + n)$

$$= T(n-1) + i\,T(1) + \propto \left(\sum_{j=0}^{i-1} (n-j) \right)$$

Observing the above recurrence, it will be noted that n – i will be less than one when i = n – 1, hence, we substitute n – 1 for i in the above equation.

$$T(1) + (n-1)T(1) + \propto \left(\sum_{j=0}^{n-2} (n-j) \right)$$

$$= nT(1) + n(n-1)T(1) + \propto \left(\sum_{j=0}^{n-2} j \right)$$

But $\sum_{j=0}^{n-2} j \sum_{j=1}^{n-2} j = (n-2)(n-1)/2$

$= nT(1) + n(n-1)T(1) - (n-2)(n-1)/2$

$= O(n^2)$

Average Case Analysis

$$T(n) \leq \frac{2}{n} \sum_{i=1}^{n-1} (ai \log i + b) + dn$$

$$\leq \frac{2}{n} \left(\sum_{i=1}^{n-1} ai \log i \right) + 2b + dn$$

$$= \frac{2}{n} \left(\sum_{i=1}^{n/2} ai \log i + \sum_{i=n/2+1}^{n-1} (ai \log i) + 2b + dn \right)$$

$$\leq \frac{2}{n} \left(\sum_{i=1}^{n/2} ai \log(n/2) + \sum_{i=n/2+1}^{n-1} (ai \log n) \right) + 2b + dn$$

$$= \frac{2}{n} \left(\sum_{i=1}^{n-1} ai \log n - \sum_{i=1}^{n/2} ai \right) + 2b + dn$$

$$= \frac{2}{n} \left(\frac{n(n-1)}{2} (a \log n) - \frac{n/2(n/2+1)}{2} (a) \right) + 2b + dn$$

$$\leq a(n-1) \log n - \frac{n}{4}(a) + 2b + dn$$

$$= an \log n + b - \frac{n}{4}(a) + b + dn$$

$\leq an \log n + b$ for $a > 4(b+d)$

$= O(n \log n)$

2.2 Bubble Sort

Bubble sort algorithm is the simplest and most used of all the sorting algorithm. It is a comparison based algorithm whereby each element is compared with the next element and their position altered accordingly.

Bubble sort works by comparing every item in the list to the next item, and if the first item is larger than the second, then they are exchanged.

If we were to bubble sort the list below:

4 2 3 1

1st Iteration
4   2   3   1
Step1:   Compare the first element with the second element, 2 is less than 4 and so an exchange occurs.
2   4   3   1
Step 2:   The second element is then compared with the third element resulting in 3 being swapped with 4 since 4 is greater than 3.
2   3   4   1
Step 3:   The third element is then compared with the fourth element which results in the following order and concludes the first iteration through the list.
2   3   1   4

2nd Iteration
2   3   1   4
Step 1: 2 and 3 is compared but no swap occurs since 2 is less than 3.
2   3   1   4
Step 2: 3 and 1 is compared and swapped since 3 is greater than 1.
2   1   3   4
Step 3: 3 and 4 is compared and no swap occurs. The resulting list at the end of the second iteration is
2   1   3   4

3rd Iteration
2   1   3   4
Step 1: 2 and 1 is compared and a swap occurs since 2 is greater than 1.
1   2   3   4
At this point, all elements are in their correct position and the list is sorted.

The process continues until the smallest number bubbles to the front of the list and the largest number bubbles to the back of the list.

At the end of the sorting process, the final list will be
1   2   3   4

The algorithmic implementation is below:

Bubblesort (A: [1…n], iteration: increments on each call)

#		Costs	Count
1.	Changed = false	c_1	1
2.	For I = 1to A.length – iteration inclusive	c_2	n
3.	If A[i] > A[i+1]	c_3	n-1
4.	SWAP(A,I,i+1)	c_4	n-1
5.	Changed = true	c_5	n-1
6.	If changed	c_6	1
7.	Bubblesort(A, iteration+1)		1

