# QuickSort
# A Historical Perspective and Empirical Study

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

***Abstract*—** In this paper a comprehensive survey and empirical study of the Quicksort algorithm is provided. The survey examines in detail all the different variations of Quicksort starting with the original version developed by Hoare in 1961 and ending with some of the most recent ones. The paper also investigates some new sorting algorithms and compares their performances to the various versions of Quicksort. The study compared each algorithm in terms of the number of comparisons performed and the running times when used for sorting arrays of integers that were already sorted, sorted in reverse order, and generated randomly.

***Keywords*—**Empirical**,** Quicksort, Sorting, Survey.

## 1. Introduction

The Quicksort algorithm developed by Hoare [9] is one of the most efficient internal sorting algorithms and is the method of choice for many applications. The algorithm is easy to implement, works very well for different types of input data, and is known to use fewer resources than any other sorting algorithm [22]. All these factors have made it very popular. Quicksort is a divide-and-conquer algorithm. To sort an array A of elements, it partitions the array into two parts, placing small elements on the left and large elements on the right, and then recursively sorts the two subarrays. Sedgewick studied Quicksort in his Ph.D. thesis [19] and it is widely described and studied in [12], [5], [6], [20] and [24].

In addition to Quicksort, the paper also examines two new sorting algorithms and compares their performances to the different versions of Quicksort. Previous surveys only studied select variations of the algorithm, and used them for sorting small sized arrays, so this work will prove to be invaluable to anyone interested in studying and understanding the algorithm and it's different versions.

Since its development in 1961 by Hoare, the Quicksort algorithm has experienced a series of modifications aimed at improving the $O(n^2)$ worst case behavior. The improvements can be divided into four categories: improvements on the choice of pivot, algorithms that use another sorting algorithm for sorting sublists of certain smaller sizes , different ways of partitioning lists and sublists, and adaptive sorting that tries to improve on the $O(n^2)$ behavior of the Quicksort algorithm when used for sorting lists that are sorted or nearly sorted. This fourth category was proposed as a research area by [27]. The Quicksort versions that fall into the first category include Hoare's original Quicksort algorithm which uses a random pivot [9], [10], Scowen's Quickersort algorithm developed in 1965, which chooses the pivot as the middle element of the list to be sorted [21]. Also included is Singleton's algorithm which chooses the pivot using the median-of-three method [23].

The second category includes all algorithms that use another sorting algorithm, normally Insertion sort [22].for sorting small sublists. [9] was the first to suggest this method for improving the performance of Quicksort. [24] suggested a technique for small sublists, whereby sublists of sizes < M should be ignored and not partitioned. After the algorithm finishes, the list will be nearly sorted, and the entire list is sorted using Insertion sort. According to Sedgewick the best value for M is between 6 and 15.

The third improvement is achieved by considering different partitioning schemes. [24] suggested a scheme that uses two approaching indices. Bentley [3] proposed a scheme where two indices start at the left end of the list/sublist and move towards the right end. This scheme is based on work by Nico Lomuto of Alsys Inc. Another variation that falls into this category is one that uses three-way partitioning instead of two-way partitioning, first suggest by [22] as a way for handling duplicate keys in sublists. [33] sorts an array of numbers by finding a pivot (using any strategy) and then recursively applies a "partial quicksort" technique to the sub-arrays. So if the pivot is smaller than m, the left sub-array ,is sorted and partial quicksort is applied to the right sub-array. If the pivot is

Manuscript received December 5, 2007
Manuscript revised December 20, 2007

greater than m, then partial quicksort' is applied to the left sub-array.

The most recent category, adaptive sorting, deals with algorithms that try to improve on the worst case behavior of Quicksort when the list of elements is sorted or nearly sorted. Adaptive sorting algorithms are algorithms that take into consideration the already existing order in the input list [15]. Insertion sort is an adaptive sorting algorithm. [26] developed Bsort, an adaptive sorting algorithm, designed to improve the average behavior of Quicksort and eliminate the worst case behavior for sorted or nearly sorted lists. Another algorithm, Qsorte, also developed by [27], performs as well as Quicksort for lists of random values, and breaks the worst case behavior of Quicksort by performing O(n) comparisons for sorted or nearly sorted lists.

The rest of the paper is organized as follows: in section II the sorting algorithms to be studied are presented by providing a detailed description of each, in section III the new sorting algorithms are described. The results are presented in section IV.

## 2. Sorting Algorithms

The original Quicksort algorithm was developed by Hoare in 1961 [9]. It is an in-place algorithm (uses a small auxiliary stack), and has an average sorting time proportional to $O(n \log_2 n)$ to sort n items. It is considered to be the most efficient internal sorting algorithm. The algorithm has been analyzed and studied extensively in [12], [5], [6], [20], [24], and [19]. The only drawback of the algorithm is its worst case time complexity of $O(n^2)$, which occurs when the list of values is already sorted or nearly sorted, or sorted in reverse order [26]. Quicksort is a divide-and-conquer algorithm. To sort a list of n values represented by a one dimensional array A indexed from 1 to n, the algorithm chooses a key called the pivot and then partitions the array into two parts, a left subarray and a right subarray. The keys in the array will be reordered such that all the elements in the left subarray are less than the pivot and all the elements in the right subarray are greater than the pivot. Then the algorithms proceeds to sort each subarray independently. The efficiency of Quicksort ultimately depends on the choice of the pivot [22]. The ideal choice for the pivot would a value that divides the list of keys near the middle. Different choices for the pivot result in different variations of the Quicksort algorithm. In Hoare's [9] original algorithm the pivot was chosen at random, and Hoare proved that choosing the pivot at random will result in 1.386nlgn expected comparisons [10].

```
void quicksort(int A[ ], int L, int R)
{
  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 Quicksort is one in which the pivot is chosen as the first key in the list. This version is often found in algorithms textbooks. The following is a C++ implementation of the partition function, where the pivot is chosen as the first element in the array:

```
  void partition( int low, int high, int& pivotpoint)
  {
        int i , j;
        int pivotitem;

        pivotitem = S[low];      // choose first item for
pivotitem.
        j = low;
        for ( i = low + 1; i <= high; i++){
            if ( S[ i ] < pivotitem) {
                 j++;
                 exchange(i,j );
            }
```

```
        }
        pivotpoint = j;
        //Put pivotitem at pivotpoint
        exchange(low, pivotpoint);
    }
```

In 1965 Scowen [18] developed a Quicksort algorithm called Quickersort in which the pivot is chosen as the middle key in the array of keys to be sorted. If the array of keys is already sorted or nearly sorted, then the middle key will be an excellent choice since it will split the array into two subarrays of equal size.

```
void partition(int A[ ],  int i, int j)
{
   int pivot =  A[(i+j)/2];
   int p = i – 1;
   int q = j + 1;

   for(;;)
   {
      do  q = q – 1; while A[q] > pivot;
      do  p = p – 1; while A[p] < pivot;
      if (p < q)
         exchange (A[p], A[q]);
      else
         return q;
   }
}
```

Choosing the pivot as the middle key, the running time on sorted arrays becomes $O(n \log_2 n)$ because the array and subarrays will always be partitioned evenly.

Another improvement to Quicksort was introduced by Singleton in 1969 [23], in which he suggested the median-of-three method for choosing the pivot. One way of choosing the pivot would be to select the three elements from the left, middle and right of the array. The three elements are then sorted and placed back into the same positions in the array. The pivot is the median of these three elements. The median-of-three method improves Quicksort in three ways [22]: First, the worst case is much more unlikely to occur. Second, it eliminates the need for a sentinel key for partitioning, since this function is served by one of the three elements that are examined before partitioning. Third, it reduces the average running time of the algorithm by about 5%. A Quicksort algorithm that partitions around a median that is computed in cn comparisons sorts n elements in cn lg n + O(n) comparisons [5]. In [25] a worst-case algorithm is given that establishes the constant c = 3, while [30] gives an expected-time algorithm that establishes the constant c = 3/2. The following is a C++ implementation for the Quicksort algorithm that uses the median-of-three method for choosing the pivot:

```
void Singleton( long unsigned a[], int left, int right )
{
    int i, j;
   long unsigned pivot;
   if (left + CUTOFF <= right) {    /* CUTOFF = 20 */
        pivot = median3( a, left, right );
        i = left; j = right-1;
        for (;;) {
            while (a[++i] < pivot);
            while (a[--j] > pivot);
           if (i < j)
               swap( i, j );
      else
            break;
   }
   swap( i, right-1 );
   q_sort( a, left, i-1 );
   q_sort( a, i+1, right);
 }
}
```

Quicksort with three way partitioning has been suggested as the method for handling arrays with duplicate keys [22]. Basically, the array of keys is partitioned into three parts: one part contains keys smaller than the pivot, the second part contains keys equal to the pivot, and the third part contains all keys that are larger than the pivot. Let S be the array of keys to be sorted. The partitioning process would split the array S into three parts: keys smaller than the pivot S[1],…,S[j]; keys equal to the pivot S[j+1],…S[i-1]; and keys larger than the pivot S[i],…,S[r]. After the three way partitioning, the sort is completed after two recursive calls. The following is a C++ implementation of the algorithm based on the algorithm provided in [22]:

```
void Median3way (int low, int high)
{
   int k;
   int l = low;
   int r = high;
   long unsigned v = S[r];
   if ( r <= l) return;
   int i = l-1, j = r, p = l-1, q = r;
   for(;;){
      while (S[++i] < v);
      while (v < S[--j]) if (j == l) break;
      if (i >= j) break;
      exchange(i,j);
      if (S[i] == v) { p++; exchange(p, i);}
      if (v == S[j]) {q--; exchange(q, j);}
```

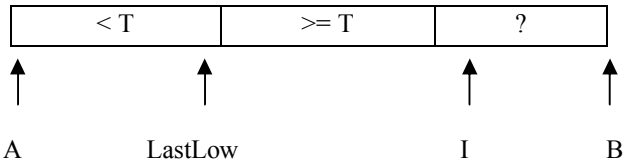
```
    }
    exchange(i, r);
    j = i-1;
    i = i+1;
    for (k = l; k <= p; k++, j--)
       exchange(k, j);
    for (k = r-1; k >= q; k--, i++)
       exchange(k, i);
    quicksort(l, j);
    quicksort(i, r)
}
```

[24] suggested a version of Quicksort that minimizes the number of swaps by scanning in from the left of the array then scanning in from the right of the array then swapping the two numbers found to be out of position. This algorithm is called *SedgewickFast* in this paper. This was declared to be a fast implementation of Quicksort. In fact, results in this paper show that for random data and data sorted in reverse, the algorithm makes $O(n \log_2 n)$ comparisons, while for sorted data the number of comparisons is linear. [14] provides a Pascal implementation for the algorithm. The following is a C++ implementation of the partition algorithm:

```
void Partition(int low, int high, int& pos)
{
  int i, j;
  int pivot;

  i = low-1;
  j = high;
  pivot = ar[high];
  for(;;)
  {
     while (ar[++i] < pivot);
     while (pivot < ar[--j]) if (j == low) break;
     if (i >= j) break;
     swap(i, j);
  }
  swap(i, high);
  pos = i;
}
```

Bentley [3] developed a version of Quicksort based on an algorithm suggested by Nico Lomuto, in which the partition function uses a for loop to roll the largest keys in the array to the bottom of the array. To partition the array around the pivot T (chosen at random) a simple scheme attributed to Nico Lomuto of Alsys, Inc is used. Given the pivot T, an index called LastLow is computed and used to rearrange the array X[A]…X[B] such that all keys less than T are on one side of LastLow, while all other keys are on the other side. Their original implementation achieves this using a simple 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 . However, if X[I]<T, the invariant is regained by incrementing LastLow by 1 and then swapping X[I] and X[Lastlow]. The original algorithm was implemented in Pearl. The algorithm will be referred to as Nico in the paper. The following is a C++ implementation [14] 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;
}
```

**Bsort** is a sorting algorithm developed by [26] which is a variation of Quicksort. It is designed for nearly sorted lists as well as lists that are nearly sorted in reverse order. The author claimed that it requires $O(n \log_2 n)$ comparisons for all distribution of keys. This claim has been disproved in [32] where examples are given that show that the algorithm exhibits $O(n^2)$ behavior. The author also claimed that for lists that are sorted or sorted in reverse order, the algorithm makes $O(n)$ comparisons. The author later corrected his statements in [32]. Our empirical study shows that for data sorted in reverse order, the algorithm makes $O(n \log_2 n)$ comparisons. The algorithm combines the

interchange technique used in Bubble sort with the Quicksort algorithm. The algorithm chooses the middle key as the pivot during each pass, then it proceeds to use the traditional Quicksort method. 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. This process ensures that at any point during the execution of the algorithm, 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. The original algorithm in [26] provided an implementation of the algorithm in Pascal. In this paper the algorithm was implemented in C++. The original Bsort algorithm [26] was implemented in Pascal.

**Qsorte** is a quicksort algorithm with an early exit for sorted arrays developed by Wainright in 1987 [27]. It is based on the original Quicksort algorithm with a slight modification in the partition phase, where the left and right sublists are checked to see if they are sorted or not. The algorithm chooses the middle key as the pivot in the partitioning phase. Initially, the left and right sublists are assumed to be sorted. When a new key is placed in the left sublist, and the sublist is still sorted, then if the sublist is not empty, the new key will be compared with its left neighbor. If the two keys are not in sorted order then the sublist is marked as unsorted, and the keys are not swapped. Similarly, when a new key is placed in the right sublist, and the sublist is still sorted, then if the sublist is not empty, the new key will be compared with its right neighbor. If the two keys are not in sorted order then the sublist is marked as unsorted, and the keys are not swapped. At the end of the partitioning phase, any sublist that is marked as sorted will not be partitioned. Qsorte still has a worst case time complexity of $O(n^2)$. This occurs when the chosen pivot is always the smallest value in the sublist. Qsorte will repeatedly partition the sublist into two sublists where one of the sublists only contains one key. The original Qsorte algorithm [27] was implemented in Pascal. The following is a C++ implementation of the algorithm:

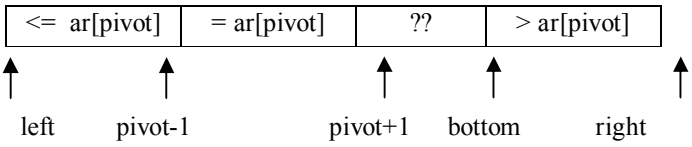
```
void Qsorte (int m, int n)
{
   int k, v;
   bool lsorted, rsorted;
   if ( m < n ){
      FindPivot (m, n, v);
      Partition (m, n, k, lsorted, rsorted);
      if (! lsorted)   Qsorte(m, k-1);
      if (! Rsorted) Qsorte(k, n);
   }
}
```

McDaniel [14] provided variations on Hoare's original partition algorithm. In one version called the **Rotate** version, the pivot is compared with the value in the bottom'th position. If the pivot is less than that value, then the bottom index is decremented. Otherwise a rotate left operation is performed using the call Rotate_Left(bottom,pivot+1,pivot). McDaniel's original code was written in Pascal. The original version has been rewritten in C++. The codes for the Rotate_Left and partition functions follow:

```
void Rotate_Left(int a,int b,int c)
{
     int t;
     t = ar[a];
     ar[a] = ar[b];
     ar[b] = ar[c];
     ar[c] = t;
}

void Partition(int left,int right, int& pos)
{
   int pivot, bottom;
   pivot = left;
   bottom = right;
   while (pivot < bottom)
   {
     if (ar[pivot] > ar[bottom])
     {
        Rotate_Left(bottom,pivot+1,pivot);
       pivot = pivot + 1;
     }
     else
       bottom = bottom - 1;
  }
  pos = pivot;
}
```

After the index bottom is decremented the following assertions are true:



## 3. New Sorting Algorithms Based on Quicsort

### 3.1 qsort7

Benteley and Mcilroy [4] developed a fast variant of Quicksort (qsort7) based on the existing qsort function that comes with the C library. The choice of the pivot is determined by the size of the array. For small sized arrays (size = 7) the pivot is chosen as the middle key, for mid-sized arrays the pivot is chosen using the median-of-three method, and finally for large sized arrays the pivot is chosen as the pseudomedian of 9. Their algorithm uses a fat partition that divides the input array into three parts (Tripartite partitioning):

| < | = | > |
|---|---|---|

After partitioning, recur on the two subarrays at the left and right ends, and ignore the equal elements in the middle. Tripartite partitioning is equivalent to Dijkstra's 'Dutch National Flag' problem [7].To make their algorithm more efficient a better fat partition version is used as shown below:

| = | < | ? | > |
|---|---|---|---|

After partitioning, the equal keys are brought to the middle by swapping the outer ends of the two left portions. A symmetric version of this partitioning process is used in their algorithm as shown below:

| = | < | ? | > | = |
|---|---|---|---|---|

a b c d

The main partitioning loop has two inner loops. The first inner loop moves up the index b. It scans over lesser elements, swapping equal elements to the element pointed to by a. The scanning stops at a greater element. The second inner loop moves down the index c in a similar manner. It scans over greater elements, swapping equal elements to the element pointed to by d. The scanning stops at a lesser element. The main loop then swaps the elements pointed to by b and c, increments b and decrements c, and continues until b and c cross paths. Afterwards, the equal keys on the edges are swapped back to the middle of the array. The pivot is chosen using the median-of-three method. The algorithm combines split-end partitioning with an adaptively sampled partitioning element. Speed up and portability is achieved by using Insertion sort to sort small arrays (7 keys). The C++ implementation of their algorithm has been adapted by [11].

### 3.2 FlashSort

In 1997 Neubert developed a new sorting algorithm, Flash sort [17], that is based on the classification of elements [29], [28], [20] instead of comparisons. In [8] it is reported that classification based sorting algorithms require O(n) time to sort n elements thereby achieving the absolute lowest time complexity for sorting n elements, and that the only disadvantage of classification based sorting algorithms is the fact that they require considerable auxiliary memory space. Flashsort requires less than 0.1n auxiliary memory to sort n elements. This is achieved by using a classification step to do the long-range ordering with in-place permutation, then the algorithm uses a simple comparison method for the final short-range ordering of each class.

The algorithm consists of three stages: classification, permutation, and straight insertion. The Classification stage determines the size of each class of elements. In the Permutation stage long-range reordering is performed to collect elements of each class together, and in the straight insertion stage final short-range ordering is done. The elements to be sorted are assumed to be stored in an array A indexed from 0 to n-1. The algorithm uses a vector L of length M called the class pointer vector. In the classification stage the elements of the array A are counted according to their key for each of the m classes. After the completion of the L VECTOR, each L[k] is equal to the cumulative number of elements A[i] in all the classes 0 through k. The last element in L. namely, L[m-1] is equal to n-1. Flashsort assumes that the elements A[i] are about evenly distributed, therefore the approximate final position of an element can be directly computed from the element value, with no comparisons. If the maximal element is Amax and the minimal element is Amin, the class of element A[i] can be computed as follows:

$$K(A[i]) = 1 + Int((m-1)(A[i] - A\min)/(A\max - A\min))$$

where K(A[i]) is a value between 1 and m. On average, there will approximately be n/m elements in each class. The class K = m only has elements equal to Amax, and the remaining classes are slightly larger.

In the first stage of the algorithm, classification, the actual number of elements per class is computed by scanning the input. The vector L is used to store the class information. Initially, each L(K) indicates the upper end of the section that will contain the elements in class K. For example, L[1] is the number of elements in class 1, and L[m] is equal to n. The scanning process takes time O(n). After the scanning process, the class sizes are added to determine the initial L[K] values, this process takes time O(m). The classification

stage is followed by the permutation stage. In this stage the elements are moved into the correct class. To accomplish this, the class index K for element A[i] is computed and A[i] is placed in L[K], and L[K] is decremented. This process ends when the first class has been filled up. Each item is moved exactly once. This process takes time O(n). The end result of the permutation stage is a partially sorted array. Straight Insertion sort is used to sort this array, and assuming that the class sizes are approximately the same, this sort will take O((n/m)2) time for each class. Neubert compared the running times of Flash sort and Quickersort using sequences of uniform random numbers. His results indicate that the running time of Flash sort increases linearly with n. Flash sort is faster than Quicker sort for m=0.1n and n > 80. However, in this study it is sown that Flashsort is not always linear. In fact, when used for sorting random data and data sorted in reverse, the algorithm made $O(n^2)$ comparisons. Only when used for sorting data that was already sorted, did it exhibit linear behavior. In [17] the algorithm was only used for sorting arrays of a maximum size of 10,000. In this work, sizes ranged from 3000, up to 500,000. The original implementation of Flashsort was in Fortran.

### 3.3 Algorithm SS06

[31] suggests a new sorting algorithm based on Quicksort which uses an auxiliary array for holding array keys during sort. No analysis or testing was done by the authors for the algorithm. The results in this paper show that SS07 requires $O(n^2)$ comparisons to sort n elements that are already sorted or sorted in reverse order. For random data it made $O(n \log_2 n)$ comparisons. The algorithm follows:

**Algorithm SS06:**

1. pivot = a [first]
2. Starting from the second element, compare it to the pivot element.
   2.1. if pivot < element then place the element in the last unfilled position of a temporary array (of same size as the original one).
   2.2. if pivot > element then place the element in the first unfilled position of the temporary array.
3. Repeat step 2 until last element of the array has been processed.
4. Finally place the pivot element in the blank position of the temporary array (remark: the blank position is created because one element of the original array was taken out as pivot)
5. partition the array into two subarrays, based on the pivot element's position.
6. Repeat steps 1-5 until the array is sorted..

This algorithm is basically Quicksort that uses an extra temporary array of the same size as the original array. Therefore, this algorithm is not an in-place algorithm.

