Item Retention Improvements to Dropsort, a Lossy Sorting Algorithm

Abram Jackson and Ryan McCulloch
Computer Science
North Dakota State University
Fargo, ND 58102
abramj@acm.org

Abstract

This paper presents the Dropsort algorithm and suggests improvements. Dropsort is a simple comparison sorting algorithm that performs in $\theta(n)$ time by simply removing those elements that are not already in the correct order. Because of this, it is not a true sorting algorithm as the output does not contain all the elements of the input. In other words it is a lossy "sorting algorithm" that performs far more quickly than true sorting algorithms. Much of the original data is lost when using this algorithm unless the numbers to sort are in a mostly ordered format already. If more than a few elements are far from their correct position nearly all of the elements are removed.

Two major improvements are proposed that greatly broaden the applicability of the Dropsort algorithm. In the first improvement, a concept of recent memory is added to the algorithm. For example, if several elements are consecutively dropped, the preceding element is assumed to be far from ordered and is removed instead. This is shown to increase the usefulness of Dropsort by increasing the number of retained elements. With a specific type of introduced error, 10,000% more of the original elements are retained over standard Dropsort. In the second improvement, each element is compared to not only determine if it is in sorted order, but if the next element is in order as well. Only if both elements are in order is the item kept. It is important to note that neither of these suggested improvements greatly increase the complexity or execution time of the Dropsort algorithm.

Finally, the applications and limitations of these improvements and Dropsort in general are explored. This improved algorithm has many applications to very quickly sort large data sets when specific elements are not as important as the assurance that the list is sorted. This could be the case when a sorted list is critical but the source providing the pre-sorted input is not trustworthy. Also, applications in fields with large data sets and where speed is of utmost importance are discussed.

1 The Dropsort Algorithm

Dropsort was first proposed as an alternative sorting algorithm by David Morgan in September of 2006. He asserts on his website that it is “a highly efficient, in-place, one-pass sorting algorithm” [1]. Traditional comparison-based sorting algorithms have been proven to perform with $\theta(n \log n)$ comparisons. However, Dropsort differs enough from the definition of a sorting algorithm that it is able to produce a sorted list by using a linear number of comparisons.

1.1 Algorithm Description

The algorithm is simple. A list of numbers to be sorted is traversed sequentially. Each element is compared to the element previous; if the current element is equal to or larger than the previous element nothing changes, but if it is smaller it is removed, or “dropped” from the list of numbers to be sorted. When a pass has been completed of all the elements in the list, the algorithm has finished. The elements remaining in the list are in increasing order. Example 1 contains pseudocode for the Dropsort algorithm.

```
int previousInt;
for (int i = 0; i < ARRAY_SIZE; i++)
    if (randomArray[i] < previousInt)
        randomArray.removeElementAt(i);
    else
        previousInt = randomArray[i];
```

Example 1: Pseudocode for Original Dropsort Algorithm

It is easy to understand that this algorithm performs with a linear number of comparisons. One comparison is performed for every element in the list except the first element, which is accepted without being compared to another element. Dropped elements are never recovered and so this algorithm can be considered lossy, which is a term borrowed from the field of data compression.

1.1 Limitations of Original Algorithm

Dropsort is effective when the elements appear in a mostly sorted form already. The algorithm requires the same number of comparisons every time a set of numbers is sorted, but the number of elements retained varies depending on the order of the list. Ideally, few elements need to be dropped from the list. The algorithm performs well when an element belongs earlier in the list. That element is removed, but if the list is mostly sorted the next element will likely be in order. However, the algorithm is much less effective when an element belongs later in the list. This element is retained in the output; however the next several elements are likely to be lost.

Also important to the number of elements that Dropsort will retain is how far an out-of-order element is from its appropriate place. For example, in the list of numbers $\{1,3,2,4,5\}$, the algorithm would remove element 2 and keep the other four. However, in the list $\{1,5,2,3,4\}$, the algorithm would remove $\{2,3,4\}$ and keep only 1 and 5. Although both lists had the second element out of order, Dropsort performed better on the first list because the out-of-order element was much closer to its correct place. In the extreme case where the elements are in reverse order only the first element is kept in the list.

This drawback greatly limits the applicability of Dropsort. Data must already be close to ordered to obtain good performance. However, the elements that are out of place are preferred to be too late in the list instead of too early, and failing that elements should be close to their appropriate place as possible.

