International Journal of Communication & Information Technology (CommIT) http://msi.binus.ac.id/commit/
Vol. 7 No. 2 Oktober 2013, pp. 41-45

# HYBRID QUICKSORT: AN EMPIRICAL STUDY

Surya Sujarwo
School of Computer Science, Bina Nusantara University
Jln. K. H. Syahdan No. 9, Jakarta 11480, Indonesia
surya.ss@binus.edu

**Abstract:** This article describes an empirical study of hybrid approach of quicksort algorithms. The hybrid approach uses various variations of quicksort partition algorithms in combination with cutoff to shellsort using several cutoff elements. The paper compares the running time performance of the existing quicksort partition algorithms with the proposed hybrid approachusing uniqueand duplicate elements. As a result, it is found that several the hybrid approaches performs faster than the existing algorithms for uniqueand duplicate elements.

**Keywords:** Hybrid approach; Cutoff; Shellsort; Quicksort

## INTRODUCTION

Sorting plays a major role in commercial data processing [1]. Many applications utilize quicksort algorithm because the algorithm works well and easy to implement for variety of different kinds of input data. The algorithm is substantially faster than other sorting methods for general purpose use [1, 2]. In addition, the algorithm uses no additional space for data storing and requires processing time that proportional to *n log (n)* on average to sort n items [1, 3].

The quicksort algorithm is a divide-and-conquer method for sorting [1]. It works by partitioning an array into two parts; then, sorting each part independently [1]. From this specification, partition plays major role in Quicksort processing time.

The common partition algorithms deployed in quicksort algorithm are Hoareand Lomuto algorithm [4, 5]. There are many improvements can be done in Quicksort such as cutoff to insertion sort [1, 6], median-of-three partitioning [1,7], median-of-five with or without random index selection [8], and multiple pivot [2].

For large data of duplicate sort key, there is a specific Quicksort algorithm that has potential to reduce processing time of sorting from linearithmic to linear [1]. The idea is to partition the data in three parts, one each for items with key smaller than, equal to and larger than partition key [1].

This paper evaluates the performance of multiple partitioning schemes of quicksort algorithm for various hybrid approaches. The hybrid algorithm approaches use cutoff to shellsort [1] for small data key in range from 6 to 32 data. The partition algorithm used in the hybrid approach is Hoare partition, modified Hoare [4], and median-of-five without random index selection [8]. The rest of the paper is organized as follows: a section describing existing algorithms used in partition for comparison, a section describing the proposed hybrid approach algorithm inspired from existing algorithms, and a section comparing the performance of existing and proposed algorithms.

## Existing Algorithms

The first existing quicksort partition algorithm used for comparison is based on Hoare algorithm described in Ref. [4]. This algorithm chooses a first key as the pivot for partition, and then moves the pivot to correct position and partition all keys smaller than the pivot to the left side of the pivot and larger than the pivot to the right side of the pivot. Then at last the algorithm returns the correct position of the pivot to partition the data to two parts. The Hoare algorithm is implemented in the C++ function below Algorithm 1 with signature int Hoare(int *data, int first, int list) where data represents the array to be sort, first represents the first location and last represents the last location performs the partition according to Hoare algorithm [4]. The function swap(int&, int&) is called to swap the value of two variables, and the function sortF(int*, int, int, int(*)(int*,int,int))is used to quicksort the Hoare partition.

Algorithm 1: Hoare algorithm

```
void sortF(int *data, int first, int
last,int (*v)(int*,int,int)) {
  if(first < last) {
    int pv = v(data, first, last);
    sortF(data, first, pv-1, v);
    sortF(data, pv+1, last, v);
  }
}
```

```
void swap(int &a, int &b) { int
c=a;a=b;b=c; }
```

```
int Hoare(int* data, int first, int
last) {
  if(first < last)  {
    int pivot = data[first], i = first, j
= last+1;
    while(true) {
      while(++i <= last && data[i] <
pivot);
      while(data[--j]>pivot);
      if(i>j) break;
      swap(data[i],data[j]);
    }
    swap(data[first], data[j]);
    return j;
  } return -1;
}
```

The following Hoare algorithm, Algorithm 2, is a modified Hoare partition algorithm which applies sentinels to cover first as well as last extremes of the array which reduce the index manipulation operations to optimum level [4]. The C++ function MHoare(int*, int, int) implements the modified algorithm.

