International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 04 Issue: 04 | April -2020 ISSN: 2582-3930

QuickSort- A Chronological and Experimental Survey

Nisha Rathi[1], Nidhi Nigam[2]

[1]*Department of Computer Science and Engineering, Acropolis Institute of Technology and Research*
[2]*Department of Computer Science and Engineering, Acropolis Institute of Technology and Research*

Abstract - Sorting is perhaps the widely studied operation on data in computer science, both because of its intrinsic theoretical importance and its use in so many applications. In this paper a complete review and experimental study of the Quicksort algorithm is provided. The survey examines in detail the variants of Quicksort starting with the original version developed by Hoare in 1961[6]. The paper compares the performances of the various versions of Quicksort based on comparison of execution time used for sorting arrays of various sizes of integers with unsorted and already sorted values.

Key Words: QuickSort, Median-of-Three Rule, Median-of-Five Rule, Dynamic Pivot Selection, Sorting, Survey, Bsort, Qsorte, qsort7, Quickersort, Singleton, SedgewickFast.

1. INTRODUCTION

QuickSort was first introduced in 1961 by Hoare [6]. It is an in-place algorithm (uses a small auxiliary stack), and has an average sorting time of O (n log 2 n) to sort n items. It is considered to be the most efficient internal sorting algorithm and is the method of choice for many applications. The algorithm is simple to put into practice, works very well for different types of input data, and is known to use less resource than any other sorting algorithm [16]. All these factors have made it widely accepted sorting algorithm. Quicksort is a divide-and-conquer algorithm. For sorting, 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 [13] and it is extensively explained and studied in [8], [3], [4], [14] and [18].

2. Enhancements in Basic Quicksort algorithm

Since its development by Hoare, the Quicksort algorithm has gone through a series of amendments intended to improve its worst case behavior of O (n^2). The enhancements can be divided into four major categories: enhancements based on the choice of pivot, enhancements based on use of another sorting algorithm by an algorithm, enhancements based on different ways of partitioning lists and sublists, and enhancements based on 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 last category was proposed as a research area by [20].

I. Enhancement based on the choice of pivot: It includes Hoare's original Quicksort algorithm based on random pivot [6], [7], Scowen's Quickersort algorithm where middle element of the list to be sorted is considered as pivot [15], and Singleton's algorithm where pivot selection is based on the median-of-three method [17].

Sedgewick [17] suggests the Median-of-Three splitting technique of random pivot selection which reduces the probability of occurrence for the worst case scenario. It proposes selecting the median of the values stored in the first, last and ((first + last) /2) indexes as a pivot. This decreases the likelihood of the worst-case state and increases the probability of the average-case performance of the algorithm. However, there is no assurance of dividing the array into equal parts. Thus, it may cause the worst case performance of QuickSort. The performance of the algorithm is responsive to small alteration for the array contents.

Janez B. [27] provides the Median-of-Five with random index selection technique by adding the values stored in two randomly picked indexes to the values stored in the first, last and ((first + last) /2) indexes. This technique may provide a more equal splitting; but it can still suffer from the same slowdowns of the prior techniques. Mohammed [28] suggests modification which aims to reduce the overhead associated with the random number generation picks the median of the five values stored in fixed indexes as the pivot; namely, first, ((first + last) /4), ((first + last) /2), (3 *(first + last) /4) and last . This technique reduces the overhead of the random number generation by using five fixed indexes, but it necessitates time to pick and choose the median of five elements at each recursive call. Mohammed [29] in 2007 mentioned Median-of-Seven and Median-of-Nine either with or without random index selection. This method of increasing in number of elements may provide a more unbiased split but the time needed to pick the pivot at each recursive call increases with the number of elements. The most important shortcoming of the earlier methods is that the selection of the pivot is based on a specific number of elements which does not necessarily reflect the nature of the array. The possibility of worst case behavior of the QuickSort algorithm is still there when using any of these pivot selection techniques.

