A Survey of Adaptive QuickSort Algorithms

Laila Khreisat *khreisat@fdu.edu*
Dept. of Math, Computer Science, and Physics
Fairleigh Dickinson University
285 Madison Ave, Madison NJ 07940 USA

Abstract

In this paper, a survey of adaptive quicksort algorithms is presented. Adaptive quicksort algorithms improve on the worst case behavior of quicksort when the list of elements is sorted or nearly sorted. These algorithms take into consideration the already existing order in the input list to be sorted. A detailed description of each algorithm is provided. The paper provides an empirical study of these algorithms by comparing each algorithm in terms of the number of comparisons performed and the running times when used for sorting arrays of integers that are already sorted, sorted in reverse order, and generated randomly.

Keywords: Algorithm, Survey, Quicksort, Sorting, Adaptive Sorting.

1. INTRODUCTION

Since its development by [1], the quicksort algorithm has been considered the most popular and most efficient internal sorting algorithm. It has been widely studied and described [2, 3, 4, 5, 6, 7]. A divide-and-conquer algorithm, Quicksort sorts an array S of n elements by partitioning the array into two parts, placing small elements on the left and large elements on the right, and then recursively sorting the two subarrays. The major drawback of the Quicksort algorithm is that when the array to be sorted is already sorted or is partially sorted, the algorithm will require $O(n^2)$ comparisons. To improve on the worst-case behavior of Quicksort, several adaptive sorting algorithms have been developed. These algorithms take into consideration the already existing order in the input list, see [8]. Insertion sort is an adaptive sorting algorithm. Wainwright [9] developed ***Bsort***, an adaptive sorting algorithm, which improves the average behavior of Quicksort and eliminates the worst-case behavior for sorted or nearly sorted lists. ***Qsorte***, is another adaptive algorithm also developed by Wainright [10], which performs as well as Quicksort for lists of random values, and performs O(n) comparisons for sorted or nearly sorted lists. Another adaptive sorting algorithm is ***Nico***, which was developed by [11]. It is a version of Quicksort in which the partition function uses a for loop to roll the largest keys in the array to the bottom of the array.

Introsort (introspective sort) [12] is a hybrid sorting algorithm that uses both quicksort and heapsort. The method starts with quicksort and when the recursion depth goes beyond a specified threshold it switches to heapsort. Thus the algorithm combines the good aspects of both algorithms.

The paper studies these four sorting algorithms by comparing each algorithm in terms of the number of comparisons performed and the running times when used for sorting arrays of integers that are already sorted, sorted in reverse order, and generated randomly. The algorithms are also compared against the original quicksort algorithm, which we call ***Hoare***.

2. SORTING ALGORITHMS

The original Quicksort algorithm developed by [1] is considered to be the most popular and most efficient internal sorting algorithm. It has been widely studied and described [2, 3, 4, 5, 6, 7]. The

algorithm's worst-case time complexity of $O(n^2)$ occurs when the list of elements is already sorted or nearly sorted, or sorted in reverse order, [9]. The efficiency of Quicksort ultimately depends on the choice of the pivot see [13]. The choice of the pivot produces different variations of the algorithm. A pivot value that divides the list of keys near the middle is considered the ideal choice. In Hoare's original algorithm the pivot was chosen at random and the choice resulted in 1.386nlgn expected comparisons see [14].

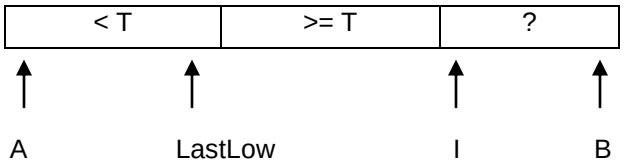
The following is the pseudo-code for the Quicksort algorithm:

```
void quicksort(int A[ ], int L, int R)
//Sorting starts by calling partition, which will choose a pivot and place the pivot A[i] in
//its correct position. Then the recursive calls to quicksort will rearrange the elements in
//the array around the pivot such that the array A will be sorted.
{
        int i;

        if (R <= L) return;
        i = partition(A, L, R);
                quicksort(A, L, i-1);
                quicksort(A, i+1, R);
}

void partition(int first,int last, int& pos)
{
        int p,l,r;
        l = first;
        r = last;
        p = l;
        swap(l,rand()%(last-first+1)+first);
        while (l < r)
        {
                 while ((l < r)&& (ar[p] <= ar[r]))
                 {   r--; }
                swap(p,r);
                p = r;
                while ((l < r)&&(ar[p] >= ar[l]))
                 {  l++; }
                swap(p,l);
                p = l;
         }
         pos = p;
}
```

