มัลติคีย์ควิกซอร์ตสำหรับการเรียงลำดับสายอักขระด้วยตัวนำหน้าและตัวตามหลังของตัวหลัก

นางสาวปุริกา บริสุทธินันท์

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิทยาศาสตรมหาบัณฑิต
สาขาวิชาวิทยาการคอมพิวเตอร์และเทคโนโลยีสารสนเทศ
ภาควิชาคณิตศาสตร์และวิทยาการคอมพิวเตอร์
คณะวิทยาศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2555

MULTIKEY QUICKSORT FOR SORTING STRINGS USING PREDECESSOR AND SUCCESSOR PIVOTS

Ms. Purika Borisuttinant

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Science Program in Computer Science and Information Technology
Department of Mathematics and Computer Science
Faculty of Science, Chulalongkorn University
Academic Year 2012

Thesis Title	MULTIKEY QUICKSORT FOR SORTING STRINGS USING PREDECESSOR AND SUCCESSOR PIVOTS
By	Ms. Purika Borisuttinant
Field of Study	Computer Science and Information Technology
Thesis Advisor	Assistant Professor Krung Sinapiromsaran, Ph.D.

Accepted by the Faculty of Science, Chulalongkorn University in Partial Fulfillment of the Requirements for the Master's Degree.

……………..…………………………………..Dean of the Faculty of Science

(Professor Supot Hannongbua, Dr.rer.nat.)

THESIS COMMITTEE

……………..…………………………………………….. Chairman

(Assistant Professor Nagul Cooharojananone, Ph.D.)

……………..…………………………………………….. Thesis Advisor

(Assistant Professor Krung Sinapiromsaran, Ph.D.)

…………………………………….……………………. Examiner

(Assistant Professor Chatchawit Aporntewan, Ph.D.)

…………………………………….……………………. External Examiner

(Assistant Professor Sarun Intakosum, Ph.D.)

ปูริกา บริสุทธินันท์ : ควิกซอร์ตสำหรับเรียงลำดับสายอักขระด้วยตัวนำหน้าและตัวตามหลังของตัวหลัก. (MULTIKEY QUICKSORT FOR SORTING STRINGS USING PREDECESSOR AND SUCCESSOR PIVOTS) อ. **ที่ปรึกษาวิทยานิพนธ์หลัก** : ผศ.ดร.กรุง สินอภิรมย์สราญ, 51 หน้า.

ควิกซอร์ตเป็นขั้นตอนวิธีการเรียงลำดับภายในที่นิยมใช้กันมาก งานวิจัยนี้เราเสนอการปรับปรุงควิกซอร์ตสำหรับจำนวนเต็มและสายอักขระด้วยตัวนำหน้าตัวหลัก ตัวตามหลังตัวหลักและการแบ่งกั้นแบบ collect-center การใช้ตัวนำหน้าตัวหลักและตัวตามหลังตัวหลักช่วยลดจำนวนครั้งที่เรียกฟังก์ชันเวียนเกิดในขณะที่การแบ่งกั้นแบบ collect-center ช่วยลดจำนวนการสลับที่ เราเปรียบเทียบประสิทธิภาพของอัลกอริทึมของเราซึ่งเรียกว่า CC5sort กับควิกซอร์ตด้วยตัวประชิด และการแบ่งกั้นต้นแบบของ collect-center เราทดสอบประสิทธิภาพของ CC5sort ในข้อมูลสี่แบบ ได้แก่ ข้อมูลที่เกือบเรียงลำดับ ข้อมูลที่เกือบเรียงลำดับแบบย้อนกลับ ข้อมูลแบบสุ่มที่มีสมาชิกซ้ำกันแบบเอกภาพและข้อมูลแบบสุ่ม จากการทดลองของเราพบว่า CC5sort ประมวลผลได้เร็วกว่า Collect-center partitioning และ APQsort สำหรับข้อมูลแบบสุ่มที่มีสมาชิกซ้ำกันเป็นจำนวนมาก

ภาควิชา วิทยาการคอมพิวเตอร์ และเทคโนโลยีสารสนเทศ ลายมือชื่อนิสิต ....

สาขาวิชา คณิตศาสตร์และวิทยาการคอมพิวเตอร์ลายมือชื่อ อ.ที่ปรึกษาวิทยานิพนธ์หลัก...

ปีการศึกษา...........2555......................

##5273608223 : MAJOR COMPUTER SCIENCE AND INFORMATION TECHNOLOGY
KEYWORDS : SORTING / QUICKSORT / PREDECESSOR PIVOT / SUCCESSOR PIVOT / COLLECT-CENTER PARTITION

PURIKA BORISUTTINANT : MULTIKEY QUICKSORT FOR SORTING STRINGS USING PREDECESSOR AND SUCCESSOR PIVOTS. ADVISOR : ASST. PROF. KRUNG SINAPIROMSARAN, 51 pp.

Quicksort is one of the most popular internal sorting algorithms. In this research, we propose multikey quicksort for sorting sequence of integers and strings using predecessor pivots, successor pivots and the collect-center partition. Predecessor pivots and successor pivots are used to reduce the recursive calls while the collect-center partitioning is used to reduce the number of swaps. We compare the performance of our algorithm, called CC5sort, with the performance of the quicksort using adjacent pivot quicksort and the original collect-center partition. We tested an efficiency of CC5sort in four different types of data sets; nearly sorted data, nearly reverse sorted data, repeated element data and random ordered data. Our experiments show that CC5sort significantly exhibits the faster running time for random ordered data with a lot of repeated elements than collect-center partitioning and APQsort.

Department : Mathematics and Computer Science	Student's Signature ....
Field of Study : Computer Science and Information Technology	Advisor's Signature ....
Academic Year : 2012	

Acknowledgements

I would like to acknowledge my thesis advisors, Assistant Professor Dr. Krung Sinapiromsaran for helpful guidance and encouragement. He has suggested the solutions to many experimental problems and helped me finish this thesis in time. Moreover, I would like to thank Siwat Rungpiphop for all his great support helping me to improve my researching skill.

Finally I would like to thank my father, mother and friends for everything they suggested and supported me.

CONTENTS

CHAPTER PAGE

List of Figures

List of Tables

Chapter I

Introduction

1.1 Backgrounds

Sorting [1, 2, 3] is one of the fundamental problems in computer science. It appears as a sub-process in many algorithms. It is one of important technique for data processing, in business, science or social science. For the sorted data, finding the element with a specific key is fast. On the other hand, the unsorted data requires looking though the whole list. Sorting is a process that purposely arranges items using specified key values. A sequence might be arranged in ascending or descending order.

For example, a library management system requires book names to be sorted in which it arranges books according to their names see Fig. 1.1. For a registration system of the university, faculty may generate a student list sorted by student names, by student ID or by student grade point average. Furthermore, sorting has been widely applied to other fields of management and sciences such as binary search [1, 2], activity selection problem [1, 2], fractional knapsack problem [1, 2], minimum spanning tree [1, 2], etc.

Sorting algorithms can be classified into 2 types according to their memory usages: the internal sorting and the external sorting.

Internal sorting algorithm performs solely within the primary memory, such as cache, RAM. Prior to the sorting process, all required data must be initially read and recorded in the primary memory. There are many algorithms that are designed to run as the internal sorting algorithm, for example, bubble sort [1, 2], selection sort [1, 2], insertion sort [1, 2, 3], shell sort [1, 2, 3], internal-merge sort [1, 2], quicksort [1, 2, 3], heap sort [1, 2, 3] and radix sort [1, 2, 3] etc.

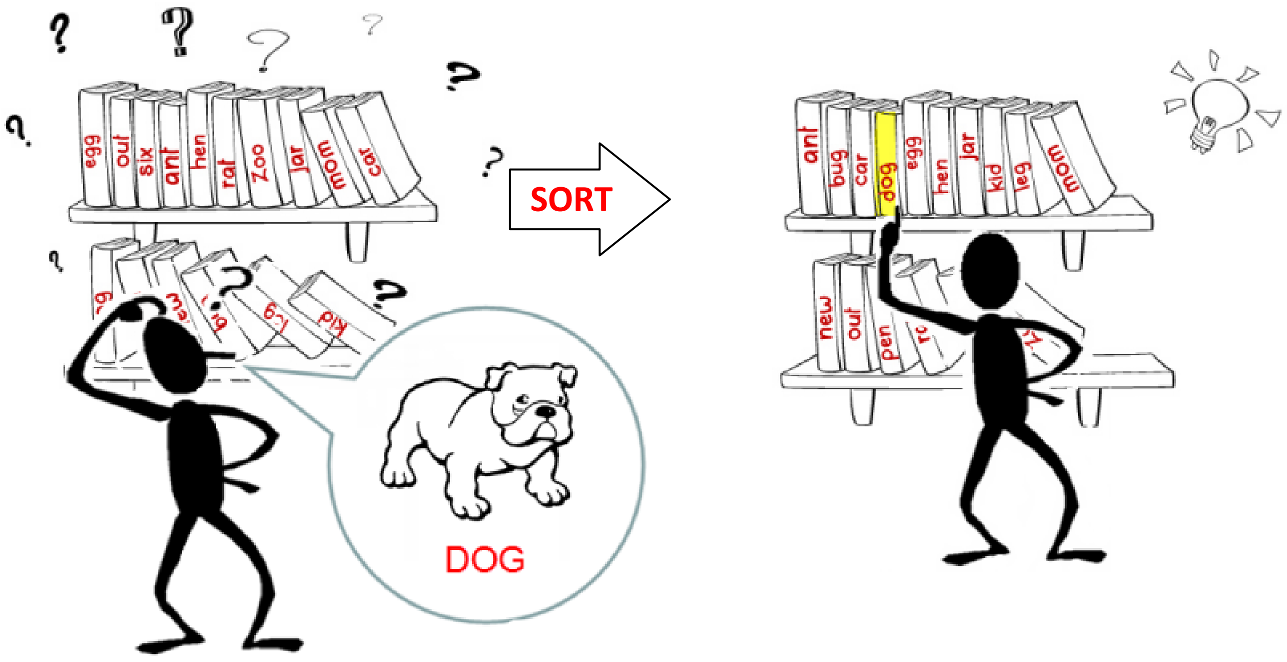


Fig. 1.1 Benefit of sorting

External sorting algorithm uses both primary memory and secondary memory. Normally, loading data to the primary memory is impossible due to the size of the data. Therefore, it requires the second memory (external memory) such as, hard disc or tape. External sorting typically sorts many chunks of small data that resides in the primary memory temporarily. Subsequently, those chunks of small data will be combined with the data in primary memory into a single large data. One common well-known external sorting based algorithm is external - merge sort.

