NEW MERGE BASED SORT ALGORITHM FOR NEARLY SORTED LISTS

(SIRALIYA YAKIN DİZİLER İÇİN YENİ BİR BİRLEŞTİRME TABANLI SIRALAMA ALGORİTMASI)

by

Orhan Can ÖZALP, B.S.

Thesis

Submitted in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

Date of Submission : January 04, 2013

Date of Defense Examination: January 28, 2013

Supervisor : Asst. Prof. Dr. Murat AKIN

Committee Members : Asst. Prof. Dr. Gülfem ALPTEKİN

Asst. Prof. Dr. A. Çağrı TOLGA

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Asst. Prof. Dr. Murat Akın, for his helpful advices and great patience during the process.

I also would like to thank my dear mother and Esra, for supporting me in every step of my life and making everything look easier.

January, 2013
Orhan Can Özalp

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Methods and algorithms, i.e. sorting algorithms, can be customized to increase system performance in large data systems. In order to do this, various parameters like data flow or data characteristics must be known.

Widely used sorting algorithms (e.g., HeapSort, MergeSort) do not consider the characteristics of data and introduce general solutions for almost all cases. Better performance algorithms can be found when some parameters (e.g., Sortedness) taken into account.

In this work, first the sorting algorithms which can provide a basis for the new method are reviewed. Also, because the new method is merge based, former merge algorithms are reviewed. Finally, based on the former methods, new sorting method is presented for nearly sorted lists.

Key words: Merge Algorithm, Adaptive Sorting Algorithm, Nearly Sorted Lists.

RÉSUMÉ

Performance des algorithmes de tri, particulièrement dans les systèmes de grandes quantités de données, est très important comme la recherche est un prérequis pour réaliser des algorithmes de tri plus performantes.

Les algorithmes de tri connues (ex: HeapSort, MergeSort) sont des algorithmes générales qui sont développés sans considérer les propriétés caractéristiques des données. Dans les cas ou les propriétés caractéristiques des données sont connues, on peut dévéloper des algorithmes plus performantes.

Dans cette recherche, on a examiné les algorithmes dans la littérature qui ont la possibilité de former la base à notre algorithme. Puis, on a amélioré la performance d'un des algorithmes et fait des comparaisons du point de vue de la consommation du temps et de l'espace.

Mots-clés: Algorithme de fusion, algorithme adaptive d'arrangement, listes des nombres presque arrangés.

ÖZET

Sıralama algoritmalarının performansı, özellikle büyük boyutta verilerin işlendiği sistemlerde, arama gibi temel algoritmaların daha performanslı gerçekleştirilebilmesinin ön koşulu olduğu için büyük önem taşımaktadır.

Sık kullanılan sıralama algoritmaları (Örn; HeapSort, MergeSort) eldeki verinin karakteristik özellikleri dikkate alınmadan oluşturulmuş genel sıralama algoritmalarıdır. Verinin karakteristik özelliklerinin (Örn: Sıralanmışlık) bilinebildiği durumlarda, bu durumlara özel daha performanslı çözümler üretilebilir.

Bu çalışmada, bu zamana kadar bulunan sıralama algoritmalarından, bu çalışmaya temel oluşturacak yöntemleri içeren algoritmalar incelenmiştir. Ayrıca çalışmanın temelini oluşturduğu için, şimdiye kadar kullanılmış birleştirme (merge) işlemlerine de değinilmiş, sonrasında da incelenen yöntemlerin sıralıya yakın diziler için uyarlanabilir hale gelmesi için iyileştirmeler yapılmaya çalışılmıştır.

Anahtar Sözcükler: Birleştirme Algoritması, Uyarlanabilir Sıralama Algoritması, Sıralıya Yakın Diziler.

1. INTRODUCTION

Sorting is the computational process of rearrangement of items into ascending or descending order [8]. It is one of the fundamental techniques in computer science, because a lot of other techniques and algorithms are based on sorting.

A sorting algorithm is adaptive if it runs faster when the input list is nearly in order (or nearly sorted) [13]. Nearly sorted lists are defined by some disorder measures. Many parameters define the sortedness of a list; e.g., *Rem, Runs* [10]. Those measures are used in further sections to define each sort algorithms characteristics.

Merge algorithms and well-known sorting algorithms are defined in the first section. Since the number of sorting algorithms is high, only the relevant algorithms are taken into account in order to make a better comparison.

The new algorithm is introduced in the second section. The key points in the algorithm are defined step by step. Run time complexity analysis of the algorithm is given in this section.

In the third section, comparison results of various algorithms are shown in terms of *Runs* measure. Also comparisons of two merge methods are given in this section.

In this work, *Runs* measure was taken into account while creating sample arrays as an input for sorting algoritms. For each size and *Runs* measure, several different lists are generated.

2. LITERATURE REVIEW

2.1. Measures of Disorder

Before analyzing performances of sorting algorithms on nearly sorted lists, the term "nearly sorted" must be clearly explained in terms of sortedness parameters (or measures of disorder). The "sortedness" of a list is a concept that specifies how this list is close to its sorted form. In this section, criteria that define "sortedness" will be explained.

2.1.1. Runs

A sorted list is an ascending list, which means it has only one ascending *run*. Based on this definition, *Runs* can be defined as the number ascending runs in a list [12]. Step-down [8] is a similar parameter with *Runs* which is defined as the number of boundaries between runs. Step-down parameter of a list *L* can be expressed as

$$stepDown(L) = \left| \{ i \mid 1 \leq i < n, L_{i+1} < L_i \} \right| \quad (2.1)$$

Runs parameter of a list L can be expressed in terms of *step-down*

$$Runs(L) = stepDown(L) + 1 \quad (2.2)$$

For example;

L : { 1 3 8 6 10 5 9 11 }

stepDown(L) : { 1 3 8 ↓ 6 10 ↓ 5 9 11 } = 2

Runs(L) : {1 3 8} {6 10} {5 9 11} = 3

2.1.2. Rem

Rem parameter can be defined as the minimum number of removal operations from an unsorted list to obtain a sorted list [4]. An unsorted list can be defined as a form of sorted list which has some elements that break sortedness.

Rem parameter can be expressed as

$$Rem(L) = |L| - LNS(L) \tag{2.3}$$

In equation (2.3), L is the length of the list and $LNS(L)$ is the length of the longest non-decreasing subsequence in the list.

2.1.3. Inv

Inv parameter can be defined as the number of pairs in the wrong order in a list [10]. Therefore, *Inv* value is smallest when the list is sorted and largest when the list is reversely sorted.

Inv parameter can be expressed as

$$Inv(L) = |\{ (i, j) \mid 1 \leq i < j \leq n \text{ and } L_i > L_j \}| \tag{2.4}$$

2.2. Merge Algorithms

Merging is an operation means combining two or more sorted lists into one sorted list [1]. For example, merge result R of L_1 and L_2 will be

L_1 : { 1, 6, 7, 12, 89 }
L_2 : { 4, 17, 28, 46 }
R : { 1, 4, 6, 7, 12, 17, 28, 46, 89 }

Regarding to previous researches, several algorithms are used to perform merge operation, e.g.; Two-Way Merge [8], Binary Merge [6], Polyphase Merge [8], Cascade Merge [8]. In this section,

- Two-Way Merge
- Binary Merge
- Prune Merge

algorithms will be reviewed[1].

2.2.1. Two-Way Merge Algorithm

Two-Way Merge algorithm is the most basic method for merging. It is a linear algorithm and it is based on comparing the smallest elements of two lists and adding the smallest element into the result list.

Algorithm 2.1: This algorithm merges two sorted lists, *List1* with size *m* and *List2* with size *n* and stores the result into the list *Result* with size *(m + n)*.

Step 1. Initialize indexes $i \leftarrow 1, j \leftarrow 1, k \leftarrow 1$
Step 2. If $List2_j < List1_i$, go to **Step 5**
Step 3. Set $Result_k \leftarrow List1_i$, $i \leftarrow i + 1$, $k \leftarrow k + 1$
Step 4. If $i > m$, set $Result_{(k,\, m+n)} \leftarrow List2_{(j,\, n)}$ and terminate, else go to **Step 2**
Step 5. Set $Result_k \leftarrow List2_j$, $j \leftarrow j + 1$, $k \leftarrow k + 1$
Step 6. If $j > n$, set $Result_{(k,\, m+n)} \leftarrow List1_{(i,\, m)}$ and terminate, else go to **Step 2**

[1] Cascade Merge and Polyphase Merge algorithms are used in external sorting algorithms thus will not be reviewed.

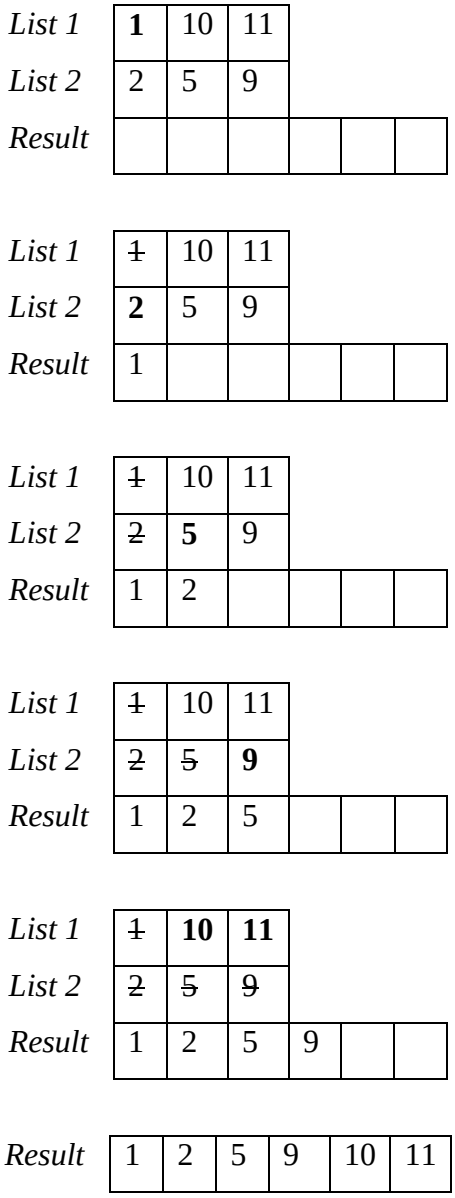

Figure 2.1 : Steps of Two-Way Merge on two sample lists.

2.2.2. Binary Merge Algorithm

Binary Merge (or Hwang-Lin Merge [6]) algorithm is a combination of Two-Way Merge algorithm and Binary Search algorithm [8]. This algorithm uses binary search to find values to compare instead of one-by-one steps like Two-Way Merge does.

Pseudocode for this algorithm is given below:

Algorithm 2.2: This algorithm merges two sorted list, *List1* with size *m* and *List2* with size *n* and stores the result into the list *Result* with size (*m* + *n*).

Step 1 : If *m* or *n* equals 0, finish
Step 2 : Set $t \leftarrow \lfloor \log(n/m) \rfloor$
Step 3 : If $m > n$, go to **Step 8**
Step 4 : Set $j \leftarrow n - 2^t + 1$
Step 5 : If $List1_m < List2_j$, add $List2_{(j, n)}$ to *Result*, set $n \leftarrow j - 1$ and go to **Step 1**
Step 6 : Set $i \leftarrow$ binary search of correct position for $List1_m$ in $\{List2_j, List2_n\}$
Step 7 : If $n \geq i$, $List2_{(i, n)}$ to *Result*
Step 8 : Add $List1_m$ to *Result*, $n \leftarrow i - 1$, $m \leftarrow m - 1$, go to **Step 1**
Step 9 : Set $j \leftarrow m - 2^t + 1$
Step 10 : If $List2_n < List1_j$, add $List1_{(j, m)}$ to *Result*, set $m \leftarrow j - 1$ and go to **Step 1**
Step 11 : Set $i \leftarrow$ binary search of correct position for $List2_n$ in $\{List1_j, List1_m\}$
Step 12 : If $m \geq i$, $List1_{(i, m)}$ to *Result*
Step 13 : Add $List1_{(i, m)}$ to *Result*, $List2_n$ to *Result*, $m \leftarrow i - 1$, $n \leftarrow n - 1$, go to **Step 1**

List1 :	1	**15**							
List2 :	2	6	9	11	**16**	18	25	33	
Result :										

List1 :	1	**15**							
List2 :	2	6	**9**	11	~~16~~	~~18~~	~~25~~	~~33~~		
Result :							16	18	25	33

List1 :	**1**	~~15~~							
List2 :	2	**6**	~~9~~	~~11~~	~~16~~	~~18~~	~~25~~	~~33~~		
Result :				9	11	15	16	18	25	33

Result :	1	2	6	9	11	15	16	18	25	33

Figure 2.2 : Steps of Binary Merge on two sample lists.

