Creation of a New Case for LUPSort : ALTERNATING

Evan B. Golub
Moshe Augenstein
Brooklyn College, City University of New York
Department of Computer & Information Science
Brooklyn, New York 11210

Abstract

The LUPSort was recently introduced in a paper presented at the 1990 ACM SIGCSE Technical Symposium. Although the algorithm was presented, only a few special cases were analyzed in detail.

In this paper we summarize the algorithm and present a new class of key distributions for analysis. These distributions are analyzed both mathematically and empirically, and the consistency of the results is shown. Finally, the significance of this class of distribution in the overall development of the case study is presented.

1 Introduction

In the February 1990 issue of SIGCSE, Merritt and Nauck published a paper titled "Inventing A New Sorting Algorithm : A Case Study."[1] The purpose of this paper was to show the path of development of a new sorting algorithm from concept, to algorithm to mathematical proof of complexity.

The algorithm suggests using a modified version of the solution to Dijkstra's Longest Upsequence Problem[2] to create a list structure which can be sorted efficiently. The modified LUP algorithm gives a set of ordered lists which is then merged by pair-wise techniques.

Merritt's paper concludes by showing that the cases of sorted and reverse sorted input data will run in $\mathbf{O(n)}$ time. Additionally, the worst case is mathematically proven to be less than $\mathbf{2nlog_2n}$. However, while the best cases are analyzed in detail, the other cases are not given a suitable review.

The purpose of this paper is to continue the case study of the invention of this new sorting algorithm. When creating a new algorithm, it is helpful to know more than just the upper and lower bounds of its complexity. In order to study the complexity in other cases, it is helpful to create special classes of input data, as well as using extensive sets of empirical results.

2 The LUPSort

The LUPSort is divided into three distinct stages. The **first stage** is to create the *m-structure* by using the solution to Dijkstra's Longest Upsequence Problem. The *m-structure* is a set of lists made up of the elements of the original data set. The heads of these lists are in increasing sorted order, and the elements on each list are sorted as well. Elements are placed on each list(the *m-structure* is built) as follows. First it must be determined whether a new number belongs on an entirely new list, the condition for this being that the new number is greater than the first, or head, element of the last list in the structure. If this is not the case, it is then determined if the new number belongs on the first list. The condition for this is that the number is less than or equal to the head of that first list. If this too is not the case, then a binary search is done on the head elements of the existing lists to determine the list on which the new number belongs.

Given Input Data : | 2 | 12 | 55 | 64 | 19 | 70 | 23 | 79 | 88 |

Stage 1 gives the m-structure : | 2 | 12 | 19 | 23 | 70 | 79 | 88 |
55 64

The **second stage** of the sort is the creation of the *m*-structure* from the *m-structure*. The definition of this process is to take any single element list in the *m-structure*, and link it to the top of the following list. This can be implemented in a two pass method. The first pass is from list K-1 to list 1, where K is the number of

lists in the *m-structure*, linking the single element lists. The second pass is from list 1 to list K, eliminating the now empty lists and obtaining the new value for K.

Stage 2 gives the m*-structure : | 2 | 23 | 70 |
12 64 79
19 88
55

The **third stage** of this sort is the pair-wise merge of the lists in the *m*-structure*. The content of the structure allows bypassing the comparison of the heads of any two lists, since the heads are already in sorted order. Further, once there is one list of a pair which is exhausted, the remainder of the other list is simply linked to the bottom of the new list holding the merge to that point.

Stage 3 gives the sorted list :
| 2 | 12 | 19 | 23 | 55 | 64 | 70 | 79 | 88 |

3 Best Case Complexity of LUPSort

To analyze the complexity of this algorithm, we consider the best and worst cases for each stage of the sort. In [1] the two best cases are discussed, these being sorted and reverse sorted data sets.

The case of reverse sorted data is the best case for the sorting algorithm described. In the first stage of the sort, all of the elements will be placed on the first list. Since the first element is automatically placed, we are left with n-1 elements to place. Two comparisons must be made to determine that an element belongs on the first list, giving us a total of 2(n-1) = **2n-2** comparisons for the first stage.

The second stage of the sort looks at 2K-1 lists during its execution, where K was the number of lists in the original *m-structure*. For reverse sorted data K=1, so that only **1** comparison is needed in this stage.

The third and final stage is the pair-wise merge, but since there is only one list, no merging is done. The net comparisons done for the sort is therefore 2n-2(Stage 1) + 1(Stage 2) + 0(Stage 3), which equals **2n-1** in the case of reverse sorted input data. This is **O(n)**, and is quite acceptable as a best case.

The other best case mentioned in [1] is that of sorted data. In this situation, each new element is placed on a new list. Since once again the first element is placed automatically, we are left with n-1 elements to place. It takes 1 comparison to determine that an element belongs on a new list, resulting in **n-1** comparisons for this stage.

When we reach the second stage, there are n lists, so that K=n. Since this stage depends entirely on K, **2(n)-1** comparisons are done here (using the same formula as we used in reversed sorted, 2K-1). However, after this stage is completed, since all of the lists were single element lists, there is only one list remaining.