In this research, we focus on improving the sorting process for repeat elements by using quicksort because of its popularity and devide-and-conquer characteristic. Form our study, many active researches are still proposed the improvement of the quicksort.

Quicksort is a powerful sorting algorithm that takes O(*n* log *n*) time on the average and the best case. However, the worst case of quicksort happens when the selected pivot is the least/highest element in each partition process. This problem leads to various ideas to improve quicksort.

1) Pivot selection: In order to improve worst case scenario of quicksort algorithm, a new pivot selection method is introduced. To avoid choosing a pivot with the least/highest value, researchers choose the pivot element at a random index as in [5], the middle index of the partition as in [6], or (especially for longer partitions) the median of the first, the middle and the last elements of the partition as in [7], the median from median-of-five as in [4].

2) Using the faster algorithm for a small sub-array: For a case that the array consists of a few elements, the other sorting algorithm might be used instead of quicksort. For example, Sedgewick [8] suggested to use Insertion sort which has a smaller constant factor and run faster on an array that consists of elements less than 10. It can increase efficiency of quicksort approximate 20 percent.

3) Mixtures of quicksort and other sorting algorithm: This technique improves quicksort performance by blending its strengths with other sorting algorithm advantages. For instance, Wainwright combined quicksort and bubble sort [9] to terminate the sorting algorithm if the array of data is "sorted". This prevents wasteful time-consuming sorting process. Another example is a "split-end" quicksort in which is suitable for elements with a repeated keys [10].

Quicksort uses a recursive "divide and conquer" technique to sort the data see Fig 1.2. Its efficiency depends on 3 different factors, 1) the number of comparisons 2) the number of swaps 3) the number of recursive calls. Comparisons and swaps occur within a single recursive call. If it can reduce number of recursive calls, then number of comparisons and number of swaps should be reducing as well.

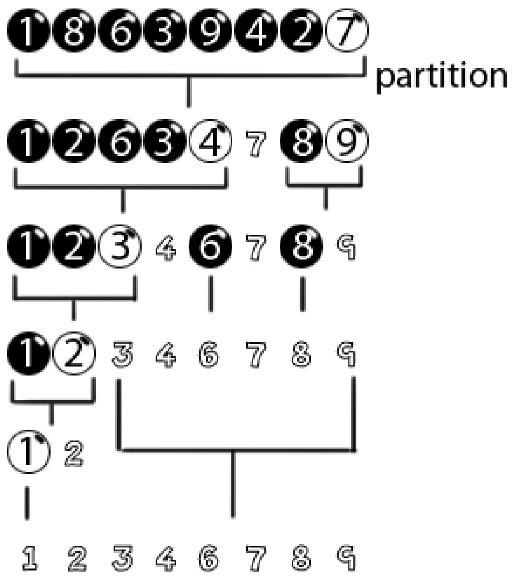


Fig.1.2 Example of quicksort

Selecting a good pivot [4] greatly improves the speed of the quicksort algorithm. There are many text books [1, 2, 3] that suggest the use of the first element in the array as the pivot; however this causes poor performance to the sorting if the data is already sorted. The better method is to utilize the median of sample elements such as median-of-three, median-of-five [4], median-of-seven and median-of-nine.

Using the insertion sort for a small sub-array helps reduce the speed of overall running time. Insertion sort consumes $O(n^2)$ time on average case. For its best case, it takes $O(n)$ time. This algorithm is more efficient for a small array [8].

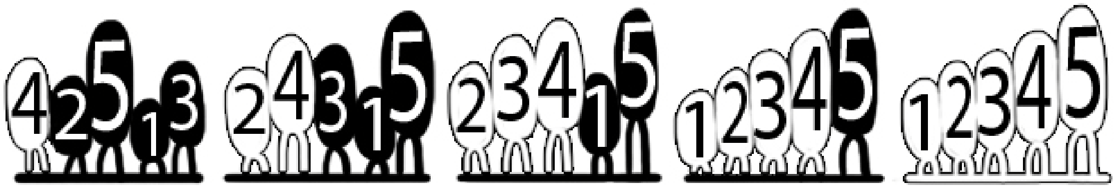


Fig.1.3 Example of insertion sort

The “Collect Center Partition” for the quicksort uses two pointers to gather the pivot elements in the middle. It swaps the elements not equal to the pivot to the left sub-array or right sub-array forcing the elements equal to the pivot to the middle.

CC5sort [11] is designed to improve efficiency of quicksort by reducing the number of recursive calls using “*predecessor pivot*” and “*successor pivot*” and selecting pivot from the median-of-five [4]. Moreover, CC5sort also applies “*Collect Center Partition* [12]” method to reduce the number of swaps within a recursive call. In addition CC5sort collects elements that values are equal to predecessor pivot and successor pivot. CC5sort will be able to identify correct positions of predecessor pivot group elements, pivot group elements, successor pivot group elements, and therefore, reduces the number of recursive calls.

1.2 Objective

This thesis explains the implementation of “CC5sort” function as the primary method for improving and enhancing quicksort algorithm. CC5sort utilizes “predecessor pivot”, “successor pivot”, and “Collect Center Partition” to reduce the number of recursive calls and accelerate the sorting time of redundant elements. In addition, CC5sort will be compared with adjacent pivot quicksort (APQsort) [13] and original collect-center-partition (CCsort) [12], to measure advantages and disadvantages of CC5sort.

1.3 Project scope

In this thesis, 1) data arrays consist of integers and strings 2) programming codes are implemented in C# language 3) four different groups of data sets are tested; nearly sorted data, nearly reverse sorted data, repeated element data are uniform distribution and random ordered data 4) performance of CC5sort will be compared with APQsort and CCsort.

1.4 Expected Result

CC5sort reduces number of times for calling recursively and process in a shorter time period for a larger number of repeated elements, comparing to the traditional quicksort, APQsort from a research by Rungpiphop S. and Sinapiromsaran K. [13] and CCsort from a research by Kim and Park [20].

Chapter II

Related Work

From chapter 1, we discuss researches related to quicksort. Many studies have been proposed in order to improve it. In this thesis, concepts of CC5sort algorithm are explained.

2.1 Improve the pivot selection

This is the process to choose the pivot to avoid the worst case of quicksort and find the best pivot. Many researches propose many techniques to improve this, such as using the Median [4, 5, 16, 17, 22, 23].

2.1.1 Median

Median is the representative of the list that is greater or equal to half of the elements. However, determining a median may require looking to the whole list. So some heuristic methods are suggested.

Median-of-three picks the median of three elements from the first, the middle and the last elements. The median-of-three uses only 2 comparisons.

Median-of-five was used in Brest et al. (2000) and Cerin (2002) [22]. It chose a pivot from five elements of the array; the first, middle, last and two other elements randomly picked through a random number between the first and last elements. These elements were selected then the pivot was chosen as a median. Their algorithm identified the rest as an element that smaller and greater than the pivot. The median-of-five used 6 comparisons.

Median-of-seven chose a pivot from seven elements of the array. The first five elements were selected as following: low, high, $\lceil (low+high)/2 \rceil$, $\lceil (low+high)/4 \rceil$ and $\lceil 3*(low+high)/4 \rceil$. While the other two elements were selected randomly.

Median-of-nine chose a pivot from nine elements of the array. The number of elements to be selected had increased by two. The nine elements were explicitly selected as following: low, high, $\lceil$(low+high)/2$\rceil$, $\lceil$(low+high)/8$\rceil$, $\lceil$(low+high)/4$\rceil$, $\lceil$3*(low+high)/8$\rceil$, $\lceil$5*(low+high)/8$\rceil$ and $\lceil$7*(low+high)/8$\rceil$. Thus, more elements were involved in the selection process.

This research, we use median-of-five to select the pivot. The detail of median-of-five is shown in the next section.

2.1.1.1 Median-of-five

Median-of-five [4] is one of many methods of selecting pivot from calculating the median of the data. This calculation requires five elements. The five elements are the first element, element in between of the first and the middle (can be written in the form of equation as $\lceil$(first+last)/4$\rceil$), the middle element, element in between of the middle and the last (can be written in the form of equation as $\lceil$3*(first+last)/4$\rceil$) and the last element.

First	$\lceil$(first+last)/4$\rceil$	Middle	$\lceil$3*(first+last)/4$\rceil$	Last

Fig.2.1 Five elements of median-of-five

Set the candidate list as V[0] = First, V[1] = Last, V[2] = (First+Last)/2, V[3] = (First+Last)/4, V[4] = 3*((First+Last)/4)

.V[0]	V[3]	V[2]	V[4]	V[1]

Fig.2.2 Five candidates for the median-of-five

Median-of-five finds the median by

1. comparing between V[0] and V[1], if V[0] is greater than V[1] then it swaps V[0] and V[1];
2. comparing between V[1] and V[2], if V[1] is greater than V[2] then it swaps V[1] and V[2];

3. comparing between V[2] and V[3], if V[2] is greater than V[3] then it swaps V[2] and V[3];

4. comparing between V[3] and V[4], if V[3] is greater than V[4] then it swaps V[3] and V[4].

