Int. J. of Computer and Communications
Vol. 1, No. 1, March 2011

www.ijcc.yolasite.com

A better approach to QuickSort implementation

D.Abhyankar*, M.Ingle**

D. Abhyankar, School of Computer Science, D.A. University, Indore
deepak.abhyankar@yahoo.co.in
M.Ingle, School of Computer Science, D.A. University, Indore
maya_ingle@rediffmail.com

Abstract

Quicksort is one of the most intriguing sorting algorithms and is a part of C, C++ and Java libraries. This paper analyzes the results of an empirical study of existing Quicksort implementations undertaken by authors. This paper formulates an alternative implementation of Quicksort. It is reported that alternative implementation is faster and simpler. Proposed implementation is based on some profound basic principles and a better partitioning algorithm.

1 Introduction

Quicksort is a phenomenally popular sorting algorithm established by C.A.R. Hoare which is on average faster than most of other sorting algorithms[2,3,4,5,6,7,8,9]. The problem for Quicksort is its worst case time complexity which is quadratic, but fortunately degradation to worst case rarely occurs. One of the salient points of Quicksort is its inner loop which is incredibly fast. Quicksort exploits the virtual memory and caches to the full and is especially effective on modern computer architectures. If we ignore the stack space needed by Quicksort then it is an inplace sorting algorithm. Because of these convincing reasons Quicksort is a leading sorting algorithm and has been a part of several standard libraries. For example C, C++ and Java libraries have long included

Quicksort algorithm to sort an integer array. Java library has long embraced Arrays.sort method to sort an integer array, which is a tuned Quicksort, adapted from Jon L. Bentley and M. Douglas McIlroy's "Engineering a Sort Function"[1]. It was tremendously useful to study the working of existing implementations of Quicksort and our experience suggests existing implementations are markedly slow.
Authors undertook an extensive study of java's Arrays.sort and found it noticeably slow. Profiling of Arrays.sort method is done on random as well as nonrandom inputs and it is observed that it is agonizingly slow even on random inputs. Arrays.sort method was taking seconds where any competitive sorting method should take milliseconds. It was found that Java library's Arrays.sort method is not good enough; we studied its strengths and weaknesses and set out to build a better implementation which ameliorates the situation and produces a measurable effect on performance. The proposed solution is based on the principles of simplicity, pragmatism and elegance. It adopts some important Quicksort optimizations which have been abandoned by Bentley and McIlroy. Compared to Arrays.sort method, presented sort algorithm is faster(typically 4 to 10 times) on random inputs. Proposed solution is intensely competitive even on nonrandom inputs.

It was observed that Java library's Arrays.sort method is excessively slow. It is already known that calls to small functions make a program tortuously slow and Java library's Arrays.sort function calls to small functions like swap and vecswap which slowed Arrays.sort method sharply. Java's Arrays.sort method is adapted from Bentley McIlroy paper which drops two important optimizations which are extremely effective in Java environment. Firstly one can place a big final insertion sort at the bottom of the recursion which replaces several bookkeeping operations by a single comparison between array elements [2]. Secondly Java's Arrays.sort avoids sentinels at the ends of the array which gain speed in Java environment. Proposed algorithm address these issues and delivers outstanding performance.

Section 2 proposes one of the most effective implementations of Quicksort. Subsection 2.1 informally describes the working of proposed implementation. Subsection 2.2 is a formal Java description of the suggested algorithm. Section 3 carries out a comparative study of proposed and existing implementation. Section 4 concludes and expresses the need of proposed implementation.

2 Proposed Implementation

Proposed implementation is a hybrid of Quicksort and insertion sort and is described by informal description 2.1 and formal description 2.2.

2.1 Informal Description of Proposed implementation

This description does not discuss Quicksort in its entirety; there are several good texts and research papers[3,4,5,6,7,8,9,10] which do that. This subsection concentrates on optimizations which we found particularly effective in delivering excellent performance. Proposed implementation dispense with small functions by manually inlining them because calling small functions have a crippling effect on the performance. For example partition function has been manually inlined by proposed implementation. Presented algorithm calls one big final insertion sort instead of little insertion sorts at the bottom of the recursion. This replaces several bookkeeping operations by a single comparison between array elements. Med3 function is another valuable function that takes the median of three elements as pivot and sets the sentinels at the array ends. Performance of Quicksort crucially depends on the selection of pivot and Med3 does the job of selecting a pivot. It takes median of first, last and middle element as pivot, so that performance does not degrade on non random input. Partitioning code gets rid of 3 instruction swap and replaces it by 2 instruction code. Partitioning code implements an amazingly fast partitioning algorithm which improves the performance in leaps and bounds.