Algorithm 2: Modified Hoare algorithm

```
int MHoare(int* data, int first, int
last) {
  if(data[first]>data[last])
    swap(data[first],data[last]);
  int pivot = data[first];
  while(true) {
    while(data[--last] > pivot);
    data[first]=data[last];
    while(data[++first] < pivot);
    if(first<last)
      data[last]=data[first];
    else {
      if(data[last+1] <= pivot)
        last++;
      data[last] = pivot;
      return last;
    }
  }
}
```

The next existing algorithm is Lomuto partition algorithm, which scans whole array and whenever an element is smaller. When the pivot is found, the element is swapped. The following C++ function Lomuto(int*,int,int) implements the Lomuto partition algorithm [4].

Algorithm 3: Lomuto partition algorithm

```
int Lomuto(int *data, int first, int r)
{
    int pivot = data[r];
    int i = first - 1;
    for (int j = first; j < r; j++)
      if (data[j] <= pivot)
        swap(data[++i], data[j]);
    swap(data[++i], data[r]);
    return i;
}
```

Following Lomuto algorithm is a modified Lomuto partition, which casts aside superfluous index manipulation and swap operations. The C++ function MLomuto int*,int,int) implements the modified Lomuto partition algorithm [4].

Algorithm 4: Modified lomuto algorithm

```
int MLomuto(int* data, int first, int
last) {
  int x = data[first], i = first, j =
last;
  while(true) {
    while(data[j] > x) j--;
    if(j <= i) break;
    data[i]=data[j];
    data[j]=data[++i];
  }
  data[i] = x;
  return i;
}
```

The next existing partition algorithm is Median-of-five without random index selection method. The pivot is a sample of size five elements of the following: first, middle, last, middle of first and middle, and middle of middle and last. The sample then sorted and the middle is used as a pivot. The C++ function M5(int*,int,int) implements this algorithm [8] and Quicksort.

Algorithm 5: Median-of-five algorithm

```
int M5(int *data, int first, int last) {
  if(first>=last) return -1;
  int i = first;
  int j = last+1;
  int range = last - first;
  if(range > 5) {
    int k[]={first,first+range/4,first+ran
ge/2,first+3*range/4,last};
    for(int i=1;i<5;i++) {
      int j=i, c=data[k[i]];
      for(j=i;j>0;j--)
        if(c<data[k[j-1]])
          data[k[j]] = data[k[j-1]];
      data[k[j]] = c;
    }
    swap(data[k[0]],data[k[2]]);
  }
  int x = data[first];
  while(true) {
    while(++i <= last && data[i] <
pivot);
    while(data[--j]>pivot);
    if(i > j) break;
    swap(data[i],data[j]);
  }
  swap(data[first],data[j]);
  return j;
}
```

The other existing Quicksort algorithm is three-way partition Quicksort which partition data to three parts using four indexes: first, last, first of equal, and last of equal. The C++ function void quick3(int*,int,int) implements this algorithm [9].

Algorithm 6: Three-way algorithm

```
void quick3(int *data, int left, int
right) {
  if (left >= right) return;
  int li = left, last = right;
  int pv = data[li];
  int ln = left;
  while (ln <= last) {
    if (data[ln] < pv) swap(data[li++],
data[ln++]);
    else if(data[ln] > pv)
swap(data[ln], data[last--]);
    else ln++;
  }
  quick3(data, left, li - 1);
  quick3(data, last + 1, right);
}
```

## Proposed Hybrid Algorithms

The proposed hybrid algorithms use existing algorithms with cutoff to shellsort algorithm if the array size between 6 and 32. The shellsort algorithm is implemented by the C++ function shell(int*,int,int). The C++ function sortHF(int* data, int first, int last, int(*v)(int*, int, int), int cutoff) implements the proposed algorithm, where data is the array to be sorted, first is the first location, last is the last location, v is the pointer to function to call existing partition algorithm, and cutoff is the size where the cutoff to shellsort will be done.

