ISSN (Print) : 0974-6846
ISSN (Online) : 0974-5645
Indian Journal of Science and Technology, *Vol 8(S8), 226–232, April 2015* DOI: 10.17485/ijst/2015/v8iS8/71491

An Optimal Sorting Algorithm for Mobile Devices

Seunghwa Lee[1*], Eunseok Lee[2] and Jeong-Min Park[3]

[1]Division of Information and Communication, Baekseok University, Cheonan, 330-04, Korea; sh.lee@bu.ac.kr
[2]Department of Computer Engineering, Sungkyunkwan University, Suwon, 440-746, Korea; eslee@ece.skku.ac.kr
[3]Department of Computer Engineering, Korea Polytechnic University, Siheung, 429-793, Korea; jmpark@kpu.ac.kr

Abstract

Data sorting is an important procedure with many applications in computer science. Thus, various methods that improve the performance of sorting algorithms have been published. Most existing algorithms mainly aim to achieve a faster processing time for sorting and require memory space to load all data in the set. Such designs can be a burden for small devices that have relatively insufficient resources. This paper proposes a new sorting method for mobile devices that finds the minimum and maximum values by reading only a portion of the data into memory, unlike existing methods that read all data into memory. This phase is repeated as often as necessary in accordance with the size of the data set. The min/max values that are determined are written at the start and end point of the data set in storage. As the process is repeated, sorting can take place even if a small amount of memory is available, such as when using a mobile device. In order to evaluate the proposed method, we implemented a prototype for a music recommendation system, and we conducted comparative experiments with representative sorting methods. The results indicate that the proposed method consumed significantly less memory, and we confirmed the effectiveness and potentiality of the proposed method.

Keywords: Data Sorting Algorithm, Mobile Device, Music Recommendation System, Selection Sort

1. Introduction

Recently, various types of portable devices have gained widespread use due to the rapid growth of information technology. Previously many tasks had to be carried out at fixed locations by using desktops equipped with wired Internet, but these can now be performed on the move. Users can access services through various applications, such as to exchange instant messages with their acquaintances and to share their favorite images, without any constraints over time and space. Most of these applications manage and store various types of data[1,2,] and they such data is often sorted to offer more useful functions. For instance, address books, e-mail, documents, chat logs, and stored images are often presented as lists. In addition, modern music players sort data to record the number of times that a track is played and recommend items according to the user's favorite music. The amount data to be sorted can increase dramatically depending on the user's propensity to use the device.

Recently, various types of portable devices have gained widespread use due to the rapid growth of information technology. Previously many tasks had to be carried out at fixed locations by using desktops equipped with wired Internet, but these can now be performed on the move. Users can access services through various applications, such as to exchange instant messages with their acquaintances and to share their favorite images, without any constraints over time and space. Most of these applications manage and store various types of data[1,2], and they such data is often sorted to offer more useful functions. For instance, address books, e-mail, documents, chat logs, and stored images are often presented as lists. In addition, modern music players sort data to record the number of times that a track is played and recommend items according to the user's favorite music. The amount data to be sorted can increase dramatically depending on the user's propensity to use the device. In fact, most applications for mp3 music players manage additional information of about 2 Kbytes per song.

*Author for correspondence

Thus, if there are 1,000 music files, the size of the playlist increases to about 2Mbytes. Even though the computing power of portable devices is continually improving, the memory constraints and battery consumption are important issues. Therefore, the size of the data may be a burden when using the device. As a result, we propose what may be a more efficient way to sort data on portable devices that have limited resources.

Data sorting is an important procedure in the field of computer science. Thus, various methods that improve the performance of sorting algorithms have been proposed. Most existing algorithms aim for a faster processing time[3], and while speed is an important factor, we have also taken note of the limited resources of portable devices. In general, well-known sorting algorithms, such as quick sort or shell sort, load all data from storage into memory and then conduct the sorting process. As it was described above, the amount of data may dramatically increase depending on the situation, and processing the data can be an excessive burden for portable devices. On the other hand, selection sort may be relatively slow, but it also allows sorting even when only part of the data has been loaded. Therefore, if the processing time can be improved, this method may be a suitable for portable devices that have small amounts of memory available.