2 Improvements

Dropsort in its current form is rarely very useful. However, with the following two improvements, which still require a linear number of comparisons, its usefulness broadens considerably.

2.1 Recent Memory

As previously discussed, misplaced elements that belong much later in a partially sorted list cause many subsequent elements to be dropped. An improved algorithm could recognize that this is happening and take steps to prevent too many elements from being removed. To this end, the algorithm could be modified to remember the number of consecutive elements that have been dropped. If the number of consecutive elements is more than x, a constant number related to the size of the list and the variability of its elements, the algorithm will remove the offensive element that belongs much later and continue iterating over the rest of the list.

As an example, consider the list {1,10,2,3,4,5,6,7,8,9}. The original Dropsort would remove the last eight elements. However, with this proposed improvement where $x = 3$ the algorithm would proceed as follows: 1 and 10 would both be kept at first, while 2 and 3 are dropped. When the algorithm compares 4 to 10 and recognizes that this would be the third consecutive element dropped, instead of dropping it 10 is removed and 4 is kept because it is larger than 1. The algorithm proceeds as normal and returns {1,4,5,6,7,8,9}. The pseudocode for Dropsort with this modification is in example 2.

```
int age = 0;
int previousInt;
int previousIndex;
for (int i = 0; i < ARRAY_SIZE; i++)
    if (randomArray[i] < previousInt)
        age++;
        if (age >= RECENCY)
            age = 0;
            randomArray.removeElementAt(previousIndex);
            previousInt = previousInt;
    else
        previousInt = randomArray[i];
        previousIndex = i;
```

Example 2: The Recent Memory Modification to Dropsort

2.2 Double Comparison

A second improvement can be made by doubling the number of comparisons performed. Instead of only comparing the current element with the previous kept element, it can be compared with both the previous and the next element. The current element will only be kept if it is properly between the two. Although this change would seem to reduce the number of elements kept, it makes up for this by dropping elements far from their appropriate index, which would cause many more elements to be dropped.

In the previous example of the list {1,10,2,3,4,5,6,7,8,9} the element 10 would be dropped even before the algorithm moves on to the next element because 10 is less than 2. The resulting list would have all the elements except for 10. The pseudocode for this improvement is in Example 3.

```
int previousInt;
for (int i = 0; i < ARRAY_SIZE; i++)
    if (randomArray[i] > previousInt &&
            randomArray[i] < randomArray[i + 1]))
        randomArray.removeElementAt(i);
    else
            previousInt = i;
```

Example 3: The Double Comparison Modification to Dropsort

3 Performance of Improvements

As each of the improvements requires an asymptotically equal number of comparisons, performance for Dropsort implementations will be measured in the number of elements retained. For the first experiment, a list of pseudorandom positive integers is sorted with the original Dropsort, Dropsort with the recent memory improvement, the double comparison improvement, and both improvements together. The numbers are generated via .NET's Random library. The average number of elements retained during 1000 trials with 10,000 elements is detailed in Table 1.

	Dropsort	Recent Memory $(x = 10)$	Double Comparison	Recent Memory and Double Comparison $(x = 10)$
Elements Retained	13	7	10	5

Table 1: Elements Retained Sorting a Randomized List

As mentioned previously, Dropsort is not very useful for sorting random data, but instead works better for lists that are partially sorted already. To make a partially sorted list, I took a sorted list and replaced a certain percentage of items (randomization factor) within it with a randomly generated number. Example 4 has pseudocode for the process used in this experiment. Table 2 has the results of 1000 trials of the four algorithms when replacing different percentages of the elements in this way.

```
for (int i = 0; i < TEST_SIZE; i++)
    if (random.NextDouble() < RANDOMIZATION_FACTOR)
        randomArray[i] = random.NextIntLessThan(TEST_SIZE);
```

Example 4: Means for Generating a Partially-Sorted List

Randomization Factor	Original Dropsort	Recent Memory $(x = 4)$	Double Comparisons	Recent Memory and Double Comparisons $(x = 4)$
0.00	10000	10000	9999	9999
0.01	220	9717	9858	9847
0.05	26	8618	717	9187
0.10	18	7357	1247	8327
0.20	17	5438	661	6736
0.30	16	3822	286	5244
0.40	13	2682	347	4030
0.50	13	1702	83	2431
0.60	12	1065	78	1423
0.70	14	466	14	506
0.80	10	195	14	154
0.90	16	4	10	1
0.99	11	2	10	9
1.00	9	1	10	3

Table 2: Elements Retained with Partial Randomization

Another important factor is the speed that the algorithms perform. Table 3 shows the average time taken during one thousand trials to sort one million elements with the four algorithms compared with .NET's implementation of the sort() function, which is based on quicksort [2]. The list used contained only positive integers less than one million.