2.2.3. Prune Merge Algorithm

This algorithm is a part of TimSort [11]. It can be called "Prune Merge" because it prunes the maximum and minimum elements and reduces list sizes before merging.

"Prune Merge" is a combination of Two-Way Merge and Binary Search; it uses binary search to find extreme intervals, prunes them and finally makes a Two-Way Merge operation on final lists.

Algorithm 2.3: This algorithm merges two sorted list, *List1* with size *m* and *List2* with size *n* and stores the result into the list *Result* with size $(m + n)$.

[Boundary Control]

Step 1: If $s1 > f1$, add $List2_{(s2,\ f2)}$ to $Result_{(s,\ f)}$ then terminate

Step 2: If $s2 > f2$, add $List1_{(s1,\ f1)}$ to $Result_{(s,\ f)}$ then terminate

[/Boundary Control]

[Prune Merge]

Step 1: Set $s \leftarrow 1$, $f \leftarrow (m + n)$, $s1 \leftarrow 1$, $s2 \leftarrow 1$, $f1 \leftarrow m$, $f2 \leftarrow n$

Step 2: If $List1_{s1} > List2_{s2}$, go to **Step 5**, if $List1_{s1} < List2_{s2}$, go to **Step 7**

Step 3: Add $List1_{s1}$ and $List2_{s2}$ to *Result*, $s1 \leftarrow s1 + 1$, $s2 \leftarrow s2 + 1$, $s \leftarrow s + 2$

Step 4: Execute ***[Boundary Control]***, go to **Step 2**

Step 5: Set $k \leftarrow$ binary search of correct position for $List1_{s1}$ in $List2_{(s2,\ n)}$

Step 6: Add $List2_{(s2,\ k-1)}$ to *Result*, $s \leftarrow s + (k - s2)$, $s2 \leftarrow k$, go to **Step 9**

Step 7: Set $k \leftarrow$ binary search of correct position for $List2_{s2}$ in $List1_{(s1,\ m)}$

Step 8: Add $List1_{(s1,\ k-1)}$ to *Result*, $s \leftarrow s + (k - s1)$, $s1 \leftarrow k$

Step 9: Execute ***[Boundary Control]***

Step 10: If $List1_{f1} > List2_{f2}$, go to **Step 13**, if $List1_{f1} < List2_{f2}$, go to **Step 15**

Step 11: Add $List1_{f1}$ and $List2_{f2}$ to *Result*, $f1 \leftarrow f1 - 1$, $f2 \leftarrow f2 - 1$, $f \leftarrow f - 2$

Step 12: Execute ***[Boundary Control]***, go to **Step 10**

Step 13: Set $k \leftarrow$ binary search of correct position for $List1_{f1}$ in $List2_{(f2,\ n)}$

Step 14: Add $List2_{(k,\ n)}$ to *Result*, $f \leftarrow f - (n - k + 1)$, $f2 \leftarrow k - 1$, go to **Step 17**

Step 15: Set $k \leftarrow$ binary search of correct position for $List2_{f2}$ in $List1_{(f1,\ m)}$

Step 16: Add $List1_{(k,\ m)}$ to *Result*, $f \leftarrow f - (m - k + 1)$, $f1 \leftarrow k - 1$

Step 17: Execute ***[Boundary Control]***

Step 18: **Two-Way Merge** $List1_{(s1,\,f1)}$ and $List2_{(s2,\,f2)}$ into $Result_{(s,\,f)}$

[/Prune Merge]

List 1:	**2**	10	17	25	28	35	43	57	90									
List 2:	**2**	5	8	12	32	50	62	75	90									
Result:																		

List 1:	~~2~~	**10**	17	25	28	35	43	57	90									
List 2:	~~2~~	**5**	**8**	12	32	50	62	75	90									
Result:	2	2																

List 1:	~~2~~	10	17	25	28	35	43	57	**90**									
List 2:	~~2~~	~~5~~	~~8~~	12	32	50	62	75	**90**									
Result:	2	2	5	8														

List 1:	~~2~~	10	17	25	28	35	43	**57**	~~90~~									
List 2:	~~2~~	~~5~~	~~8~~	12	32	50	**62**	**75**	~~90~~									
Result:	2	2	5	8												90	90	

List 1:	~~2~~	**10**	**17**	**25**	**28**	**35**	**43**	**57**	~~90~~									
List 2:	~~2~~	~~5~~	~~8~~	**12**	**32**	**50**	~~62~~	~~75~~	~~90~~									
Result:	2	2	5	8											62	75	90	90

Result:	2	2	5	8	10	12	17	25	28	32	35	43	50	57	62	75	90	90

Figure 2.3 : Steps of Prune Merge on two sample list

2.3. Sorting Algorithms

Sorting algorithm is a process of rearranging a list of items into ascending or descending order [8]. Several sorting algorithms developed so far and they can be grouped in 2 main types of being comparison based sort algorithm or not. Although there are some integer sorting algorithms e. g., Counting Sort [3], Radix Sort [3], sorting algorithms are mostly comparison based and can be implemented for generic uses.

Comparison based sorting algorithms can be grouped in 5 types;

- Insertion based sort
- Exchange based sort
- Selection based sort
- Merge based sort
- Hybrid sort

Insertion based sort algorithms are basically keeps sorted elements in a place and inserts every element into the correct position sequentially. Straight Insertion Sort and ShellSort are examples for this type.

Exchange based sort algorithms are based on exchanging unsorted pairs of elements. Exchange (or swapping) operations continue until list is sorted. QuickSort and BubbleSort are examples for exchange based sort algorithms.

Selection based sort algorithms are based on removing the minimum element from the list and adding it to the result list until there is no element left in the list. HeapSort and SelectionSort are selection based selection sort algorithms.

Merge based sort algorithms are based on merging separate sorted sublists into one sorted lists. These algorithms use different techniques to create sorted sublists. MergeSort, Polyphase Merge Sort and StrandSort [15] are examples to this type.

Hybrid sort algorithms are combinations of other types of sorting algorithms. Cook-Kim Sort [2] and TimSort [11] are examples of hybrid sort algorithms.

2.3.1. Straight Insertion Sort

Straight Insertion Sort is an insertion based sort algorithm. Straight Insertion Sort has the same working principle with to sort a deck of cards in a card game [14]. In a card game, player starts with an empty and all the cards are face down. Player takes all the cards one by one and for each card, he founds the correct place for the card and inserts it to this place.

Straight insertion sort has the worst case time complexity of $O(n^2)$ and a best case complexity of $O(n)$. Its average complexity is $O(n + i)$, where i is the number of inversions, therefore Straight Insertion Sort can be defined as *Inv*-optimal adaptive sort algorithm.

```
procedure insertionsort
begin
        for i := 2 to n
        begin
                j := i
                value := List_i
                while j > 1 and List_j-1 > value
                begin
                        List_j := List_j-1
                        j := j - 1
                end

                List_j := value
        end
end
```

Figure 2.4 : Pseudocode of Straight Insertion Sort

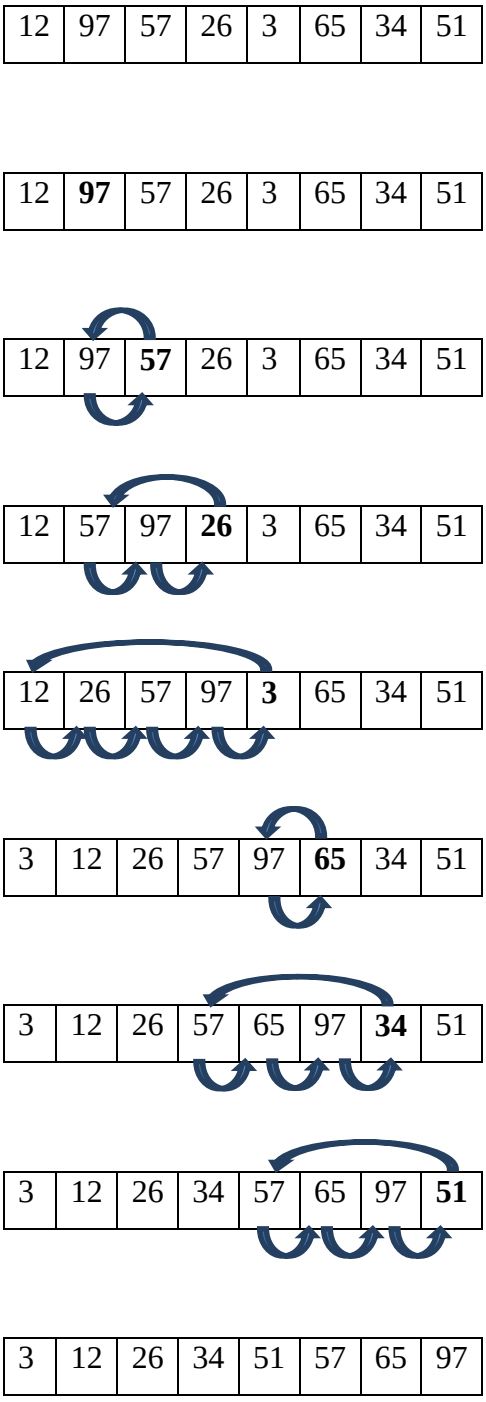

Figure 2.5 : Steps of Straight Insertion Sort on a sample list

2.3.2. Merge Sort

Merge Sort is a merge based sort algorithm and uses Divide & Conquer principle [1]. It is a recursive process of partitioning elements into two groups and sorting both partitions recursively until the size of the recursive call is 1 (Recursive call with 1 element will return the value itself). After both of the partitions are sorted, they are merged and recursive call terminates (Formula 2.5).

$$MergeSort(L_{(1,\,n)}) = Merge(MergeSort(L_{(1,\,\lfloor n/2 \rfloor)}), MergeSort(L_{(\lfloor n/2 \rfloor + 1,\,n)})) \quad (2.5)$$

Merge Sort has a constant time complexity *O(nlog(n))* for worst, best and average case, therefore it is not adaptive. Standard merge sort has an extra space complexity of *O(n)*, because it allocates a list size of *n* to store the result. An in-place merge sort [16] exists but since the classic approach has better performance [16], in-place merge sort will not be analyzed.

```
procedure mergesort (start, end)
begin
        if end = start
        begin
                result := create list (1)
                result1 := Liststart
                return result
        end
        begin
                mid := (start + end) / 2

                left := mergesort (start, mid)
                right := mergesort (mid + 1, end)

                return merge(left, right)
        end
end
```

Figure 2.6 : Pseudocode of Merge Sort

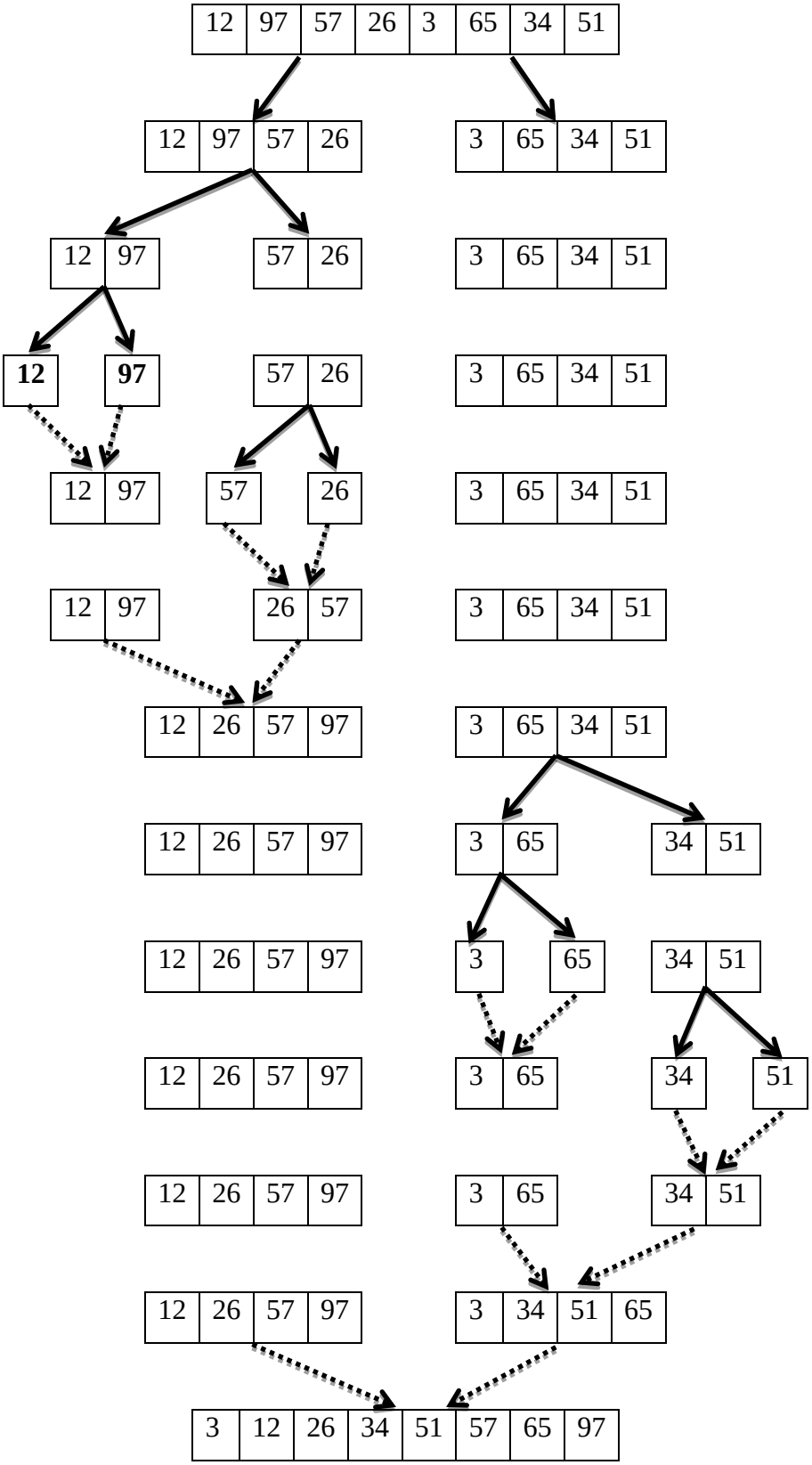

Figure 2.7 : Steps of Merge Sort on a sample list.