Another variation of the Quicksort algorithm was developed by [11] based on an algorithm suggested by Nico Lomuto. In this algorithm the partition function uses a for loop to roll the largest keys in the array to the bottom of the array. A pivot T is chosen at random, and an index called LastLow is computed and used to rearrange the array X such that all keys less than the pivot T are on one side of the index LastLow, while all other keys are on the other side of the index. This achieved using a for loop that scans the array from left to right, using the variables I and LastLow as indices to maintain the following invariant in array X:



If X[I] >= T then the invariant is still valid . If X[I]<T, the invariant is regained by incrementing LastLow by 1 and then swapping X[I] and X[Lastlow]. The following is a C++ implementation of the partition function:

```
void Partition(int low, int high, int& pos)
{
        int i, j;
        int pivot;
        swap(low,rand()%(high-low+1)+low);
        pivot = ar[low];

        i = low;
        for (j = low+1; j <= high; j++)  //j = I
        {
                if (ar[j] < pivot)
                {
                        i = i +1;         // i = Lastlow
                        swap(i, j);
                }
        }
        swap(low, i);
        pos = i;
}
```

Wainwright [9] developed ***Bsort,*** a variation of Quicksort, designed for nearly sorted lists and lists that are nearly sorted in reverse order. The author claimed that the algorithm performs $O(n\log_2 n)$ comparisons for all distribution of keys. However, the claim was disproved in a technical correspondence in 1986 [15] that showed that the algorithm exhibits $O(n^2)$ behavior. For lists that are sorted or sorted in reverse order, the algorithm performs $O(n)$ comparisons. ***Bsort*** uses the interchange technique from Bubble sort in combination with the traditional Quicksort algorithm. During each pass of the algorithm the middle key is chosen as the pivot and then the algorithm switches over to Quicksort. Each key that is placed in the left subarray will be placed at the right end of the subarray. If the key is not the first key in the subarray, it will be compared with its left neighbor to make sure that the pair of keys is in sorted order. If the new key does not preserve the order of the subarray, it will be swapped with its left neighbor. Similarly, each new key that is placed in the right subarray, will be placed at the left end of the subarray and if it is not the first key, it will be compared with its right neighbor to make sure that the pair of keys is in sorted order, if not the two keys will be swapped [16]. This process ensures that the rightmost key in the left subarray will be the largest value, and the leftmost key in the rightmost subarray will be the smallest value, at any point during the execution of the algorithm.

Qsorte is a quicksort algorithm developed by Wainwright [10] that includes the capability of an early exit for sorted arrays. It is a variation of the original Quicksort algorithm with a modified partition phase, where the left and right subarrays are checked if they are sorted or not.

During the partitioning phase the middle key is chosen as the pivot. Initially, both the left and right subarrays are assumed to be sorted. When a new key is placed in the left subarray, and the subarray is still sorted, then if the subarray is not empty, the new key will be compared with its left

neighbor. If the two keys are not in sorted order then the subarray is marked as unsorted, and the keys are not swapped. Similarly, when a new key is placed in the right subarray, and the subarray is still sorted, then if the subarray is not empty, the new key will be compared with its right neighbor. If the two keys are not in sorted order then the subarray is marked as unsorted, and the keys are not swapped. At the end of the partitioning phase, any subarray that is marked as sorted will not be partitioned, [16]. ***Qsorte***'s worst-case time complexity is $O(n^2)$ which happens when the chosen pivot is always the smallest value in the subarray. During this scenario ***Qsorte*** will repeatedly partition the subarray into two subarrays with only one key in one of the subarrays.

```
void Qsorte (int m, int n)
{
        int k, v;
        bool lsorted, rsorted;

        if ( m < n ){
                FindPivot (m, n, v);
                Partition (m, n, k, lsorted, rsorted);
                if (! lsorted)  Qsorte(m, k-1);
                if (! Rsorted) Qsorte(k, n);
        }
}
```

Introsort (introspective sort) [12] is a hybrid sorting algorithm that uses both quicksort and heapsort. The algorithm starts by using quicksort and switches to heapsort when the recursion depth exceeds a specified threshold, thus avoiding the $O(n^2)$ worst-case behavior of quicksort.