An intelligent QuickSort algorithm based on a dynamic pivot selection technique was proposed by Dalhoum [30]. He enhanced the average case and eliminates the worst case behaviors of the algorithm. This technique is data-dependent to increase the chances of splitting the array or list into relatively equal sizes in order to reduce the number of recursive calls made for the Quicksort algorithm. In addition; the modified algorithm converts the worst case state into a best case state with Θ (n) execution time. The algorithm is sufficiently intelligent to distinguish a sorted array or sub-array so doesn't require further processing.

Another useful modification is the use of median-of-medians or Blum-Floyd-Pratt-Rivest-Tarjan (BFPRT)

algorithm [31], the pivot selection algorithm in the linear median finding algorithm.

II. Enhancements based on use of another sorting algorithm by an algorithm: It includes all algorithms that use another sorting algorithm for sorting small sublists, usually Insertion sort [16]. The concept was first suggested for improving the performance of Quicksort by Hoare ,. A more feasible technique for small sublists was proposed by Sedgewick [18], whereby sublists of sizes < M, where M is between 6 and 15, should be ignored and not partitioned. After the algorithm come to an end, the list will be almost sorted, and the whole list is sorted using Insertion sort.

III. Enhancement based on different ways of partitioning lists and sublists: It is accomplished by considering different partitioning schemes. A scheme was suggested by Sedgewick [18] that uses two approaching indices. Another scheme was proposed by Bentley [1] where two indices start at the left end of the list/sublist and move towards the right end. For handling duplicate keys in sublists, a method to use three-way partitioning instead of two-way partitioning was suggested by Sedgewick [16]. Martinez [24] in 2004 suggested to sorts an array of numbers by finding a pivot and then recursively apply a "partial Quicksort" technique to the sub-arrays. In case of smaller value of pivot as compared to M, the partial Quicksort is applied to the right sub-array as the left sub-array is sorted. In case pivot is greater than M, the partial Quicksort is applied to the left sub-array.

IV. Enhancements based on adaptive sorting: It tries to improve on the worst case behavior, O (n^2), of the Quicksort algorithm when used for sorting lists that are sorted or nearly sorted. Adaptive sorting algorithms, like Insertion sort, considered the already existing order in the input list [10].

An adaptive sorting algorithm, called Bsort, was developed by Wainwright [19] to improve the average behavior of Quicksort and eliminate the worst case behavior for sorted or nearly sorted lists. Wainright [20] developed another adaptive sorting algorithm, Qsorte, which performs for lists of random values and breaks the worst case behavior of Quicksort by performing O(n) comparisons for sorted or nearly sorted lists.

This paper is organized as follows. Section 3 presents detailed description of few of the enhanced QuickSort algorithms. Sections 4 and 5 discuss behavioral analysis of few of the QuickSort algorithms. This paper concludes with section 6.

3. Sorting Algorithms

The original Quicksort algorithm was developed by Hoare in 1961 [6]. It is considered to be the most efficient internal sorting algorithm. The major drawback of original Quicksort is that worst case time complexity of naïve implementation of Quicksort is O (n^2) with input size n. The algorithm has been analyzed and studied extensively in [8], [3], [4], [14], [18], and [13] research work.

QuickSort follows the technique of divide and conquer by recursively splitting each array into two sub arrays, which makes it easier to solve smaller problems than a single larger one [25], [26]. In QuickSort, a pivot is selected from the unsorted array and used to split the array into two sub arrays for which the same algorithm is called recursively until the sub arrays have size one or zero. The QuickSort algorithm has an average runtime complexity of Θ (n log n) for an input size n, and a worst case complexity of Θ (n^2) . The worst case arises when the input is an already sorted list, thus, the selected pivot is always a largest element.
The QuickSort algorithm's runtime depends on the splitting of the array and the consecutive sub arrays. If splitting constantly results in a small reduction in the size of the array or sub array, the resultant runtime will be:

T(n) = n + T(n-c), where c is a constant. This recurrence relation results to

$$T(n) = \Theta(n^2) \quad (1)$$

If splitting of array results almost equal size sub-arrays, the runtime complexity of QuickSort will be reduced to:

T(n) = n + T(n/2).