2.3.3. Quicksort

Quicksort is an exchange based sorting algorithm. It is based on a Divide & Conquer principle [14]; it works by partitioning the list into 2 and recursively sorting both sublists (Figure 2.8).

```
function quicksort (start, end)
begin
        if end > start
        begin
                index := partition(start, end)
                quicksort(start, index - 1)
                quicksort(index + 1, end)
        end
end
```

Figure 2.8 : Basic pseudocode of Quicksort

"Partition" function is the key point of Quicksort. This function selects a list value as a pivot value and creates two separate lists which contain smaller and greater values from the pivot value. This work can be done with an in-place method (Figure 2.9).

```
function partition (start, end)
begin
        pivot := determine pivot index()
        value := List_pivot
        i := start
        j := end
        while j > i
        begin
                while List_i <= value
                        i := i + 1
                while List_j >= value
                        j := j - 1

                swap(i, j)
        end
        return i
end
```

Figure 2.9 : In-place partition pseudocode method of Quicksort

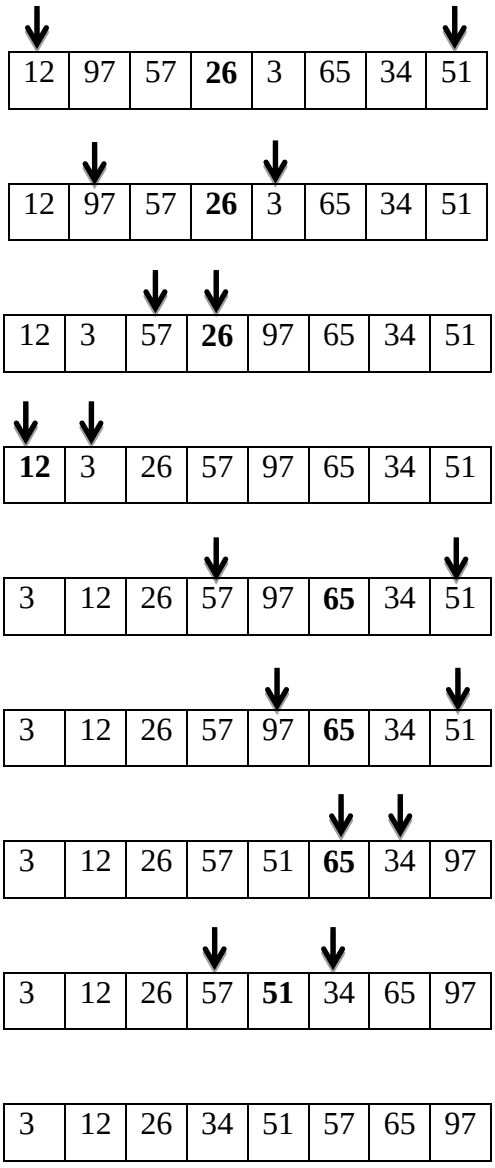

Figure 2.10 : Steps of Quicksort on a sample list

Quicksort has an average case and best case time complexity of *O(nlogn)* and a worst case complexity $O(n^2)$. Worst case for Quicksort occurs in a few permutations; e.g., when the list is sorted ascending or descending order.

2.3.4. Cook-Kim Sort

Cook-Kim (CK) Sort is an adaptive hybrid sort algorithm [2]. It is a combination of Quickersort (Quicksort algorithm with an optimization on selecting the pivot index), Straight Insertion Sort and merging.

CK Sort algorithm basically extracts the unordered pairs in the list, stores unordered pairs in another list and sorts the second list. If there are more than 30 values in the second list, it uses Quickersort otherwise Straight Insertion Sort to sort the values [17]. In the final step, two ordered lists are merged.

```
procedure cksort
begin
        List2 := create empty list()
        start := 1
        i := start + 1

        while i <= n and start > 0
        begin
                while List_i < List_start and start > 0 and i <= n
                begin
                        add(List2, List_i, List_start)
                        List_i := +∞
                        List_start := +∞
                        i := i + 1
                        start := start - 1
                end
        end

        n2 := size of List2

        if n2 <= 30
                insertionsort (List2)
        else
                quickersort (List2)

        return merge List and List2 ignoring sentinel values
end
```

Figure 2.11 : Pseudocode of CK Sort

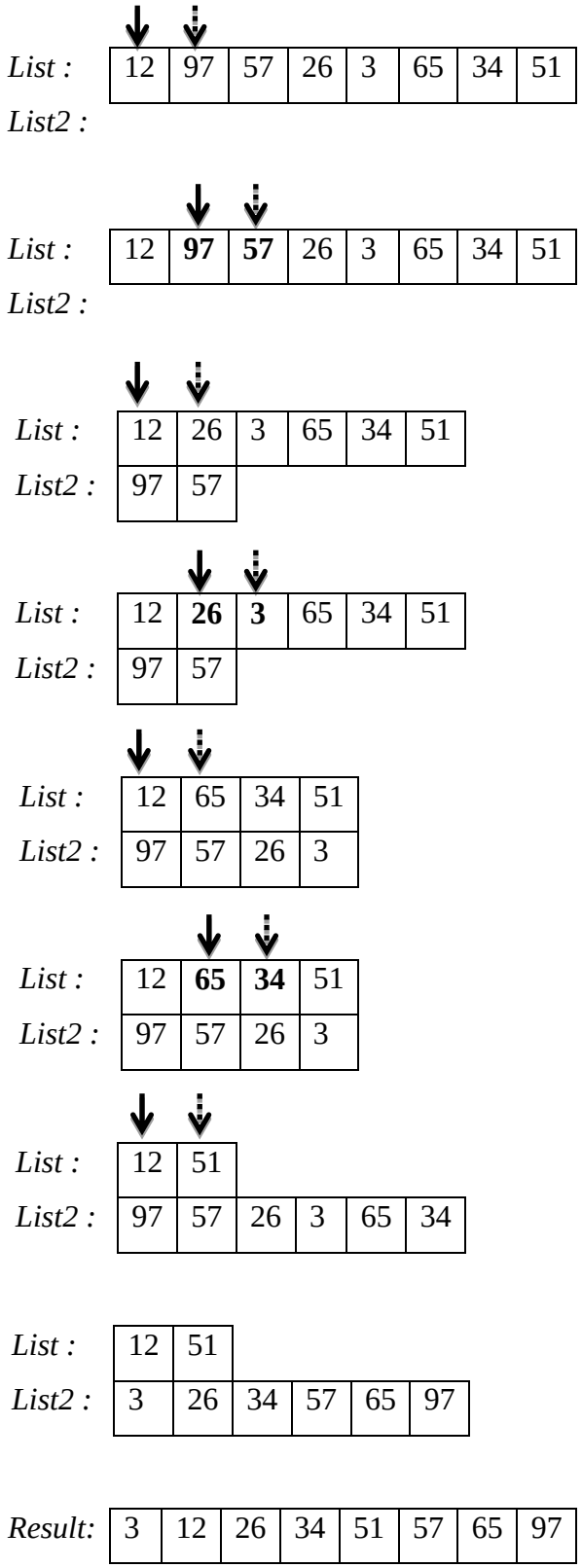

Figure 2.12 : Steps of CK Sort on a sample list

Because only the unordered pairs are being removed from list and the indexes of the unordered pairs are not taken into account, CK Sort is *Rem*-optimal. Because of the direct correlation between *LNS* (Longest Nondecreasing Subsequence) function and *Rem* parameter (Formula 2.3), performance of CK sort will decrease when the size of the remaining list decreases.

CK Sort has a best case complexity *O(n)* and a worst case complexity *O(nlogn)* for lists with size>30 and $O(n^2)$ otherwise. Average case complexity of CK Sort is *O(n + rlogr)* where *r* equals to *Rem* value.

2.3.5. Splitsort

Splitsort is a merge based adaptive sorting algorithm [9]. It basically rearranges original list, recursively sorts unsorted sublists and merges all the sublists. It needs an extra *O(n)* memory space for this rearrangement (split) operation.

```
procedure splitsort (start, end)
begin
        buffer := create empty list (end – start + 1)
        splitindex := split(start, end)

        if splitindex < end
        begin
                mid := (end + splitindex) / 2

                splitsort(splitindex + 1, mid)
                splitsort(mid + 1, end)

                merge (buffer, start, splitindex, splitindex + 1, mid)
                merge (buffer, start, mid, mid + 1, end)

                copy (buffer, List, start, end)
        end
end
```