Algorithm 7: Shellsort algorithm

```
void shell(int *data, int first, int
last) {
  const int k[]={1,7,19};
  int l=0, range=last-first;
  while(l<3 && k[l] < range) l++;
  while(--l >= 0) {
    int t = k[l];
    for(int i=first+t;i<last;i++) {
      int kt = data[i], j=i;
      for(;j>first;j-=kt)
        if(t<data[j-kt])
          data[j]=data[j-kt];
      data[j]=t;
    }
  }
}
```

Algorithm 8: Sort HF algorithm

```
void sortHF(int *data, int first, int
last,int (*v)(int*,int,int), int
cutoff) {
  if(first < last) {
    if(last - first <= cutoff) {
      shell(data, first, last);
      return;
    }
    int pv = v(data, first, last);
    sortF(data, first, pv-1, v);
    sortF(data, pv+1, last, v);
  }
}
```

The sort HF function call three different partitions: Median-of-five, Hoare and modified Hoare.

```
// Median-of-five
for(int i=6; i<=32; i++) {
 // read array and start timer
   sortHF(array, 0, arraycount, M5, i);
// stop timer
}
// Hoare
for(int i=6; i<=32; i++) {
 // read array and start timer
   sortHF(array, 0, arraycount, Hoare,
i);
// stop timer
}
//Modified Hoare
for(int i=6; i<=32; i++) {
 // read array and start timer
   sortHF(array, 0, arraycount, MHoare,
i);
// stop timer
}
```

## EMPIRICAL TESTING AND RESULTS

The performance of the sorting algorithms describe in section Existing Algorithms and section Proposed Algorithms was studied using number generated randomly from 100,000 to 1,000,000 elements with 100,000 increments. The experiments were conducted on a computer with Intel Xeon (TM) CPI 3.00 GHz, and 1 GB of RAM. To study the behavior of the algorithms on arrays of random elements, each algorithm was used to sort five sequences of random numbers of a specific size N for each distinct element and duplicate elements of maximum N/1000 elements, and the average running time were calculated.

Table 1 shows the running time of the proposed proposed algorithms with the cutoff values from 6 up to 32 unique elements. The best cutoff value for the partition algorithm for unique elements is as the following: Median-of-five using cutoff of 11 elements with average running time 182.8ms, Hoare using cutoff of 30 elements with average running time 174.34ms, and modified Hoare using cutoff of 19 elements with average running time 181.86ms. Table

2 shows running time of each proposed algorithms with cutoff value from 6 to 32 duplicate elements. Each best cutoff value from the partition algorithm for unique elements is as follow: Median-of-five using cutoff of 27 elements with average running time 149.06 ms, Hoare using cutoff of 29 elements with average running time 146.82 ms, and modified Hoare using cutoff of 22 elements with average running time 140.62.

Table 1: Running Time of Proposed Algorithm on Median-of-five, Hoare and Modified Hoare in milliseconds for unique elements