	Original Dropsort	Recent Memory $(x = 4)$	Double Comparisons	Recent Memory and Double Comparisons $(x = 4)$	.NET sort (based on quicksort)
Time in ms	7,814	16,995	7,918	39,846	57,330

Table 3: Time to Sort One Million Integers with a Randomization Factor of 0.30

3.1 Analysis of Results

The first thing we can notice is that none of the Dropsort algorithms or improvements perform very well for completely unsorted data. In the trial documented in Table 1 Dropsort without any improvements kept 13 elements, but that is still less than 2% retention. The improvements slightly lower performance. None of the algorithms perform well enough on random data to be useful.

However, when the input data is partially sorted the algorithms perform much better. The recent memory improvement increases the number of elements kept more than one hundred-fold until the list has 40% randomization. The modification is less effective as more randomness is introduced. This is because the number of elements dropped after an out-of-place element is more likely to be reset by another element that is even farther out of place. As the randomization approaches 100% the modification yields the same number of elements kept as completely random data, as expected.

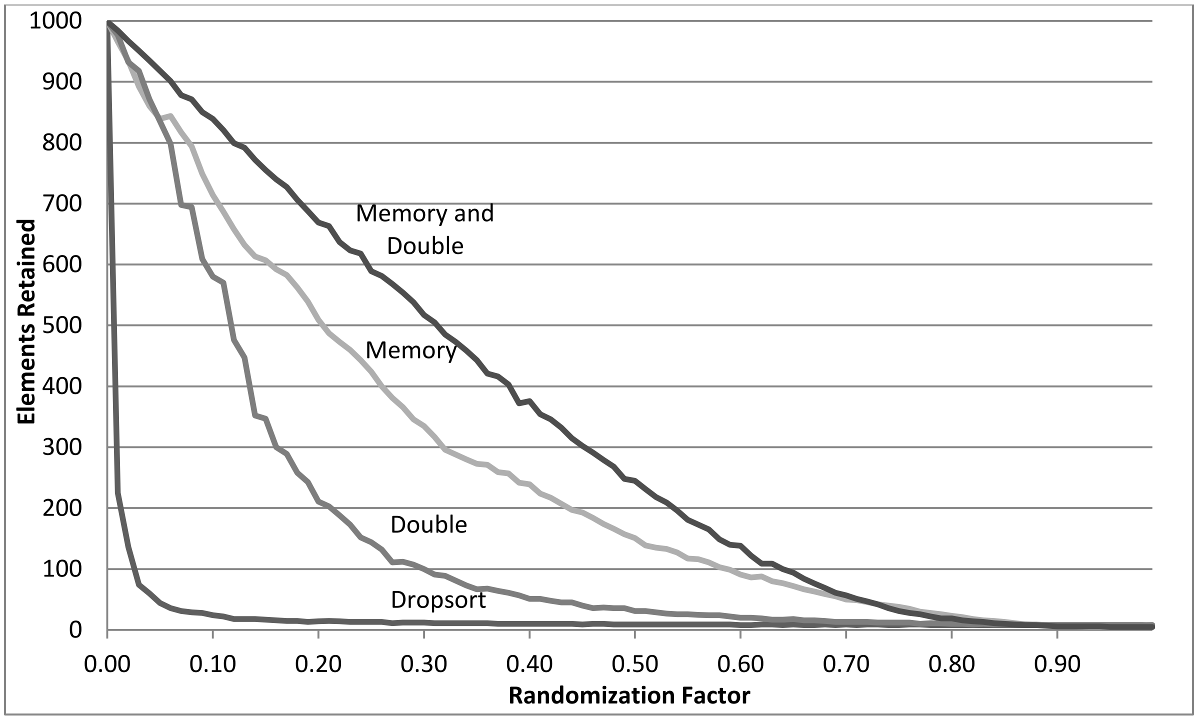

Figure 1: Comparison of Algorithm Performance

The modification to compare each element with both the previous and the next element in the list is less effective than the recent memory improvement, but still provides an excellent improvement over the original algorithm. At 10% randomization, this improvement is more than one hundred times more effective than the original. However, this gain is lost more quickly than the improvement with recent memory. The double comparison modification is not useful for sorting an element if there are two sequential out-of-place elements, which quickly becomes more common when the data is more random.