Figure 2.13 : Pseudocode of Splitsort

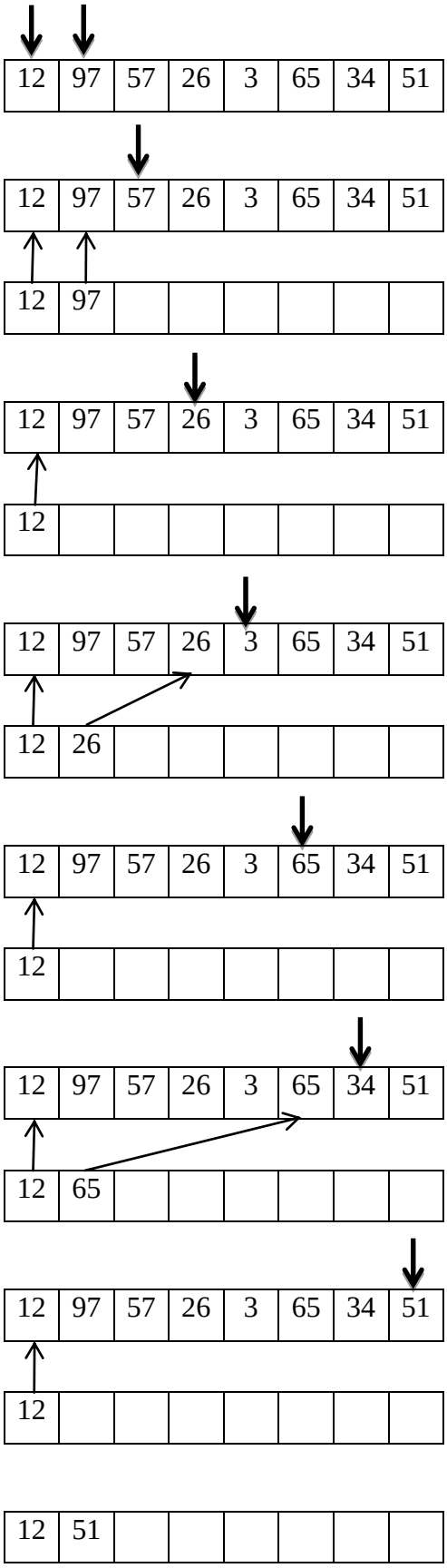

Figure 2.14 : Splisort - Extracting an ascending sequence from list into buffer

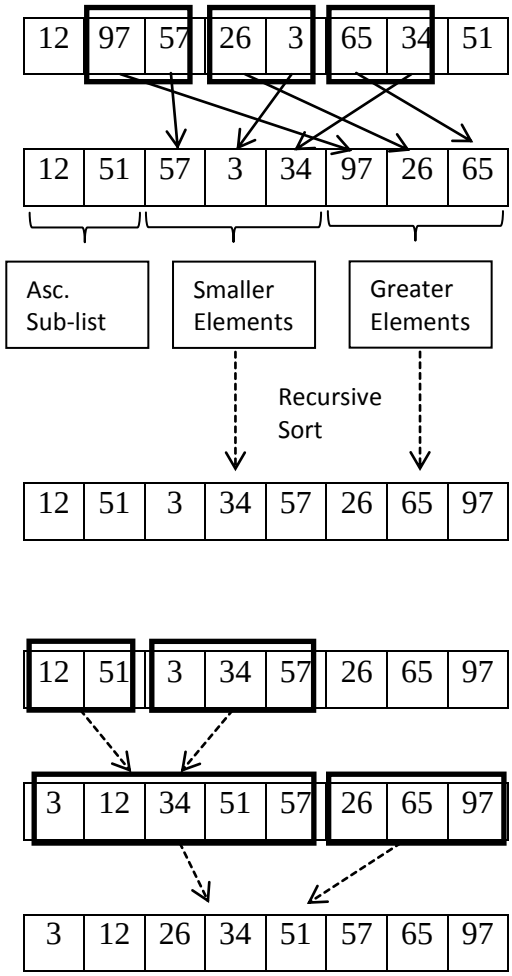

Figure 2.15 : Splitsort – Merging ordered sub-lists

Splitsort has a best case time complexity of *O(n)*, when the list is already sorted. In this case, there will be only (*n – 1*) comparison and the algorithm will terminate. Worst case will occur when the list is reversely sorted. In this case, there will be *nlogn* recursive calls and in each call and there will be *O(n)* comparisons in total in merge and split methods, so the worst case is *O(nlogn)*.

Splitsort removes all the unordered pairs from the list and recursively sorts these pairs, therefore it is a *Rem*-optimal adaptive sorting algorithm. In the average case, number of elements in both smaller and greater elements list is *Rem(L)*. *Rem(L)log(Rem(L))* comparisons need to be done in order to sort these lists and O(n) comparisons for merge operations, therefore the average case complexity is *Rem(L)log(Rem(L)) + O(n)*.

3. A NEW ADAPTIVE SORTING ALGORITHM

Adaptive sorting algorithms take advantage of already sorted subsequences in the list. It is possible to calculate the longest non-decreasing (ascending) subsequence in $O(n^2)$ [5] but since the operation itself contains a sorting procedure and its complexity is higher than $O(nlogn)$ boundary, it is not feasible to extract the longest subsequence. Therefore all adaptive algorithms use different methods to extract an ascending subsequence.

Consider the list L = { 1, 2, 3, 6, 7, 5, 4, 8, 9 }. It is a nearly sorted list with $Rem(L)$ = 2. Longest ascending subsequence ({ 1, 2, 3, 6, 7, 8, 9 }) in this list can be calculated using the algorithm defined in (Fredman, 1975). An extraction method is good as its result is similar to the longest ascending subsequence of the input list.

As described in Section 2.3.4, Cook-Kim Sort uses a linear method to extract an ascending list. It removes unordered pairs from the original lists and stores in another list to sort.

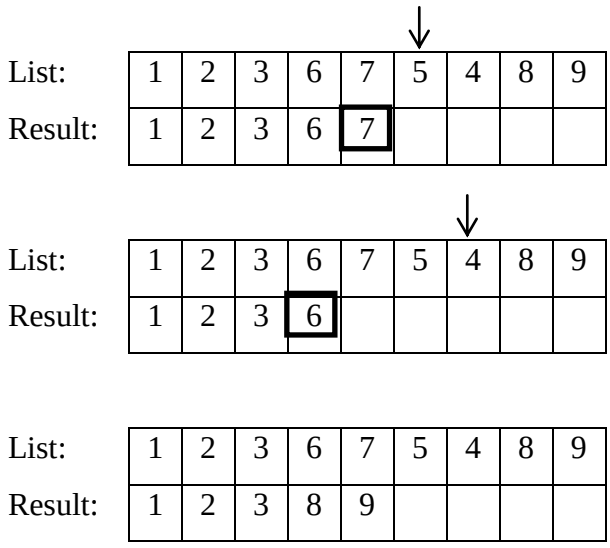

Figure 3.1 : Extracting sorted subsequence with Cook-Kim method

Splitsort extracts an ascending subsequence using the same method with Cook-Kim Sort. The difference between these algorithms is the sorting method of the remaining parts; Splitsort recursively sorts the remaining parts and Cook-Kim Sort uses a different algorithm to sort depending on the size of the list.

3.1. The New Method for Sorting Nearly Sorted Lists

The new method uses a linear algorithm for extracting ascending sequences. Cook-Kim Sort and Splitsort use a different method; they move the unsorted value into another list and after the scanning of list finished, second list is sorted. The main aspect of new method is to continue extending sorted subsequence even there is an unsorted element found.

New method uses an *O(n)* buffer for storing current ascending list. With each unordered element, the buffer is merged with the current ascending subseqence and stored into buffer. After each merge operation, maximum element of the sequence reset to -∞, therefore new method is also *Runs*-optimal.

Figure 3.4 shows a psedocode of the new method. First it creates a new *buffer* with the size *n*. After that it starts to iterate over the input list and hold the current sequence in a *frame* (Figure 3.2). A *frame* is simply a *start* and a *finish* point in the list.

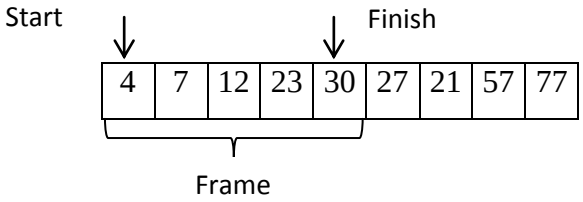

Figure 3.2 : Frame logic in new method

After an unsorted element is found, *buffer* is extended with the next value of the current element, until an unsorted element is found again.

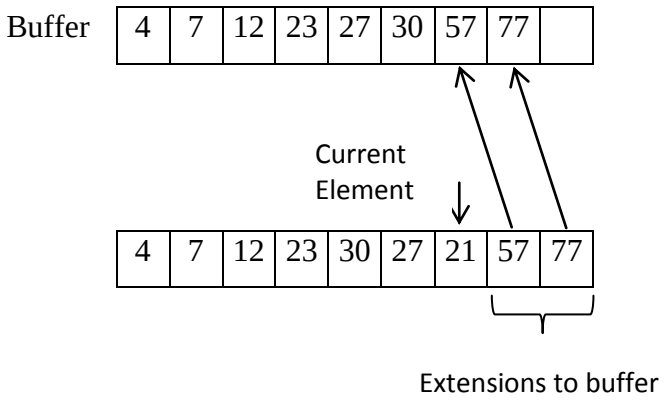

Figure 3.3 : Extending *buffer* elements in new method

Final step is the merge step; after extension of the *buffer* has finished, *frame* and *buffer* will be merged and the result will be stored in *buffer*.

```
procedure newmethod
begin
        buffer := create new list()
        i := 1
        arraystart := 1
        arraymax := -∞
        buffercount := 0
        buffermax := -∞

        while i < n
        begin
                if List_i >= arraymax
                begin
                        arraymax := List_i
                        i := i + 1
                end
                else
                begin
                        arrayfinish := i - 1
                        while i < n and List_i >= buffermax
                        begin
                                buffer_buffercount := List_i
                                buffermax := List_i

                                buffercount := buffercount + 1
                                i := i + 1
                        end

                        merge (List_(arraystart, arrayfinish)), buffer) into buffer

                        arraystart := i
                        buffercount := i
                        buffermax := buffer_buffercount
                        arraymax := -∞
                end
        end
        if buffercount < n
                merge (List_(arraystart, n)), buffer) into buffer

        return buffer
end
```

Figure 3.4 : Pseudocode of new method

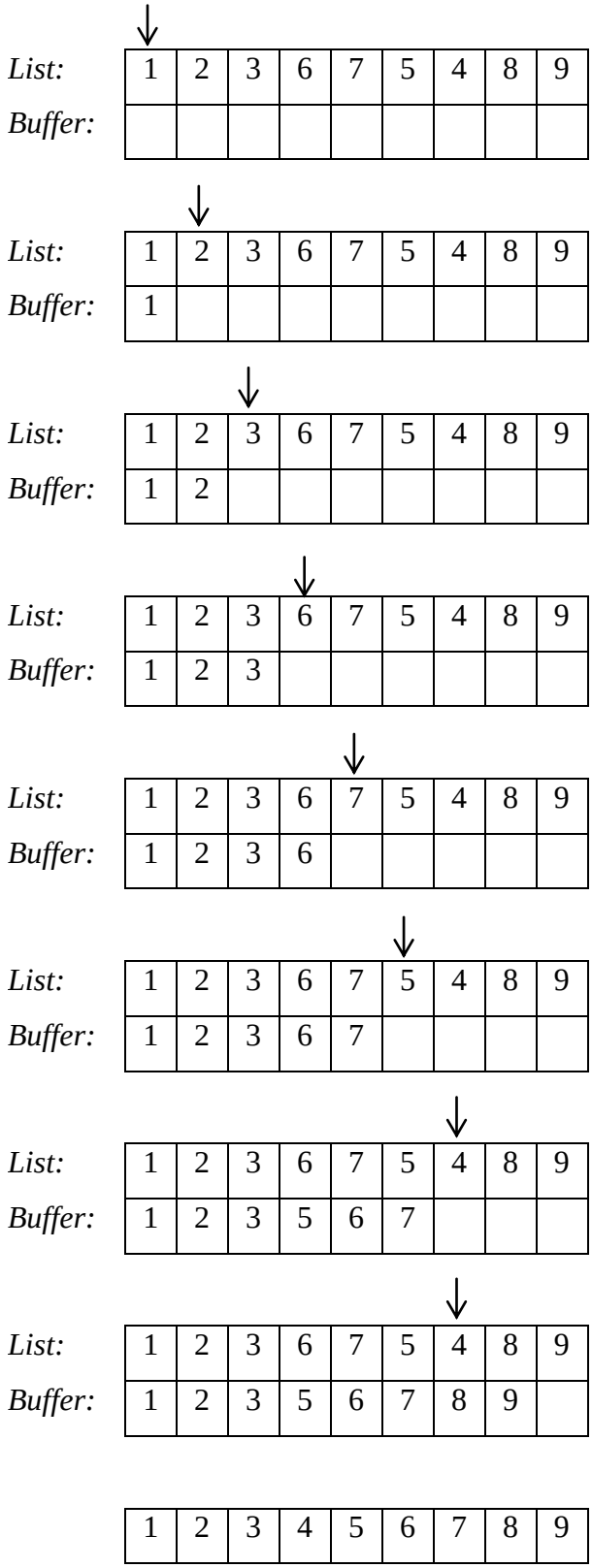

Figure 3.5 : Steps of new method on a sample list with low *Inv* value.