Best Case Analysis
The best case situation is when the array is already sorted, in which case line 4,5 and 7 will never be executed. The total runtime will therefore be:

$$T(n) = c_1 + c_2 . n + c_3 . (n-1) + c_6$$
$$T(n) = n. (c_2 + c_3) + c_1 + c_6 - c_3$$
$$T(n) = O(n)$$

Worst Case Analysis
The worst case scenario occurs when the first element is the maximum. In this case the total runtime will be:
c1 + c2.n + (n-1). (c3 + c4 + c5) + c6 +c7
But since the line 7 reiterates several times, the equation becomes

$$c1 + c2.n + (n-1).(c3 + c4 + c5) + c6 + \sum_{j=0}^{n}(n-1)$$

$$= n.(c2 + c3 + c4 + c5) + c1 + c6 - (c3 + c4 + c5) + \sum_{j=0}^{n}(n-i)$$

$= O(n^2)$

2.3 RADIX SORT
Radix sort is a non-comparative sorting algorithm that makes use of keys to sort list of values. Keys are usually integers and sometimes it may consider an alphabet as key for sorting strings. There are two classifications of radix sort: Least Significant Digit (LSD) and Most Significant Digit (MSD) radix sort. The LSD first sort proceeds from the least significant digit and moves from the right to left.

Consider a list L = {489, 358, 145, 909, 690, 820, and 527}. The number of elements on the list $n = 7$, the number of digits $l = 3$ and radix = 10. To sort using radix sort, the algorithm will require 10 bins and the sorting will be done in 3 passes, explanation is shown in Fig. 2 below.

L = {489, 358, 145, 909, 690, 820, 527}
Pass 1

820 690					145		527	358	909 489
0	1	2	3	4	5	6	7	8	9

Pass 2

909		820 527		145	358			489	690
0	1	2	3	4	5	6	7	8	9

Pass 3

	145		358	489	527	690		820	909
0	1	2	3	4	5	6	7	8	9

Fig. 2: *Explanation of Radix sort*

At the end of the third pass, the sorted list is
L= {145, 358, 489, 527, 690, 820, 909}

Radix Performance Analysis
The efficiency of radix sort is a somewhat difficult one to establish when compared to other algorithms. Theoretically, the average runtime complexity of radix sort is O ($d.n$), where d is the number of digits in each element and n is the number of elements. In practical, the efficiency of radix sort depends on the value of d.

3. EMPERICAL EXPERIMENTS AND RESULTS

The algorithms were implemented over two programming paradigms namely, Procedural and Object Oriented Programming. C programming language was the preferred procedural language used, due to its standard library concept, large repertoire of operators and syntax but more importantly, its ready access to hardware when needed. Java was preferred as our Object Oriented Programming language due to its portability and large class library.

The OOP Programming paradigm

The three sorting algorithms were implemented using the java programing language. The java system.currentTimeMillis() function was used to time the sorting process during program execution. This function returns the current system time in milliseconds. The inbuilt java random number generating function was used to generate random number of integers with size varying from 1000 to 100000. The data collected was the running time of each algorithm on the different size of input used. The test were carried out five times and the average was calculated. Results are shown in Tables and Figures below.

Table 1: Average time taken to sort different list size using different sorting algorithms in java programming language.