3. EMPIRICAL TESTING AND RESULTS

To study the performance of the sorting algorithms described in section 2, all algorithms were used for sorting arrays of integers that were already sorted, sorted in reverse order, and generated randomly. The experiments were conducted on a computer with an Intel i7 processor with a speed of 2.6 GHz, and 16 GB of RAM. The sizes of the arrays ranged from N = 3000 to N = 400,000 elements. For the case of arrays of random numbers, each algorithm was used to sort three sequences of random numbers of a specific size N, and the average running time and number of comparisons were calculated. The Mersenne Twister random number generator developed by Matsumoto and Nishimura [17] was used to generate the sequences of random numbers. The generator has a period of $2^{19937}-1$.

Figure 1 below shows the running times for the sorting algorithms when used to sort arrays of random numbers of different sizes. The fastest algorithm is ***Qsorte***, giving the best performance for sorting arrays of random integers overall. This is followed by ***Bsort.*** ***Hoare*** is faster than ***Nico*** for data sizes between N= 9000 and N=262144, after which the two algorithms are comparable. The slowest algorithm was ***Introsort***.

In terms of the number of comparisons, ***Nico*** and ***Introsort*** were comparable and had the best performance compared to the other algorithms. ***Qsorte*** required on average 1.15 more comparisons than ***Nico,*** and ***Hoare*** performed on average 1.23 more comparisons than ***Qsorte.*** The worst performance was exhibited by ***Bsort*** requiring on average 1.7 more comparisons than ***Hoare***, and 2.3 more comparisons than ***Nico***. The performance of all the algorithms in terms of the number of comparisons was of order $O(N \log_2 N)$, see figure 2.

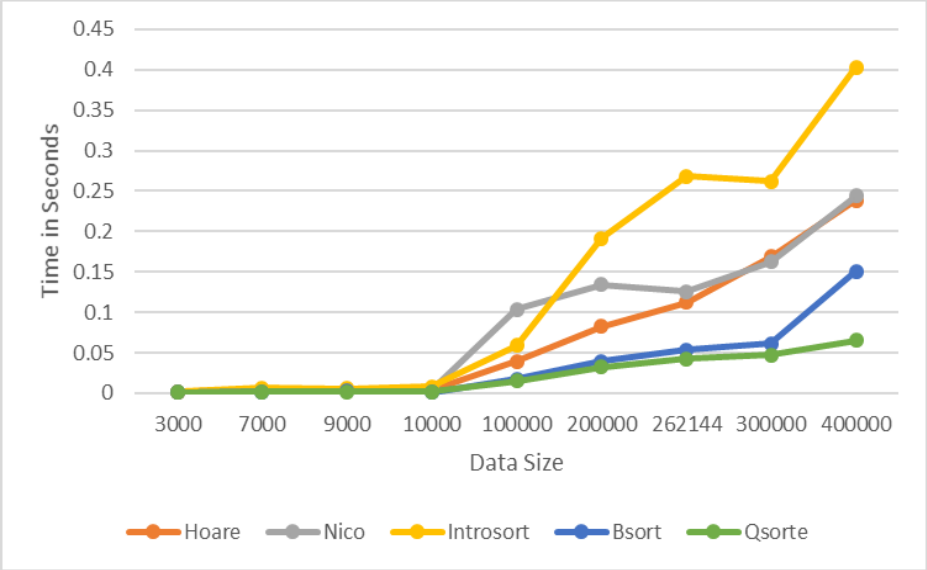


FIGURE 1: Average Running Times for Random Data.