2.2 Formal description of Proposed Implementation

```
class SCS_SORT{

// Hybrid of Quicksort and Insertion sort

 public static void sort(int a[], int n)
    {
  Quicksort(a,0,n-1);
  InsertionSort(a,n);  //one big final insertion sort
    }

 public static void Quicksort(int a[], int p, int r)
{

    if((r-p)>10)  //  Entertains subarrays  larger than size
12
    {
     // Inlined partitioning code
      int q;
      int p1 = p;
      int r1 = r;
      Med3(a,p1,r1); // Chooses pivot
     int x = a[p1];
     while(true)
          {
```

```
            do{
             r1--;
            }while(a[r1]>x);
    a[p1]=a[r1];
    do{
       p1++;
     }while(a[p1]<x);
    if(p1<r1)
    a[r1]=a[p1];
    else{
      if(a[r1+1]<=x)
      r1++;
      a[r1]=x;
      q =  r1;
      break;
    }

   }
        Quicksort(a,p,q-1);
        Quicksort(a,q+1,r);
    }
}

 public static void Med3(int a[], int p, int r)
    {
           // Selects the median and sets the sentinels

         int mid = (p+r)/2;
        int largest;
        if(a[p]>a[mid])
            largest = p;
        else largest = mid;
        int temp;
        if(a[largest]>a[r])
        {
            temp = a[r];
            a[r]=a[largest];
            a[largest]=temp;
        }
        if(a[mid]>a[p])
        {
            temp = a[p];
            a[p] = a[mid];
            a[mid]=temp;
        }

    }

 public static void InsertionSort(int a[], int n)
    {
     // Implementation of Insertion sort
```

```
    int j = 1; int key;
    while(j<n)
     {
        key = a[j];
        int i = j-1;
        while(i>-1)
        {
           if(a[i]>key)
           a[i+1] =a[i];
           else
               break;
           i--;

        }
        a[i+1]=key;
        j++;
     }
    }

}
```