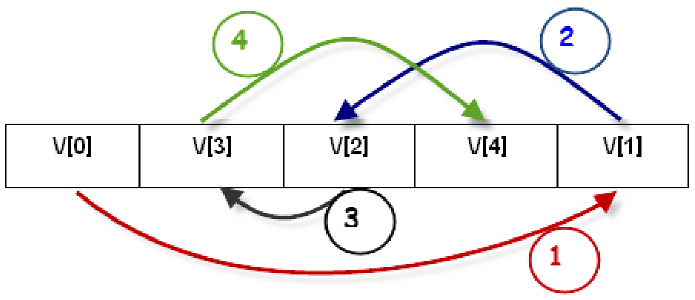


Fig.2.3 Sequence of comparison among five elements

As a result, median value of the 5 elements repositions to the middle position. This pivot selection method tends to return the actual median of whole array. Regarding this principle, it can be translated into pseudo code as following,

```
Procedure Median of five (A[ ], first, last)
    V[0] = first;
    V[1] = Last;
    V[2] = (first+last)/2;
    V[3] = (first+last)/4;
    V[4] = 3*((first+last)/4);

    For(i = 0  ; i< 5; i ++)
        For ( j = 0; j < 4 ; j++)
            IF ( A[V[j]] > A[V[j+1]] ) THEN
                Swap V[j] and V[j+1]
            ENDIF
        Return V[2];
        ENDFOR
    ENDFOR
```

END Median of five

2.2 Proposed the location of pivot elements.

One way to improve quicksort is to swap pivot elements toward the end (Split-end-partition [10]) or the middle (Collect-center-partition [12]) of the array. We are interested in improving collect-center-partition.

2.2.1 Collect-Center-Partitioning

Collect-center-partitioning [7] has been proposed by Kim and Park in 2009. It is a method for managing the redundant elements using four pointers i, j, ipL, ipR. The ipL pointer is used for partition elements less than pivot to the left. The ipR pointer is used for partition elements greater than pivot to the right. The i and j pointers are similar to pointers in quicksort. If every element of the array has the same value, CCsort will terminate immediately. This technique is suitable for data that has numerous repetitions of elements. Initially, CCsort begins the process of partitioning by scanning through elements that have values greater/less than pivot. Therefore, if *i* points to an element that has value less than pivot, that element will be repositioned to the ipL of the pivot. On the other hand, if *j* points to an element that has value greater than the pivot, that element will be repositioned to the ipR of the pivot. This process depicts in Fig. 2.4.

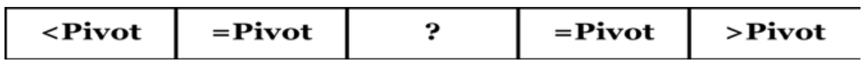


Fig. 2.4 Collect-center-partitioning

When partitioning is complete, collect-center-partitioning achieves 3 sub-groups of elements 1) the first group contains elements that have values less than the pivot 2) the second group contains elements that have values are equivalent to pivot and 3) the third group contains elements that have values are greater than the pivot.

< pivot	= pivot	> pivot

Fig. 2.5 Three sub-groups of elements after partitioning process is complete

From previous paragraph, it can be translated into the pseudo code as following (in this case, selected pivot is the rightmost array),

```
Function Split-end partition (A[ ], left, right)
    i = pl = left;
    j = pr = right – 1;
    pivot = A[right];

    LOOP
        WHILE (A[i] < pivot AND $i \le j$) DO
                Swap A[i] and A[pl];
                pl = pl + 1;
            ENDIF
            i = i + 1;
        ENDWHILE
        WHILE (A[j] > pivot AND $i \le j$) DO
                Swap A[j] and A[pr];
                pr = pr – 1;
            ENDIF
            j = j – 1;
        ENDWHILE
        IF (i> j) THEN
            Exit LOOP;
        ENDIF
        Swap A[i] and A[pr];
        Swap A[j] and A[pl];
    END LOOP
```

Consider the list of elements: 6, 8, 3, 6, 4, 6, 9, 6, 6. The collect-center-partition performs step-by-step as follow.

First, it selects the right element of array as the pivot. Then, it assigns *i* and *ipL* as a pointer at the first position. Then, it assigns *j* and *ipR* as a pointer at the last position.

6	**8**	**3**	**6**	**4**	**6**	**9**	**6**	**6**
i, ipL								*j, ipR*

It compares element at pointer *i* with the pivot and moves pointer *i* to the right until it founds an element that does not equal to the pivot (value 6). In this case, pointer *i* stops at 2nd element which value is 8.

6	**8**	**3**	**6**	**4**	**6**	**9**	**6**	**6**
ipL	*i*							*j, ipR*

It compares element at pointer *j* with the pivot and moves pointer *j* to the left until it founds an element that does not equal to the pivot (value 6). However, the element at 7th position which value 9 is greater than the pivot. As a result, it swaps between the element at 7th and the pivot. After that, pointer *j* keeps moving to the left until it finds an element that does not equal to the pivot (value 6). Note that, the element at 5th position corresponds to this condition. Therefore, pointer *j* stops at the element at 5th position.

6	**8**	**3**	**6**	**4**	**6**	**6**	**6**	**9**
ipL	*i*			*j*			*ipR*	

It swaps elements at pointer *i* with value 8 and at pointer *ipR* with value 6 and moves pointer *ipR* one position to the left. Then, it swaps elements at pointer *j* with value 4 and at pointer *ipL* with value 6 and moves pointer *ipL* one position to the right.

4	**6**	**3**	**6**	**6**	**6**	**6**	**8**	**9**
	i, ipL			*j*		*ipR*		

It compares elements at pointers *i* with the pivot and moves pointer *i* to the right until it finds an elements that does not equal to the pivot. However, the element at 3rd position with value 3 is less than the pivot. Therefore, it swaps this element with element at pointer *ipL* and moves pointer *ipL* one position to the right. Then, it moves pointer *i* to the right as it stops at the element 8th position with value 8.

4	**3**	**6**	**6**	**6**	**6**	**6**	**8**	**9**
		ipL		*j*		*ipR*	*i*	

It compares elements at pointers *i* with *j*. As pointer *i* points at the element at 8th position while pointer *j* points at the element at 5th position. Since pointer *i* is at the position after pointer *j*, this concludes the partitioning process.

4	**3**	**6**	**6**	**6**	**6**	**6**	**8**	**9**

At the end of the partitioning process, every element that has value equal to the pivot is gathered in the middle part of the array. Elements that have values less than pivot are gathered at the left sub-array while elements that have values greater than pivot are gathered at the right sub-array. The process repeats for sub-arrays until the stopping criteria is met. Consequently, the data will be sorted in a sequential order.

2.2.2 Adjacent Pivot Quicksort

Adjacent pivot quicksort (APQsort) uses predecessor pivot or successor pivot and pivot for partitioning. If both predecessor and successor pivot are used for partitioning, more comparisons and swaps are needed. APQsort uses three patterns for partitioning; partition by using predecessor pivot (PD partition), partition by using successor pivot (SC partition) and partition by split-end-partition [7]. It identifies pseudo-predecessor pivot (p-PD) and pseudo-successor pivot (p-SC).

1) First case: If the pivot equals to pseudo-predecessor pivot and pseudo-successor pivot, then it uses split-end-partition.
2) Second case: If the difference of pivot and pseudo-predecessor pivot is less than the difference of pivot and pseudo-successor pivot, then

a. If the pivot equals to pseudo-predecessor pivot, then it uses SC partition.

b. Otherwise, it uses PD partition.

3) Third case: If the difference of pivot and pseudo-successor pivot is less than the difference of pivot and pseudo-predecessor pivot, then

a. if the pivot equals to pseudo-successor pivot, then it uses PD partition;

b. otherwise, it uses SC partition.

The PD partition and SC partition are shown below.

The PD partition

1) For each element that is less than the pivot, it moves that element to the left sub-array and if it is greater than or equal to pseudo-predecessor pivot then it is defined as a new pseudo-predecessor pivot and moves to the leftmost position.

2) For each element that is greater than or equal to the pivot, it moves that element to the right sub-array.

After finishing PD partition, the value of pseudo-predecessor pivot will equal to the real predecessor pivot. It moves the elements equal to the pivot and predecessor pivot to the correct position as shown in Fig.2.6. In case of the predecessor pivot position adjacent to the pivot position, all elements in left sub-array are sorted and no recursive call is needed.

<PD	**=PD**	**<Pivot**	**>Pivot**	**=Pivot**

Fig. 2.6 APQsort with predecessor pivot partition

The SC partition

1) For each element that is less than or equal to the pivot, it moves that element to the left sub-array.

2) For each element that is greater than the pivot, it moves that element to the right sub-array and if it is less than or equal to pseudo-successor pivot then it is defined as a new pseudo-successor pivot and moves to the rightmost position.

After finishing SC partition, the value of pseudo-successor pivot will equal to the real successor pivot. It moves the elements equal to pivot and successor pivot to the correct position as shown in Fig.2.7. In case of the successor pivot position adjacent to the pivot position, all elements in the right sub-array are sorted and no recursive call is needed.