In this paper, we propose a new sorting method that finds the min and max values by reading only a portion of the data unlike existing methods that read all data to memory. This phase is repeated as many times as necessary according to the size of the data set. The min/max values that are determined are written in storage at the start and end points of the data set. As the process is repeated, the size of the data set is reduced by as much as n-2. Therefore, sorting can take place even if the amount of memory is small, such as in mobile devices, and the results after sorting can be obtained more quickly when compared to selection sort, which only finds the minimum value.

We conducted various experiments to evaluate the proposed method, and we compared the proposed method with existing representative methods in terms of the memory workload and processing time. In the series of experiments, the proposed method consumed significantly less memory and also exhibited a reasonable processing time. These results indicate that proposed method is effective and has potential for use.

The remainder of the paper is organized as follows. In Section 2, we introduce the characteristics of the existing representative sorting algorithms. In Section 3, we propose a new sorting method that is designed to overcome the resource limitations of mobile devices and also describe a music player application that implements the proposed method. Section 4 explains the experiments and the evaluation results for the proposed sorting method. Finally, Section 5 presents the conclusion.

2. Related Work

Quick sort is a well-known traditional sorting algorithm that was developed in 1960 by T. Hoare[4,5]. It is a representative divide and conquer algorithm that first divides a large array into two smaller sub-arrays and recursively sorts the sub-arrays. It makes O(n log n) comparisons to sort n items on average6. The advantage of using this method is that it has a high processing speed. The method is often faster in practice than other O(n log n) algorithms, and the pseudo code to sort elements i through k (inclusive) of an array A can be expressed compactly as follows[7]:

```
quick sort(A, i, k):
        if i < k
            p := partition(A, i, k)
        quicksort(A, i, p – 1)
    quicksort(A, p + 1, k)
```

Many researchers have proposed a number of variant algorithms that improve the performance of this method. Bentley and McIlroy[8] show how to mitigate the worst case by making a careful selection of a pivot. They propose using a dynamic method based on the array size instead of picking the first or a random pivot. After many years, Yaroslavskiy[9] proposed the dual-pivot quicksort to improve both the average and the worst cases by picking two pivots instead of one pivot[10]. In addition, there are many methods, such as three-way radix quicksort (which is also known as the multikey quicksort) that combines some of the characteristics of radix sort and quick sort[11]. Williams and Floyd proposed heapsort, a classical sorting algorithm that provides good performance[12,13]. The method uses a heap, which is a simple data structure, to efficiently support the priority of the data set. The method starts by constructing a heap, and once the heap is constructed, the element in the first position of the array will

be defined as the maximum and can be swapped into its correct position. The method can be thought of as an improved selection sort. Although in practice it is somewhat slower on most machines than a well-implemented quicksort, it has the advantage of having a more favorable worst-case O(n log n) runtime[14]. In addition, it has many variant algorithms, such as bottom-up heapsort[15] and weak-heapsort[16,17].

These methods mainly aim for a faster processing time and provide good performance. However, they require a memory space to load all of the data set that is to be sorted. As the services provided by applications continue to diversify, the data that is managed can often grow enormously. This can be a burden for small devices that have relatively insufficient resources. Moreover, when the method is implemented with an array type, the array size should be designated as the maximum in the light of the fact that the amount of data may become more massive.

Although the selection sort algorithm has a lower processing time with O(n2) time complexity, it can perform ordering without loading all data into memory at the same time. The method looks for the minimum value in a list and interchanges the first element with it. Then it looks for the second minimum value in the list excluding the first element, which was found in the previous step. This process is continued until the list is arranged. The pseudo code for the method can be expressed as follows:

```
classical selection sort(A, n):
    for i ← 0 to n – 1
    // where n is the size of the data set
    do
        min_val ← A[ i ]
        for j ← i + 1 to n – 1
        do
            if A[ j ] < min_val
                min_val ← A[ j ]
        done
        if min_val != A[ i ]
            exchange A[ i ] min<—>val
    done
```

The method is notable for its simplicity. It can also sort if even one element is loaded from the list, and all of the data to be sorted does not load into memory at once because it finds a minimum value. If the processing time improves, it may be suitable for portable devices with limited resources.