Thereby results to logarithmic time complexity:

T(n) = n + T(n/2).

$$T(n) = \Theta(n \log n) \quad (2)$$

```
int Hoare-Partition (int a[], int beg, int end)
{
int pivot = a[beg];
int p = beg-1;
int q = end+1;
for (;;)
{
do {q= q-1;} while (a[q] > pivot);
do {p= p+1;} while (a[p] < pivot);
if (p < q)
swap (a[p], a[q]);
else
return q;
}
}
```

Fig -1: The Hoare-Partition Algorithm

The best performance of the QuickSort algorithm came by splitting the array into almost equal size subarrays. These almost equal halves reduce the number of recursive calls and eventually reduce the execution time. The efficiency of Quicksort ultimately depends on the choice of the pivot [16]. The perfect selection for the pivot would be the one that divides the list elements nearly to the half. Different variations of the Quicksort algorithm have aroused from diverse choices for the pivot. In Hoare's [6] original algorithm the pivot was chosen at random, and Hoare proved that choosing the pivot at random will result in 1.386nlgn expected comparisons [7].

```
int random_partition(int a[], int beg, int end)
{
   int pivotIdx = beg + rand() % (end-beg+1);        int pivot = a[pivotIdx];
   swap(a[pivotIdx], a[end]); // move pivot element to the end
   pivotIdx = end;      int i = beg -1;
   for(int j=beg; j<=end-1; j++)
   {
      if(a[j] <= pivot)
      {
         i = i+1;
         swap(a[i], a[j]);
   }   }
   swap(a[i+1], a[pivotIdx]);
   return i+1;
}

void random_quick_sort(int a[], int beg, int end)
{
   if(beg < end)

{
      int mid = random_partition(a, beg, end);
      random_quick_sort(a, beg, mid-1);
      random_quick_sort(a, mid+1, end);

 }}
```

Fig -2: **The Random--Partition Algorithm**

Another variation of Quicksort is one in which the pivot is chosen as the first or last key in the list. The following is a C++ implementation of the partition function, where the pivot is chosen as the last element in the array:

```
int pivotasLastPartition(int a[],int beg,int end)
 {  int pivot=a[end];   int j=beg;    int i=j-1;
                    while(j<end)
            {
                    if(a[j]<= pivot)
                    {
                            swap(a[++i],a[j]);
                    }
                     j++;
             }
            swap(a[++i],a[end]);
            return i;
      }
```

Fig -3: **The PivotasLastPartition Algorithm**

In 1965 Scowen [12] proposed Quickersort in which the pivot is selected as the middle key. If the array is already sorted or nearly sorted, then the middle key will be a brilliant choice since it will divide the array into two subarrays of equal size. As the array and subarrays will always be partitioned evenly on opting the pivot as the middle element, the running time on sorted arrays becomes $O(n \log_2 n.)$

```
int Scowenpartition(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)
swap (a[p], a[q]);
else
return q;
}}
void QS(int a[],int beg,int end)
 {
            if(beg<end){
            int pos=partition(a,beg,end);
 QS(a,beg,pos);
 QS(a,pos+1,end);
 }
```