Therefore, in third stage(as in the reverse sorted case), no comparisons are necessary. The net comparisons done for this data set is therefore n-1(Stage 1) + 2n-1(Stage 2) + 0(Stage 3) = **3n-2**. Again, **O(n)** and clearly a suitable best case.

4 Development of a New Case(ALTERNATING)

To find LUPSort's mathematical average case one would need to calculate the probability and complexity of every possible input scenario. However, by observing LUPSort's intriguing symmetry, and its self-balancing properties, we can develop a case which reflects how LUPSort works on a general, randomly created data set. This is possible due to the fact that the general ordering of a random set of numbers will not resemble either of the best cases(sorted/reverse sorted), and in this situation the best and worst cases for each stage have a counter balancing effect on each other. Empirical results from over 50,000 test runs on random data sets showed the range of the number of comparisons done to be well within +-5% of the median for those runs. The following explains the various counter-balancing features which lead to the parameters for the formal definition of our new case to be referred to as ALTERNATING.

Within the first stage, the creation of the *m-structure*, there is a balancing effect between best and worst case situations. The best case for inserting a number into the *m-structure* is if it needs to be placed on a new list, where it will take 1 comparison. The worst case is when the value needs to be inserted on the last list of the structure. Here 2 comparisons would be needed prior to the binary search, then an additional $\lfloor \log_2 r \rfloor + 1$, where r="the number of lists - 2", giving a total of $\lfloor \log_2 r \rfloor + 3$. However, the more frequently the "best case" occurs, the larger the "worst case" will be when it occurs, since the best case increases r's value.

For the first stage, where the worst case would be when extensive binary searches need to be done to insert elements into the *m-structure*, one such worst case would be as follows. If the first $^1/_2$n elements were in sorted order, and then the next $^1/_2$n elements all had to be placed in the next to last list. Here it would take $\mathbf{^1/_2 n + {}^1/_2 n[\log_2(^1/_2 n) + 1]}$ comparisons to accomplish the first stage of the sort.

For the second stage of the sort, the worst case would be where K=n, such as the case of sorted data. However, when we have single element lists, they do not need to be merged. Therefore, with either the worst case for the first or second stage, the third stage would not be at its worst case.

The worst case for the third stage needs for there to be many lists to merge, none of which will be exhausted too early in a particular merge. If for each pair of lists being merged, one list were exhausted while the other were still full, this could be seen as a best case. If both lists are on their last element when one is exhausted, this could be

seen as a worst case. A hypothetical average here would be where one list is half-way through when the other is exhausted.

From this information we can see that the data set we want needs to have no single element lists generated from it, yet still have a high number of lists. For our ALTERNATING case we therefore need a set which will give us $^1/_2$n lists, which will each have 2 elements, in an order which will allow the second of each pair of lists being merged to be exhausted when the first list is only half way through.

5 Formal Definition and Complexity of Case

The data set ALTERNATING can be defined as : For any position K, if K is odd, then K+1's number is less than K's number, and if K is even, then K+1's number is greater than K's number. Additionally, every second number beginning with the first is in non-increasing order, and starting with the second is in increasing order.

***e.g.* ALTERNATING :**
500,1,499,2,498,3,497,4,496,5,495,6,494,7,493,8

If we trace through the sort algorithm, using an ALTERNATING data set of size n, where n is a power of 2, we can come up with a formula for the number of comparisons done by any set of ALTERNATING numbers. This formula can then serve as the theoretical median of random data set runs on some data set of size n.

The first stage of the LUPSort is to create the m-structure. In the case of ALT, $^1/_2$ of the numbers, beginning with the first of the set, and continuing as every second number in the set, belong on a new list. Since the sort automatically puts the first number into a new list, and to place the other numbers into new lists takes 1 comparison each, then these $^1/_2$n numbers take $\mathbf{^1/_2n - 1}$ comparisons to place.

For the other $^1/_2$n numbers, they will be placed on an existing list. The first two of these numbers will take 2 comparisons each to place, giving **4** comparisons. The remainder are placed with the binary search. For each of these insertions, there are initially 2 comparisons done before reaching the binary search. The binary search itself then takes $\lceil \log_2 r \rceil + 1$ comparisons, where r="the number of lists - 2." Therefore the insertion of one of these numbers will take $2 + (\lceil \log_2 r \rceil + 1)$, or $\lceil \log_2 r \rceil + 3$ comparisons. With ALT, for the R^{th} number of these $^1/_2$n numbers, to be placed, there are R lists in the m-structure. The solution for these $^1/_2$n numbers is then 4*(first 2 numbers)* + Σ $(\lceil \log_2 r \rceil + 3)$ as r goes from 1 to $(^1/_2n)-2$ *(representing #s 3 to $^1/_2n$)*. The solution to this equation is $\mathbf{^1/_2 n \log_2 n - \log_2 n + 2}$. Adding the comparisons for the two groups of $^1/_2$n numbers gives us :

$$(^1/_2n - 1) + (^1/_2 n \log_2 n - \log_2 n + 2) =$$
$$^1/_2 n \log_2 n + ^1/_2 n - \log_2 n + 1$$

