INVENTING A NEW SORTING ALGORITHM: A CASE STUDY

Susan M. Merritt
Pace University, School of Computer Science and Information Systems
1 Martine Avenue, White Plains, NY 10606

Cecilia Y. Nauck
The Masters School
49 Clinton Avenue, Dobbs Ferry, NY 10522

ABSTRACT

Dijkstra's algorithm for finding the length of the longest upsequence within a given sequence of numbers is used as the basis for a case study in the development of a new sorting algorithm. The case study has pedagogic value for several reasons: the motivation for each step in the development of the algorithm is explained; interesting data structures are introduced; the complexity of the algorithm is analyzed using mathematics familiar to first year computer science students; the appendices provide detailed description of the implementation of the algorithm as well as several interesting examples of its use. The proposed algorithm merits attention since it uses O(nlogn) compares in the worst case (actually less than 2nlogn) and O(n) for both ordered or nearly ordered and reverse ordered or nearly reverse ordered sequences.

1.0 INTRODUCTION

We provide a comprehensive and pedagogical description of an algorithm development process. The algorithm is new. It uses a variety of building blocks from computer science including well known techniques such as list merging, and lesser known techniques such as the efficient solution of the longest upsequence problem. It incorporates a variety of interesting data structures. It demonstrates how basic concepts in computer science and mathematics are used to develop and demonstrate an interesting and respectable algorithm.

We describe the longest upsequence problem and the elegant solution given by Dijkstra [2]. We then propose an extension to Dijkstra's algorithm to develop a sorting algorithm which uses O(nlogn) compares in the worst case and O(n) compares for an interesting class of data. Substantial analysis of the algorithm is demonstrated with respect to compares. The algorithm development provides a good case study for a first year computer science course.

2.0 THE LONGEST UPSEQUENCE PROBLEM (LUP)

Given a sequence s of n elements, an upsequence in s is a subsequence of s whose elements appear in s in non decreasing order. A subsequence of s is any subset of s in which the original order is retained (hence there are 2^n possible subsequences). The longest upsequence problem is:

> Give an algorithm which will return the length of the longest upsequence of a sequence s. (Note that there may be more than one longest upsequence of the same length. For example, the sequence [1,3,4,1,3,5] has the following upsequences of length 4: [1,3,4,5], [1,1,3,5], and [1,3,3,5].)

A solution with O(nlogn) worst case performance has been given by Dijkstra [2] and modifications have been given by Dewar, Merritt and Sharir [1].

Dijkstra's algorithm can be described as follows: Elements in the original sequence are examined from left to right. As each new element is examined it is inserted into the m-sequence, a sequence consisting of minimum rightmost elements of upsequences of length 1,2,... New sequence elements are inserted into the m-sequence either by being added to the right end (if x is larger than the rightmost element of the partial m-sequence) or by "bumping" an element already in the m-sequence (where the element being bumped is the smallest element in the m-sequence which is larger than the new element to be inserted).

Consider the following example, with input sequence s of length 8: s: 8, 1, 6, 5, 4, 7, 2, 3

```
step1 (insert 8) m:8
step2 (insert 1) m:1      (bump 8)
step3 (insert 6) m:1 6    (add 6)
step4 (insert 5) m:1 5    (bump 6)
step5 (insert 4) m:1 4    (bump 5)
step6 (insert 7) m:1 4 7  (add 7)
step7 (insert 2) m:1 2 7  (bump 4)
step8 (insert 3) m:1 2 3  (bump 7)
```

The length of the longest upsequence is 3 (and m contains the minimum rightmost elements of upsequences of length 1, 2, 3, respectively).

3.0 THE COMPLEXITY OF THE LUP

Dijkstra uses a binary search technique to determine which element of the m-sequence is to be replaced, which makes the worst case complexity of the algorithm O(nlogn); without the binary search the worst case complexity is $O(n^2)$.