Fig -4: **The ScowenPartition Algorithm**

Another improvement to Quicksort was introduced by Singleton in 1969 [17], in which he suggested the Median-of-Three Rule for choosing the pivot that improves Quicksort. The worst case is now more unlikely to take place; the Median-of-Three Rule is served by one of the three elements that are observed prior to partitioning so this new approach removes the need for a key to partition the array and it decrease the average running time by about 5%. This method selects the three elements from the left, middle and right of the array. The three elements are then sorted and positioned back into the same location in the array. The pivot is the median of these three elements.

```
int Medianof3_partition(int a[], int beg, int end, long pivot)
{
int begPtr = beg;           // end of first elem
int endPtr = end - 1;       // beg of pivot

while(true)
  {
  while( a[++begPtr] < pivot )  // find bigger
    ;                           //    (nop)
  while( a[--endPtr] > pivot ) // find smaller
    ;                           //    (nop)
  if(begPtr >= endPtr)      // if pointers cross,
    break;                  //    partition done
  else                      // not crossed, so
    swap(a,begPtr, endPtr);  // swap elements
  }  // end while(true)
swap(a,begPtr, end-1);        // restore pivot
return begPtr;               // return pivot location
}  // end partitionIt()
```

Fig -5: **The Medianof3_Partition Algorithm**

Sedgewick [16] suggested three-way partitioning as the method for handling arrays with duplicate keys. The array is partitioned into three parts: one containing keys smaller than the pivot, the other containing keys equal to the pivot, and the last part containing all keys that are larger than the pivot. The sort is completed after two recursive calls in the three-way partitioning,

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

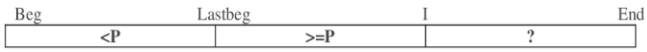
Fig -6: **The Median3way Algorithm**

Sedgewick [18] suggested a version of Quicksort, *SedgewickFast*, 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. The algorithm makes $O\ n\ (\log_2 n)$ comparisons for random data and data sorted in reverse while it is linear for sorted data.

```
int SedgewickFastPartition(int a[], int beg, int end)
{
int i, j;
int pivot;
i = beg-1;
j = end;
pivot = a[end];
for(;;)
{
while (a[++i] < pivot);
while (pivot < a[--j]) if (j == beg) break;
if (i >= j) break;
swap(a[i],a[j]);
}
swap(a[i], a[end]);
int pos = i;
return pos;
}
```

Fig -7: **The SedgewickFastPartition Algorithm**

Bentley [1] suggested an algorithm which the partition function that rolls the largest keys in the array to the bottom of the array using 'for' loop. For the pivot P (chosen at random), Lastbeg is computed that rearrange the array A [Beg]…A [End] such that all keys less than T are on one side of Lastbeg and all other keys are on the other side. A simple for loop scans the array from left to right, using the variables I and Lastbeg as indices, maintain the following invariant in array A. If A [I] >= P then the invariant is still valid. However, if A [I] < P, the invariant is regained by incrementing LastLow by 1 and then swapping A [I] and A [Lastbeg]




```
int BentleyPartition(int a[], int beg, int end)
{
int i, j;
int pivot;
swap(a[beg],a[rand()%(end-beg+1)+beg]);
pivot = a[beg];
i = beg;
for (j = beg+1; j <= end; j++) //j = I
{
if (a[j] < pivot)
{
i = i +1; // i = Lastbeg
swap(a[i], a[j]);
}
}
swap(a[beg], a[i]);
int pos = i;
return pos;
}
```

Fig -8: **The BentleyPartition Algorithm**

Wainwright [19] suggested Bsort, a variation of Quicksort for nearly sorted lists as well as lists that are nearly sorted in reverse order. He combined the swapping technique used in Bubble sort with the Quicksort algorithm. It chooses the middle key as the pivot during each pass, and then it continues to use the conventional 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 and it will be swapped with its left neighbor if the new key does not preserve the order of the subarray. Similarly, keys are placed in right subarray. It 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.