The second stage of the LUPSort is the creation of the m*-structure from the m-structure. In the case of ALT, there are $^1/_2$n lists. Therefore, by the same equation as in the reverse sorted and sorted discussion, 2K-1, the stage looks at $2(^1/_2n)-1$ lists. So, the number of comparisons done here are :

$$\mathbf{n - 1}$$

The third and final stage of this sort is the pair-wise merge of the lists in the m*-structure. For the i^{th} pass in the pair-wise merge of the m*-structure created with ALT data, each pair of lists will take $1.5(2^i)-1$ comparisons to merge. The 2^i comes from the number of elements in each list during that pass. The 1.5 comes from the fact that the second of each pair of lists will be fully exhausted when the other list is half way through. The -1 comes from the previously stated fact that the heads need not be compared during the pair-wise merge. Additionally, for pass i, there are $^n/_{2(i+1)}$ *[n over 2 to the i+1]* pairs to merge. So, for each pass i, there are $\{^n/_{2(i+1)}\}*\{1.5(2^i)-1\}$ comparisons done. In all, there are $\log_2 n - 1$ passes through the lists. The sum of these passes can be written as the summation:

Σ $\{^n/_{2(i+1)}\}*\{1.5(2^i)-1\}$, as i goes from 1 to $\log_2 n - 1$

This summation is solved to be :

$$^3/_4 n \log_2 n - ^5/_4 n + 1$$

By adding the sub-totals from each stage of the sort, we come up with the total number of comparisons done to sort a file of ALT through the use of LUPSort to be :

Stage 1 $(\mathbf{^1/_2 n \log_2 n + ^1/_2 n - \log_2 n + 1}) +$
Stage 2 $(\mathbf{n - 1}) +$
Stage 3 $(\mathbf{^3/_4 n \log_2 n - ^5/_4 n + 1})$

$$\equiv \mathbf{^5/_4 n \log_2 n + ^1/_4 n - \log_2 n + 1}$$

6 Empirical Support for ALTERNATING Case

Now we have the ALTERNATING case of $^5/_4 n \log_2 n + ^1/_4 n - \log_2 n + 1$. In order to test its accuracy, we need empirical random results. By doing 1500 sorts on data sets created by a random number generator(Table 1), of sizes 64 and 512 elements, we can obtain a run-time average case.

For the size of 64, tests runs gave 484 comparisons as the average. By replacing n with 64 in the formula for ALTERNATING, we come up with 491. For the size of 512, test runs gave 5865 comparisons as the average. The ALT formula gives 5880. The range for the number of comparisons done on a data set of 512 random numbers

was from ~5600 to ~6100, no where near O(n), yet within a reasonable range of 5880. These results serve as justification to classify the case of ALTERNATING as the median value for the complexity of random data set LUPSort runs, and this case is $\mathbf{O(nlog_2n)}$.

7 Conclusion

While the sort does have good upper and lower bounds, $\mathbf{O(nlog_2n)}$ and $\mathbf{O(n)}$, they are extremely dependent on the input data. Additionally, our ALTERNATING case, representing the complexity of random tests, is in excess of $\mathbf{nlog_2n}$. By looking at the different possible situations, we can see that this sort is not a particularly useful one for real world applications.

If we know that the data will be almost sorted or reverse sorted order, we can use other sorts which are simpler and shorter, which take up less memory, that will still run in $\mathbf{O(n)}$ time. If the data is presented in a random, mixed up ordering, then using a straight merge will do the sort with less work than the LUPSort, as far as the ALTERNATING case has shown.

The sort is, however, useful as part of a case study for computer science students. Merritt's paper shows the steps taken in developing this complex sort. However, it is also important to show that after you have written an algorithm, and defined its best and worst cases, you need to continue. You have to see how it works in random data set situations, and how often a random run will fall near the best case. The process of establishing a data set which will result in an empirically supportable average for the random data set, as well as solving for its precise complexity are important steps for a computer science student to study. They will help develop the student's ability to better utilize his knowledge of mathematical induction and intuition.

REFERENCES

1 Susan Merritt and Cecilia Nauck. *Inventing a New Algorithm : A Case Study.* SIGCSE, 181-184(1990).

2 Edsger Dijkstra. *Some beautiful arguments using mathematical induction.* Acta Informatica, 13, 1-8(1980).

Table 1
Sample Empirical Results

N	# files	min comps	max comps	average	ALT
64	100	444	511	485	
	500	417	517	485	
	1000	417	517	485	
	1500	417	519	484	
					491
512	100	5657	6024	5879	
	500	5624	6100	5865	
	1000	5624	6129	5866	
	1500	5624	6129	5865	
					5880

Notes :

1) Special thanks to The Ford Foundation for their guidance and support

2) Pascal-coded interpretation of LUPSort algorithm may be obtained from the authors of this paper.