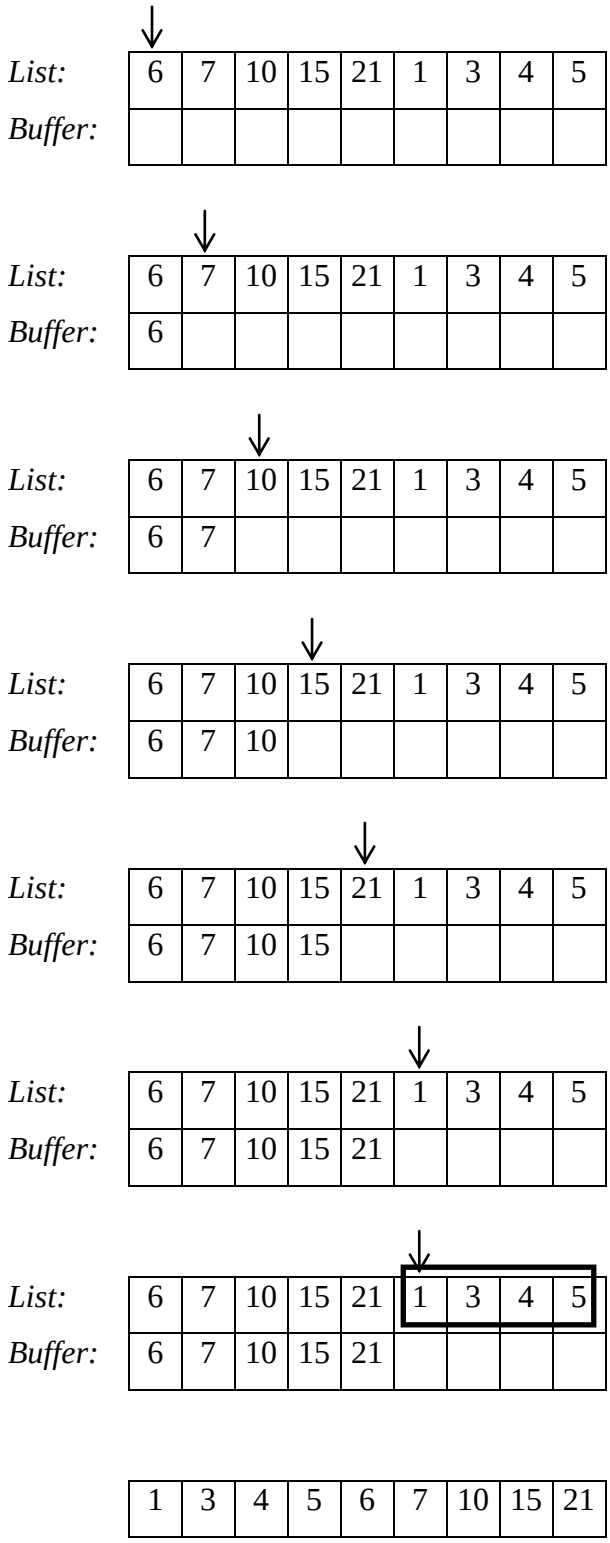

Figure 3.6 : Steps of new method on a sample list with low *Runs* value.

3.2. Analysis of the New Method

The new method finds ascending sequences in the list and merges the sequences, therefore it will be analyzed in terms of *Runs*.

If the list is already sorted, algorithm will terminate after *O(n)* comparisons and 0 exchange operations, so the best case time complexity is *O(n)*. If the list is reversely sorted, there will be *O(n)* comparisons for sequential step, *O(n)* comparisons before merging and $O(n^2)$ memory exchanges in merge operations, so the worst case complexity is $O(n^2)$.

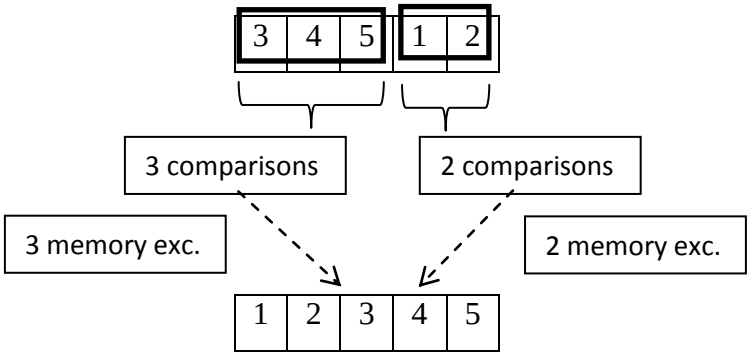

Figure 3.7 : Counts of operations in the new method on a n=5, *Runs(L)*=2 list

For a *Runs(L)* = 2 list (Figure 3.7), there will be *O(n)* comparisons total for sequential step, *O(n) + (Runs(L) – 1)* comparisons for merge operations and *O(n)* operations for memory exchanges. This formula can be generalized to

$$O(n) + Runs(L) \text{ comparisons} + O((Runs(L)- 1)n) \text{ exchanges} \quad (3.1)$$

In Formula 3.1 the limiting factor is number of exchanges occur in merge operations. With the use of Prune Merge instead of Two-Way Merge, number of comparisons will reduce (will be equal in worst case).

4. RESULTS

4.1. Parameters

Although several sorting algorithms have been developed in many years, only adaptive algorithms which are similar to the new method were tested. Straight insertion sort is included because of its good performance on small lists and Mergesort is included because the new method uses a merge technique. Apart from these Quicksort, Splitsort and Cook-Kim Sort included.

All algorithms are tested with different *Runs* and *size* parameters. Since we are focusing on nearly sorted lists, *Runs* parameter has a value between 1 and 20 and array sizes change from 500 to 100,000. Tests input files are generated with a custom C program.

All algorithms are implemented in C++ and compiled with GNU C. All programs are compiled with –O3 parameter. Memory usage values are obtained by "*/proc/self/stat"*, a standard Linux stream.

All tests were executed on a 4GB RAM and 2.20 GHz Dual Core CPU and Ubuntu 12.04 operation system.

4.2. Test Results

4.2.1. Comparison of Algorithms

Test results generated from the parameters described in Section 4.1 can be found below. For each case (*Runs* parameter) there is a mean runtime table for all algorithms and array sizes. Also there is a graph for all algorithms and three adaptive (Cook-Kim Sort, Splitsort and New Method) algorithms separately.

In Table 4.1, there are results for *Runs(L)* = 1, which means input is an already sorted array. This is the best case for all adaptive algorithms but it causes Quicksort run in worst case time. Straight insertion sort especially runs faster on smaller arrays and also terminates fastest between all algorithms.

Table 4.1: Execution times (ms) for *Runs(L)* = 1.

	500	5000	10000	100000
Insertion	0,00001160	0,00006260	0,00012540	0,00126280
QuickSort	0,00008660	0,00063220	0,00135860	0,01704140
CK-Sort	0,00002780	0,00015160	0,00028480	0,00342880
MergeSort	0,00018720	0,00134800	0,00275120	0,03159900
SplitSort	0,00001600	0,00015180	0,00032240	0,00365200
New Method	0,00000560	0,00005260	0,00010580	0,00136080

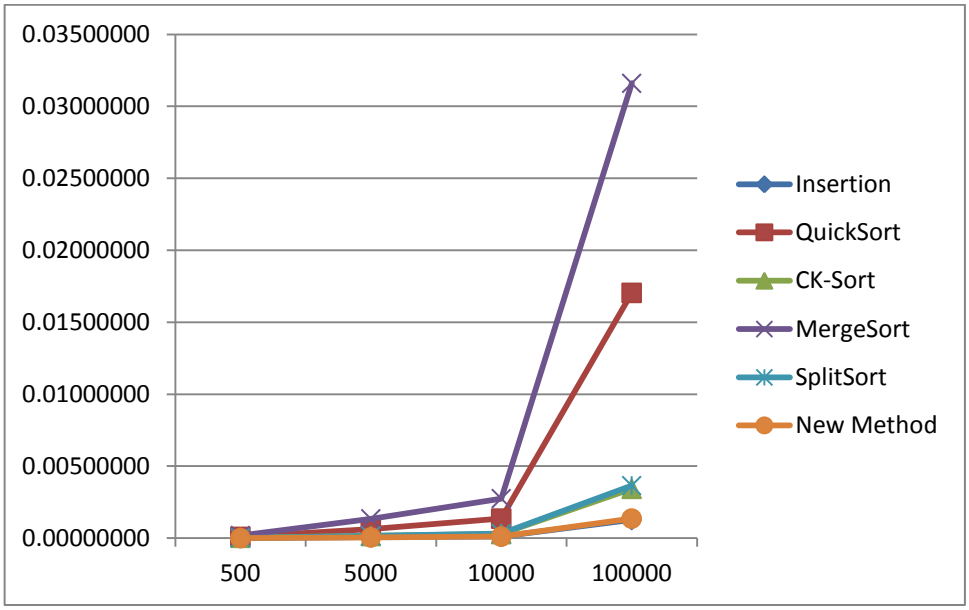

Figure 4.1 : Line graph for *Runs(L)* = 1.

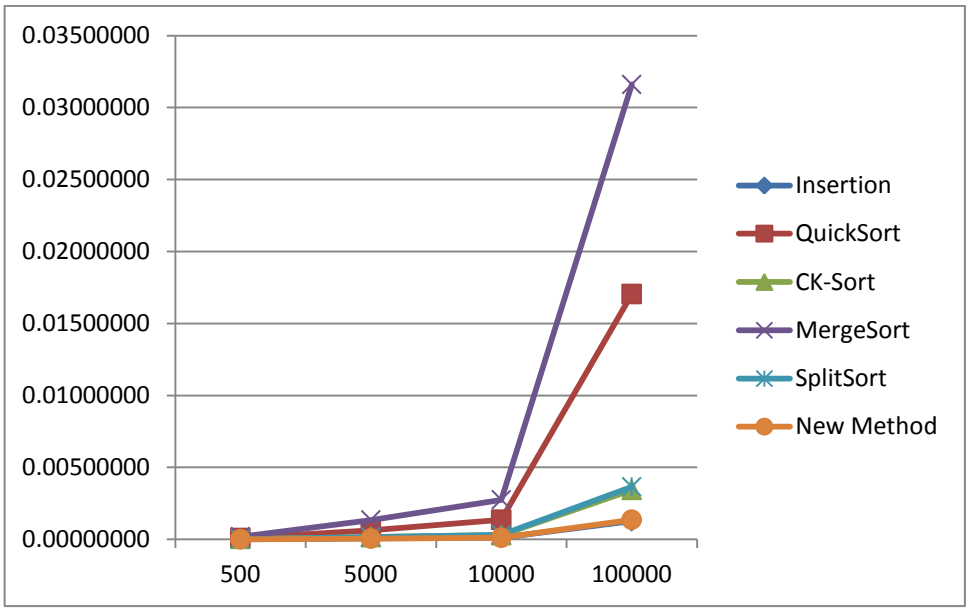

Figure 4.2 : Line graph of adaptive algorithms for *Runs(L)* = 1.

In Table 4.2, results for *Runs(L)* = 2 can be seen. Although there is a slight difference between this case and the previous case, differences of execution times can be seen clearly. For longer lists, straight insertion sort will make much more memory exchange operations, therefore its runtime will increase.

Table 4.2 : Execution times (ms) for *Runs(L)* = 2.

	500	5000	10000	100000
Insertion	0,00000840	0,00012120	0,00016480	0,00155100
QuickSort	0,00004720	0,00063560	0,00135480	0,01704040
CK-Sort	0,00001560	0,00015720	0,00030740	0,00359140
MergeSort	0,00011520	0,00134520	0,00278320	0,03150920
SplitSort	0,00001540	0,00015080	0,00030000	0,00449560
New Method	0,00000960	0,00009500	0,00017240	0,00211520

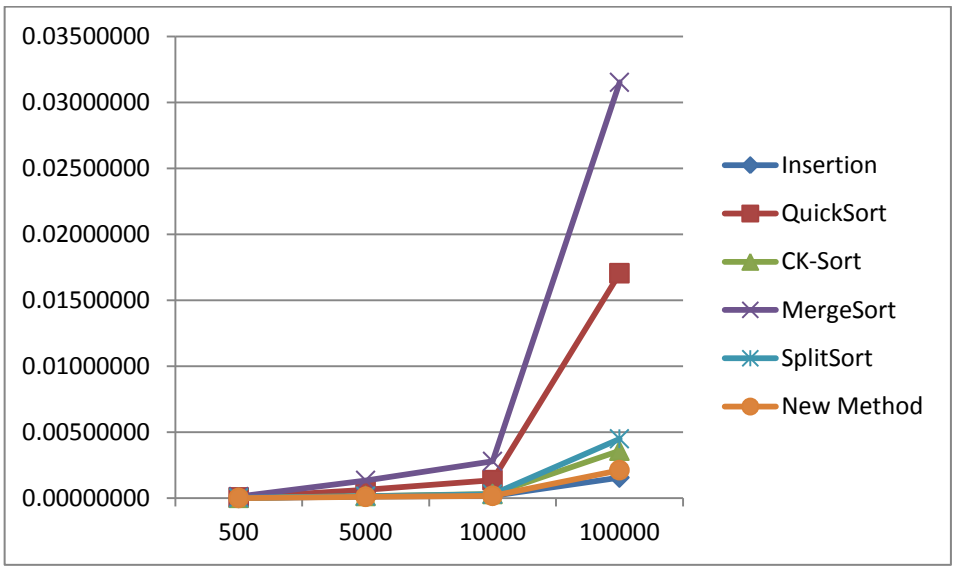

Figure 4.3 : Line graph for *Runs(L)* = 2.