Wainright in 1987 [20] gave Qsorte, an algorithm with an early exit for sorted .arrays. It is original Quicksort algorithm with a slight modification in the partition phase i.e. checking by the left and right sublists for sorting. The middle key is chosen as the pivot in the partitioning phase and the left and right sublists are assumed to be sorted in the beginning. While placing a new key in the left sublist, if the sublist is still sorted and if the sublist is not empty, the new key will be match up with its left neighbor. The sublist is marked as unsorted if the two keys are not in sorted order and the keys are not swapped. Similar processing will be done for right sublist. Any sublist that is marked as sorted will not be partitioned in the end of partitioning phase. When the chosen pivot is always the smallest value in the sublist, Qsorte has a worst case time complexity of $O\ (n^2)$

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

Fig -9: **The Qsorte Algorithm**

A variation on Hoare's original partition algorithm '**Rotate'** version was provided by McDaniel [9] where the pivot is compared with the value in the bottom'th position. The bottom index is decremented when the pivot is less than that value otherwise a rotate left operation is performed using the call Rotate_Left (bottom, pivot+1, pivot). After the index bottom is decremented the following declaration are true:

beg	pivot-1	pivot+1	bottom	end
<= a[pivot]	= a[pivot]	??	> a[pivot]	

```
void Rotate_Left(int i, int j, int k)
{
int t;
t = a[i];
a[i] = a[j];
a[j] = a[k];
a[k] = t;
}
```

```
void McDanielPartition(int beg, int end, int& pos)
{
int pivot, bottom;
pivot = beg;
bottom = end;
while (pivot < bottom)
{
if (a[pivot] > a[bottom])
{
Rotate Left (bottom, pivot+1, pivot);
pivot = pivot + 1;
}
else
bottom = bottom - 1;
}
pos = pivot;
}
```

Fig -10: **The McDanielPartition Algorithm**

Benteley and Mcilroy [2] developed a fast qsort7 based on the existing qsort function that comes with the C library. The size of the array is the basis for selecting pivot. The pivot is chosen as the middle key for small sized arrays, the pivot is chosen using the median-of-three method for mid-sized arrays and the pivot is chosen as the Pseudo median of 9 for large sized arrays. Fat partitioning divides the input array into three parts (Tripartite partitioning (< = >)). After partitioning, the equal elements in the middle are ignored and the left and the right sub arrays are recur. A better fat partition version is used (= <? >) where after partitioning, the equal keys are brought to the middle by swapping the outer ends of the two left portions. The combination of split-end partitioning and an adaptively sampled partitioning element is considered in this algorithm. Here the partitioning has two loops, first inner loop go up the index b, scans over lesser elements, exchange equal elements to the element pointed to by a and bring to an end at a greater element while the second inner loop go down the index c, scans over greater elements, exchange equal elements to the element pointed to by d and bring to an end at a lesser element. The main loop then exchange ar [b] and ar [c], increment b and decrement c, and continues until b and c cross paths. Then the equal keys on the boundaries are swapped back to the middle of the array. Median-of-three method is used to select pivot.

Neubert [11] suggested Flashsort that consider classification of elements [22], [21], [14] instead of comparisons. Dobosiewicz [5] suggested that only O (n) time is required by classification based sorting algorithms to sort n elements thus accomplishing the absolute lowest time complexity, but they require considerable auxiliary memory space. Flashsort is able to reduce this factor as 0.1n auxiliary memory by using a classification step for long-range ordering with in-place permutation. Afterwards the algorithm uses a simple comparison method for the final short-range ordering of each class.

Chakraborty [23] suggested an algorithm which uses an auxiliary array for holding array keys during sort. Unfortunately author had not done any analysis or testing. This algorithm is not an in-place algorithm as it is essentially a Quicksort that uses an extra temporary array of the same size as the original array.