=Pivot	**<Pivot**	**>Pivot**	**=SC**	**>SC**

Fig.2.7 APQsort with successor pivot partition

Both processes generate three sub-arrays; the left sub-array, the middle sub-array (contains only pivot) and the right sub-array but the characteristics of elements in sub-arrays after partitioning are shown in table 2.1

Table 2.1 APQsort partition characteristics

Subarray	PD partition	SC partition	Split-end partition
The left sub-array	element <PD	Element < pivot	element < pivot
The middle sub-array	element = PD and element = pivot	element = pivot and element = SC	element = pivot
The right sub-array	element > pivot	element >SC	element > pivot

The pseudo code of APQsort is as followed.

```
Function APQsort (A[ ], left, right)
    IF size of list A$\leq M$ THEN
        Call InsertionSort(A, left, right);
    ELSE
        Select pivot with Median of three;
        IF (pivot – p-PD == p-SC – pivot) THEN
            {ind_L, ind_R} = Split-end partition(A, left, right);
        ELSE IF (pivot – p-PD $\leq$ p-SC – pivot) THEN
            IF (pivot – p-PD $\neq$ 0) THEN
```

```
                {ind_L, ind_R} = PD partition(A, left, right);
            ELSE
                {ind_L, ind_R} = SC partition(A, left, right);
            ENDIF
        ElSE
            IF (p-SC – pivot $\neq$ 0) THEN
                {ind_L, ind_R} = SC partition(A, left, right);
            ELSE
                {ind_L, ind_R} = PD partition(A, left, right);
            ENDIF
        ENDIF

        Recursively call APQsort(A, left, ind_L);
        Recursively call APQsort(A, ind_R, right);
    ENDIF
End APQsort

Function PD partition (A[ ], left, right)
    Set values of i, j, bound left (indL), bound right (indR), pivot and predecessor pivot (PD);

    LOOP
        WHILE (A[i] < pivot AND i$\leq$ j) DO
            IF (A[i] $\geq$ PD) THEN
                IF (A[i] == PD) THEN
                    Swap A[i] and A[indL];
                    indL =  indL + 1;
                ELSE
                    Update value of PD and Swap A[i] and A[indL];
                    indL = indL + 1;
                ENDIF
            ENDIF
            i = i + 1;
        ENDWHILE
        WHILE (A[j] >= pivot AND i≤ j) DO
            IF (A[j] == pivot) THEN
                Swap A[j] and A[indR];
                indR = indR – 1;
            ENDIF
            j = j – 1;
        ENDWHILE
        IF (i≥ j) THEN
            Exit LOOP;
        Swap A[i] and A[j];
    ENDLOOP

    Move the elements are equal to PD and pivot to the correct positions;
    Return index of PD and pivot;
End PD partition

Procedure SC partition (A[ ], left, right)
    Set values of i, j, bound left (indL), bound right (indR), pivot and successor pivot (SC);

    LOOP
        WHILE (A[i] <= pivot AND i≤ j) DO
            IF (A[i] == pivot) THEN
                Swap A[i] and A[indL];
                indL = indL + 1;
            ENDIF
            i = i + 1;
        ENDWHILE
        WHILE (A[j] > pivot AND i≤ j) DO
            IF (A[j] <= SC) THEN
                IF (A[j] == SC) THEN
                    Swap A[j] and A[indR];
                    indR = indR – 1;
                ELSE
                    Update value of SC and Swap A[j] and A[indR];
                    indR = indR – 1;
                ENDIF
            ENDIF
            j = j – 1;
        ENDWHILE
        IF (i≥ j) THEN
            Exit LOOP;
        Swap A[i] and A[j];
    ENDLOOP

    Move the elements are equal to pivot and SC to the correct positions;
    Return index of pivot and SC;
End SC partition
```

2.3 Related Researches

In 1962, Hoare [5] was the pioneer of proposing quicksort with the random pivot from the list of elements.

In 1965, Scowen [16] proposed “Quickersort” that applied similar principle of quicksort but it selected pivot at the middle of the data. This algorithm achieved O(n log n) if that data had been sorted, which meant the middle element was the median of the list.

In 1969, Singleton [17] proposed the new heuristic method of selecting pivot close to the median of the list. This estimation of data came from calculating median from three different elements: the first element, the middle element, and the last element. This method was known as “median-of-three”. Certainly, this method of selecting pivot ensured a better median approximation.

In 1978, Sedgewick [8] proposed quicksort without recursion and applied with median-of-three and call insertion sort when sub-array less than 9 or 10 elements. This method improved quicksort competency for approximately 20 percent.

In 1980, Cook and Kim [18] proposed the new sorting method. This method was purposely for data that nearly sorted. It, initially, scanned through elements and collects all elements that were not in the right order. If collected elements were more than 30 elements, it called quicksort to rearrange these elements. If collected elements were less than 30 elements, it applied insertion sort. Finally, it combined two sets of sorted data into a single complete sorted data.

In 1983, Motzkin [19] proposed “Meansort”. It used the mean instead of the median; however, it included function of calculating average of each sub-group of data during the partition process. This meansort algorithm surprisingly reduced the probability of occurrence of quicksort worst case since selected pivot was the average of the data.

In 1985, Wainwright [20] proposed “Bsort”. It included function of sequence inspection whether data was correctly in a sequential order. This algorithm was designed base on the principle of bubble sort. It swop adjacent elements immediately. Bsort worked effectively on data that almost sorted.

In 1987, Wainwright [21] proposed “QSORTE” as a development from bsort algorithm. Qsort omitted process of swapping elements and improved the method of data checking.

In 1993, Bentley and Mcilroy [10] proposed a new method of selecting pivot from median-of-median. First it split data into three groups and selected the median-of-three in each group. Second it determined the median of these three numbers. This method additionally proposed a new way of partitioning that dealing with redundant elements.

In 1996, Sarwar and colleagues [22] proposed method of selecting pivot from median-of-three, median-of-five, median-of-nine, and median-of-seventeen. This method of selecting pivot depended on numbers of elements in the data set.

In 2006, Edmondson [23] proposed “M Pivot Sort”. It used several pivots instead of a single pivot and if sub-groups had elements less than 15, then it applied insertion sort instead of M Pivot sort.

In 2007, Jidol and colleagues [24] proposed method of developing quicksort by applied “Successive Difference” principle as to check the sequence whether the sequence was sorted. This “Successive Difference” principle was capable of applying with ordinary sort and reversed sort.

In 2007, Aminu Mohammed and Mohammed Othman [4] proposed method of selecting pivot by applying “Random Index for minimizing the execution time” principle.

In 2009, Kim and Park [12] proposed “collect-center”. It improved the ‘split-end’ partitioning in case of many equal elements by moving the pivot to the middle sub-array. It reduced the numbers of recursive calls. The running time was better than “Split-end” partitioning.

In 2010, Rungpiphop and Sinapiromsaran [13] proposed adjacent pivot quicksort (APQsort) which improved the performance of quicksort for data with repeated elements by reducing the number of recursive calls. APQsort utilized additional elements called “pivot predecessor” or “pivot successor” combined with the split-end partition.

Chapter III

DESIGN AND METHODOLOGY

This research proposes quicksort by using predecessor and successor pivot. It improves quicksort in case of repeat elements. This algorithm decreases the number of recursive calls by using the element in front of the pivot called "predecessor (PD)" and the element behind the pivot called "successor (SC)". This chapter describes CC5sort by using predecessor and successor pivot, instead of using Split-end partition, it uses collect-center-partition.

3.1 Collect-center partition using predecessor and successor pivot

The concept of CC5sort is similar to APQsort but CC5sort using collect-center partition; which it moves the element that equals to the pivot to the middle instead of swapping it to the left or right sub-array. Because of its complexity, it uses more comparisons and swaps. It selects the pivot by using median-of-five in order to avoid equal value of the predecessor pivot and successor pivot. Due to the repeat elements, CC5sort outperforms APQsort and CCsort. CC5sort chooses the partition from CC5sort PD partition, CC5sort SC partition and collect-center partition. It determines the pseudo-predecessor pivot (p-PD) and pseudo-successor pivot (p-SC) according to the following cases;

1) First case: If the pivot equals to pseudo-predecessor pivot and pseudo-successor pivot, then it uses collect-center partition.
2) Second case: If the difference of pivot and pseudo-predecessor pivot is less than the difference of pivot and pseudo-successor pivot, then
    a. if the pivot equals to pseudo-predecessor pivot, then it uses CC5sort SC partition:
    b. otherwise, it uses CC5sort PD partition.
3) Third case: If the difference of pivot and pseudo-successor pivot is less than the difference of pivot and pseudo-predecessor pivot, then
    a. if the pivot equals to pseudo-successor pivot, then it uses CC5sort PD partition;

b. otherwise, it uses CC5sort SC partition.

The CC5sort PD partition and CC5sort SC partition are shown below.

The CC5sort PD partition

1) For each element that is less than the pivot, it moves that element to the left sub-array and if it is greater than or equal to pseudo-predecessor pivot then it is defined as a new pseudo-predecessor pivot and moves to the leftmost position.

2) For each element that is greater than the pivot, it moves that element to the right sub-array. If the elements that equals to the pivot it moves to middle by collect-center partition process.

After finished the partition, the value of pseudo-predecessor pivot will equal to predecessor pivot. It moves the elements that equal to pivot and predecessor pivot to the correct position as show in Fig. 3.1. In case of predecessor pivot position nearly to pivot position, all elements in the left sub-array are sorted and no recursive call is needed.

<PD	**=PD**	**<Pivot**	**=Pivot**	**>Pivot**

Fig. 3.1 CC5sort with predecessor pivot partition

The CC5sort SC partition

1) For each element that is less than pivot, it moves that element to the left sub-array. If the elements that equals to the pivot it moves to middle by collect-center partition process.

2) For each element that is greater than the pivot, it moves that element to the right sub-array and if it is less than or equal to pseudo-successor pivot then it is defined as a new pseudo-successor pivot and moves to the rightmost position.

After finished the partition, the value of pseudo-successor pivot will equal to successor pivot. It moves the elements equal to pivot and successor pivot to the correct position as shown in Fig 3.2. In case of successor pivot position nearly pivot position, all elements in the right sub-array are sorted and no recursive call is needed.

<Pivot	**=Pivot**	**>Pivot**	**=SC**	**>SC**

Fig. 3.2 CC5sort with successor pivot partition

After finishing partition, both processes generate three sub-arrays; left sub-array, sorted sub-array and right sub-array but characteristics of sub-arrays after are shown in table 3.1.