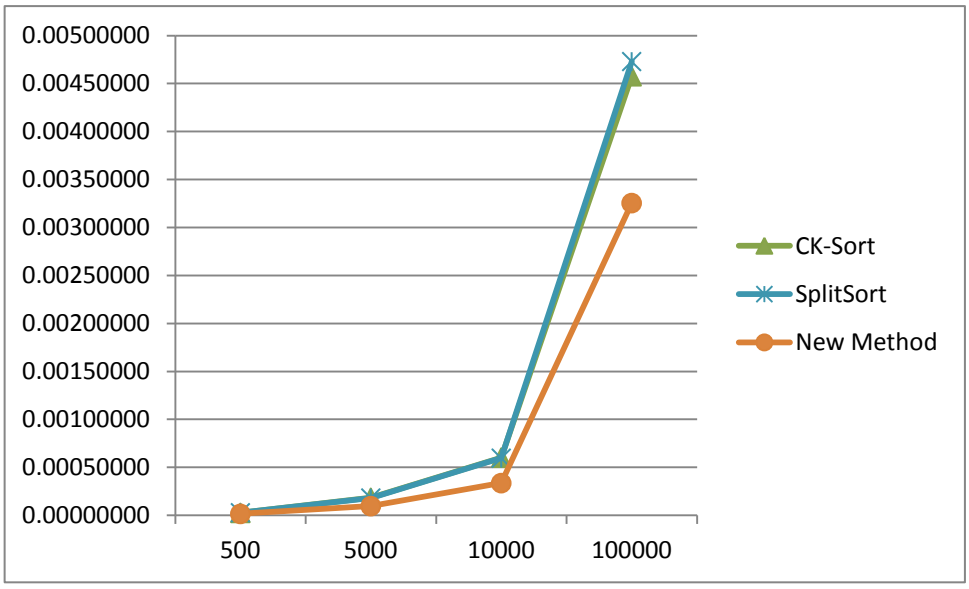

Figure 4.4 : Line graph of adaptive algorithms for *Runs(L)* = 2.

In Table 4.3, results for *Runs(L)* = 5 can be seen. Using these it can be said that the new method performs better than Cook-Kim Sort and Splitsort for larger data sizes.

Table 4.3 : Execution times (ms) for *Runs(L)* = 5.

	500	5000	10000	100000
Insertion	0,00001320	0,00012580	0,00025320	0,00253000
QuickSort	0,00004860	0,00064160	0,00137700	0,01718780
CK-Sort	0,00001680	0,00014860	0,00030160	0,00361960
MergeSort	0,00011780	0,00134200	0,00278720	0,03183960
SplitSort	0,00001840	0,00015660	0,00031840	0,00385940
New Method	0,00001240	0,00010760	0,00020900	0,00280160

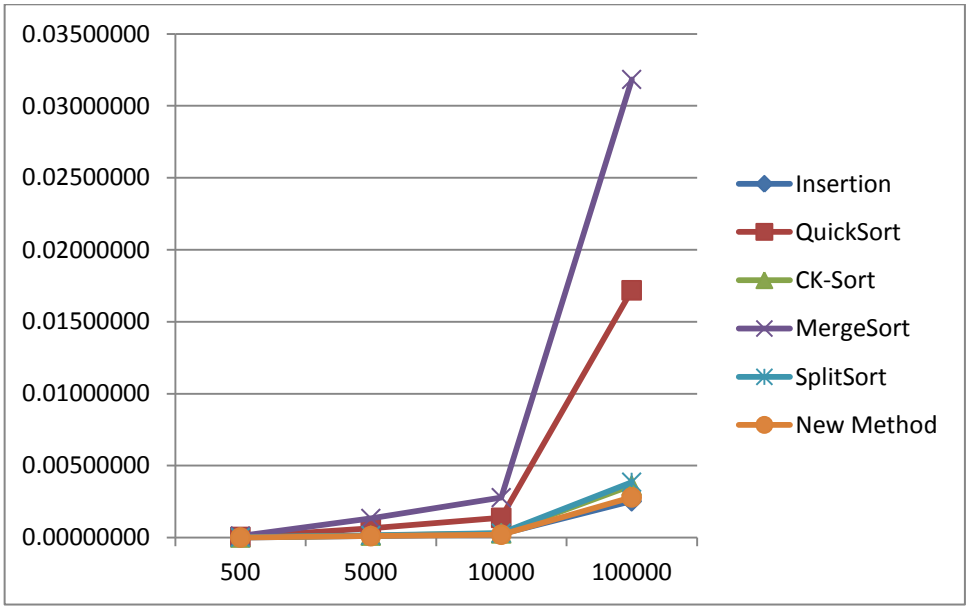

Figure 4.5 : Line graph for *Runs(L)* = 5.

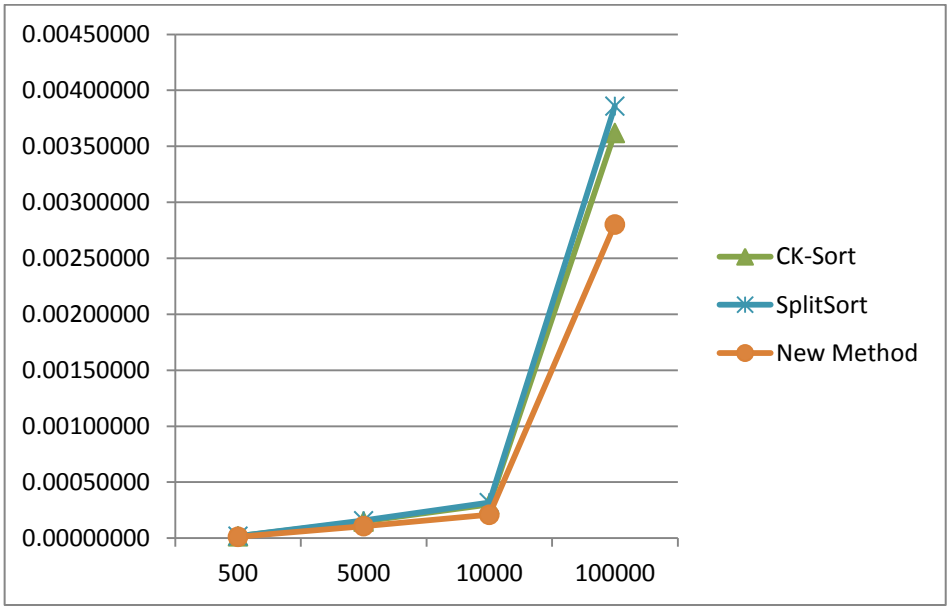

Figure 4.6 : Line graph of adaptive algorithms for *Runs(L)* = 5.

In Table 4.4, results for *Runs(L)* = 10 can be seen. Comparing to previous cases, run times of adaptive sorting algorithms are changed ~0.001 milliseconds. Mergesort nearly has the same runtime for all cases, because of its stable runtime.

Table 4.4 : Execution times (ms) for *Runs(L)* = 10.

	500	5000	10000	100000
Insertion	0,00002100	0,00020480	0,00041140	0,00401900
QuickSort	0,00005020	0,00063540	0,00137220	0,01712740
CK-Sort	0,00001660	0,00014940	0,00029960	0,00361560
MergeSort	0,00011740	0,00134680	0,00280220	0,03175980
SplitSort	0,00001920	0,00015760	0,00034100	0,00386780
New Method	0,00001500	0,00011400	0,00023300	0,00323340

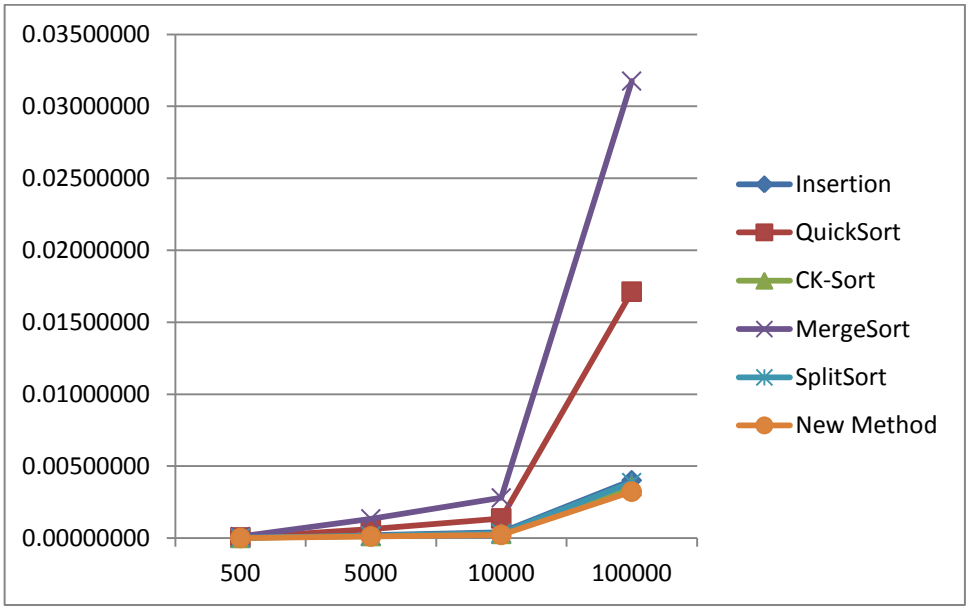

Figure 4.7 : Line graph for *Runs(L)* = 10.

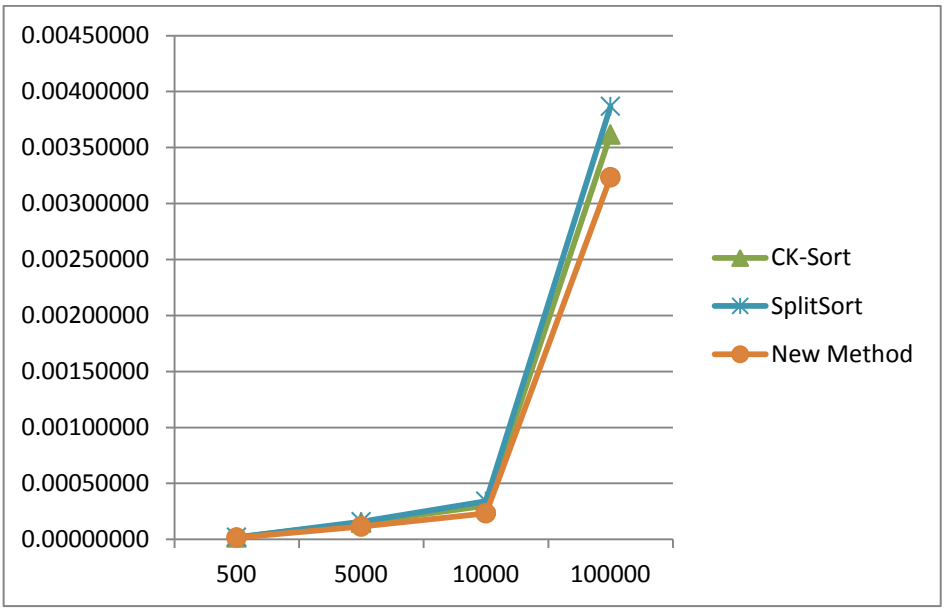

Figure 4.8 : Line graph of adaptive algorithms for *Runs(L)* = 10.

Table 4.5 contains runtimes for *Runs(L)*=20. Insertion Sort has a small run time value with small array sizes. With the efficient use of Prune Merge, New Method handles large array sizes better than other algorithms.

Table 4.5 : Execution times (ms) for *Runs(L)* = 20.

	500	5000	10000	100000
Insertion	0,00003520	0,00035100	0,00071020	0,00711460
QuickSort	0,00005060	0,00064380	0,00137720	0,01719840
CK-Sort	0,00001940	0,00015200	0,00030240	0,00360180
MergeSort	0,00011880	0,00135680	0,00280120	0,03193300
SplitSort	0,00002040	0,00015880	0,00034120	0,00379260
New Method	0,00002440	0,00012500	0,00027080	0,00346940

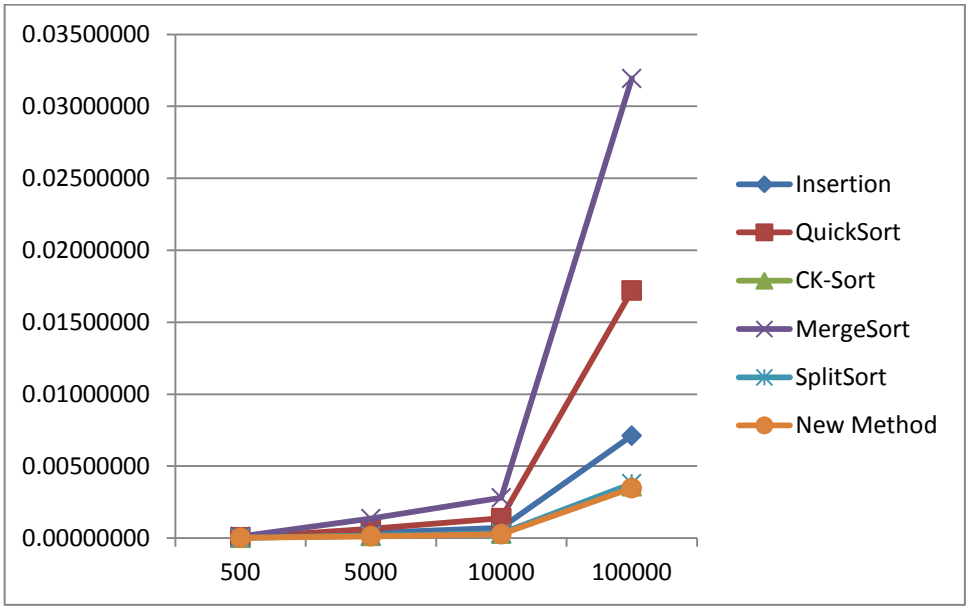

Figure 4.9 : Line graph for *Runs(L)* = 20.