Dynamic Pivot Selection Technique by Nisha [32] is the modified algorithm which is based on splitting the array into relatively equal halves so that for each recursive call there will be reduction in number of recursive calls and the overall execution time of QuickSort. In addition, if the array is already sorted it will not be processed any further which reduces the O (n2) complexity into the best case behavior of the algorithm; i.e. O (n). Firstly, the rightmost element of the array is chosen as pivot. Each element value is compared with the pivot value. The two counters, CountLess and CountLarger, made use to count the number of elements with values smaller than the pivot and the number of elements with values larger than the pivot. The variables SumLess and SumLarger are used to store the sum of the values of the elements smaller than the pivot and the sum of those larger than the pivot. These variables are then used to calculate the next pivots for the recursive calls. In the recursive call for the left sub array, the integer average of the values smaller than the pivot is passed as the pivot value. Similarly, in the recursive call for the right sub array, the integer average of the values larger than the pivot is passed as the pivot value. This way to choose pivot helps in consecutively dividing the array into nearly equal halves thus improves the competence of the QuickSort algorithm. A Boolean variable is utilized by the algorithm to recognize an already sorted array or sub array which reduces the number of recursive calls.

```
void QS(int a[],int beg,int end, int pivot)
{
int i=beg, j=end, flag=0,p1=0,p2=0,temp,c1=0,s1=0,c2=0,s2=0,z;
        if(beg<end)
        {
        temp=a[end];
        while(i<=j)
        {
                if(a[i]<=pivot)
                {
                        c1++; s1+=a[i];
                        if(flag==0 && temp>=(pivot-a[i])) temp=pivot-a[i];
                        else flag=1;
                                i++;
                }
                else
                {
                c2++;  s2+=a[i];
                z=a[i];
                a[i]=a[j];
                a[j]=z;
                j--;
                }
        }
        if(c1!=0)
                {
                p1=s1/c1;
                if(flag!=0) QS(a,beg,i-1, p1);
                }

        if(c2!=0)
                {
                p2=s2/c2;
                QS(a,i,end, p2);
                }

        }
}
```

Fig -11: **The Qsort Algorithm**

Table -1: Assessment of time (in seconds) required to execute unsorted arrays with varying sizes for the Hoare-Partition Algorithm, the Random--Partition Algorithm, the Median3way Algorithm, the Scowen-Partition Algorithm, the Medianof3_ Partition Algorithm, The SedgewickFast Partition Algorithm, The Bentley Partition Algorithm and the Qsort Algorithm

No. of elements in array (Unsorted)	*The Hoare-Partition Algorithm*	*The Random--Partition Algorithm*	*The Median3way Algorithm*	*The Scowen-Partition Algorithm*	*The Medianof3_ Partition Algorithm*	*The Sedgewick Fast Partition Algorithm*	*The Bentley Partition Algorithm*	*The Qsort Algorithm*
100	0.01532	0.124	0.1202	0.04129	0.04658	0.1508	0.1217	0.01475
200	0.03948	0.06304	0.09342	0.0307	0.01611	0.07339	0.05913	0.009376
300	0.05855	0.05503	0.09342	0.09564	0.08117	0.05196	0.1342	0.03299
400	0.01427	0.06452	0.06857	0.2157	0.06036	0.08464	0.07737	0.01155
500	0.01621	0.05035	0.1151	0.08506	0.02082	0.1104	0.1412	0.01194
1000	0.03514	0.1204	0.1388	0.07727	0.03975	0.08045	0.1366	0.05433
2000	0.05783	0.2443	0.1115	0.192	0.06043	0.07209	0.1017	0.03083
5000	0.07183	0.2092	0.2111	0.1235	0.04501	0.2076	0.2861	0.04339
10000	0.06299	0.157	0.3261	0.09393	0.06211	0.2267	0.3229	0.04447

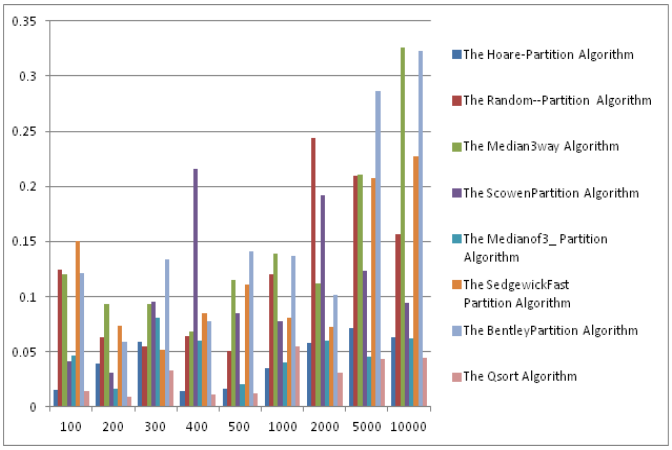


Fig -11: **The Comparison Chart of assessment time for eight algorithms required to execute unsorted arrays with various sizes**