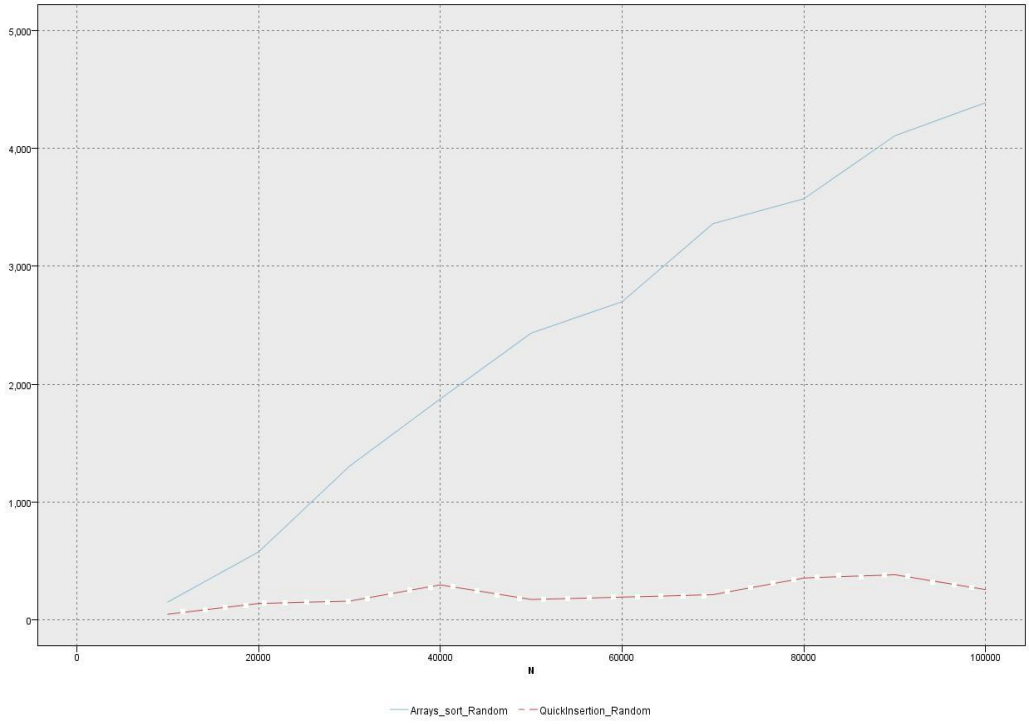
3 A Case Study

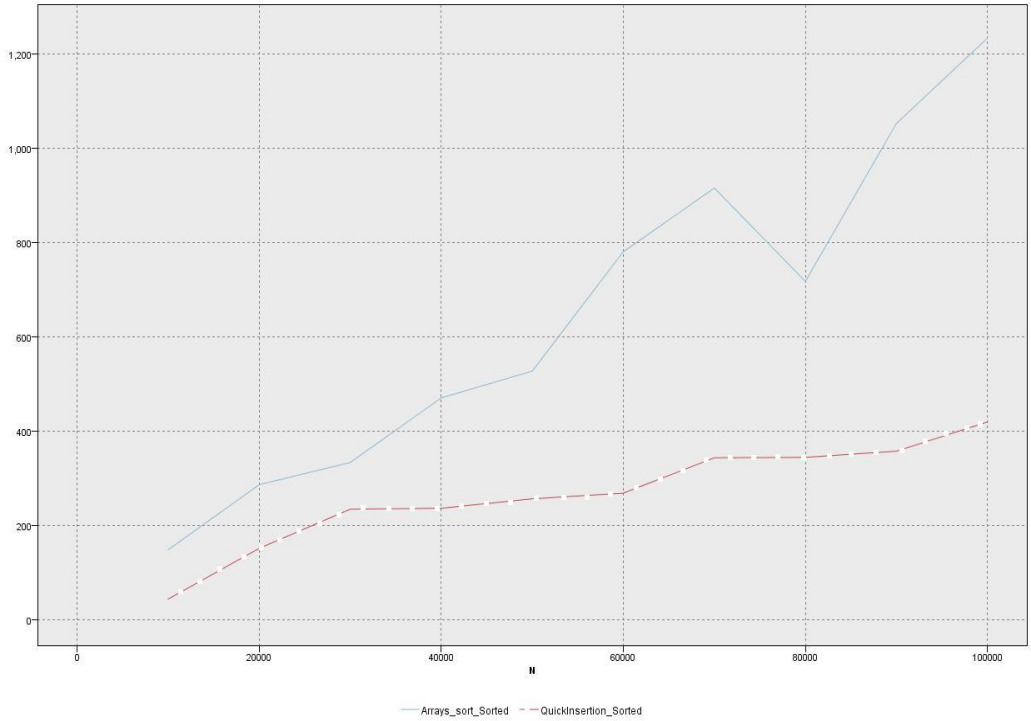
This research adopted an empirical approach to conduct a comparative case study of library method and proposed method. Netbeans 6.7 was used for profiling and was highly instrumental in preparing reliable statistics. We have generated random input and sorted input for our empirical study and on every test data proposed algorithm beats the library method. Case study revealed that proposed implementation is remarkably faster than library method on random data. Even on non random data proposed solution convincingly beats the library method.

Table 1 (***Comparison on Random and sorted Input)***

N	Time Taken in ***Milliseconds***			
	Arrays.sort()		QuickInsertion()	
	Random	Sorted	Random	Sorted
10000	153	149	48.9	44.3
20000	578	287	140	152
30000	1305	334	160	235
40000	1876	471	298	237
50000	2434	528	174	257
60000	2700	781	194	269

70000	3362	916	215	344
80000	3573	718	356	345
90000	4107	1053	384	358
100000	4388	1234	258	420


5,000
4,000
3,000
2,000
1,000
0
0
20000
40000
60000
80000
100000
N
Arrays_sort_Random
QuickInsertion_Random


1,200
1,000
800
600
400
200
0
0
20000
40000
60000
80000
100000
N
Arrays_sort_Sorted
QuickInsertion_Sorted

4 Conclusions

Study of existing Quicksort implementations revealed us that existing implementations are interminably slow and therefore we needed a fresh implementation which overcomes the weaknesses of existing implementations. To addresses the need of a fresh and fast implementation an industrial strength Java sorting program was developed which produces enormous improvement in performance. Proposed work has adopted some pragmatic optimizations which have been ignored by Java library. We ignored some optimizations which were not elegant and were the cause of increased complexity and abstruseness. Results of this research work suggest that simplicity, elegance and pragmatism are keys to excellent performance.

References

[1] J. L. Bentley and M. D. Mcilroy "Engineering a sort function," Software—practice and experience, VOL. 23(11), 1249–1265 (NOVEMBER 1993).

[2] R. Sedgewick, ‘Quicksort’, PhD Thesis, Stanford University (1975).

[3] C. A. R. Hoare, "Partition: Algorithm 63, " "Quicksort: Algorithm 64," Comm. ACM 4(7), 321-322, 1961.

[4] D. E. Knuth, *The Art of Computer Programming*, Vol. 3, Pearson Education, 1998.

[5] C. A. R. Hoare, "Quicksort," *Computer Journal5(1)* , 1962, pp. 10-15.

[6] S. Baase and A. Gelder, *Computer Algorithms:Introduction to Design and Analysis*, Addison-Wesley, 2000.

[7] J. L. Bentley, "Programming Pearls: how to sort," *Communications of the ACM*, Vol. Issue 4, 1986, pp. 287-ff.

[8] R. Sedgewick, "Implementing quicksort Programs," *Communications of the ACM*, Vol. 21, Issue10, 1978, pp. 847-857.

[9]T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001.

[10] G. S. Brodal, R. Fagerberg and G. Moruz, "On the adaptiveness of Quicksort," *Journal of Experimental AlgorithmsACM*, Vol. 12, Article 3.2, 2008.