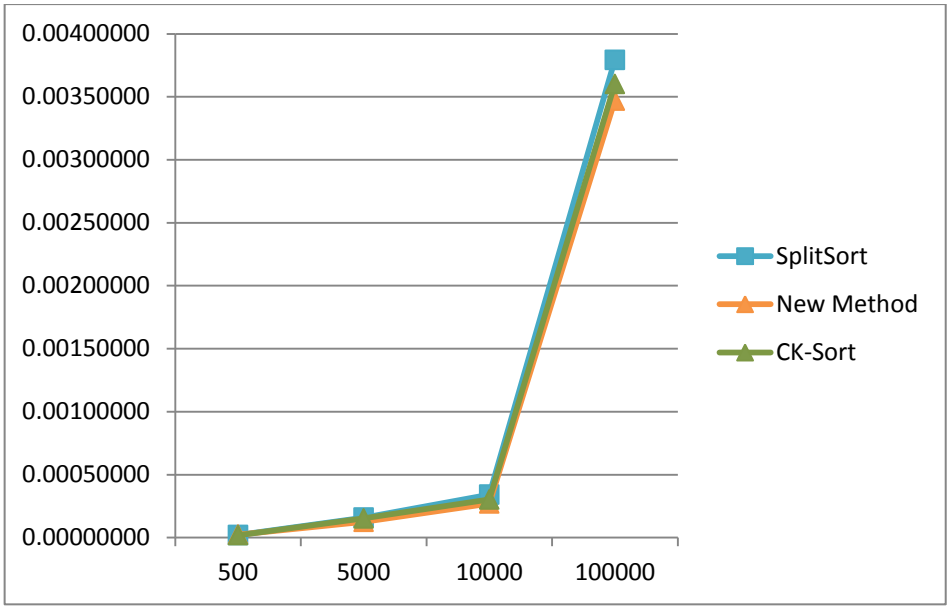

Figure 4.10 : Line graph of adaptive algorithms for *Runs(L)* = 20.

Table 4.6 contains execution times for *Runs(L) = 200*. This is an extreme case for the new method because the list is much more unsorted when compared to Runs(L) = 20. The performance of the new method is slower than Insertion Sort in this case. This case shows that it is better to use the new method for low-*Runs* lists.

Table 4.6 : Execution times (ms) for *Runs(L)* = 200.

	500	5000	10000	100000
Insertion	0,00001000	0,00010260	0,00018940	0,00197420
QuickSort	0,00004860	0,00063080	0,00136380	0,01703800
CK-Sort	0,00001560	0,00015080	0,00030080	0,00355100
MergeSort	0,00012440	0,00133560	0,00276140	0,03158700
SplitSort	0,00001780	0,00015660	0,00031860	0,00381200
New Method	0,00001080	0,00010660	0,00020420	0,00259960

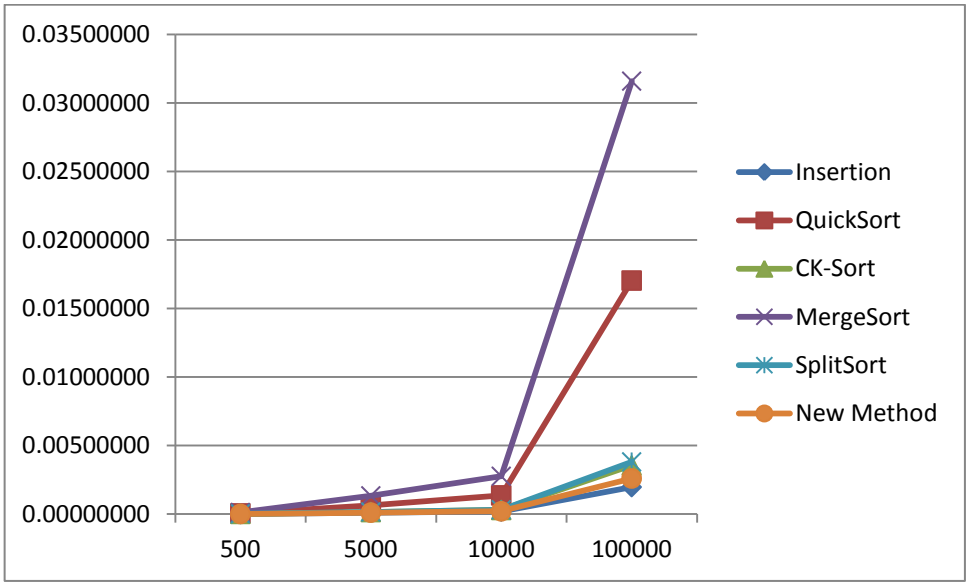

Figure 4.11 : Line graph for *Runs(L)* = 200.

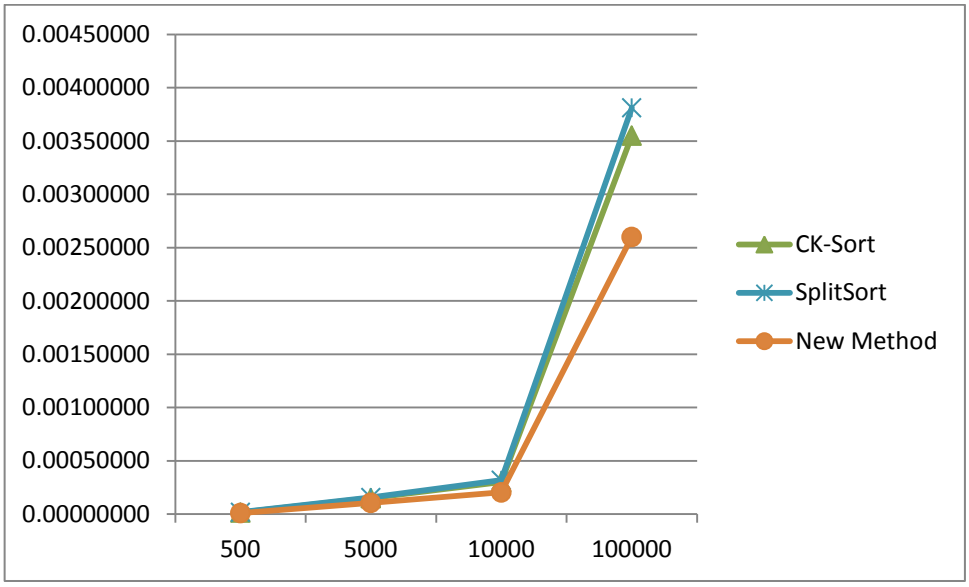

Figure 4.12 : Line graph of adaptive algorithms for *Runs(L)* = 200.

4.2.2. Comparison of Merge Methods

In Section 4.2.1, results are generated using Prune Merge algorithm for the New Method. In this section, the runtimes of New Method that uses Prune Merge and uses Two-Way Merge will be compared.

In Table 4.6, run times of both implementation can be seen for *Runs(L)* = 2. In this case both algorithms make 1 merge operation. Prune Merge method is slightly faster even for 1 merge operation.

Table 4.7 : Execution times (ms) of two implementations for *Runs(L)* = 2.

	500	5000	10000	100000
New Method	0,00000960	0,00009500	0,00017240	0,00211520
Two-Way Merge	0,00002528	0,00016001	0,00031703	0,00394314

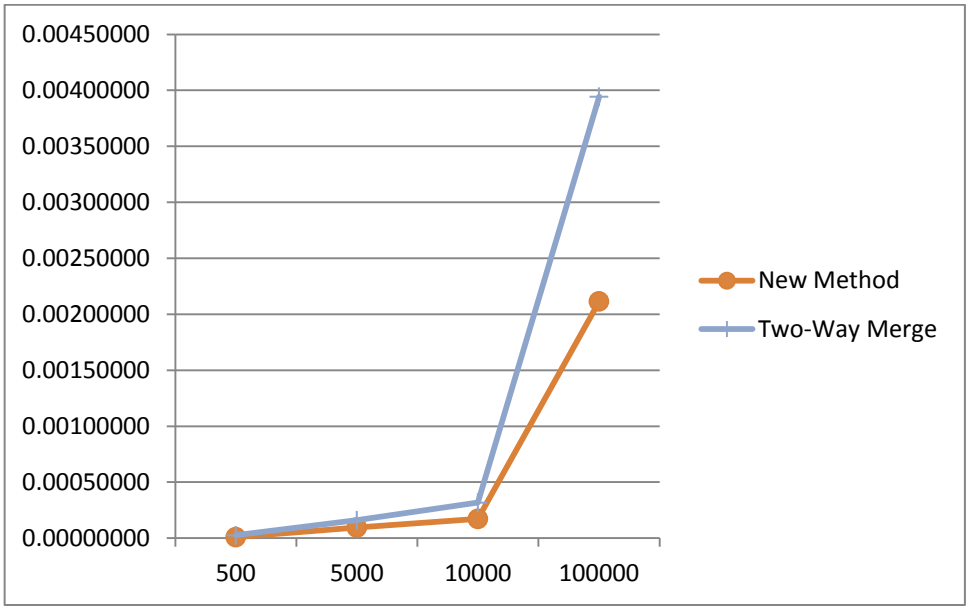

Figure 4.13 : Line graph for Table 4.7.

Tabe 4.7, Table 4.8 and Table 4.9 show run times of three implementation for *Runs(L)* = 5, 10 and 20. In these cases, number of merge operations increases and the difference of performances can be seen more clearly.

Figure 4.8 : Execution times (ms) of two implementations for *Runs(L)* = 5.

	500	5000	10000	100000
New Method	0,00001240	0,00010760	0,00020900	0,00280160
Two-Way Merge	0,00005290	0,00035426	0,00063836	0,00636465

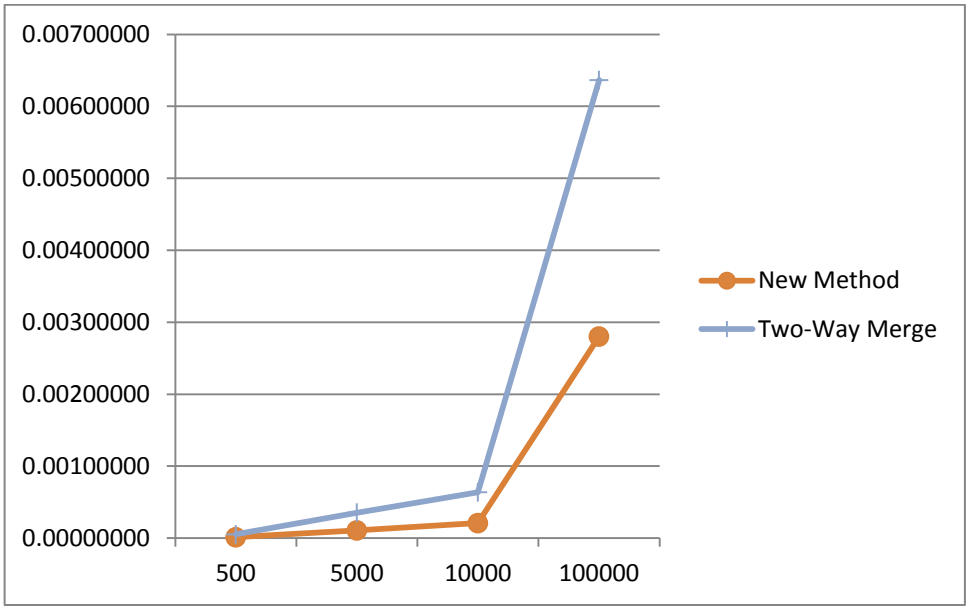

Figure 4.14 : Line graph for Table 4.8.

Table 4.9 : Execution times (ms) of two implementations for *Runs(L)* = 10.