| Cutoff | Median-of-Five | Hoare | Modified Hoare |
|---|---|---|---|
| 6 | 185.02 | 177.20 | 187.54 |
| 7 | 183.96 | 178.80 | 184.10 |
| 8 | 186.02 | 178.08 | 188.40 |
| 9 | 183.44 | 174.74 | 184.34 |
| 10 | 184.06 | 176.24 | 184.84 |
| 11 | 182.80 | 178.78 | 184.98 |
| 12 | 186.60 | 176.16 | 184.40 |
| 13 | 185.98 | 176.02 | 183.72 |
| 14 | 185.30 | 174.96 | 187.80 |
| 15 | 184.02 | 177.84 | 185.02 |
| 16 | 184.34 | 175.92 | 183.42 |
| 17 | 186.68 | 177.76 | 183.78 |
| 18 | 183.42 | 175.96 | 185.90 |
| 19 | 185.28 | 175.94 | 181.86 |
| 20 | 184.34 | 77.78 | 189.46 |
| 21 | 183.74 | 178.78 | 187.62 |
| 22 | 188.04 | 176.58 | 185.88 |
| 23 | 186.42 | 178.08 | 183.74 |
| 24 | 184.36 | 174.78 | 186.60 |
| 25 | 186.24 | 177.14 | 183.42 |
| 26 | 185.04 | 175.08 | 185.24 |
| 27 | 185.02 | 174.40 | 186.60 |
| 28 | 185.66 | 175.56 | 184.34 |
| 29 | 185.28 | 175.92 | 186.88 |
| 30 | 185.70 | 174.34 | 184.06 |
| 31 | 186.22 | 177.18 | 187.46 |
| 32 | 183.98 | 175.88 | 186.08 |

Table 2: Running Time of Proposed Algorithm on Median-of-five, Hoare and Modified Hoare in milliseconds for duplicate elements

| Cutoff | Median-of-Five | Hoare | Modified Hoare |
|---|---|---|---|
| 6 | 151.02 | 148.94 | 142.28 |
| 7 | 151.24 | 150.30 | 141.80 |
| 8 | 153.44 | 150.58 | 145.38 |
| 9 | 151.92 | 150.36 | 143.52 |
| 10 | 152.10 | 150.92 | 142.14 |
| 11 | 152.58 | 150.34 | 143.36 |
| 12 | 154.10 | 150.28 | 142.58 |
| 13 | 152.24 | 149.44 | 144.30 |
| 14 | 155.30 | 148.92 | 143.78 |
| 15 | 150.66 | 150.04 | 142.24 |
| 16 | 154.02 | 147.86 | 141.86 |
| 17 | 151.90 | 148.46 | 142.52 |
| 18 | 152.72 | 148.38 | 145.64 |
| 19 | 152.26 | 149.36 | 143.16 |
| 20 | 152.14 | 148.54 | 141.26 |
| 21 | 153.16 | 148.72 | 144.04 |
| 22 | 153.40 | 151.60 | 140.62 |
| 23 | 150.04 | 150.00 | 145.92 |
| 24 | 152.16 | 148.62 | 143.40 |
| 25 | 150.94 | 149.14 | 142.48 |
| 26 | 151.84 | 149.08 | 143.44 |
| 27 | 149.06 | 148.40 | 145.98 |
| 28 | 150.28 | 150.38 | 141.82 |
| 29 | 151.58 | 146.82 | 146.32 |
| 30 | 150.02 | 148.84 | 144.02 |
| 31 | 149.08 | 149.02 | 143.78 |
| 32 | 152.46 | 150.82 | 144.46 |

From the best cutoff of each proposed algorithm, median-of-five with cutoff 11 and 27 elements, Hoare with cutoff 30 and 29 elements, and modified Hoare with cutoff 19 and 22 elements, the algorithms then compared with algorithm 3-way partition, median-of-five, Hoare, Lomuto, modified Hoare, and modified Lomuto. Tables 3 and 4 show the average running times for each Nofthe respective algorithms. Table 3 shows that the slowest algorithms for unique elements are 3-way partition followed by modified Hoare. Table 4 shows that the slowest algorithms for duplicate elements are modified Hoare followed by modified Lomuto. The three best running time for unique elements algorithms are the proposed Hoare cutoff 30 elements with average time 174.34ms, proposed Hoare cutoff 29 elements with average time 175.92ms, and Hoare with average time 176.28ms. The three best running time for duplicate elements are proposed modified Lomuto cutoff 22 elements with average time 140.62ms, Lomuto with average time 141.48ms, and Lomuto cutoff 19 elements with average time 143.16ms. The best three algorithms from category unique numbers and duplicate numbers are combined to see the detail running time in figure 1 and 2.