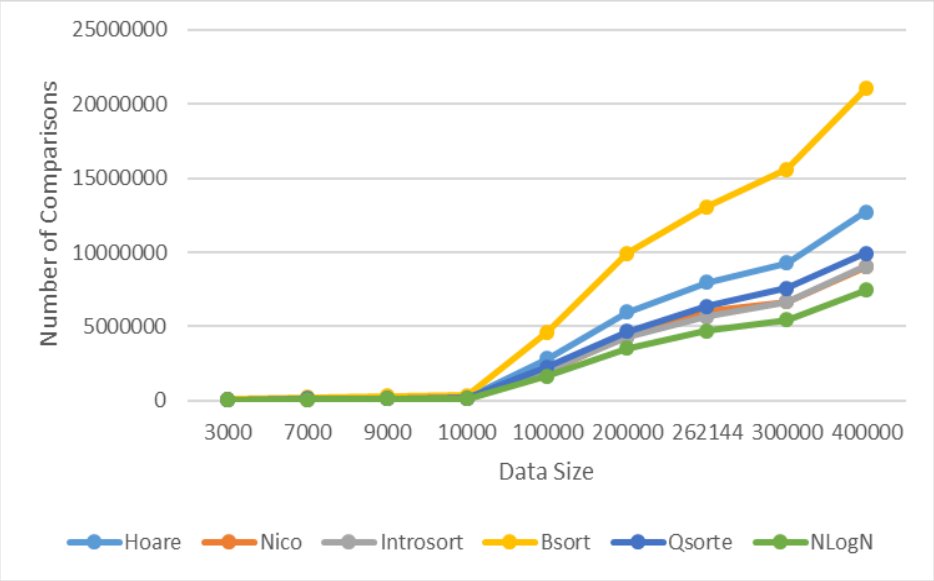


FIGURE 2: Average Number of Comparisons for Random Data.

For data sorted in reverse order, ***Bsort*** was the fastest for sorting a list of size 10,000 or less, followed by ***Qsorte***, then Hoare, with ***Introsort*** having the slowest performance, see figure 3-1. For lists of size 100,000 and above, figure 3-2 clearly shows that ***Qsorte*** had the best performance, while ***Introsort*** exhibited the slowest running time.

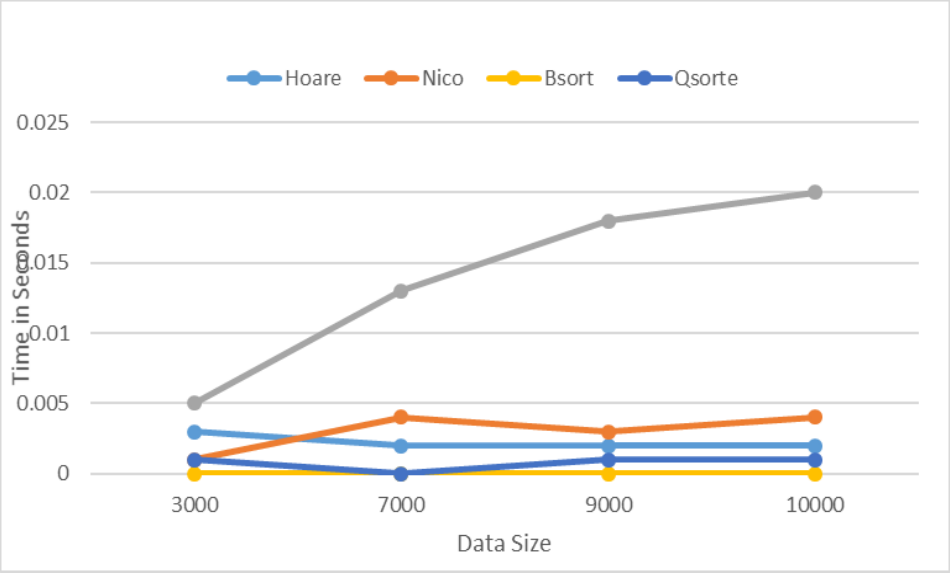


FIGURE 3-1: Running times for Data Sorted in Reverse.