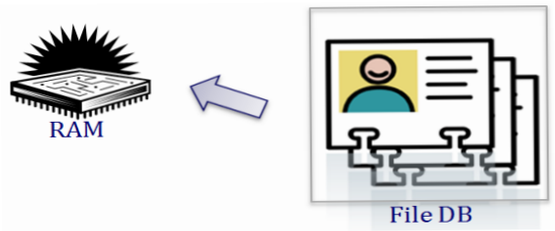

Figure 1. Conventional methods should read the all data to be sorted into memory.

Researchers have proposed many algorithms to improve thecf the list by a factor of 2, eliminating some loop overhead without actually decreasing the number of comparisons or swaps. It more often refers to a bidirectional variant of bubble sort.

Our ideas for the proposed system also began by finding the minimum and maximum values at the same time, and we made an attempt to improve the existing algorithm in order to suit devices with limited resources.

3. Proposed System

The proposed method is designed to perform efficient sorting in small devices that have relatively insufficient resources. The core idea is very simple. The method reads only one element from a data list and sets up the data as the initial minimum and maximum values. Subsequently, it finds actual minimum and maximum values by comparing the initial values with the remaining elements. The values that are found are exchanged with each data at the start/end point of the list, and this process is repeated for the remaining elements until all data is sorted.

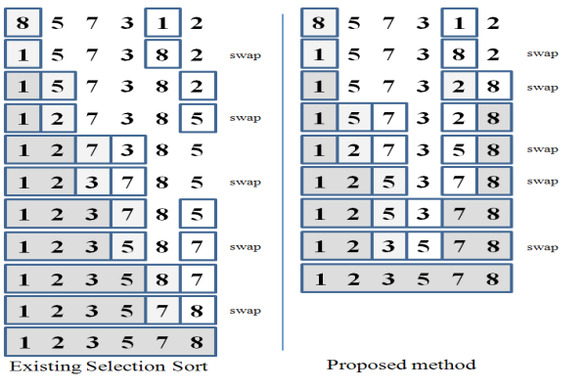

Figure 2. Comparative case for existing selection sort and proposed method.

The pseudo code for the method is shown as follows:

```
proposed selection sort(A, n):
    for i ← 0 to n – 1
    // where n is the size of the data set
    do
        max ← min ← i
        for j ← i + 1 to n – 1
        do
            if A[ j ] < A[ min ]
                min ← j
            else if A[ j ] > A[ max ]
                max ← j
        done
        if min != i
            exchange A[ i ] <—> A[ min ]
        if max != n – 1
            exchange A[ n - 1 ] <—> A[ max ]
        n ← n - 1
    done
```

The algorithm needs size-1 comparisons in a manner similar to selection sort. However, the actual processing time can greatly improve due to decrease in size for the data set and for the number of times each step is repeated. Figure 2 presents a comparison between the existing selection sort, which finds the minimum value and exchanges it with the value of the first element, and the proposed method.

The proposed method allows reading only one instead of all entities at once, which would be loaded into memory by using the min/max value. In order to explain the efficacy of the method, we have designed a music recommendation system that adopts such functionality.

When the user selects the 'recommend' button, the system executes a function that rearranges the playback order according to the number of times that a track has been played. This application produces a personalized playlist of music that reflects the user's preferences. During the initial operation, the system reads the information from mp3 music files within the specified path and makes a playlist. As in most existing music player applications, the playlist conforms to the XML format and includes various description of the recorded song.