	500	5000	10000	100000
New Method	0,00001500	0,00011400	0,00023300	0,00323340
Two-Way Merge	0,00005533	0,00040548	0,00079062	0,01042674

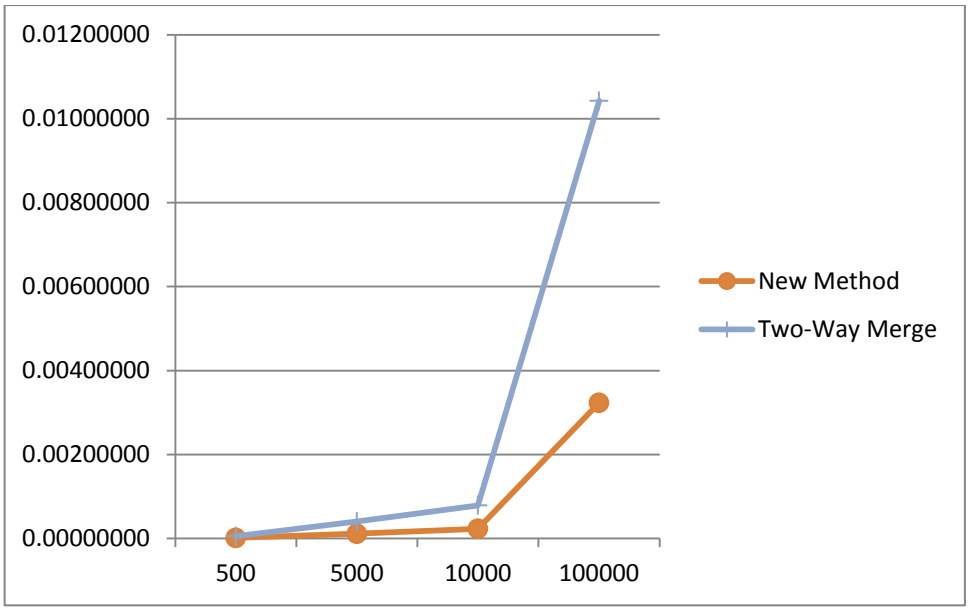

Figure 4.15 : Line graph for Table 4.9.

Table 4.10 : Execution times (ms) of two implementations for *Runs(L)* = 20.

	500	5000	10000	100000
New Method	0,00002440	0,00012500	0,00027080	0,00346940
Two-Way Merge	0,00010608	0,00071514	0,00184034	0,01544227

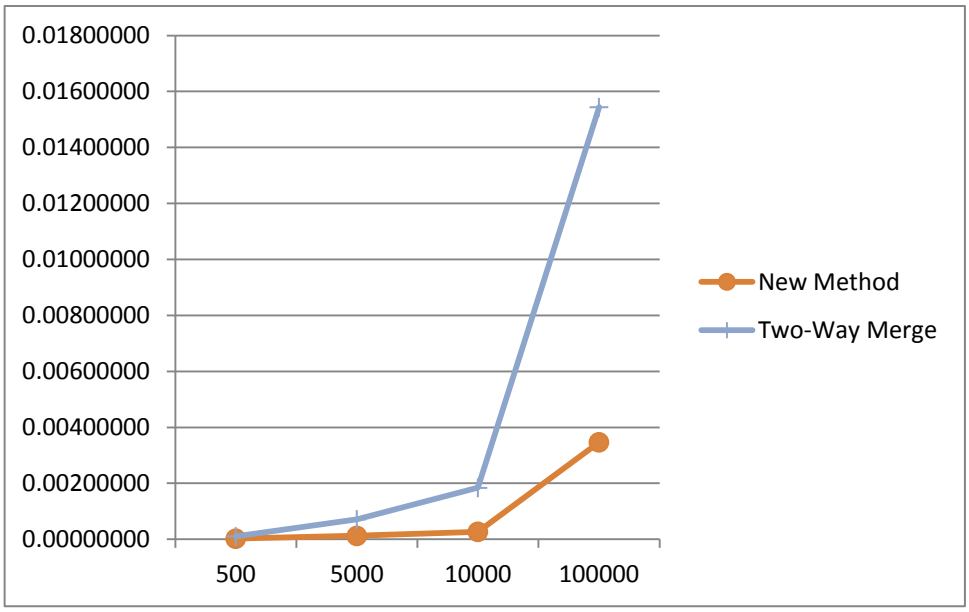

Figure 4.16 : Line graph for Table 4.10.

4.2.3. Comparison of Space Complexities

Space complexities of reviewed sorting algorithms can be seen in Table 4.11.

Table 4.11 : Space Complexities of Algorithms.

Insertion	*O(1)*
QuickSort	*O(1)*
CK-Sort	*O(n)*
MergeSort	*O(n)*
SplitSort	*O(n)*
New Method	*O(n)*

Insertion Sort and Quicksort algorithms are in-place sorting algorithms therefore they do not use any extra memory space for the list. CK-Sort uses an extra *O(n)* memoy space to store unsorted pairs. Splitsort can be implemented as an in-place algorithm but in-place version will scan the list twice therefore it will be slower. MergeSort uses *O(n)*

space to store merged array. The new method uses an *O(n)* memory space for buffering and an *O(n)* memory space for merge operations.

Table 4.12 : Memory Usage (bytes) of Algorithms.

	500	5000	10000	100000
Insertion	2024	20034	40112	400236
QuickSort	4128	8144	80208	800256
CK-Sort	6552	65054	120054	1200056
MergeSort	6634	63076	125624	1250768
SplitSort	2192	20288	40312	400384
New Method	6304	60416	120444	1200528

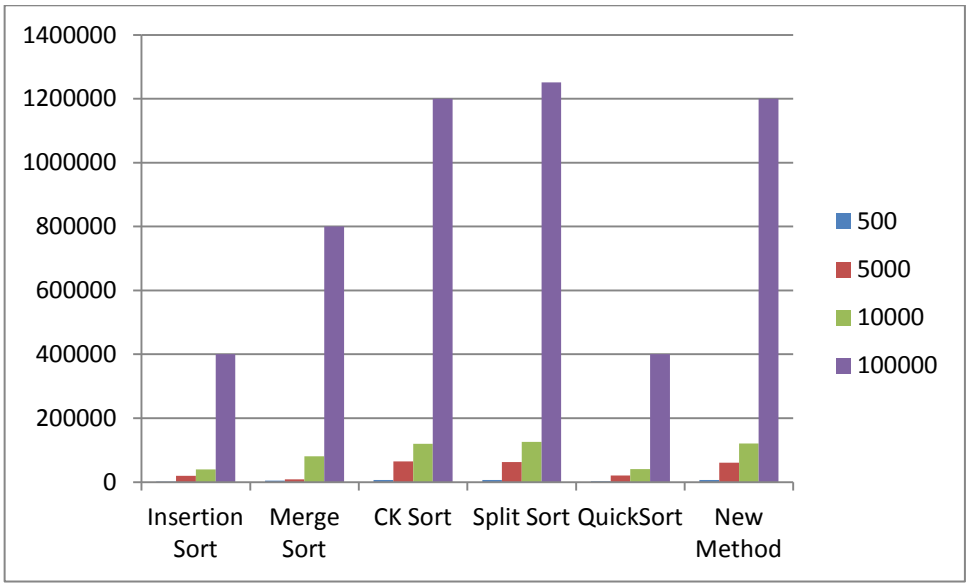

Figure 4.17 : Memory Usage (bytes) of Algorithms.

Table 4.12 shows average memory usage of reviewed algorithms. All algorithms are tested with 1000 sample lists for each list size. In Figure 4.17 it can be seen that memory usage of adaptive algorithms are nearly equal to each other because of the extra memory space used for storing sorted sub-lists.

The new method allocates memory related to the size of list therefore it is recommended to use the new method with smaller lists. Also in previous results (Table 4.1 – Table 4.6) it is clear that the new method performs better with low *Runs* values in proportion to the list size. Based on experiments, the maximum recommended *Runs* value is ~10%. With the values greater than 10%, as seen in Table 4.6, the new method shows worse performance than the other algorithms.

5. CONCLUSION

Sorting is one of the fundamental techniques in computer science. Several generic and adaptive sorting algorithms are designed to meet the requirements.

Previous works proved that the theoretical lower bound of a comparison based sort algorithm is *O(nlogn)* [8] [3] and it is not possible to design such generic sort algorithm. When characteristics of the data are taken into account, it is possible to design an adaptive algorithm which performs better than *O(nlogn)* for specific cases.

In this work, the term sortedness is analyzed in several parameters and a *Runs*-optimal adaptive algorithm for nearly sorted lists is designed. As seen in results, the new algorihm performs better from similar adaptive algorithms such as Cook-Kim Sort and Splitsort for some sortedness parameters.

However, the new algorithm is a pure merge based algorithm. It is adaptive for *Inv* measure but it could be changed as a hybrid sorting algorithm like Cook-Kim Sort and can be made more adaptive for *Rem* measure and could take advantage of powerful generic sorting algorithms like Quicksort.

Herein, the new algorithm is analyzed for only single computing system but it could be developed to work for distributed systems as a future research work.

REFERENCES

[1] Black, P. E (2008). Strand Sort. Dictionary of Algorithms and Data Structures, U.S. National Institute of Standards and Technology.

[2] Cook, C. R, Kim, D. J. (1980), Best Sorting Algorithm for Nearly Sorted Lists. Communications of the ACM, 23, 11, p.620 – 624.

[3] Cormen, T. H., et al (2009). Introduction to Algorithms, 3rd Edition. The MIT Press.

[4] Estivill-Castro, V., Wood, D. (1992). A Survey of Adaptive Sorting Algorithms. ACM Computing Surveys, Vol.24 Issue 4, p.441 – 476.

[5] Fredman, M. L. (1975). On Computing the Length of Longest Increasing Subsequences. Discrete Mathematics 11, p.29 – 35.

[6] Hwang, F. K., Lin, S. (1971). Optimal Merging of 2 elements with n elements. Acta Informatica 1, p.145 – 158.

[7] Katajainen, J., et al (1996). Practical In-Place Merge Sort. Nordic Journal of Computing Issue 3, p.27 – 40.

[8] Knuth, D. E. (1998). The Art of Computer Programming Vol. 3: Sorting and Searching 2nd Edition. Reading: Addison – Wesley.

[9] Levcopoulos, C., Petersson, O. (1990). Splitsort – An Adaptive Sorting Algorithm. MFCS ’90 Proceedings of the Mathematical Foundations of Computer Science, p.416 – 422.

[10] Mannila, H. (1985). Measures of Presortedness and Optimal Sorting Algorithms. IEEE Transactions on Computer, Vol. C-34 No. 4, p.318 – 325.

[11] Martelli, A. (2006). Python in a Nutshell, O’Reilly.

[12] Mehlhorn, K. (1984). Data Structures and Algorithms, Vol. 1: Sorting and Searching. Monographs in Theoretical Computer Science, An EATCS Series. Springer – Verlag.

[13] Özalp, O. C., Akın, M. (2012). Optimization of Merge Based Sort Algorithms on Nearly Sorted Lists. Proceedings of the 12th WSEAS International Conference on Advances in Mathematical and Computational Methods, p.136 – 140.

[14] Sedgewick, R. (1983). Algorithms. Addison-Wesley.

[15] Seward, H. H. (1954). Information Sorting in the Application of Electronic Digital Computers to Business Operations. MIT Digital Computer Laboratory.

[16] Skiena, S. S. (2008). The Algorithm Design Manual, 2nd Edition. Springer.

[17] Wainwright, R. L. (1985). A Class of Sorting Algorithms Based on Quicksort. Communications of the ACM, 28, 4, p.396 – 402.

BIOGRAPHICAL SKETCH

Özalp was born in İzmir on June 27th 1985. In 2003 he graduated from Selçuk Lisesi, İzmir. He began his undergraduate studies at Ege University Computer Engineering Department same year. In 2007, he received his BSc degree in Computer Engineering from Ege University. He began to work as a Software Engineer at the Software Architecture team at *IBTech*, which is the IT subsidiary of *Finansbank* at the same year. In 2009, he enrolled in MSc studies in Galatasaray University Computer Engineering department. In 2011, he began to work at *Intertech*, which is the IT subsidiary of *Denizbank*, as a Software Developer and later as a Software Development Team Leader. He is currently working as a Software Developer at Safkan Yazılım. He is the co-author of the paper entitled "*Optimization of Merge Based Sort Algorithms in Nearly Sorted Lists*" which was published in the *Proceedings of the 14th WSEAS International Conference on Advances in Mathematical and Computational Methods* held at Sliema, Malta in September 7-9, 2012.