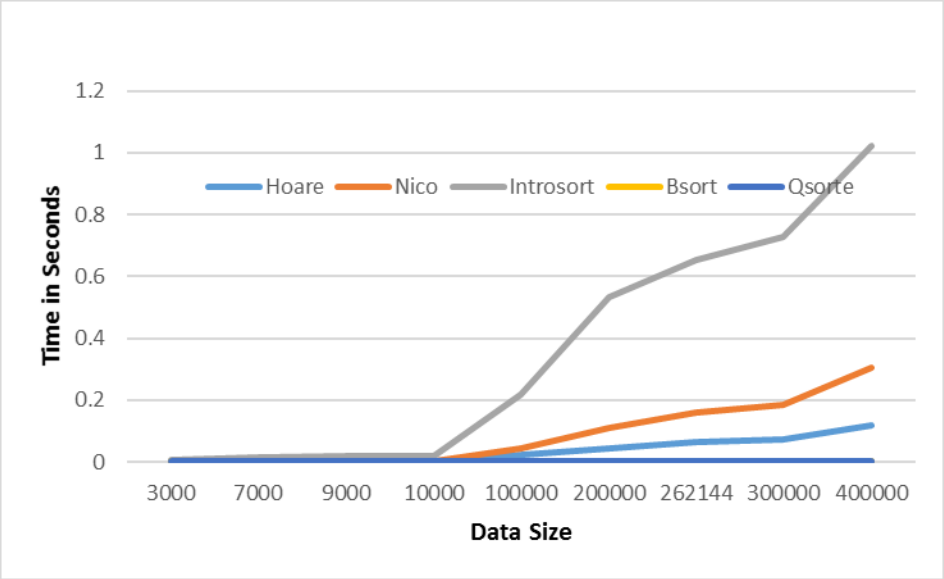


FIGURE 3-2: Running times for Data Sorted in Reverse.

In terms of the number of comparisons, ***Qsorte*** required the smallest number of comparisons followed by ***Bsort***, both of which were of order $O(N)$. Comparing ***Nico*** and ***Hoare***, it is apparent that ***Hoare*** requires more comparisons than ***Nico*** and both are of order $O(N \log_2 N)$. ***Hoare*** made the largest number of comparisons overall, and on average ***Hoare*** made 14 more comparisons than ***Qsorte***, which exhibited the best performance. The behavior of all three algorithms, namely, ***Hoare***, ***Nico***, and ***Introsort***, in terms of the number of comparisons was of order $O(N \log_2 N)$, see figures 4-1 and 4-2.

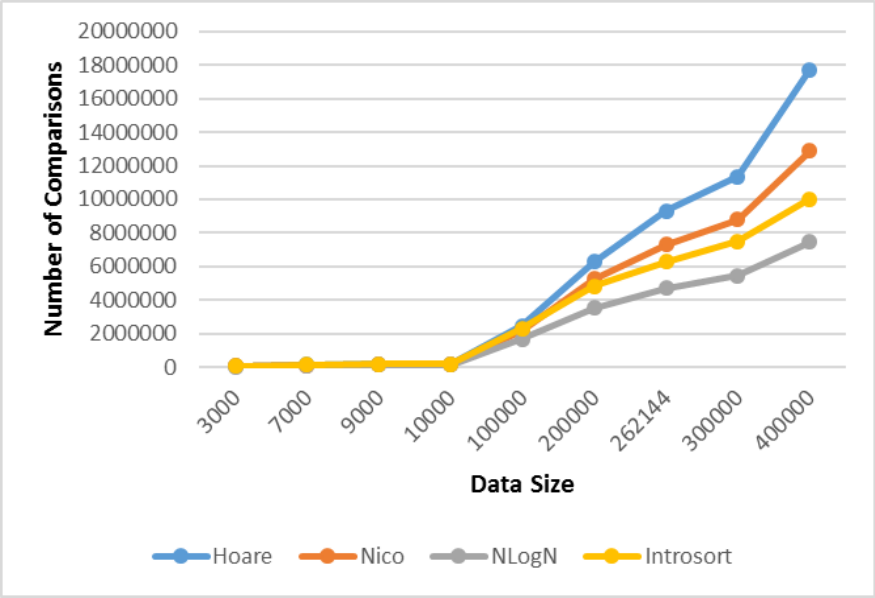


FIGURE 4-1: Number of Comparisons for Data Sorted in Reverse.