The file includes not only basic information, such as the song title, artist, and genre of the music, but also a unique number to manage the song, path, and play/skip count. We use the line number of the song in the playlist file as a unique number. The skip count may be considered as negative feedback from the user for the recommendation. The size of the data that is managed is of about 1–2Kbytes per song. As mentioned above, if about 1,000 songs are managed, the size of the playlist file may grow to about 2Mbytes. Figure 3 shows a sample of the structure of the playlist that has been determined.

```
<song>
    :
</song>
<song>
    <key> ID</key>
    <integer>102</integer>
    <key>Name</key>
    <string>I'm Yours </string>
    <key>Artist</key>
    <string>Jason Mraz</string>
    <key>Location</key>
    <string>file://localhost/....</string>
                :
    <key>PlayCount</key>
    <integer>23</integer>
    <key>SkipCount</key>
    <integer>2</integer>
</song>
<song>
    :
</song>
```

Figure 3. Playlist sample.

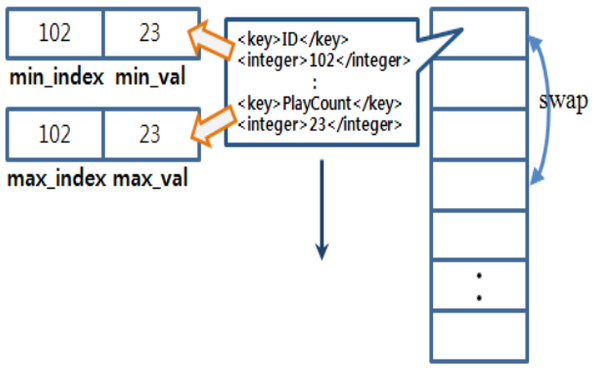

Figure 4. Playlist sorting.

The information for the song used in the sorting process is the song's ID (line number, index) and play count. The proposed system reads the first element of the playlist, sets the initial index, and the min/max variables. Then, the system loads the next element, compares it with the min/max variables, and renews the variables if necessary.

After all of the elements have been checked, and the actual min/max values are found, the system exchanges the first/last elements in the list with the element that has the value that was found. The size of the list is reduced by as much as a factor of 2, and this process is repeated when the size of the list is zero. As a result, although the proposed method executes many comparison processes, it is able to sort even if only four integer variables and one structure variable are to be recorded as an element in the swap process.

4. System Evaluation

A desktop computer (Intel Core i5 CPU 750, 2.67GHz, 2GB RAM, Linux operating system) was used to compare the performance of the proposed sorting method to that of existing methods. We also implemented a prototype for the music recommendation system. For the first experiment, we measured the processing time by changing the size of the data set. Randomly generated integers were used as data for sorting, and the time was calculated as the average speed of 10 repetitions. The processing times for the bubble, insertion, and selection algorithms were very slow, and those of the shell and quick algorithms were remarkable, as expected. The quick sort algorithm exhibited the fastest speed, but its relative performance changed significantly for the worst case, e.g., for the reverse order. Moreover, sometimes the sorting process did not run because the increase in data resulted in memory overflow since the method is generally implemented as a type of recursive function. The processing time for the proposed method was slower than that of the shell or quick methods. However, when compared to the results of selection sort, it is reasonable if the lower memory footprint is considered. Table 1 and Figure 5 show a comparison of the results. Next, we sorted the playlist and compared the results of the proposed system with those of the selection and quick algorithms. The time was calculated like in the preceding tests as the average speed for 10 repetitions. The proposed system exhibited faster results than the existing selection sort algorithm.

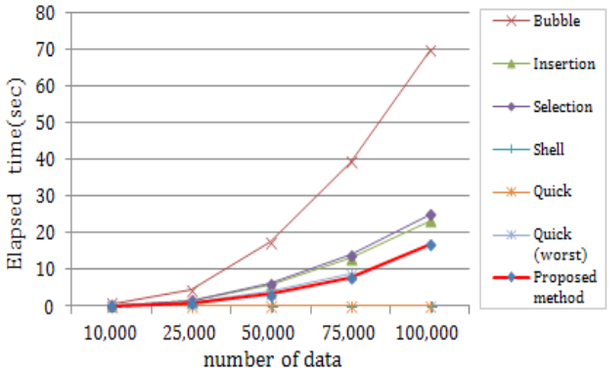

Figure 5. Comparisons of processing time (integer).