Figure 1 provides a dramatic comparison of the four algorithms. It details the results of one thousand tests of sorting a 1000 element array. Each of the improvements is far more efficient than the original Dropsort algorithm. The double comparison improvement by itself is only effective at low levels of randomization. However, when combined with the recent memory improvement it retains far more elements than either could alone. In fact, there is synergy with the two improvements. For example at 40% randomization the combined number of elements with the two improvements alone is 3029, but the combination of the two algorithms retains 4030 elements.

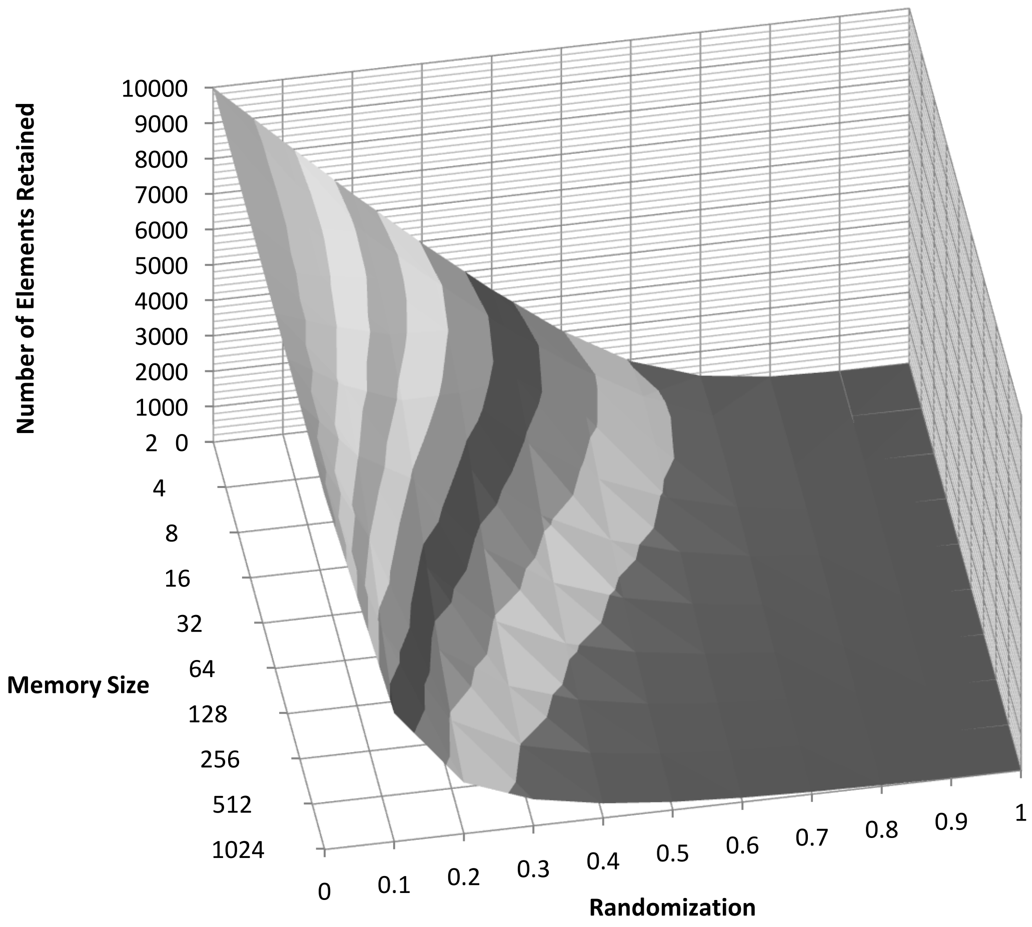

Figure 2: Relationship Between Recent Memory-Size and Randomization

Figure 2 was generated by performing 1000 trials of sorting 10,000 elements with different factors of randomization and values for x by using the algorithm modified by both the recent memory and double comparison improvements. The data shows a bowing out at small memory sizes and a concave shape at larger memory sizes. This suggests that although large memory sizes are effective for low amounts of randomized elements, a large memory performs very poorly with the data is fairly random. Conversely, a small memory, although slightly more effective for any amount of randomization, provides a much greater advantage for data that is more random.