## 4. Emprical Testing and Results

The performance of the sorting algorithms described in section II was studied by using these algorithms 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 Pentium M processor with a speed of 1500 MHz, and 512 MB of RAM. The sizes of the arrays ranged from N= 3,000 to N = 500,000 elements. To study the behavior of the algorithms on arrays of random elements, each algorithm was used to sort three sequences of random numbers of a specific size N, and the average running time and the average number of comparisons were calculated. The sequences of random numbers were generated using the Mersenne Twister random number generator [13]. The generator has a period of $2^{19937}-1$. The C++ implementation provided in [2] was used. Figs. 1 and 2 below show the running times for the sorting algorithms when used to sort arrays of random numbers of different sizes. It is evident that the sorting algorithm of Benteley and Mcilroy [4] (qsort7) is the fastest, giving the best performance for sorting arrays of random integers. Quickersort and Qsorte are comparable for values of N ranging from 3000, up to 200,000. For values of N greater than 200,000, Quickersort becomes slightly slower than Qsorte. Qsorte outperformed both Hoare's algorithm and QuickFirst for N>=9000. The new sorting algorithm SS07 developed by [31] is comparable to SedgewickFast, Singleton and Bsort, for values of N ranging from 3000 up to 200,000. Bsort is faster than Rotate, Nico, QuickFirst and Hoare for values of N ranging from 10,000 up to 500,000. QuickFirst is much slower than Hoare's algorithm, Nico and Rotate, see fig. 2. The worst performers for sorting random numbers were the Flashsort algorithm and Median3way, see fig. 3.

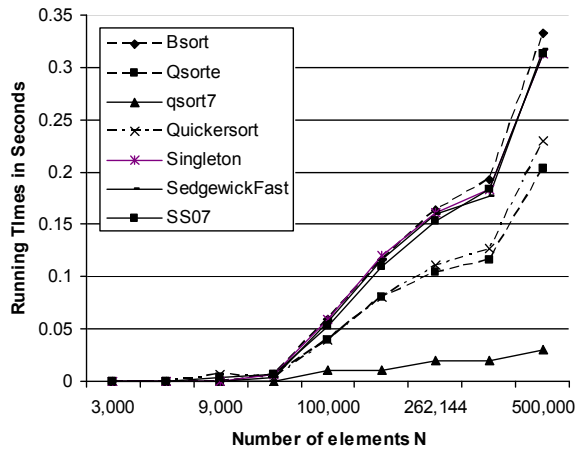



Fig. 1 Average Running Times for Random Data

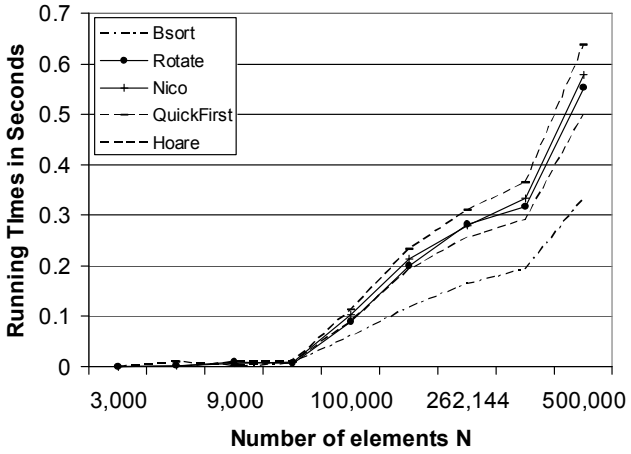



Fig. 2 Average Running Times for Random Data

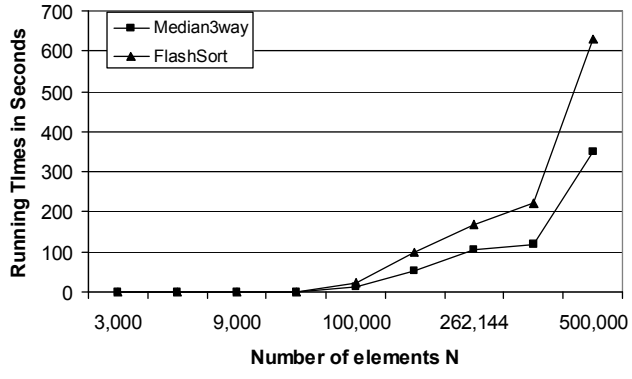



Fig. 3 Average Running Times for Random Data

Looking at the number of comparisons, Flashsort required $O(N^2)$ ($0.25\,N^2$ exactly) comparisons to sort N integers, see fig. 4 below. This contradicts the results in [17] which show that the running time of Flashsort increases linearly with N. The results in [17] were computed for N<=10,000. By looking at the number of comparisons for values of N larger than 10,000, it is evident that Flashsort increases quadratically in N when used for sorting arrays of random numbers.

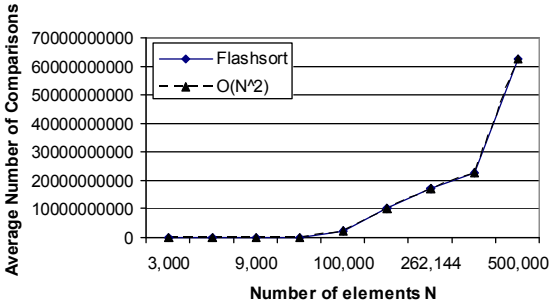



Fig. 4 Average Number of Comparisons for Random Data

The number of comparisons performed by Bsort and Hoare were of order $O(N \log_2 N)$, with Bsort requiring more comparisons than Hoare at an average rate of 1.6, see fig. 5.