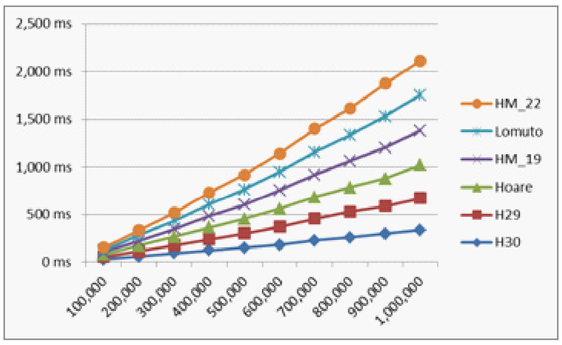

Fig. 1: Running time of selected algorithms in unique elements

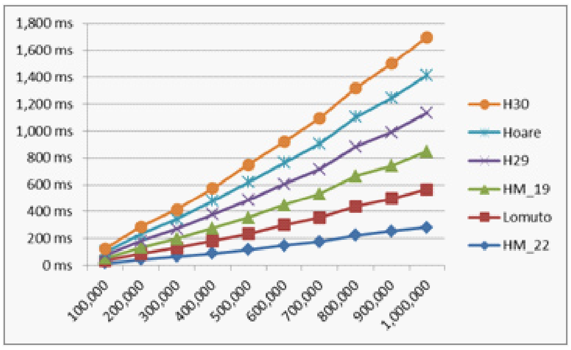

Fig. 2: Running time of selected algorithms in duplicate elements

Table 3: Running Time of Algorithms for 100,000 to 1,000,000 unique elements

| Algorithm | 100,000 | 200,000 | 300,000 | 400,000 | 500,000 | 600,000 | 700,000 | 800,000 | 900,000 | 1,000,000 |
|---|---|---|---|---|---|---|---|---|---|---|
| 3way | 47 | 90.6 | 146.8 | 200 | 256 | 325 | 418.6 | 478.2 | 569 | 677.8 |
| median 5 | 31.4 | 65.6 | 93.8 | 134.4 | 169 | 199.8 | 237.6 | 268.6 | 300 | 340.8 |
| HM5_11 | 31.2 | 62.4 | 100 | 128.2 | 162.4 | 200 | 234.2 | 265.8 | 306.2 | 337.6 |
| HM5_27 | 31.2 | 62.6 | 103.2 | 131 | 168.6 | 199.8 | 231.4 | 272 | 309.4 | 341 |
| Hoare | 25 | 59.2 | 87.6 | 121.8 | 152.8 | 187.8 | 228.4 | 256.2 | 293.8 | 350.2 |
| Lomuto | 21.8 | 59.4 | 84.4 | 122 | 156.4 | 196.6 | 240.6 | 271.8 | 328 | 371.8 |
| MHoare | 37.6 | 71.8 | 109.4 | 149.8 | 196.8 | 234.2 | 274.8 | 322 | 369 | 421.8 |
| MLomuto | 31.2 | 56.4 | 93.8 | 125.2 | 156.4 | 188 | 228 | 265.4 | 303 | 356 |
| H29 | 28 | 56.2 | 87.8 | 118.8 | 150.2 | 190.8 | 227.8 | 271.8 | 290.4 | 337.4 |
| H30 | 25 | 56.2 | 90.4 | 122 | 149.8 | 181.2 | 228.4 | 259.4 | 296.6 | 334.4 |
| HM_19 | 25 | 50 | 81.4 | 121.8 | 152.8 | 193.8 | 234.6 | 278 | 322 | 359.2 |
| HM_22 | 28 | 49.8 | 87.4 | 125 | 156.2 | 190.6 | 240.6 | 278 | 344 | 359.2 |