Table 3.1. Collect-center partition characteristics

Sub-array	PD partition	SC partition	Collect-center partition
The left sub-array	element < PD	element < pivot	element < pivot
The middle sub-array	element = PD and element = pivot	element = pivot and element = SC	element = pivot
The right sub-array	element > pivot	element > SC	element > pivot

From the previous paragraph, it can be translated into the pseudo code as following.

```
Function CC5sort (A[ ], left, right)
    IF size of list A $\leq$ $M$ THEN
        Call InsertionSort (A, left, right);
    ELSE
        Select pivot with Median of three;
        IF (pivot – p-PD == p-SC – pivot) THEN
            {ind_L, ind_R} = Split-end partition(A, left, right);
        ELSE IF (pivot – p-PD $\leq$ p-SC – pivot) THEN
            IF (pivot – p-PD $\neq$ 0) THEN
                {ind_L, ind_R} = PD partition(A, left, right);
            ELSE
                {ind_L, ind_R} = SC partition(A, left, right);
            ENDIF
        EISE
```

```
            IF (p-SC – pivot $\neq$ 0) THEN
                {ind_L, ind_R} = SC partition(A, left, right);
            ELSE
                {ind_L, ind_R} = PD partition(A, left, right);
            ENDIF
        ENDIF

        Recursively call CC5sort(A, left, ind_L);
        Recursively call CC5sort(A, ind_R, right);
    ENDIF
End CC5sort

Function PD partition (A[ ], left, right)
    Set values of i, j, bound left (indL), bound right (indR), pivot and predecessor pivot (PD);

    LOOP
        WHILE (A[i] < pivot AND i $\leq$ j) DO
            IF (A[i] $\geq$ PD) THEN
                IF (A[i] == PD) THEN
                    Swap A[i] and A[indL];
                    indL =  indL + 1;
                ELSE
                    Update value of PD and Swap A[i] and A[indL];
                    indL = indL + 1;
                ENDIF
            ENDIF
            i = i + 1;
        ENDWHILE
        WHILE (A[j] > pivot AND i $\leq$ j) DO
                Swap A[j] and A[indR];
                indR = indR – 1;
        ENDWHILE
        IF (i $\geq$ j) THEN
```

```
            Exit LOOP;
        Swap A[i] and A[j];
    ENDLOOP

    Return index of PD and pivot;
End PD partition

Procedure SC partition (A[ ], left, right)
    Set values of i, j, bound left (indL), bound right (indR), pivot and successor pivot (SC);

    LOOP
        WHILE (A[i] <= pivot AND i ≤ j) DO
                Swap A[i] and A[indL];
                indL = indL + 1;
        ENDWHILE
        WHILE (A[j] > pivot AND i ≤ j) DO
            IF (A[j] <= SC) THEN
                IF (A[j] == SC) THEN
                    Swap A[j] and A[indR];
                    indR = indR – 1;
                ELSE
                    Update value of SC and Swap A[j] and A[indR];
                    indR = indR – 1;
                ENDIF
            ENDIF
            j = j – 1;
        ENDWHILE
        IF (i ≥ j) THEN
            Exit LOOP;
        Swap A[i] and A[j];
    ENDLOOP

    Return index of pivot and SC;
End SC partition
```

Consider the list of elements; 1, 6, 3, 8, 8, 4, 9, 6, 6. The collect-center partition performs step-by-step as follow.

It uses median-of-five to select the pivot from the list. It chooses five elements from;

1) The first element: the first array that is the element with value 1.
2) The second element: It selects from element in between the first and the middle array ($\lceil$(first+last)/4$\rceil$) that is the element with value 3.
3) The third element: It selects from the middle array ((first+last)/2) that is the element with value 8.
4) The fourth element: It selects from element in between the middle and the last array ($\lceil$3*(first+last)/4$\rceil$) that is the element with value 9.
5) The fifth element: It selects from the last array that is the element with value 6.

Then it sorts five elements and selects the pseudo-predecessor pivot, pseudo-successor pivot and pivot. It assigns the pivot equal to the element with value 6, pseudo-predecessor pivot equal to the element with value 1 and pseudo-successor pivot equal to the element with value 9. It finds that the difference of pseudo-successor pivot and the pivot is greater than the difference of pseudo-predecessor pivot and the pivot. So, it uses SC partition, the pivot is on the middle of array, pseudo-successor pivot is on the right array. Let *i* and *ipL* be indices point to the first position, *j* and *ipR* be indices point to the second to the last position.

1	**6**	**3**	**8**	**6**	**4**	**8**	**6**	**9**
i, ipL				**Pivot**			***j, ipR***	**p-SC**

It compares elements at index *i* and the *pivot* by moving index *i* to the right, so it checks an element against the pivot (pivot value =6), it finds the element at 1st position with value 1 less than the pivot, then it swaps the element at 1th position and the element at index point to *ipL* and moves index *ipL* to the right one position. It compares elements at index *i* and the *pivot* by moving index *i* to the right again and finds the element at 3th position with value 3, then it swaps the elements at 3th position and the element at index point to *ipL* and moves index *ipL* to the right. Index *i* stop at 4th position with value 8.

1	**3**	**6**	**8**	**6**	**4**	**8**	**6**	**9**
		ipL	**i**	**Pivot**			***j, ipR***	**p-SC**

It compares the elements at index *j* and the *pivot* by moving index *j* to the left position until it finds the elements not equal to the pivot (pivot value =6) and finds the elements greater than or equal to pseudo-successor pivot, it finds the element 7th position with value 8 which is greater than the pivot, then it compares element at 7th position and pseudo-successor pivot. It finds that the pseudo-successor pivot is greater than the element at 7th position, then it swaps elements at 7th position and index *ipR*, and then moves indices *ipR* and ***p**-SC* to the left position. It compares the elements at index *j* and the *pivot* by moving index *j* to the left position again and finds the element at 6th position with value 4 is greater than the pivot, index *j* stops at this index.

1	**3**	**6**	**8**	**6**	**4**	**6**	**8**	**9**
		ipL	**i**	**Pivot**	**j**	**ipR**	**p-SC**	

Then, it swaps an elements at index *i* and index *j*.

1	**3**	**6**	**4**	**6**	**8**	**6**	**8**	**9**
		ipL	**i**	**Pivot**	**j**	**ipR**	**p-SC**	

Then, it compares elements at index *i* and the *pivot*, it checks an elements against 6, it finds the element at 4th position with value 4 is less than the pivot then it swaps the element at 4th position and index *ipL* and moves index *ipL* to the right. Index *i* stops at 6th position with value 8.

1	**3**	**4**	**6**	**6**	**8**	**6**	**8**	**9**
			ipL	**Pivot**	**j, i**	**ipR**	**p-SC**	

It compares an elements at index *j* and the *pivot* by moving index *j* to the left until it finds the elements not equal to the pivot (pivot value = 6) and finds the elements are greater than or equal to pseudo-successor pivot. It finds the element 6th position with value 8 is greater than the pivot. Then, it compares elements at index *j* and index *p-SC*, Index *j* is equal to index *p-SC*. Then, it swaps elements between index *j* and index *ipR,* it moves *ipR* to left one index, next it checks the element against the pivot to the right. Index *j* stops at 5th position with value 6. It stops SC partition because index *i* is greater than index *j*.

1	**3**	**4**	**6**	**6**	**6**	**8**	**8**	**9**
			ipL	**Pivot, j**	**i, ipR**		**p-SC**	

After finishing SC partition, the process generates three sub-arrays; left sub-array which elements are less than the pivot, the middle sub-array which elements are equal to the pivot and the predecessor pivot and the right sub-array which elements are greater than the successor pivot. Then it partitions the remaining element similar to the concept above until it can not partition, CC5sort will stop. The result is the element that sorted from the minimum to maximum.

1	**3**	**4**	**6**	**6**	**6**	**8**	**8**	**9**

Chapter IV

Experiment and Result

From previous chapter, we explained CC5sort. This chapter, we describe our experiments and analyze our results on the effectiveness of our algorithm comparing to other sorting algorithms.

4.1 The experiments

4.1.1 Tools

All sorting algorithms were implemented with C#.net from Microsoft Visual C# 2010. We run experiments on Intel® CPU Core™ 2 DUO 2.2 GHz and 3GB of RAM under Microsoft window XP system.

4.1.2 The dataset

Our experiments run on string type data and integer type data.

String type has 4 data sets. All data sets contain 9 characters per string except the fourth data set has 5 characters per string.

1) Data set 1 uses uniform random distribution. It contains three different sizes (1000, 5000 and 10000 elements). Each size consists of 10 data files.
2) Data set 2 uses random data that is nearly sorted. This data set consists of 10000 elements with five un-sorted ratios, 0, 0.01, 0.03, 0.1 and 0.3. Each size consists of 10 data files. The un-sorted ratio can be calculated from the number of un-sorted elements divided by the number of all elements.
3) Data set 3 uses random data that nearly reverse sorted. This data set consists of 10 elements and five reverse sorted ratio, 0, 0.01, 0.03, 0.1 and 0.3. Each ratio consists of 10 data files.

4) Data set 4 uses random data that have many repeated elements. This data set consists of 10000 elements. It has 5 characters per string and four distinct ratios, 0.0032, 0.0243, 0.1024 and 0.3125. Each ratio consists of 10 data files. The distinct ratios can be calculated from the number of probability per position of string.

Integer type has 3 data sets as below;

1) Data set 1 uses uniform random distribution. It generates a range of numbers between 100 and 200 and consists of 10000 elements per file.
2) Data set 2 uses random data that nearly sorted. This data consists of 10000 elements and un-sorted ratio is 0.3.
3) Data set 3 uses random data that nearly reverse sorting. This data consists of 10000 elements and reverse sorting ratio is 0.3.

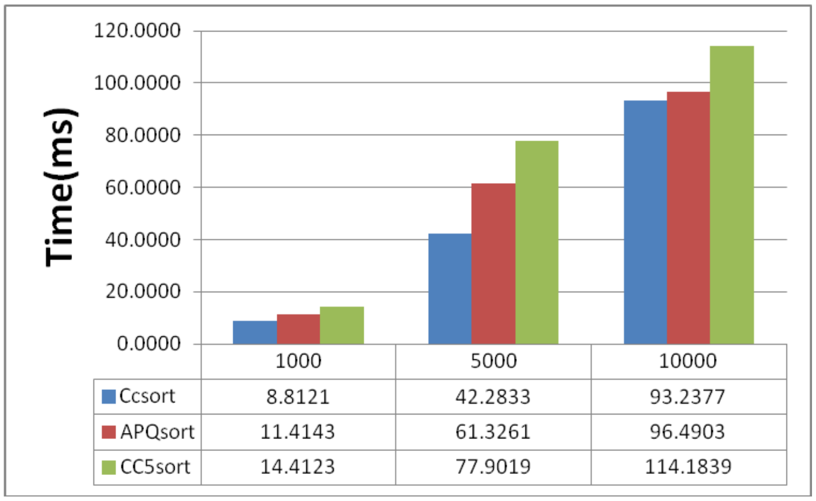
Each data set contains 10 data files.