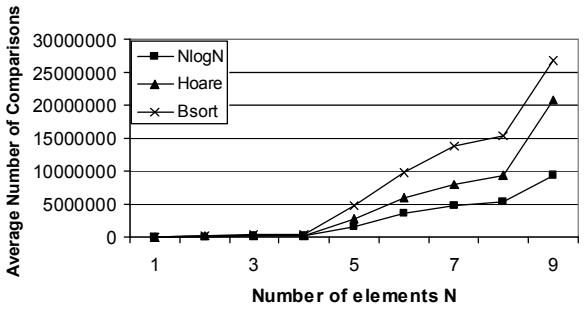



Fig. 5. Average Number of Comparisons for Random Data.

For Median3way, Nico and Qsorte the number of comparisons are comparable and are of order $O(N \log_2 N)$, and they are collectively better than Bsort and Hoare, see fig. 6.

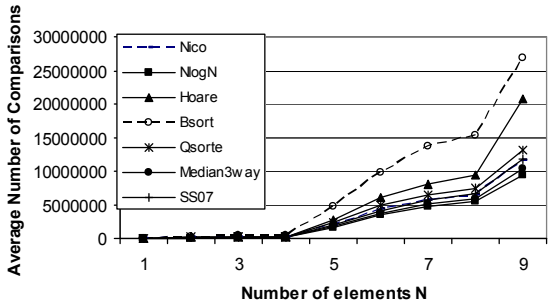



Fig. 6. Average Number of Comparisons for Random Data

qsort7 requires the smallest number of comparisons among all the algorithms and is of order $O(N \log_2 N)$, this is followed by SedgewickFast and Singleton which also exhibit $O(N \log_2 N)$ behavior, and are both better than Median3way, see fig. 7.

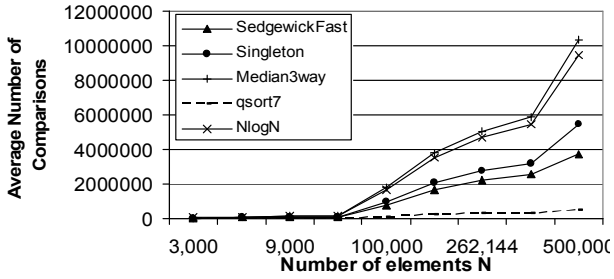



Fig. 7 Average Number of Comparisons for Random Data

From fig. 8 it is apparent that QuickFirst, SS07, Nico, Rotate and Quickersort have very similar performance in terms of the number of comparisons and are all of order $O(N \log_2 N)$. They are collectively better than Qsorte.

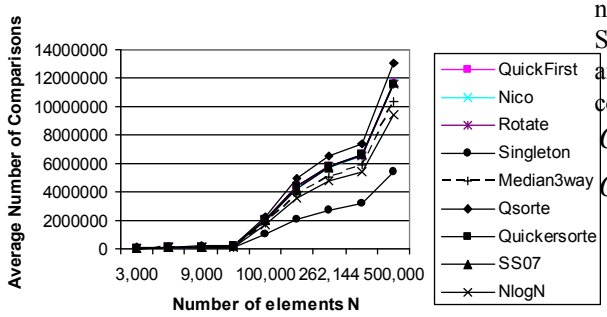



Fig. 8 Average Number of Comparisons for Random Data

For data that is already sorted, Qsorte and Bsort have the same running times and proved to be the fastest. Flashsort is faster than Median3way, Singleton, qsort7, and Quickersort which are all comparable for values of N between 3,000 and 300,000. and faster than Hoare's algorithm, which in turn is faster than Nico, see fig. 9. The worst performer was SS07 resulting in stack overflow for values of N >= 10,000. SedgewickFast, Rotate and QuickFirst were much slower than Nico and all resulted in stack overflow for values of N>=100,000, see fig. 10. The running time for Median3way is on average 4% less than the running time of Hoare, while Singleton resulted in an average reduction of approximately 5% and qsort7 gave an average reduction of 6% compared to Hoare. In terms of the number of comparisons, Bsort and Qsorte required the smallest number of comparisons which is of order O(N). This is the expected behavior of Bsort and Qsorte for sorted lists. Singleton and Flashsort also exhibit linear behavior and require more comparisons than Bsort and Qsorte, see fig. 11. Quickersort and qsort7 are comparable and have an order of $O(N \log_2 N)$. Median3way requires fewer comparisons than both Quickersort and qsort7 and is also of order $O(N \log_2 N)$.

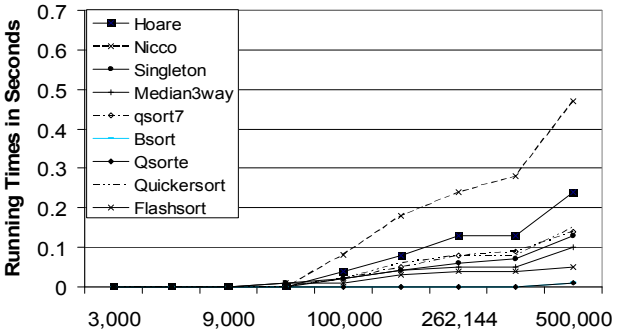



Fig. 9. Running times for Sorted Data

.SedgewickFast and Bsort are very similar in terms of the number of comparisons for N <= 10,000. For N >10,000, SedgewickFast results in stack overflow. Comparing Nico and Hoare, it is apparent that Nico requires fewer comparisons than Hoare and both are of order $O(N \log_2 N)$. QuickFirst, Rotate and SS07 are of order $O(N^2)$.