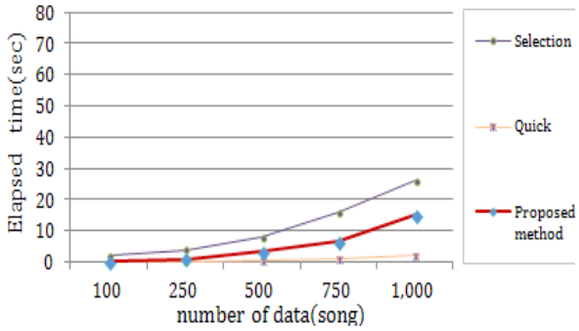

Figure 6. Comparisons of processing time (playlist).

The processing time for the proposed method was slower than that of the shell or quick methods. However, when compared to the results of selection sort, it is reasonable if the lower memory footprint is considered. Table 1 and Figure 5 show a comparison of the results. Next, we sorted the playlist and compared the results of the proposed system with those of the selection and quick algorithms. The time was calculated like in the preceding tests as the average speed for 10 repetitions. The proposed system exhibited faster results than the existing selection sort algorithm. The second experiment compared memory consumption. To this end, we wrote a simple script (i.e., ps aux | grep './sort' >> mem.txt) to continually monitored changes. The results for the existing methods changed as the size of the data set changed because the data were loaded by using the dynamic allocation. In contrast, the space complexity of the proposed method is O(1). As a result, we confirm that the proposed method can perform data sorting by using only significantly less memory. Table 1 shows the experiment results.

Table 1. Comparisons of processing time (integer)

# of data	10,000	25,000	50,000	75,000	100,000
Bubble	0.709	4.377	17.441	39.347	69.652
Insertion	0.247	1.475	5.867	13.169	23.364
Shell	0.004	0.009	0.035	0.044	0.059
Quick(w)	0.165	0,995	3,941	8,825	Overflow
Proposed method	**0.156**	**0.987**	**3.245**	**7.837**	**16.728**

Table 2. Comparisons for memory consumption

# of data	10,000	25.000	50,000	75,000	100,000
Bubble	336	488	728	900	1054
Insertion	334	464	712	856	1036
Quick(w)	840	1160	2472	3784	4236
Proposed method	**204**	**204**	**204**	**204**	**204**

5. Conclusion and Implications

As IT is further developed, the performance of mobile devices is constantly being improved. However, device resources are still limited, and to address such limitations, this paper presents a new sorting algorithm for mobile devices. The method finds the min and max values by reading only a portion of the data unlike existing methods that read all the data into memory.

In order to evaluate the proposed method, we have conducted comparative experiments with representative sorting methods. These results indicate that the method consumed significantly less memory, and we confirmed the effectiveness and the potential use of the proposed method. The proposed method is expected to be applicable for mobile applications, such as for e-mail, messenger, and image viewing services. Furthermore, we evaluated the proposed method by implementing a prototype for a music recommendation system. The intelligent music recommender uses contextual information, which is currently being studied by many researchers[22–25]. Our recommendation system may be further developed to incorporate more meaningful information through combination contextual information, such as the weather at the moment that the user plays or skips such music. In future work, we will study a way to improve the processing time and will include an adaptive method that executes different sorting methods in accordance with the device's resource availability.