If x=s[k] is the next element of the input sequence and m[j] is the rightmost element of the m-sequence, the heart of Dijkstra's algorithm is:

```
if m[j] <= s[k]
  then n := n + 1; m[n] := s[k];
  else "establish c such that"
   m[c-1] <= s[k] < m[c];
   m[c] := s[k]
end if;
```

Binary search uses a divide and conquer approach to find the correct placement of a given item within a given sorted list and is described in many places, for example [3]. From a pedagogical perspective it is interesting to review the mathematics of this procedure. The solution to the equation given below provides the number of times a given set can be so divided:

$(1/2)^x n = 1.$

or $2^{-x} n = 1$. Taking the log of each side using 2 as the base of the logarithm and using logn to mean the log of n base 2, the following result is obtained:

$\log 2^{-x} + \log n = \log 1$

or $-x + \log n = 0$

or $x = \log n.$

Since there are n elements that need to be inserted, the order of complexity for the entire process in the worst case is O(nlogn).

4.0 DEVELOPMENT OF THE SORTING ALGORITHM

In the solution to the LUP outlined above, bumped elements are discarded. It should be noted that if bumped elements could be saved, some useful ordering information would be preserved.

In the proposed sorting algorithm the m-array produced by Dijkstra's algorithm together with the saved bumped elements form something that we could call an m-structure. The building of the m-structure is the first phase of the proposed sort. In the m-structure each m[i] is the head of a list and initially set to point to nil. If the original m[i] of Dijkstra's m-array has not been "bumped", then it will continue to point to nil. Otherwise, it will be the head of a list of elements that have been "bumped" from that position. Such a list shall be referred to as a history queue labelled Li, with length li. Clearly, if Li is a history queue its corresponding length, li, must be at least 1 (no bumped elements) and at most n (every new element bumps the previous one). For example, the m-structure produced by applying Dijkstra's algorithm to the sequence 8, 1, 6, 5, 4, 7, 2, 3 is:

L1	L2	L3
1	2	3
8	4	7
	5	
	6	

For the case above, the length of the longest upsequence is 3. The m-structure would be initialized as an array of 8 lists of integers initialized to point to nil. After completion of the first phase of the algorithm, three of the lists would not be empty. The number of nonempty lists will be referred to as r. Clearly, r will be at least 1 (the case for reverse order sequences) and at most n (the case for ordered sequences). Here r=3. Note that the m-structure not only contains a sorted first row which is the m-array of Dijkstra's algorithm but also a series of history queues which are themselves sorted lists.

The second or sorting phase of the proposed technique will involve merging L1, consisting of the list 1 -> 8 ->nil, with L2, consisting of 2 -> 4 -> 5 -> 6 ->nil, to produce the list 1 -> 2 -> 4 -> 5 -> 6 -> 8 ->nil. This would then be merged with L3, which contains 3 -> 7 ->nil to produce a single list of ordered elements.

5.0 THE SORTING PHASE

In discussing the merge operation required to transform the m-structure into a single sorted array it is interesting to consider the various "shapes" the m-structure can have. In the case of reverse ordered data it will consist of a single m[i], namely m[1] pointing to a list of ordered elements. The work involved in forming this array has been of order n and no further compares are necessary to create the final sorted sequence. Every other case will produce an m-array with r>1 and the question of the best way to merge these r history queues is a central issue to be addressed.

Consider the simple case of merging two lists L1 and L2 of ordered data. In general, given lists of length p and q, the largest number of compares required to merge the two lists is well known (see [3], for example). In the worst case neither list would be exhausted early. Since the maximum number of compares in this case is always 1 less than the sum of the lengths of the two lists, there are p+q-1 compares in the worst case. Consider the merge of L1 and L2 which is begun by comparing 3 and 5,

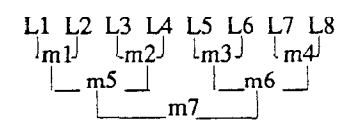
L1	L2	
3	5	then 5 and 8,
8	10	then 8 and 10,
12	15	then 10 and 12,
17		then 12 and 15,

and finally 15 and 17. The number of compares is 6 = 4 + 3 - 1. Furthermore, if it is known that the first element of L1 is always smaller than the first element of L2 (as is the case in the m-structure), the first compare can be avoided and thus the total number of compares can be reduced in the worst case to p+q-2.