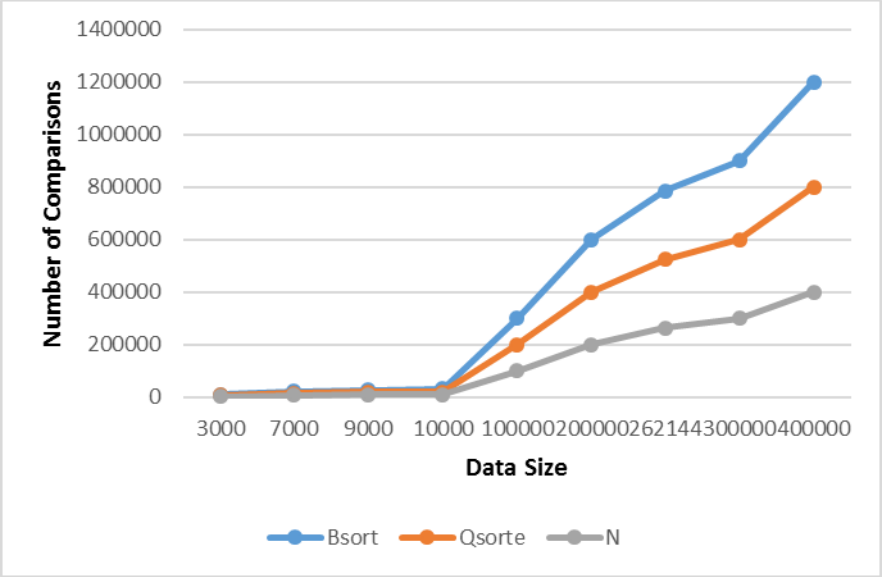


FIGURE 4-2: Number of Comparisons for Data Sorted in Reverse.

For sorted data, both ***Bsort*** and ***Qsorte*** achieved the fastest running times. ***Bsort*** and ***Qsorte*** were both faster than ***Hoare*** which was faster than ***Nico***, and ***Introsort*** exhibited the worst behavior, see figure 5.

Qsorte and ***Bsort*** performed the same number of comparisons, which was of order $O(N)$, see figure 6-1. This is the expected behavior of ***Bsort*** and ***Qsorte*** for sorted lists. ***Introsort*** performed the smallest number of comparisons for N>=200,000 compared to ***Hoare*** and ***Nico***. All three algorithms exhibited $O(N \log_2 N)$ behavior, see figure 6-2.

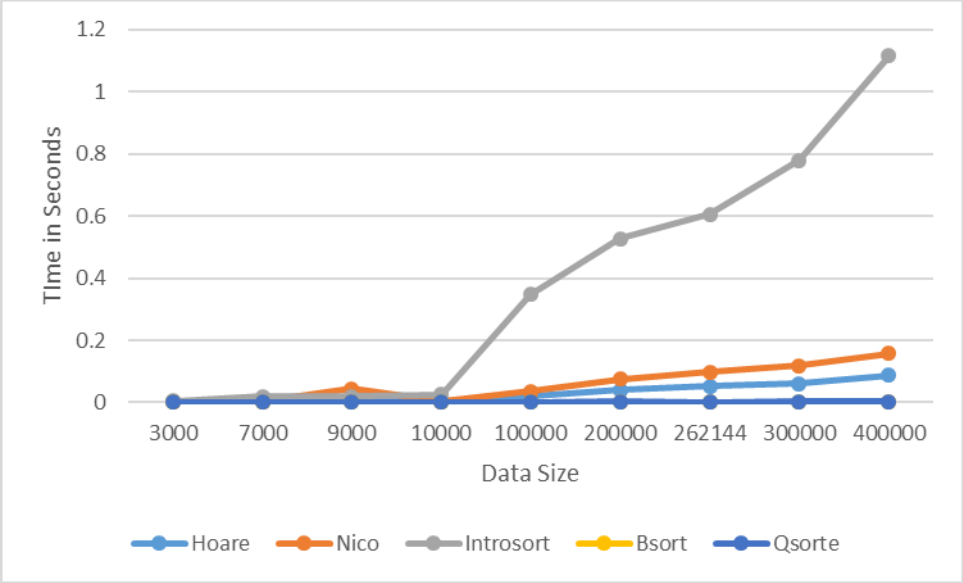


FIGURE 5: Running Times for Sorted Data.