6. References

1. Yang Y, Chow K, Hui L, Wang C, Chen L, Chen Z, Chen J. Forensic analysis of popular chinese internet applications. Advances in Digital Forensics. 2010 Jan; 6:285–95.
2. Liu J, Meng F, He J, Wu S. Analysis of DB files based on compound document format. PACIIA 2009: Proceedings of the Second Asia-Pacific Conference on Computational Intelligence and Industrial Applications, vol 1; 2009 Nov. p. 87–90.
3. Knuth DE. The art of computer programming – sorting and searching. Addison-Wesley Professional; 1998 May.
4. Hoare CAR. Algorithm 64: Quicksort. Comm ACM. 1961 Jul; 4(7):321–2.
5. Hoare CAR. Quicksort. Comput J. 1962 Apr; 5(1):10–6.
6. Sedgewick R. Implementing quicksort programs. Comm ACM. 1978 Oct; 21(10):847–57.
7. Kushagra S, Lopez-Ortiz A, Munro JI, Quao A. Multi-pivot quicksort: theory and experiments. ALENEX 14: Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments; 2014 Jan. p. 47–60.
8. Bentley JL, McIlroy MD. Engineering a sort function. Software Pract Ex. 1993 Nov; 23(11). p. 1249–65.
9. Yaroslavskiy V. Dual-pivot Quicksort. Research Disclosure RD539015. 2009 Sep; Available from: http://www.iaroslavski.narod.ru/quicksort/ DualPivotQuicksort.pdf
10. Cormen TH, Leiserson CE, Rivest RL, Stein C. Introduction to algorithms. Cambridge: The MIT Press and McGraw-Hill Book Company; 2001. p. 2.
11. Bentley JL, Sedgewick R. Fast algorithms for sorting and searching strings. Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms; Society for Industrial and Applied Mathematics; 1997. p. 360–9.
12. Williams JWJ. Algorithm 232: Heapsort. Comm ACM. 1964 Jun; 7(6):347–8.

13. Floyd RW. Algorithm 245: Treesort 3. Comm ACM. 1964 Dec; 7(12):701.
14. Skiena SS. The algorithm design manual. 2nd ed. Springer-Verlag; 2008. ISBN: 978-1-84800-069-8.
15. Wegener I. Bottom-up heapsort, a new variant of heapsort beating, on an average, quicksort (if n is not very small). Theor Comput Sci. 1993 Sep; 118(1):81–98.
16. Dutton RD. Weak-heap sort. BIT Numerical Mathematics. 1993 Sep; 33(3):372–81.
17. Edelkamp S, Wegener I. On the performance of weak-heapsort. Lect Notes Comput Sci. 2000 Mar; 1770:254–66.
18. Jadoon S, Solehria SF, Rehman S, Jan H. Design and analysis of optimized selection sort algorithm. IJECS. 2011 Feb; 11(1):16–22.
19. Chand S, Chaudhary T, Parveen R. Upgraded selection sort. Int J Comput Sci Eng. 2011 Apr; 3(4):1633–7.
20. Min W. Design and analysis on bidirectional selection sort algorithm. ICETC: Proceedings of the 2nd International Conference on Education Technology and Computer; 2010 Jun; 4:380–3.
21. Kapur E, Kumar P, Gupta S. Proposal of a two way sorting algorithm and performance comparison with existing algorithms. International Journal of Computer Science, Engineering and Applications. 2012 Jun; 2(3):61–78.
22. Wang M, Kawamura T, Sei Y, Nakagawa H, Tahara Y, Ohsuga A. Music recommender adapting implicit context using 'renso' relation among linked data. J Inform Process. 2014 Apr; 22(2):279–88.
23. Domingues MA, Rezende SO. The impact of context-aware recommender systems on music in the long tail. BRACIS: Proceedings of the IEEE 2013 Brazilian Conference on Intelligent Systems; 2013 Oct. p. 119–24.
24. Park MKS, Moon N. The effects of personal sentiments and context on the acceptance of music recommender systems. MUE: Proceedings of the IEEE 5th FTRA Intenational Conference on Multimedia and Ubiquitous Engneering; 2011 Jun. p. 28–30.
25. Dror G, Koenigstein N, Koren Y. Web-scale media recommendation systems. Proceedings of the IEEE. 2012 Sep; 100(9):2722–36.