There is a straightforward approach to merging the lists L1, L2, L3, L4,..., Lr. L1 and L2 are merged, and then the resulting queue is merged with L3 and the resulting queue is merged with L4, continuing in this manner until Lr is merged with the result of all the previous merges. A bit of reflection reveals that L1 is used in all r-1 merges. If the m-structure is such that most of the n numbers are in L1 then it is clear that this merge operation uses $O(n^2)$ compares in the worst case. Since there are many good sorting algorithms with O(nlogn) compares [3], a $O(n^2)$ technique for the merge alone is not acceptable.

A better way to merge the lists is to merge L1 and L2, L3 and L4, ..., L(r-1) and Lr and then to do pair-wise merges of the merged lists. The complexity for this kind of merge is less dependent on the "shape" of the m-stucture. The simplest case arises when r is a power of two. Consider the case L1, L2, L3, L4, L5, L6, L7, L8. Recalling that li represents the length of list Li, it is clear that the maximum number of compares required to merge L1 and L2 is l1+l2-2. Similarly, the subsequent merges yield the following results for the maximum number of compares per merge ("m1=>l1+l2-2" means that the maximum number of compares required to merge L1 and L2 is the sum of their respective lengths, l1 and l2, minus 2):

m1=>l1+l2-2
m2=>l3+l4-2
m3=>l5+l6-2
m4=>l7+l8-2
m5=>l1+l2+l3+l4-2
m6=>l5+l6+l7+l8-2
m7=>l1+l2+l3+l4+l5+l6+l7+l8-2



The total number of compares required for the merge is 3n-2(7). From the information above it is clear that the number of compares in this case is a function of n and not of the "shape" of the m-structure. This information can be used to arrive at a formula for the number of compares required to merge r lists when r is a power of two.

Since the shape of the m-structure is immaterial in the case that r is a power of two, suppose the n elements of the original array are evenly divided among r queues where r is a power of two. Then each list contains n/r elements and the maximum number of compares required to merge any pair is n/r + n/r -2 or 2(n-r)/r. Since the first pass requires r/2 such merges the maximum total number of compares for the first pass is (2(n-r)/r)(r/2) = n-r compares.

For the second pass, each queue is now of length 2n/r and hence the maximum number of compares required to merge two such lists is 2n/r + 2n/r -2. Since there are r/2 lists after the first pass, r/4 merges are required so the maximum number of compares for pass two is(2n/r + 2n/r - 2)(r/4) = (2n - r)/2.

Similarly, for pass three each queue is of length 4n/r. Since r/8 merges are required, the maximum total number of compares is (4n/r + 4n/r -2)(r/8) = (4n - r)/4.

For the fourth pass the maximum number of compares is (8n - r)/8.

For the kth pass the maximum number of compares is $(2^{k-1}n - r)/2^{k-1}$.

Adding all of the compares for each pass together the following is obtained:

If c = the total number of compares up to the kth pass, then

$c= (n-r)/1 + (2n-r)/2 + (4n-r)/4 + (8n-r)/8+...+(2^{k-1}n-r)/2^{k-1}$

$c= n-r/1 + n-r/2 + n-r/4 + n-r/8 + ... + n-r/2^{k-1}$

$c= (n+n+...+n) - (r/1 + r/2 + r/4 + r/8 + ... + r/2^{k-1})$

$c= kn - r(1 + 1/2 + 1/4 + 1/8 + ... + 1/2^{k-1})$

$c= kn - \frac{r(1 - (1/2)^k)}{(1/2)}$ (sum of a geometric series)

$c= kn - 2r + 2r2^{-k}$ (A)

Under the assumption that r is a power of two let k = log r base 2, denoted as logr. Using this value in formula (A) obtained above it follows that