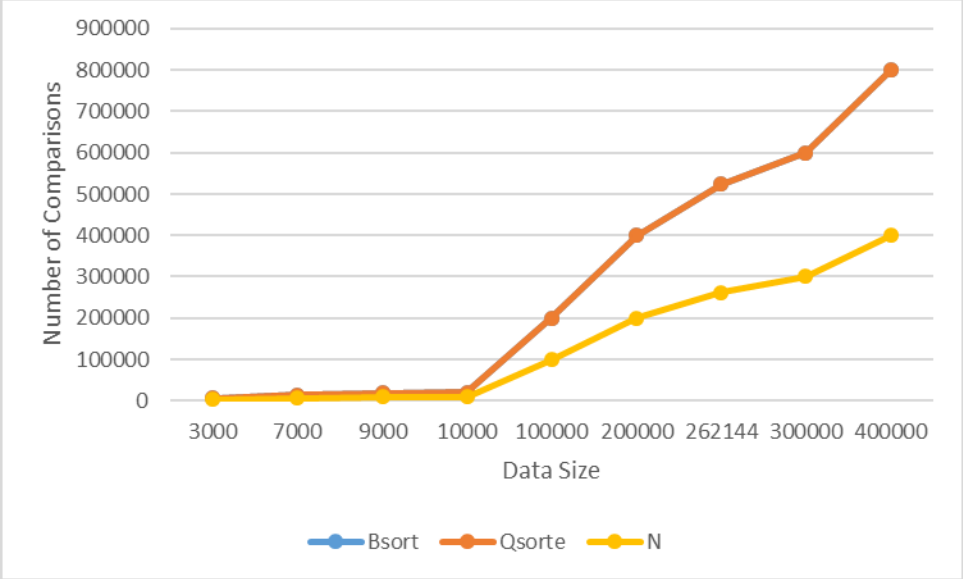


FIGURE 6-1: Number of Comparisons for Sorted Data.

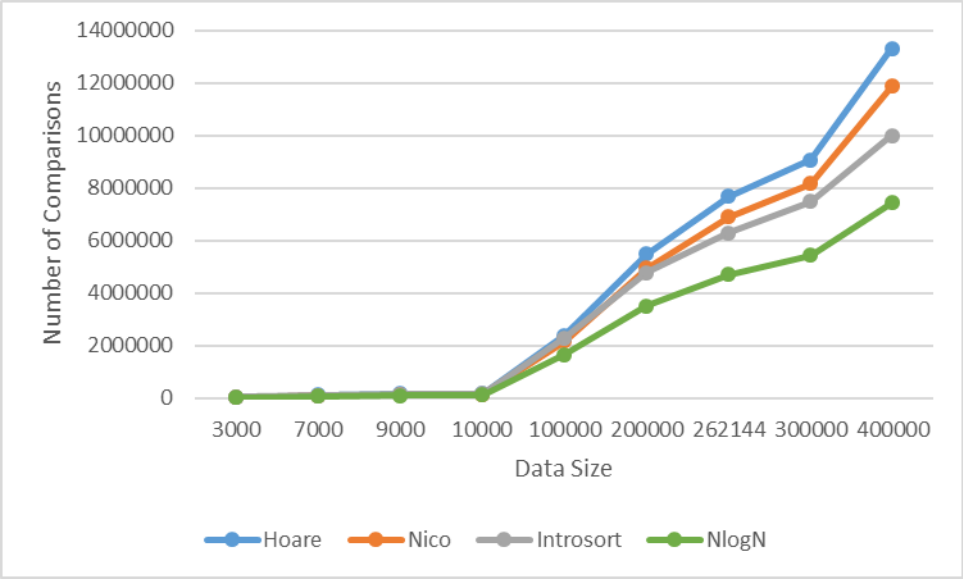


FIGURE 6-2: Number of Comparisons for Sorted Data.