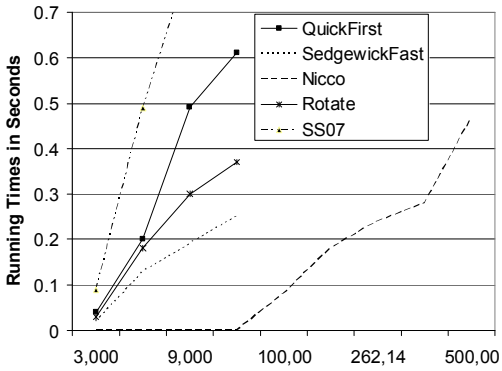



Fig. 10. Running times for Sorted Data

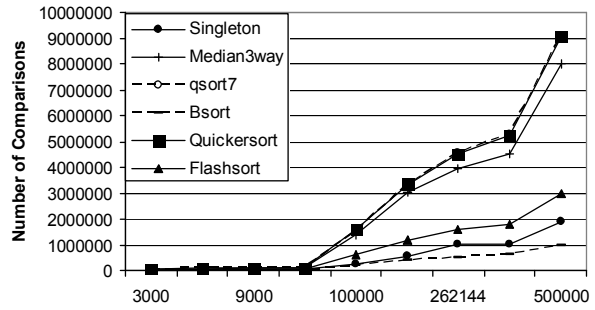



Fig. 11. Number of Comparisons for Sorted Data

For data sorted in reverse order, the fastest running times were achieved by both Bsort and Qsorte. Quickersort is faster than qsort7 which in turn is faster than Singleton for N>=100,000. Bsort, Qsorte and Quickersort are all faster than Hoare for N >= 9000. Nico is on average 2.6 times slower than Hoare for N >= 100,000. SedgewickFast and Rotate are much slower than Nico and both result in stack overflow for N >= 100,000. SS07 resulted in stack overflow for N>= 10,000 and was much slower than Rotate for N between 3000 and 9000, and faster than QuickFirst which also resulted in stack overflow for N >= 9000. See figs 12, 13, 14.

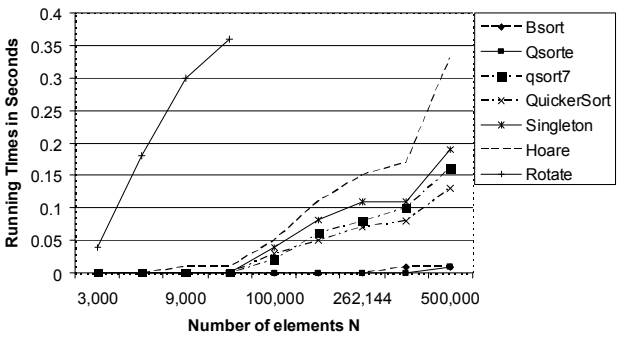



Fig. 12. Running times for Data Sorted in Reverse.

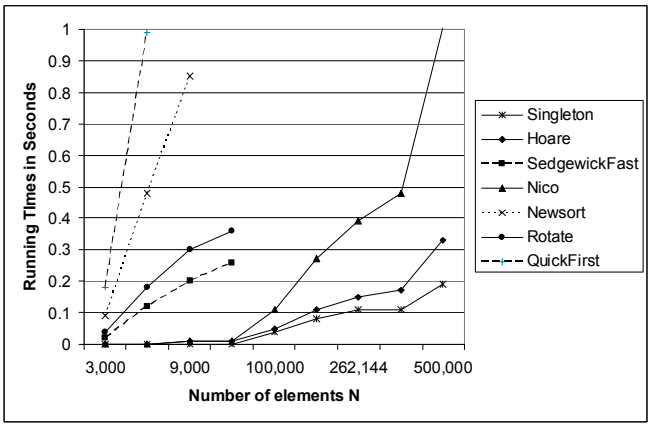



Fig. 13. Running times for Data Sorted in Reverse.

The worst performers were Median3way and Flashsort which had very similar running times, see fig. 14. Qsorte performed the smallest number of comparisons followed by Bsort, and then Singleton. For N <= 10,000 the number of comparisons performed by SedgewickFast is very close to the number of comparisons done by Qsorte, however, for N > 10,000 SedgewickFast results in stack overflow, see fig. 15 Quickersort and qsort7 are comparable. All of the aforementioned algorithms are of order $O(N \log_2 N)$.

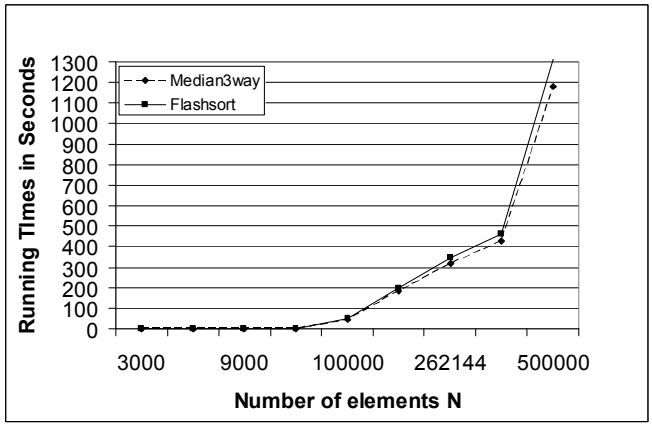



Fig. 14. Running times for Data Sorted in Reverse.

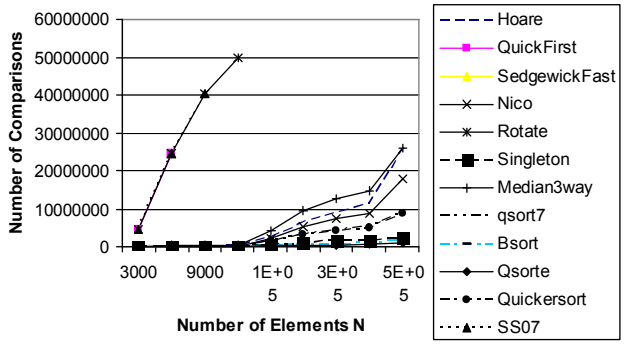



Fig. 15. Number of Comparisons for Data Sorted in Reverse.