4.2 Results

The process of CC5sort uses predecessor or successor pivot combine with collect-center partitioning can reduce the times of recursive calls by selects pivot by using median-of-five. We report the average time of 10 experiments per data set by using the command called stopWatch.Start() and stopWatch.End(); in Microsoft Visual C# 2010. The sorting algorithms used to compare with our concept are CCsort and APQsort.

4.2.1 Results for strings

The results from the data set 1 show that CCsort is the fastest while APQsort is faster than CC5sort on average as see in Fig. 4.1 due to the number of small repeated elements. This is because CCsort does not determine SC and PD. Also, APQsort does not perform collect-center partition. Therefore, both processes take more time to find the elements equal to the pivot, real PD and SC see in Fig 4.1. CCsort uses more swaps, comparisons and recursive calls than APQsort and CC5sort. Therefore, CCsort places only pivot to the correct position.



	1000	5000	10000
🟦 Ccsort	8.8121	42.2833	93.2377
🟥 APQsort	11.4143	61.3261	96.4903
🟩 CC5sort	14.4123	77.9019	114.1839

Fig. 4.1 Time complexity for random sort data

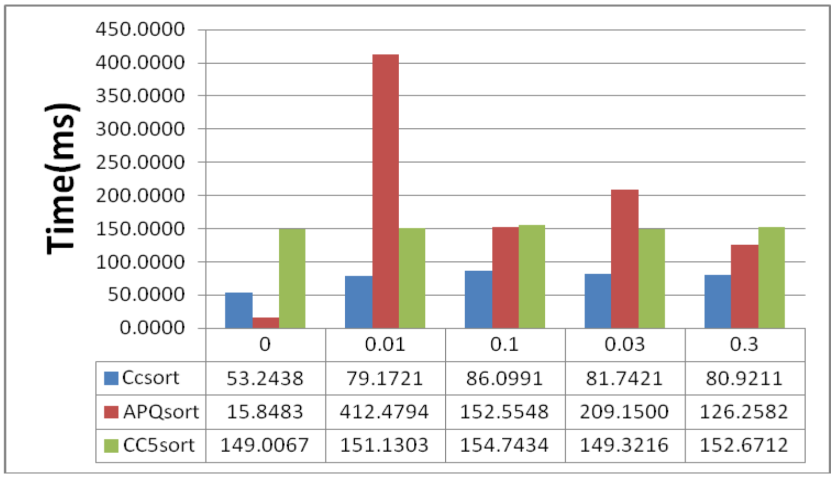
Table 4.1 Show 3 processes for random data set of string

Size	Method	Compare	Swap	Recursive
1000	Ccsort	54,976	60,684	904
	APQsort	14,723	2,320	131
	CC5sort	23,925	8,257	133
5000	Ccsort	362,783	406,329	4,550
	APQsort	96,791	14,331	655
	CC5sort	158,349	54,984	672
10000	Ccsort	828,496	927,106	9,133
	APQsort	212,525	31,066	1,310
	CC5sort	346,298	121,124	1,343

The results from data set 2 can be concluded into two points.

1) For data with un-sorted ratio equal to zero, APQsort is the fastest while CCsort is faster than CC5sort on average which can be seen in Fig. 4.2 because APQsort can detect the sorted sub-array for every partition.

2) For data with un-sorted ratios nearly to zero (0.01, 0.03, 0.1 and 0.3), CCsort is the fastest among the others because APQsort and CC5sort need to find the difference of pseudo-successor pivot and the pivot and the difference of pseudo-predecessor pivot and the pivot. However CCsort does not require this computation. Although our method has a slow running time but the number of swaps, comparisons and recursive calls are less than CCsort. The number of recursive calls of CC5sort is less than APQsort and CCsort above 50% because the pivot position is near the middle of the data as shown in Table.4.2 and Fig.4.2.



	0	0.01	0.1	0.03	0.3
🟦 Ccsort	53.2438	79.1721	86.0991	81.7421	80.9211
🟥 APQsort	15.8483	412.4794	152.5548	209.1500	126.2582
🟩 CC5sort	149.0067	151.1303	154.7434	149.3216	152.6712

Fig. 4.2 Time complexity for nearly sort data

Table 4.2 the number of 3 processes for nearly sort data

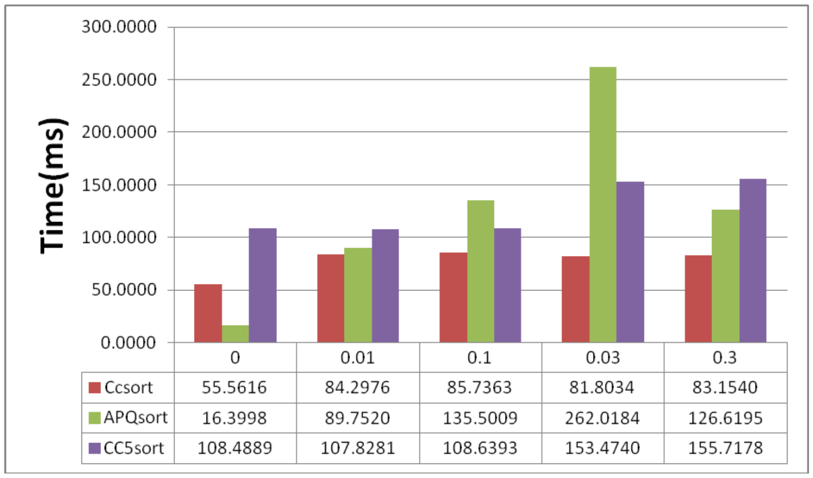
Unsorted Ratio	Method	Compare	Swap	Recursive
0	Ccsort	555,682	538,802	5,621
	APQsort	25,221	9,991	9
	CC5sort	299,294	75,326	1,022
0.01	Ccsort	745,315	785,680	8,647
	APQsort	901,297	357,913	3,018
	CC5sort	312,241	95,488	1,285
0.03	Ccsort	784,842	836,890	7,942
	APQsort	382,476	26,180	2,508
	CC5sort	314,446	100,624	1,289
0.1	Ccsort	811,806	878,068	8,570
	APQsort	272,259	26,773	1,641
	CC5sort	322,327	107,105	1,309
0.3	Ccsort	703,252	786,508	9,166
	APQsort	212,608	28,591	1,361
	CC5sort	336,331	113,298	1,328

The results from data set 3 can be concluded into two points. This data set is not related from data set type 2.

1) For data from un-sorted ratio equal to zero, APQsort is the fastest processing while CCsort is faster than CC5sort on average which can be seen in Fig. 4.2 because APQsort can detect the sorted sub-array for every partition.

2) For data from un-sorted ratios nearly to zero (0.01, 0.03, 0.1 and 0.3), CCsort is the fastest among the others because APQsort and CC5sort need to find the difference of pseudo-successor pivot and the pivot and the difference of pseudo-predecessor pivot and the pivot. However CCsort does not require this computation. Although our method has a slow running time but the number of swaps, comparisons and recursive calls are less than CCsort. The number of recursive calls of CC5sort is less than APQsort and CCsort above 50% because the pivot position is near the middle

of the data. The number of comparisons of CC5sort is less than APQsort and CCsort because it does not compare the elements that equal to the pivot to the correct position after finishing the process like APQsort as shown in Table.4.3 and Fig.4.3.



	0	0.01	0.1	0.03	0.3
🟥 Ccsort	55.5616	84.2976	85.7363	81.8034	83.1540
🟩 APQsort	16.3998	89.7520	135.5009	262.0184	126.6195
🟪 CC5sort	108.4889	107.8281	108.6393	153.4740	155.7178

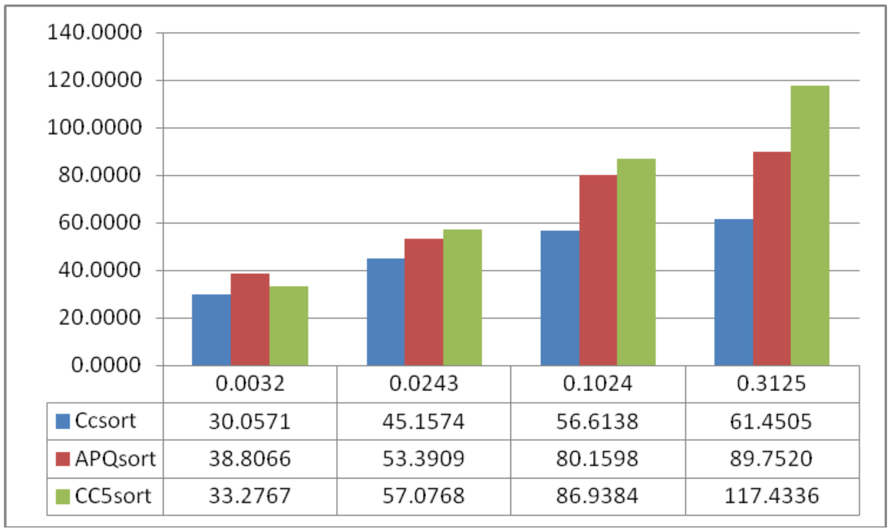
Fig 4.3 Time complexity for nearly reverse sort data

Table 4.3 the number of 3 processes for nearly reverse sort data

Unsorted Ratio	Method	Compare	Swap	Recursive
0	Ccsort	555,682	566,297	5,621
	APQsort	39,132	14,994	9
	CC5sort	310,320	80,569	1,022
0.01	Ccsort	708,890	765,185	9,250
	APQsort	944,042	25,915	2,454
	CC5sort	319,778	104,786	1,289
0.03	Ccsort	750,569	822,166	8,387
	APQsort	537,630	28,356	2,436
	CC5sort	322,873	107,999	1,298
0.1	Ccsort	783,595	868,878	8,299
	APQsort	290,014	28,776	1,701
	CC5sort	330,054	111,247	1,311
0.3	Ccsort	725082.5	818480.6	9030.7
	APQsort	944041.5	25914.7	2453.9
	CC5sort	319777.7	104786.3	1289.3

The results from data set 4 can be concluded as following.