4. CONCLUSIONS

In this paper the results of an empirical study of adaptive variations of the quicksort were presented. All the sorting algorithms were studied by computing their running times and number of comparisons made when used for sorting random data, sorted data and data sorted in reverse. ***Qsorte*** was the fastest when used for sorting random data, while ***Bsort*** was the fastest for sorting lists of data already sorted in reverse of size 10,000 or less. For lists of size 100,000 and above, ***Qsorte*** had the best performance. For sorted data, both ***Bsort*** and ***Qsorte*** achieved the fastest running times.

In terms of the number of comparisons for random data, ***Nico*** and ***Introsort*** were comparable and had the best performance, which was of order $O(N \log_2 N)$. For data sorted in reverse, ***Qsorte*** required the smallest number of comparisons followed by ***Bsort***, both of which were of order $O(N)$. For sorted data ***Qsorte*** and ***Bsort*** performed the same number of comparisons, which was of order $O(N)$. ***Introsort*** performed the smallest number of comparisons for N>=200,000 compared to ***Hoare*** and ***Nico***. All three algorithms exhibited $O(N \log_2 N)$.

5. REFERENCES

[1] C.A.R. Hoare. “Algorithm 64: Quicksort.” *Communications of the ACM*, vol. 4, pp. 321, Jul. 1961.

[2] R. Loeser. “Some performance tests of: quicksort: and descendants.” *Communications of the ACM*, vol. 17, pp. 143–152, Mar. 1974.

[3] J.L. Bentley, R. Sedgewick, R. “Fast algorithms for sorting and searching strings,” in Proc. 8th Annual ACM-SIAM symposium on Discrete algorithms, 1997, pp. 360–369.

[4] R. Chaudhuri, A. C. Dempster. “A note on slowing Quicksort,” in Proc. SIGCSE , 1993, vol. 25.

[5] R. Sedgewick. “Quicksort.” PhD dissertation, Stanford University, Stanford, CA, USA, 1975.

[6] R. Sedgewick. “The Analysis of Quicksort Programs.” *Acta Informatica*, vol. 7, pp. 327–355, 1977.

[7] R. Sedgewick. “Implementing Quicksort programs.” *Communications of the ACM*, vol. 21(10), pp. 847–857, 1978.

[8] K. Mehlhorn. “Data Structures and Algorithms,” in *EATCS Monographs on Theoretical Computer Science*, vol. 1. Sortzng and Searchzng, 1984.

[9] R.L. Wainwright. “A class of sorting algorithms based on Quicksort.” *Communications of the ACM*, vol. 28(4), pp. 396-402, April 1985.

[10] R.L. Wainright. “Quicksort algorithms with an early exit for sorted subfiles.” *Communications of the ACM*, pp. 183-190, 1987.

[11] J. Bentley. “Programming Pearl: How to sort.” *Communications of the ACM*, vol. 27(4), 1984.

[12] D.R. Musser. *Introspective Sorting and Selection Algorithms. Software: Practice and Experience*. Wiley, Vol. 27(8), 1997, pp. 983–993.

[13] R. Sedgewick. *Algorithms in C++*. Addison Wesley, 1998.

[14] C.A.R. Hoare. “Quicksort.” *Computer Journal*, vol. 5, pp. 10–15, 1962.

[15] CORPORATE Tech Correspondence, Technical correspondence. *Communications of the ACM*, vol. 29(4), pp. 331-335, Apr. 1986.

[16] L. Khreisat. “QuickSort: A Historical Perspective and Empirical Study.” *International Journal of Computer Science and Network Security*, vol. 7(12), Dec. 2007.

[17] M. Matsumoto, T. Nishimura. “Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator.” ACM Trans. on Modeling and Computer Simulation, vol. 8(1), pp.3-30, Jan. 1998.