Rotate and SS07 perform the same number of comparisons, with Rotate resulting in stack overflow for N> 10,000, while SS07 results in stack overflow for N > 9000. The number of comparisons performed by Rotate and SS07 is of order $O(N^2)$. Flashsort is very similar to Rotate and SS07 before they result in stack overflow. The number of comparisons performed by Flashsort is also of order $O(N^2)$, which contradicts the claim made by the author in [17] that the algorithm is linear.

## References


[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data structures and algorithms.* Reading, Mass: .Addison-Wesley, 1983.

[2] www.agner.org/random/theory/.

[3] J. Bentley, "Programming Pearl: How to sort", *Com.. ACM,* Vol. 27 Issue 4, April 1984.

[4] J. L. BENTLEY, M. D. McILROY, "Engineering a Sort Function" *SOFTWARE—PRACTICE AND EXPERIENCE,* Vol. 23(11), Nov. 1993, pp 249 – 1265 .

[5] J. L. Bentley and R. Sedgewick, "Fast algorithms for sorting and searching strings", *In Proc. 8th annual ACM-SIAM symposium on Discrete algorithms,* New Orleans, Louisiana, USA, 1997, pp 360 - 369 .

[6] R. Chaudhuri and A. C. Dempster, "A note on slowing Quicksort", *SIGCSE* Vol . 25, No . 2, Jane 1993.

[7] E. W. Dijkstra, *A discipline of programming.,* Englewood Cliffs, NJ Prentice-Hall, 1976.

[8] W. Dobosiewicz, "Sorting by distributive partitioning," *Information Processing Letters* 7, 1 – 5., 1978.

[9] C.A.R. Hoare, "Algorithm 64: Quicksort," *Comm. ACM* 4, 7 , 321, July 1961.

[10] C. A. R. Hoare, "Quicksort," *Computer Journal*, 5, pp 10 - 15 1962.

[11] http://lampsvn.epfl.ch/svn-repos/scala/scala/tags/R_2_5_0 RC1/src/library/scala/util/Sorting.scala.

[12] R. Loeser, "Some performance tests of :quicksort: and descendants," *Comm. ACM* 17, 3 , pp 143 – 152, Mar. 1974.

[13] M. Matsumoto, and T. Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator," *ACM Trans. on Modeling and Computer Simulation* Vol. 8, No. 1, January pp.3-30, 1998.

[14] B. McDaniel, "Variations on Put First," *Conference on Applied Mathematics* , University of Central Oklahoma, Spring 1991.

[15] K. Mehlhorn, *Data Structures and Algorithms*, Vol. 1, Sortzng and Searchzng, 1984 EATCS Monographs on Theoretical Computer Science, Berlin/Heidelberg: Springer-Verlag.

[16] D. Motzkin, "Meansort," *Comm. ACM* 26, 4, pp 250-251, Apr. 1983.

[17] K.D. Neubert, "The FlashSort algorithm," *In Proc. of the euroFORTH'97 –Conf.*, Oxford, England, Sept. pp 26 – 28, 1997.

[18] R.S. Scowen, "Algorithm 271: Quickersort," Comm. ACM 8, 11, pp 669-670, Nov. 1965.

[19] R. Sedgewick, "Quicksort," PhD dissertation, Stanford University, Stanford, CA, May 1975. Stanford Computer Science Report STAN-CS-75-492.

[20] R. Sedgewick, "The Analysis of Quicksort Programs," *Acta Informatica* 7, pp 327 – 355,, 1977.

[21] R. S. Scowen, "Algorithm 271: quickersort," *Comm. of the ACM*, 8, pp 669 – 670, 1965.

[22] R. Sedgewick, *Algorithms in C++*, 3rd edition, Addison Wesley, 1998.

[23] R. C. Singleton, "Algorithm 347: An efficient algorithm for sorting with minimal storage," *Comm. ACM* 12, 3, pp 186-187, Mar. 1969.

[24] R. Sedgewick, "Implementing Quicksort programs," *Comm. of ACM*, 21(10), pp 847 – 857, Oct. 1978.

[25] A.M. Schoenhage, M. Paterson, and N. Pippenger, "Finding the median," *Journal of Computer and Systems Sciences* 13, pp 184 - 199, 1976.

[26] R. L. Wainwright, "A class of sorting algorithms based on Quicksort," *Comm. ACM,* Vol. 28 Number 4, April 1985.

[27] R. L Wainright, "Quicksort algorithms with an early exit for sorted subfiles," *Comm. ACM*, 1987.

[28] N. Wirth, *Algorithm und Datenstrukturen*, B. G. Teubner, 1983 .

[29] D. E. Knuth, *The Art of Computer Programming*, Vol. 3: Sorting and Searching, Addison Wesley Publ. Co., 1973.
[30] R.W. Floyd, and R. L Rivest, "Expected time bounds for selection," *Comm. of the ACM* 18, 3, pp 165 - 172., March 1975.
[31] K. K. Sundararajan, and S. Chakraborty, " A new sorting algorithm", *InterStat, Statistics on the Internet*, 2006.
[32] Technical Correspondence, Comm. ACM, Vol. 29, No. 4, April 1986.
[33] C. Martinez, Partial quicksort. In *Proceedings of the First ACM-SIAM Workshop on Analytic Algorithmics and Combinatorics (ANALCO),* 2004.