Table -2: Assessment of time (in seconds) required to execute sorted arrays with varying sizes for the Hoare-Partition Algorithm, Median of three Method, Bentley Partition Algorithm, Median3way Algorithm, SedgewickFast Algorithm The Random--Partition Algorithm and the Qsort Algorithm

No. of elements in array (sorted)	The Hoare-Partition Algorithm	Median of three Method	Bentley Partition	Median3way	SedgewickFast	The Random--Partition Algorithm	QSort
100	0.02708	0.04533	0.1307	0.08169	0.08501	0.05775	0.02032
500	0.03255	0.03658	0.2195	0.07684	0.1028	0.1406	0.02848
1000	0.4775	0.04698	0.1968	0.07707	0.07897	0.0848	0.02951
5000	0.2133	0.06411	0.2583	0.2165	0.1946	0.1004	0.04136
10000	0.6298	0.08402	0.1759	0.5632	0.6165	0.2546	0.09452

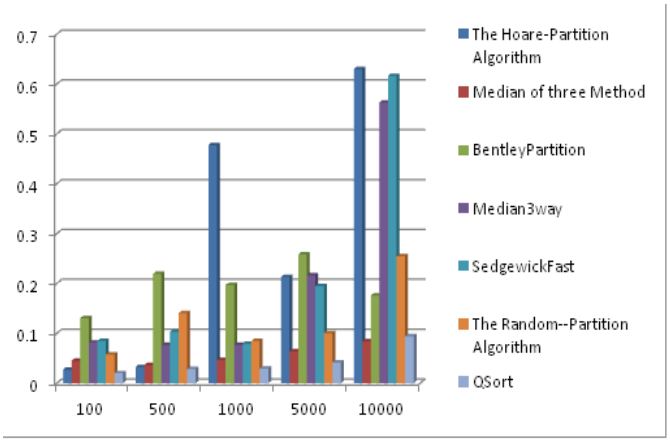


Fig -12: **The Comparison Chart of assessment time for seven algorithms required to execute sorted arrays with various sizes**

3. CONCLUSIONS

In this paper a broad review and experiential study of the Quicksort algorithm is presented. The survey studies in detail the variations of Quicksort from the original version developed by Hoare in 1961 to few of the latest approaches for Quicksort. The paper also investigates the concept behind selecting pivot and evaluates their performances to the variety of versions of Quicksort. The study compared each algorithm in terms of the running times that execute unsorted and sorted arrays with various sizes.