$c = nlogr - 2r + 2r2^{-logr}$

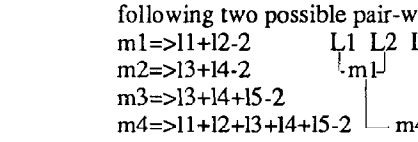
$c = nlogr - 2r + 2r^{-1}$

$c = nlogr - 2r + 2$

$c = nlogr - 2(r - 1)$

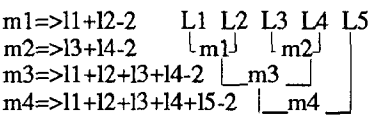
Note that in the example above the number of compares required to merge eight queues was 3n - 2(7).

To better understand the case when r is not a power of two, an example using 5 queues can be investigated. Consider the following two possible pair-wise mergings of L1, L2, L3, L4, L5:



the maximum total compares = 2n + (l3+l4) - 2(4)

On the other hand, if the 5 lists are merged in a different manner, the following is obtained:



the maximum total compares = 2n + (l1+l2+l3+l4-l5) - 2(4)

In this case the shape of the m-structure is relevant, but in any case it is clear that the maximum number of compares is less than 3n - 2(4).

The result can be arrived at analytically by using result (A) from above. If r is not a power of two the number of passes required to accomplish the r-1 required merges is log p where p is defined as the smallest power of two bigger than r. Letting k = log p where p is chosen as the smallest power of 2 bigger than r (recall we are assuming r is not a power of 2), it follows that log p = [llog rl] + 1. It is also true that r/p is less than 1. Therefore, (A) can be reduced to:

c = nlogp - 2r + 2(r/p) < nlogp - 2r + 2(1) = nlogp - 2(r - 1).

This is already a satisfactory result since it produces the sorted array in O(nlogn) compares in any case. It does have one problem, however. In the case of sorted data the m-array produced would have r equal to n and hence would require nlogn + n - 2n + 2 or nlogn -n + 2 compares to complete the merges. Although it is O(nlogn) in the worst case, it is a less than satisfying result for inorder data. To alleviate this problem and improve the performance of the process in general, an intermediate step is introduced between the formation of the original m-array and the final merge. This step can best be described as follows:

A) Suppose Dijkstra's algorithm is applied to the sequence

0 1 3 5 4 7 10 2 15 20

B) The m-structure produced is

0	1	2	4	7	10	15	20
		3	5				

Since r=8 is a power of two the maximum number of compares required to produce a single list is nlogr - 2(r -1) = 10log 8 - 2(8-1) =10(3)-2(7)=16.

C) Produce a new m-structure by looking at the pointer of each m[i]. If it points to nil let m[i] point to m[i+1]. This procedure will yield the following structure which shall be referred to as an m*-structure:

0	4	7
1	5	10
2		15
3		20

D) The corresponding r value shall be referred to as r*. Since r* = 3 we can use the formula for the non-power of two case which says the number of compares < nlog p -2(r-1). In this case p=4, r=3 and n=10 giving the result that the number of compares < 10(log 4) - 2(3-1) = 10(2) - 2(2) = 16. In fact we can compute the number of compares to be

m1 => 4 + 2 - 2 = 4

m2 => 6 + 4 - 2 = 8

total compares = 12 < 16.

The most important reason for this improvement is in the case of sorted data where an m-array such as 1 2 3 4 5 would yield the m*-structure

1
2
3
4
5

which gives the final result requiring no further merging.

This discussion would seem to imply that the worst case

scenario would be an m-structure which is a perfect n/2 by 2 rectangle since it would have the largest possible r with no hope of improvement by arrangement into a m*-structure. That intuitive result can be further explored as follows using the ordinary methods of the calculus. If we let r be a real number then we can study the behavior of the function f(r) = nlogr - 2(r-1) realizing that whenever r is a positive integer which is a power of two the value of f(r) accurately gives the maximum number of compares required for the merge operation.