Table 4: Running Time of Algorithms for 100,000 to 1,000,000 duplicate elements

| Algorithm | 100000 | 200000 | 300000 | 400000 | 500000 | 600000 | 700000 | 800000 | 900000 | 1000000 |
|---|---|---|---|---|---|---|---|---|---|---|
| 3way | 31.4 | 56.4 | 75 | 106 | 146.8 | 190.8 | 218.8 | 274.8 | 306 | 387.4 |
| median 5 | 21.6 | 53.4 | 78.2 | 109.2 | 137.4 | 165.4 | 212.6 | 218.8 | 256.4 | 278.4 |
| HM5_11 | 28.2 | 50 | 81.4 | 106.4 | 134.2 | 162.4 | 228 | 222.2 | 244 | 269 |
| HM5_27 | 25.2 | 52.4 | 87.4 | 109.6 | 131.4 | 162.4 | 187.4 | 218.8 | 240.6 | 275.4 |
| Hoare | 21.8 | 50 | 72 | 100.2 | 134.4 | 159.4 | 190.8 | 218.8 | 253.2 | 281.2 |
| Lomuto | 19.2 | 43.2 | 65.8 | 93.8 | 118.8 | 152.8 | 177.8 | 215.4 | 243.8 | 284.2 |
| MHoare | 78 | 250 | 519 | 881 | 1374.8 | 1962.6 | 2584.6 | 3372.2 | 4194 | 5168.8 |
| MLomuto | 47 | 163 | 334.4 | 553.4 | 866 | 1222 | 1612.4 | 2131.6 | 2609.2 | 3209.2 |
| H29 | 21.6 | 46.6 | 72 | 103 | 131.2 | 156.2 | 184.6 | 218.8 | 249.8 | 284.4 |
| H30 | 25.2 | 56.6 | 75.2 | 97 | 131.4 | 156.2 | 190.6 | 215.8 | 256.2 | 284.2 |
| HM_19 | 21.6 | 47 | 65.8 | 93.8 | 118.8 | 149.8 | 178.4 | 228.2 | 243.8 | 284.4 |
| HM_22 | 15.4 | 44 | 65.6 | 87.6 | 115.4 | 146.8 | 175 | 222 | 253.2 | 281.2 |

## CONCLUSION

To conclude the results, for the case of duplicate elements which is common in indexing of multiple keys in database application, we can use the proposed hybrid algorithm with modified Lomuto partition algorithm and cutoff value of 22 elements for the best average running time. As the case of unique elements such as primary key of data base application, we can use the proposed hybrid algorithms with modified Hoare and cutoff value of 30 elements for best average running time.

## REFERENCES

[1] R. Sedgewick and K. Wayne, Algorithms (4th Edition), Cloth: Addison-Wesley Professional, 2011.

[2] S. F. Solehria and S. Jadoon, "Multiple Pivot Sort Algorithm is Faster than Quick Sort Algorithms: An Empirical Study," IJECS: International Journal of Electrical and Computer Science, 11, pp. 14–18, June 2011.

[3] G. S. Brodal, R. Fagerberg and G. Moruz, "On the adaptiveness of Quicksort," Journal of Experimental Algorithmics, 12,(3.2), doi:http://dx.doi.org/10.1145/1227161.1402294, 2008

[4] D. Abhyankar and M. Ingle, "Engineering of a Quicksort Partitioning Algorithm," Journal of Global Research in Computer Scienc,2, pp. 17–23, 2011.

[5] L. Khreisat, "QuickSort A Historical Perspective and Empirical Study," International Journal of Computer Science and Network Security,7, pp. 54–65, 2007.

[6] D. Abhyankar and M. Ingle, "A better approach to QuickSort implementation," International Journal of computers and communications,1, pp. 39–48, 2011.

[7] R. Sedgewick, "Implementing Quicksort programs," Communications of the ACM,21, pp. 847–857, 1978.

[8] A. Mohammed and M. Othman, "Comparative Analysis of Some Pivot Selection Schemes for Quicksort Algorithm," Information Technology Journal, 6, pp. 424–427, 2007.

[9] R. Sedgewick and J. Bentley, Quicksort is Optimal, Knuthfest: Stanford University, 2002.