REFERENCES

1. J. Bentley, "Programming Pearl: How to sort", *Com ACM, Vol*. 27 Issue 4, April 1984.
2. J. L. Bentley, M. D. Mcilroy, "Engineering a Sort Function", *SOFTWARE—PRACTICE AND EXPERIENCE,* Vol. 23(11), Nov. 1993, pp 249 – 1265.
3. 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 .
4. R. Chaudhuri and A. C. Dempster, "A note on slowing Quicksort", SIGCSE Vol . 25, No . 2, Jan 1993.
5. W. Dobosiewicz, "Sorting by distributive partitioning," Information Processing Letters 7, 1 – 5, 1978.
6. C.A.R. Hoare, "Algorithm 64: Quicksort," Comm. ACM 4, 7, 321, July 1961.
7. C. A. R. Hoare, "Quicksort," Computer Journal, 5, pp 10 – 15, 1962.
8. R. Loeser, "Some performance tests of : quicksort: and descendants," Comm. ACM 17, 3 , pp 143 – 152, Mar. 1974.
9. B. McDaniel, "Variations on Put First," Conference on Applied Mathematics, University of Central Oklahoma, spring 1991
10. K. Mehlhorn, Data Structures and Algorithms, Vol. 1, Sortzng and Searchzng, 1984 EATCS Monographs on Theoretical Computer Science, Berlin/Heidelberg Springer-Verlag.
11. K.D. Neubert, "The FlashSort algorithm," In Proc. of the euroFORTH'97 –Conf., Oxford, England, Sept. pp 26 – 28, 1997.
12. R.S. Scowen, "Algorithm 271: Quickersort," Comm. ACM 8, 11, pp 669-670, Nov. 1965.
13. R. Sedgewick, "Quicksort," PhD dissertation, Stanford University, Stanford, CA, May 1975. Stanford Computer Science Report STAN-CS-75-492.
14. R. Sedgewick, "The Analysis of Quicksort Programs," Acta Informatica 7, pp 327 – 355,, 1977.
15. R. S. Scowen, "Algorithm 271: quickersort," Comm. of the ACM, 8, pp 669 – 670, 1965.
16. R. Sedgewick, Algorithms in C++, 3rd edition, Addison Wesley, 1998.
17. R. C. Singleton, "Algorithm 347: An efficient algorithm for sorting with minimal storage," Comm. ACM 12, 3, pp 186-187, Mar. 1969.
18. R. Sedgewick, "Implementing Quicksort programs," Comm. of ACM, 21(10), pp 847 – 857, Oct. 1978.
19. R. L. Wainwright, "A class of sorting algorithms based on Quicksort," Comm. ACM, Vol. 28 Number 4, April 1985.
20. R. L Wainright, "Quicksort algorithms with an early exit for sorted subfiles," Comm. ACM, 1987.
21. N. Wirth, Algorithm und Datenstrukturen, B. G. Teubner, 1983
22. D. E. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison Wesley Publ. Co., 1973.
23. K. K. Sundararajan, and S. Chakraborty, " A new sorting algorithm", InterStat, Statistics on the Internet, 2006.
24. C. Martinez, Partial quicksort. In Proceedings of the First ACM-SIAM, Workshop on Analytic Algorithmics and Combinatorics (ANALCO), 2004.
25. Dean C. (2006). "A Simple Expected Running Time Analysis for Randomized Divide and Conquer Algorithms". Computer Journal of Discrete Applied Mathematics, 154(1), 1-5
26. Ledley R. (1962). "Programming and Utilizing Digital Computers." McGraw Hill. Bell D. (1958). The Principles of Sorting. The Computer Journal, 1(2), 71-77
27. Janez B., Aleksander V. and Viljem Z. (2000). "A sorting algorithm on a pc cluster", ACM Symposium on. Applied Computing, 2-19
28. Mohammed, A. and Othman M. (2004). "A new pivot selection scheme for Quicksort algorithm". Suranaree. J. Sci. Technol., 11, 211-215
29. Mohammed, A. and Othman M. (2007). "Comparative analysis of some pivot selection schemes for Quicksort algorithm". Inform. Technol. J., 6, 424-427
30. Abdel Latif Abu Dalhoum1* (Corresponding author), Thaeer Kobbaey1, Azzam Sleit1*,Manuel Alfonseca2,Alfonso Ortega2. "Enhancing QuickSort Algorithm using a Dynamic Pivot Selection Technique", Wulfenia Jounal, Klagenflert, Austria, ISSN: 1561-882X, Vol 19, No. 10;Oct 2012
31. M. Blum, R.W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan, "Time bounds for selection", J. Comput. Syst. Sci. 7, 448–461 (1973).
32. Mrs. Nisha Rathi, "QSort – Dynamic Pivot in Original Quick Sort", International Journal of Advance Research and Development, July 2018, Vol 3, Issue 7, Page 125-128