For a given n, the maximum number of compares is a function of r given by

$f(r) = n\log r - 2(r - 1)$

$f(r) = n\log r - 2r + 2$

If r = 1 the number of compares required for the merge operation is 0. If r = n the maximum number of compares is nlogn -2n + 2. To investigate critical points of this function consider taking derivatives. $f'(r) = n/r - 2$. Setting this equal to 0 we obtain r = n/2. $f''(r) = -n/r^2$. Since this is always negative the critical point at r=n/2 is a relative maximum.

6.0 CONCLUSION

The sorting procedure described above consists of:

1) the creation of an m-structure, an extension of Dijkstra's algorithm for finding the LUP;

2) the transformation of the m-structure into an m*-structure as described above;

3) the pair-wise merge of the queues of the m*-structure.

It is a relatively efficient method of sorting all kinds of data. It has a lovely symmetry in that it treats ordered and reverse ordered sequences (and nearly ordered and reverse ordered sequences) equally well, performing the sort in O(n) compares. Other cases have O(nlogn) compares, with the maximum number of compares less than 2nlog n. We have described the motivation for various parts of this algorithm in some detail, as well as the results of various design decisions. The inventive process uses a creative data structure, motivates a binary search, and compares different merge techniques. Complexity is discussed on an intuitive level and is supported with fairly elementary mathematics. The result is a case study which can be used either to demonstrate the development process of an algorithm or to enrich/reinforce the basic concepts of a first year course in computer science.

REFERENCES

[1] Dewar, R.B.K., Merritt, S. M. and Sharir, Micha. Some modified algorithms for Dijkstra's longest upsequence problem. *Acta Informatica*,18, 1-15 (1982).

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

[3] Knuth, D.E. *The Art of Computer Programming* , Vol 3: Sorting and Searching. Addison Wesley, Reading, Mass. 1973

[4] L'Ecuyer, P. Efficient and portable combined random number generators. *Communications of the ACM*, Vol 31, Number 6 (1988).

The algorithm described above has been coded in Pascal. Copies of the code may be obtained from Cecilia Nauck.

APPENDIX: EMPIRICAL DATA

Sample output from several runs of the program are given below. Special test cases have been used and the trace has been set so that the different steps of the algorithm are shown.

```
Example: n = 8
  0  1  3  5  4  7  9  2
compares needed to form history queues = 13
heads point to  0 1 2 4 7 9
  1)  0
  2)  1
  3)  2 3
  4)  4 5
  5)  7
  6)  9
after transformation:
  1)  0 1 2 3
  2)  4 5
  3)  7 9
total number of compares = 21
```

```
Example: n = 8
  1  2  3  4  5  6  7  8
compares needed to form history queues = 7
heads point to  1 2 3 4 5 6 7 8
  1)  1
  2)  2
  3)  3
  4)  4
  5)  5
  6)  6
  7)  7
  8)  8
after transformation:
  1)  1 2 3 4 5 6 7 8
total number of compares = 7
```

```
Example: n = 8
  8  7  6  5  4  3  2  1
compares needed to form history queues = 14
heads point to  1
  1)  1 2 3 4 5 6 7 8 9
after transformation:
  1)  1 2 3 4 5 6 7 8 9
total number of compares = 14
```

For n=8, 64, 128, 256, and 512, four runs of the LUPSORT program were performed using random numbers generated by a RandomGenerator program. A summary of the results obtained is given below:

```
n=8  nlogn=24
average total compares needed to form history queues = 13
average total compares needed for sort = 22

n=64  nlogn=384
average total compares needed to form history queues = 258
average total compares needed for sort = 509

n=128 nlogn=896
average total compares needed to form history queues = 615
average total compares needed for sort = 1276
```

n=256 nlogn=2048
average total compares needed to form history queues = 1399
average total compares needed for sort = 2833

n=512 nlogn=4608
average total comapres needed to form history queues = 3138
average total compares needed for sort = 6601

In the last four of the five cases the total number of compares required for the sort is between nlong and 2nlogn. In the first case the total is less than nlogn.