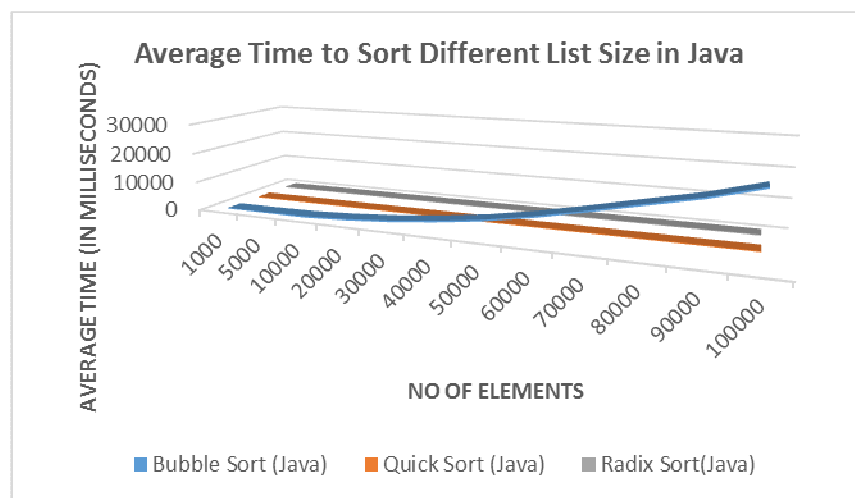| No. of elements	Average Time Taken in milliseconds		
	Bubble Sort (Java)	Quick Sort (Java)	Radix Sort(Java)
1000	22	0.8	0.8
5000	78.2	1.0	1.4
10000	262.6	1.6	3.6
20000	966.6	3.2	5.4
30000	2101.6	4.6	6.2
40000	3714.2	5.2	8.8
50000	5722	6.6	10.6
60000	8490.6	8.0	12
70000	11805.6	8.6	13.6
80000	15323.2	10.4	15.6
90000	18745	10.6	18.2
100000	23102.2	12.2	20.2

Fig 3: Time taken to sort different list size using the different sorting algorithms in java programming language.

Table 2: Time taken to sort different list size using the different sorting algorithms in C programming language

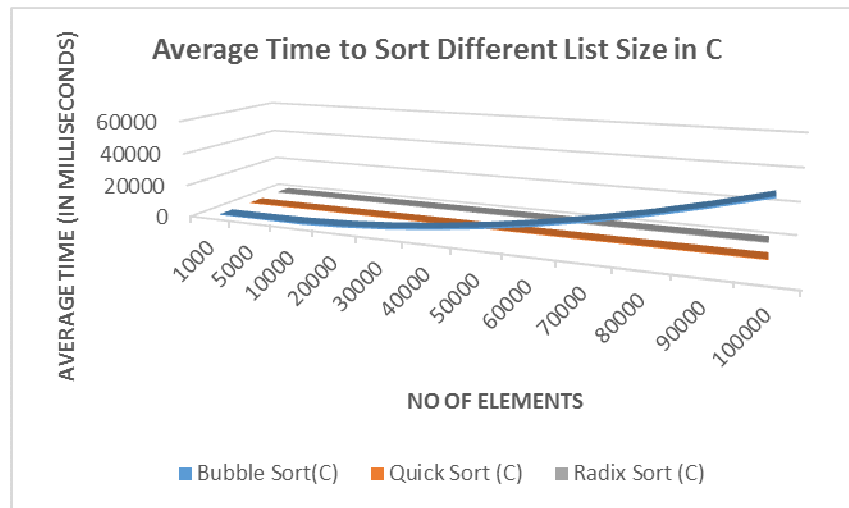| No. of elements	Average Time Taken in milliseconds		
	Bubble Sort(C)	Quick Sort (C)	Radix Sort (C)
1000	3	0	0
5000	79	1	0.8
10000	369.6	2	1
20000	1561	3.8	3
30000	3607	5.2	4
40000	6497.4	7	5
50000	10210.2	9	6
60000	14739	10.6	7.2
70000	20176.4	12	8
80000	26364	14	9.2
90000	33538	16	11
100000	41535	17	12

Fig 4: Time taken to sort different list size using the different sorting algorithms in C programming language

Table 2: Average time taken to sort different list size in both C and Java programming language.