1) For four distinct ratios, 0.0032, 0.0243, 0.1024 and 0.3125, CCsort is the fastest process among the others because APQsort and CC5sort need to find the difference of pseudo-successor pivot and the pivot and the difference of pseudo-predecessor pivot and the pivot. However CCsort does not require this computation. Although our method has a slow running time but the number of swaps, comparisons and recursive calls are less than CCsort. The number of recursive calls of CC5sort is less than APQsort and CCsort because the pivot position is near the middle of the data as shown in Table.4.4 and Fig.4.4.



	0.0032	0.0243	0.1024	0.3125
🟦 Ccsort	30.0571	45.1574	56.6138	61.4505
🟥 APQsort	38.8066	53.3909	80.1598	89.7520
🟩 CC5sort	33.2767	57.0768	86.9384	117.4336

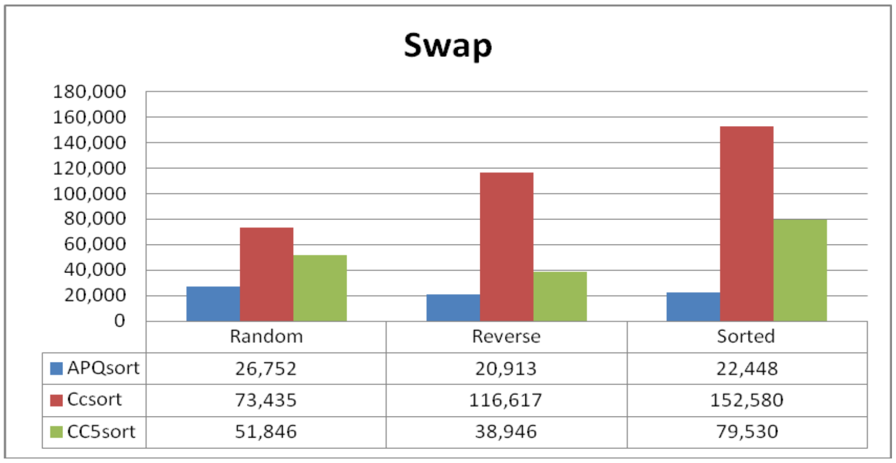
Fig 4.4 Time complexity for repeat element data

Table 4.4 the number of 3 processes for random data that have many repeat elements

Unrepeated	Method	Compare	Swap	Recursive
0.0032	CCsort	232,011	202,703	171
	APQsort	67,788	27,409	23
	CC5sort	108,361	43,236	17
0.0243	CCsort	385,659	355,320	1,331
	APQsort	105,724	31,060	182
	CC5sort	185,925	76,958	142
0.1024	CCsort	486,160	464,043	3,855
	APQsort	142,678	34,917	588
	CC5sort	247,545	98,877	490
0.3125	CCsort	537,406	530,422	5,578
	APQsort	157,264	34,187	796
	CC5sort	285,099	107,542	886

4.2.2 Results for integer

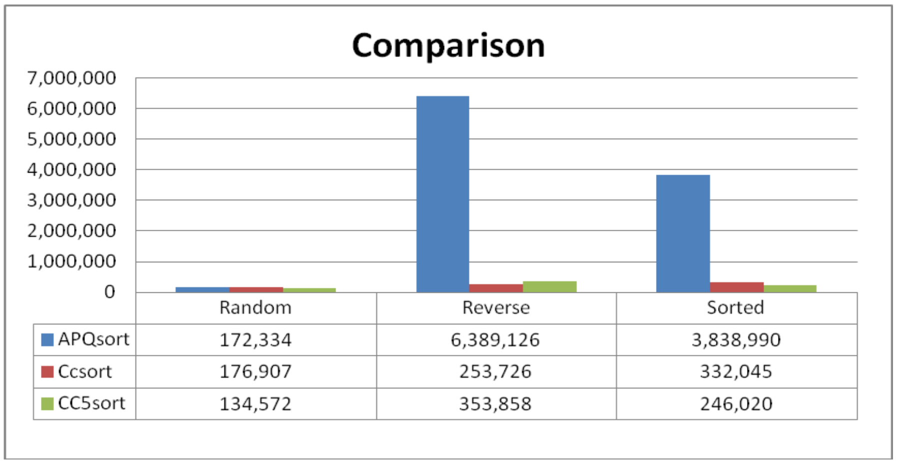
From three data sets as shown in Fig.4.5, APQsort has less the number of swaps than CCsort and CC5sort because these methods swap all elements not equal to pivot to the left or right sub-array for forcing pivots to the middle.



	Random	Reverse	Sorted
🟦 APQsort	26,752	20,913	22,448
🟥 Ccsort	73,435	116,617	152,580
🟩 CC5sort	51,846	38,946	79,530

Fig 4.5 Number of swap for integer

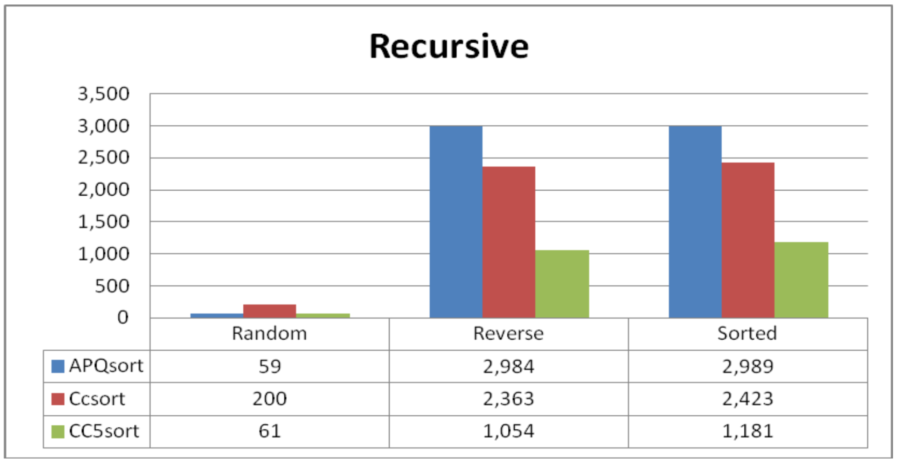
From three data sets as shown in Fig.4.6, APQsort has more number of comparisons than CCsort and CC5sort because after the partition is done, it needs to move a group of pivots to the middle, a group of PD elements to the front of pivot and a group of SC elements behind pivot.



	Random	Reverse	Sorted
🟦 APQsort	172,334	6,389,126	3,838,990
🟥 Ccsort	176,907	253,726	332,045
🟩 CC5sort	134,572	353,858	246,020

Fig 4.6 Number of comparisons for integer

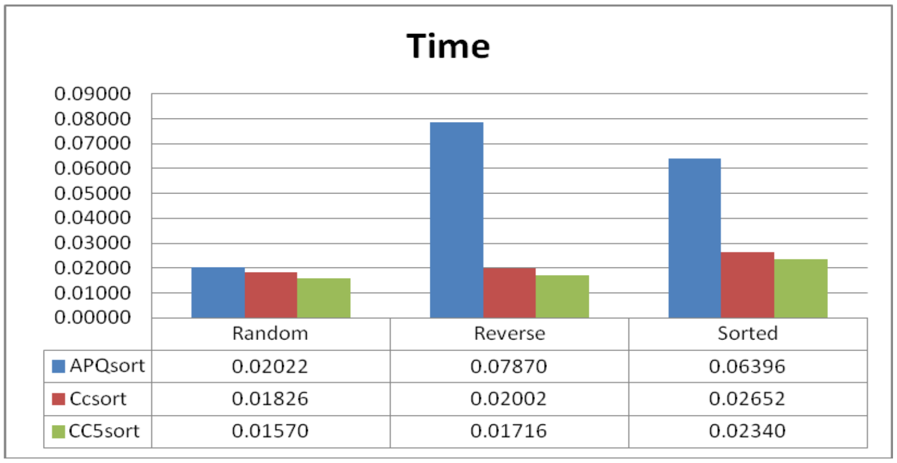
From three data sets as shown in Fig.4.7, CC5sort can reduce the number of recursive calls above 50% because it uses median-of-five to select the pivot position closer to the middle of the data.



	Random	Reverse	Sorted
🟦 APQsort	59	2,984	2,989
🟥 Ccsort	200	2,363	2,423
🟩 CC5sort	61	1,054	1,181

Fig 4.7 Number of recursive for integer

From three data sets as shown in Fig.4.8, CC5sort has the best time complexity because it can decrease the number of recursive calls.



	Random	Reverse	Sorted
🟦 APQsort	0.02022	0.07870	0.06396
🟥 Ccsort	0.01826	0.02002	0.02652
🟩 CC5sort	0.01570	0.01716	0.02340

Fig 4.8 Time complexity for integer

Chapter V

CONCLUSIONS

This research has proposed quicksort by using the predecessor and the successor pivot with collect-center partitioning called CC5sort. CC5sort improves quicksort in case of repeat elements. This algorithm decreases the number of recursive calls by using elements in front of pivot, called “predecessor (PD)” and the element behind the Pivot, called “successor (SC)”. It reduces the number of comparisons by moving the pivot during the partition process instead of swapping pivots to the left or right sub-array.

From the experiments, we can conclude the following.

1) The number of swaps: The number of swaps from CC5sort is greater than APQsort in all data types because the data sets contain small repeated elements. The number of swaps used by CC5sort is less than CCsort in all data types because CC5sort uses predecessor (PD) and successor (SC) pivots for keeping the elements.

2) The number of comparisons: The number of comparisons from CC5sort is greater than CCsort and APQsort in nearly sorted data (unsorted ratio 0.01 and 0.03) and nearly reverse sorted data (unsorted ratio 0.01 and 0.03) because CC5sort moves group of pivot to the middle and group of PD or SC to the correct position.

3) The number of recursive calls: The number of recursive calls from CC5sort is less than CCsort and APQsort above 50% in all data sets except random data because it uses the median-of-five to select the pivot. Therefore, it gets the pivot value close to the median.