4 Applications

It is difficult to find direct applications of the Dropsort algorithm simply because there has been little in the way of research or development for lossy sorting algorithms. However, the advantages of using a sorting algorithm with a linear number of comparisons may outweigh the drawbacks in many cases, especially with the two improvements, which retain more than 90% of the input data that has 5% randomization.

Most data gleaned from the real world is not perfect. Sensors can retrieve bad values, users will type data incorrectly, and other programs may have errors. If you consider the randomized data to be “bad” the improvements retain 9187 out of 9500 good elements at 5% randomization, which is 96% successful. In this manner, then, Dropsort can be used as a kind of error-correction to obtain a sorted list.

4.1 Performance

As an algorithm that only requires a linear number of comparisons, Dropsort can be an excellent choice when speed is of the utmost importance. The results in Table 3 show a clear improvement over quicksort. Even still, much of the time is lost in overhead and is not a sufficient judge of comparison performance. A more dramatic test would be sorting complex objects over multiple fields.

Consider a case where a large amount of data to be sorted exists in a flat text file. Dropsort when sorting the lines in the file must only delete a row. Other sorting algorithms require that lines be deleted, stored, and inserted into another place in the file – often many times. This is more difficult in a text file than an in-memory array, and can require even more time in other structures. For example, it is much easier in a linked list to remove an element than to move an element to another location.

A further advantage of Dropsort and which also applies to the suggested improvements is that it sorts the elements in-place. For most implementations of the algorithm, only enough memory for one element is required.

4.2 Simplicity

Dropsort is a very simple algorithm for obtaining a sorted list of elements. It can be implemented in just a few short lines of code and is easy to understand or teach. Even bubble sort in comparison is more complex. With just one loop and one conditional, it is possible to describe Dropsort in a single sentence. This makes it an excellent example to use with students first learning about algorithms and concerning the topics of sorting, asymptotic complexity, and approximate solutions.

The improvements described in this paper are also easy to understand and implement. The double comparison modification in particular may easily come to mind when a student traces the execution of the original algorithm. This makes these improvements a valuable tool to discuss refining algorithms to find better solutions. In contrast, the bidirectional improvement to bubble sort is easy to understand but difficult to implement.

4.2 Poor Data

There are many cases when a program requires that incoming data is sorted but the source cannot be trusted. This is particularly important when the source is human, but also should be considered when the source is another program. If the data is input as sorted, Dropsort will iterate over it cheaply in linear time. If the data is not sorted, but is close, Dropsort with these improvements will correct the data to be sorted in linear time as well. In any application then where the data must be sorted before it can be used Dropsort should be considered as a means of fixing the input data.

4.3 Further Research

There are several fields where lossy algorithms are acceptable or even preferable. Image, audio, and video data is often compressed for ease of storage or transfer. They can be compressed with lossless algorithms, but these are usually not as space efficient. Lossy algorithms can also be used for data conversion where only approximate equivalents are known of a different data format.

Although Dropsort is the first lossy “sorting” algorithm, further study should be conducted into the usefulness of approximation with new or existing sorting algorithms. There are no doubt applications in sorting similar to the field of compression, particularly when the data set is very large or difficult to compare.

5 Conclusion

Dropsort is an algorithm that on its own does not lend itself to many uses because of its lossy nature. However, with the two improvements discussed in this paper it becomes significantly more useful to sort or clean data. For a list that is 99% sorted, the modified Dropsort algorithm keeps more than 98% of the data. Meanwhile, it performs asymptotically faster than the fastest comparison-based sorting algorithms. The improvements also follow very neatly from the original algorithm and so can be used as an example to teach students about asymptotic complexity and modifying algorithms to improve them.

References

[1] Morgan, D. (2006, September 03). Dropsort. Retrieved from http://www.dangermouse.net/esoteric/dropsort.html

[2] List(T).Sort Method (Comparison(T)). (2008). Microsoft Developer Network. Retrieved March 16, 2011, from http://msdn.microsoft.com/en-us/library/w56d4y5z.aspx