| No. of elements	Average Time Taken in milliseconds					
	Bubble Sort (Java)	Quick Sort (Java)	Radix Sort (Java)	Bubble Sort (C)	Quick Sort (C)	Radix Sort (C)
1000	22	0.8	0.8	3	0	0
5000	78.2	1.0	1.4	79	1	0.8
10000	262.6	1.6	3.6	369.6	2	1
20000	966.6	3.2	5.4	1561	3.8	3
30000	2101.6	4.6	6.2	3607	5.2	4
40000	3714.2	5.2	8.8	6497.4	7	5
50000	5722	6.6	10.6	10210.2	9	6
60000	8490.6	8.0	12	14739	10.6	7.2
70000	11805.6	8.6	13.6	20176.4	12	8
80000	15323.2	10.4	15.6	26364	14	9.2
90000	18745	10.6	18.2	33538	16	11
100000	23102.2	12.2	20.2	41535	17	12
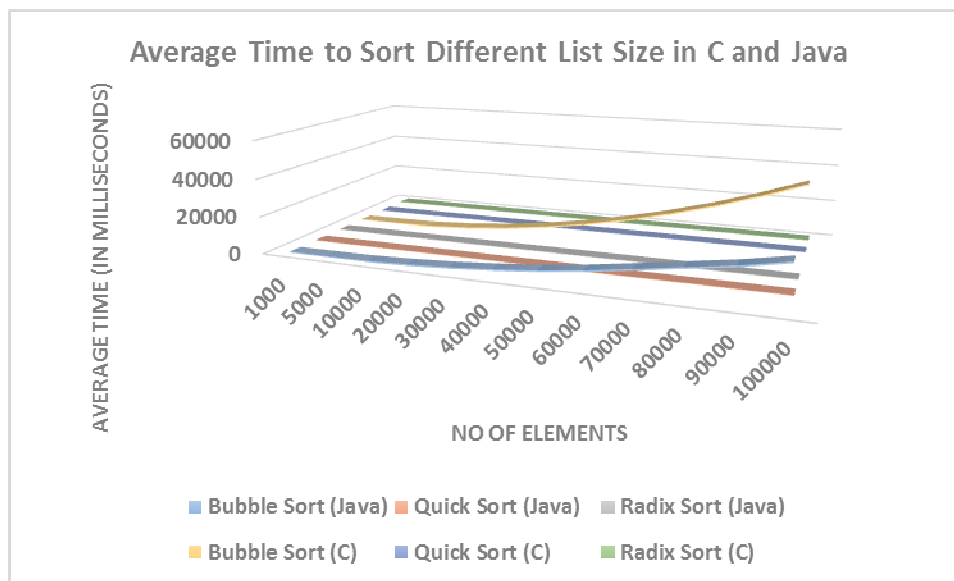
Fig 5: Time taken to sort different list size using the different sorting algorithms in C and Java programming languages

4. CONCLUSION

From the above analysis, when comparing the algorithms, it can be concluded that in a list of 1000 to 100000, bubble sort takes more time to sort as compared to quick sort and radix across the two programming paradigms. Quick sort, when compared to radix sort produced an almost identical performance but quickly gets better as the input size increases. If we consider the worst case complexity of the three algorithms, then quick and radix sort gives the result of the order of n^2. We also can infer from the analysis that sorting generally takes less time to run in Procedural Languages than in Object Oriented Program (OOP) paradigm. The marginal exception observed in quick sort is nullified by the performance of radix sort which is the best across the two paradigm. This explains why Procedural Programming Paradigm produces better runtime efficiency than the OOP paradigm [5].

REFERENCES

[1]. Pooja K. C. and Jayashree S. S. (2013). Comparative Analysis & Performance of Different Sorting Algorithm in Data Structure. International Journal of Advanced Research in Computer Science and Software Engineering vol. 3(11). Pp500-507.

[2]. Nidhi C. and Simarjeet S. B. (2013) A Comparison based Analysis of different types of Sorting Algorithms with their Performances. Indian Journal of Research Vol. 2(3), pp10-13.

[3]. Kamlesh K. P., Rajesh K. B. and Kamlesh K. R. (2014). A Comparative Study of Different Types of comparison Based Sorting Algorithms in Data Structure. International Journal of Advanced Research in Computer Science and Software Engineering, vol.4(2). Pp304-309.

[4]. Thomas H. C., Charles E. L., Ronald. L. R. and Clifford. S.(2001). Introduction to Algorithms, 2ed. Cambridge, Massachusetts, MIT Press.

[5]. Kuan C. C. (2004). Comparison of Object-Oriented and Procedure-Based Computer Languages. Issues in Information System. Vol 5(1) pp 70-76.