4) The number of time complexity: The number of time complexity from CC5sort is worse than CCsort and APQsort in case of string type because CC5sort finds the difference of pseudo-successor pivot and the pivot and the difference of pseudo-predecessor pivot

and the pivot. Those processes require the conversion of strings to numbers which causes CC5sort perform slowly.

REFERENCES

[1] Cormen, T.H., Leiserson C.E., Rivest, R.L. and Stein, C. 2001. INTRODUCTION TO ALGORITHMS. Cambridge, MA : MIT Press.

[2] Levitin A, C. 2007. Introduction to the design & analysis of algorithms, 2nd edition. Pearson Education, Inc.

[3] Prof. Dr. Chidchanok L, C. 2000. Analysis & Design of Algorithms, SUM System Company Limited.

[4] Mohammed A, Othman M, 2007. Comparative Analysis of Some Pivot Selection Schemes for Quicksort Algorithm. Asian Network for Scientific Information.

[5] Hoare, C.A.R. 1962. Algorithm 63 Partition and Algorithm 64 Quicksort. Communications of the ACM 4(7) : 321.

[6] Scowen, R.S. 1965. Algorithm 271 Quickersort, Communications of the ACM 8(11) : 669 – 670.

[7] Singleton, R.C. 1969. Algorithm 347 An efficient algorithm for sorting with minimal storage. Communications of ACM 12(3) : 186 – 187.

[8] Sedgewick, R. 1978. Implementing quicksort programs. Communications of the ACM 21(10) : 847 – 857.

[9] Wainwright, R.L. 1985. A class of sorting algorithms based on quicksort. Communication of the ACM : 28(4), 396 – 402.

[10] Bentley, J.L. and Mcilroy, M.D. 1993. Engineering a sort function. Software – Practice and Experience 23(11) : 1249 – 1265.

[11] Borisuttinant, P. and Sinapiromsaran, K. 2011. Multikey Quicksort For Sorting Numeric Using Predecessor And Successor Pivots. The 3rd International Conference on Computer Design and Applications 2011.

[12] Eunsang Kim, Kunsoo Park, Improve multikey Quicksort for sorting strings with many equal elements, ELSEVIER B.V. (January 2009) 454-459.

[13] Rungpiphop, S. and Sinapiromsaran, K. Adjacent Pivot Quicksort, AMM 2010 185-195

[14] Knuth, D.E. 1973. The art of Computer Programming Vol. 3: Sorting and Searching. CA : Addison-Wesley Press.

[15] Khreisat, L. 2007. Quicksort A Historical Perspective and Empirical Study. International Journal of Computer Science and Network Security 7(12) : 54 – 65.

[16] Scowen, R.S. 1965. Algorithm 271 Quickersort, Communications of the ACM 8(11) : 669 – 670.

[17] Singleton, R.C. 1969. Algorithm 347 An efficient algorithm for sorting with minimal storage. Communications of ACM 12(3) : 186 – 187.

[18] Cook, C.R. and Kim, D.J. 1980. Best sorting algorithm for nearly sorted lists. Communications of the ACM 23(11) : 620 – 624.

[19] Motzkin, D. 1983. Pracniques : Meansort. Communications of the ACM 26(4) : 250 – 251.

[20] Wainwright, R.L. 1985. A class of sorting algorithms based on quicksort. Communication of the ACM : 28(4), 396 – 402.

[21] Wainwright, R.L. 1987. Quicksort algorithms with an early exit sorted subfiles. The 15th annual conference on Computer Science 1987 : 183 – 190.

[22] Sarwar, S.M., Sarwar, S.A., Jaragh, M.H.A. and Brandeburg, J. 1996. Engineering quicksort, Elsevier Science 22(1) : 39 – 47.

[23] Edmonson, J. 2006. M Pivot Sort – Faster than Quick Sort!. McNair Research Review 4(6) : 59 – 69.

[24] Jidol, J., Sinapiromsaran, K. and Sinthupinyo, S. 2007. A Successive Difference Quicksort. The 4th International Joint Conference on Computer Science and Software Engineering 2007 : 321 – 326.

APPENDIX

Appendix
Sorting Program

Swap Function

```
protected void Swap(IComparable[] elements, int indexA, int indexB)
    {
        IComparable tmp = elements[indexA];
        elements[indexA] = elements[indexB];
        elements[indexB] = tmp;
    }
```

Insertion sort

```
 protected void InsertSort(IComparable[] array, int left, int right)
    {
        int i, j;
         for (i = left; i <= right; i++)
        {
            IComparable value = array[i];
            j = i - 1;
            while ((j >= 0) && (array[j].CompareTo(value) > 0))
            {
                array[j + 1] = array[j];
                j--;
            }
            array[j + 1] = value;
        }
    }
```

Adjacent Pivot Quicksort (APQsort)

```
static void APQSort(IComparable[] elements, int left, int right)
        {
           APQSortAction xx = new APQSortAction();

              if(right - left <= 9)  xx.InsertSort(elements,left,right);

             else{
             //start if
                 int mid;
                  Int64 difL, difR;
                  string Left_NS, Mid_NS, Right_NS;
                  pivot new_index = new pivot();

                 // Median of three
                 xx.HashMain(elements, ref left, ref right, out Left_NS, out Mid_NS, out
Right_NS, out mid);
                  Int64 LeftN = System.Convert.ToInt64(Left_NS);
                  Int64 MidN = System.Convert.ToInt64(Mid_NS);
                  Int64 RightN = System.Convert.ToInt64(Right_NS);
//find diferance between left group and right group
                difL =  MidN - LeftN;
                difR = RightN - MidN;
                if(difR < difL){
                     if (difR != 0) new_index = xx.SC_partition(elements, left, mid, right);
                     else new_index = xx.PD_partition(elements, left, mid, right);
                     }
                 else if(difL < difR){
```

```
            if (difL != 0) new_index = xx.PD_partition(elements, left, mid, right);
            else new_index = xx.SC_partition(elements, left, mid, right);
        }
        else{
            if (difL != 0) new_index = xx.PD_partition(elements, left, mid, right);
            else new_index = xx.Split_end_partition(elements, left, mid, right);
        }

        if (new_index.left - left <= 9) xx.InsertSort(elements, left, new_index.left);
        else APQSort(elements, left, new_index.left);

        if (right - new_index.right <= 9) xx.InsertSort(elements, new_index.right, right);
        else APQSort(elements, new_index.right, right);
       }
  }
```

SC_partition

```
pivot SC_partition(IComparable[] elements, int left, int mid, int right)
    {
        IComparable pivot, SC;
        int i, j, k, indL, indR, isc;
         pivot index = new pivot();
          Swap(elements, mid, left);
         //Set value and pointer of pivot and pivot successor
         i = left + 1; indL = left + 1; j = right - 1; indR = right - 1;
         pivot = elements[left]; SC = elements[right]; isc = right;
        while (i <= j)
        {
            while (elements[i].CompareTo(pivot) <= 0 && i<=j)
            {
```

```
        if (elements[i].CompareTo(pivot) == 0)
            Swap(elements, i, indL++);
        i++;
    }

    while (elements[j].CompareTo(pivot) > 0 && i <= j)
    {
        // Update pivot successor
        if (elements[j].CompareTo(SC)<=0)
        {
            if (elements[j].CompareTo(SC) == 0)
                Swap(elements, j, indR--);
            else
            {
                isc = indR; SC = elements[j];
                Swap(elements, j, indR--);
            }
        }
        j--;
    }
    if (i >= j) break;
    Swap(elements, i, j);
}
if (indL <= indR)
{
    // Move pivot to the correct position
    k = left;
    while (k < indL)
    {
        while (elements[j].CompareTo(pivot) == 0 && j >= k) { j--; if (j < 0) break; }
```

```
                if (j <= k) break;
                Swap(elements, k++, j--);
            }

            // Move pivot successor to the correct position
            if (i != isc)
            {
                for (k = isc; k > indR; k--)
                    Swap(elements, k, i++);
            }
            else i = right;
            index.left = j; index.right = i;
        }
        else
        {
            index.left = left; index.right = right;
        }

        return index;
    }
```

PD_Partition

```
pivot PD_partition(IComparable[] elements, int left, int mid, int right)
    {
         IComparable   pivot, PD;
         int i, j, k, indL, indR, ipd;
         pivot index = new pivot();
        Swap(elements, mid, right);
```

```
// Set value and pointer of pivot and pivot predecessor
i = left + 1; indL = left + 1; j = right - 1; indR = right - 1;
pivot = elements[right]; PD = elements[left]; ipd = left;
while (i <= j)
{
   while (elements[i].CompareTo(pivot) < 0)
   {
      if (elements[i].CompareTo(PD) > 0)
      {
         if (elements[i].CompareTo(PD) == 0)
           Swap(elements, i, indL++);
         else
         {
            ipd = indL; PD = elements[i];
           Swap(elements, i, indL++);
         }
 }
      i++;
   }

   while (elements[j].CompareTo(pivot) >= 0 && i <=j)
   {
      if (elements[j].CompareTo(pivot) == 0)
         Swap(elements, j, indR--);
      j--;
   }
     if(i >= j)  break;
    Swap(elements, i, j);
}
j = i - 1;
```

```
    if(indL <= indR){
        // Move pivot predecessor to the correct position
        if(i != indL){
                for(k = ipd; k < indL; k++)
                    Swap(elements, k, j--);
        }
        else  j = left;

        // Move pivot to the correct position
            k = right;
         while(k > indR){

             while (elements[i].CompareTo(pivot) == 0 && i <= k)
            {
                if (i == elements.Length - 1) break;
                i++;
            }

                if(i >= k)  break;
                Swap(elements, k--, i++);
        }
        index.left = j;  index.right = i;
    }
    else{
        // case all elements are equal
        index.left = left;  index.right = right;
    }
    return index;
}
```

VITAE

Miss Purika Borisuttinant was born on November 21st, 1984, in Nakornpathom, Thailand. She obtained her Bachelor’s Degree in Computer Science from the Faculty of Information Communication and Technology , Silpakorn University in 2004.