MERGE SORT SYSTEM

Abhishek, Amit Sharma, Nishant Mishra
Department Of Electronics And Communication
Dronacharya College Of Engineering, Gurgaon

***Abstract*- Given an assortment with n rudiments, we dearth to reshuffle them in ascending order. Cataloging procedures such as the Bubble, Insertion and Selection Sort all have a quadratic time difficulty that limits their use when the number of rudiments is very big. In this paper, we introduce Unify Sort, a divide-and- conquer procedure to sort an N element assortment. We evaluate the O(NlogN) time complexity of unify sort theoretically and empirically. Our results show a large enhancement in efficiency over other procedures.**

I. INTRODUCTION:

Search engine is basically using sorting procedure. When you search some key word online, the pointer information is brought to you sorted by the importance of the web page.

Bubble, Selection and Insertion Sort, they all have an O(N2) time complexity that limits its usefulness to small number of element no more than a few thousand data points.

The quadratic time complexity of existing procedures such as Bubble, Selection and Insertion Sort limits their performance when assortment size increases.

In this paper we introduce Unify Sort which is able to rearrange rudiments of a list in ascending order. Unify sort works as a divide-and-conquer procedure. It recursively divide the list into two halves until one element left, and unify the already sorted two halves into a sorted one.

Our main contribution is the introduction of Unify Sort, an effective procedure can sort a list of assortment rudiments in O(NlogN) time. We assess the O(NlogN) time complexity theoretically and empirically.

The next section describes some existing sorting procedures: Bubble Sort, Insertion Sort and Selection Sort. Section 3 provides a details clarification of our Unify Sort procedure. Section 4 and 5 discusses empirical and theoretical evaluation based on efficiency. Section 6 recaps our study and gives a conclusion.

Note: Assortments we mentioned in this article have the size of N.

II. RELATED WORK:

Selection sort [1] works as follows: At each repetition, we identify two regions, sorted region (no element from start) and unsorted region. We "select" one smallest element from the unsorted region and put it in the sorted region. The number of rudiments in sorted region will increase by 1 each repetition. Repeat this on the rest of the unsorted region until it is pooped. This method is called selection sort because it works by repeatedly "selecting" the smallest remaining element.

We often use Insertion Sort [2] to sort bridge hands: At each repetition, we identify two regions, sorted region (one element from start which is the smallest) and unsorted region. We take one element from the unsorted region and "insert" it in the sorted region. The rudiments in sorted region will increase by 1 each repetition. Repeat this on the rest of the unsorted region without the first element. Experiments by Astrakhan [4] sorting strings in Java show bubble sort is roughly 5 times slower than pullout sort and 40% slower than selection sort which shows that Accumulation is the fastest among the three. We will evaluate attachment sort compared with unify sort in empirical evaluation.

Bubble sort works as follows: keep passing through the list, swapping adjacent element, if the list is out of order; when no talks are required on some pass, the list is sorted.

In Bubble sort, Selection sort and Insertion sort, the O(N2) time difficulty limits the performance when N

gets very big. We will introduce a "divide and conquer" procedure to lower the time complexity.

III. APPROACH:

Unify sort uses a divide-and-conquer approach: 1) Divide the range frequently into two halves 2) Stop dividing when there is single element left. By fact, single element is previously sorted. 3) Unifies two already sorted sub assortments into one. Pseudo Code:

a) Input: Assortment A[1…N], indices p, q, r ($p \le q < r$). A[p…r] is the assortment to be divided

A[p] is the beginning element and A[r] is the ending element Output: Assortment A[p…r] in ascending order

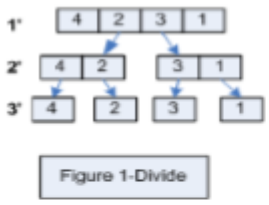
```
MERGE-SORT(A,p,q,r)
1        if p <r
2         then   q ← (r + p)/2
3         MERGE-SORT(A, p, q )
4         MERGE-SORT(A,q+1,r)
5         MERGE(A, p, q, r)
```

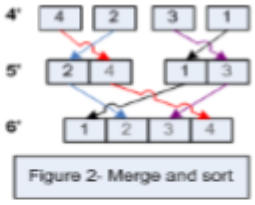
Figure 1. The merge sort algorithm (Part 1)

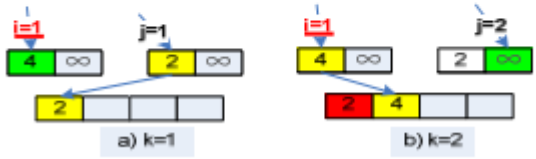
```
MERGE(A, p, q, r)
6     n1←q-p+1
7     n2←r-q
8     create arrays L[1...N1+1] and R[1...N2+1]
9     for i←1 to N1
10        do L[i]  ←  A[p+i-1]
11    for j  ←  1 to n2
12        do R[j]  ←  A[q+j]
13        L[N1+1]  ←  ∞
14        R[N2+1]  ←  ∞
15  i  ←  1
16  j  ←  1
17  for k  ←  p to r
18     do if L[i] ≤ R[j]
19              then A[k]  ←  L[i]
20                   i  ←  i+1
21              else A[k]  ←  R[j]
22                   j  ←  j+1
```

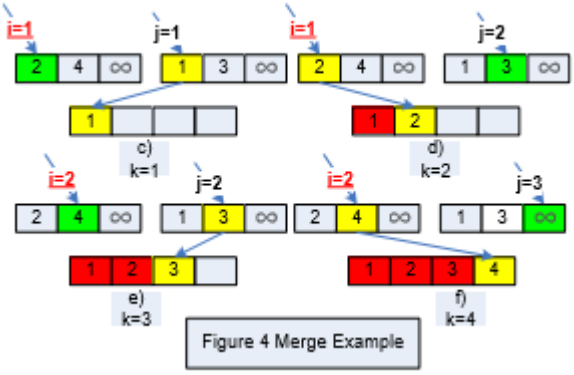
Figure 2. The merge sort algorithm(Part 2)

In figure 1, Line 1 panels when to stop separating – when there is single element left. Line 2-4 divides collection A[p…r] into two halves. Line 3', by fact, 2, 1, 4, 3 are sorted element, so we stop dividing. Line 5 unify the sorted rudiments into an assortment. In figure 2, Line 6-7, N1, N2 calculate numbers of essentials of the 1st and 2nd halve. Line 8, two blank assortments L and R are created in order to store the 1st and 2nd halve. Line 9-14 copy 1st halve to L and 2nd halve to R, set L[N1+1], R[N2+1] to ∞. Line 15-16, baton i, j is pointing to the first rudiments of L and R by default (See Figure 3,a);Figure 4,c)). Line 17, after k times judgment (Figure 3, 2 times; Figure 4, 4 times), the assortment is sorted in ascending order. Line 18-22, compare the rudiments at which i and j is "pointing". Append the smaller one to assortment A[p…r].(Figure 3,a) Figure 4 a)). After k times contrast, we will have k rudiments sorted. Finally, we will have assortment A[p…r] sorted.(Figure 3,4)









IV. THEORETICAL EVALUATION:

Comparison between two assortment rudiments is the key operation of bubble sort and unify sort. Because before we sort the assortment we need to compare between assortment rudiments.

The worst case for unify sort occurs when we unify two sub assortments into one if the biggest and the second biggest rudiments are in two separated sub

assortment. Let N=8, we have assortment {1,3,2,9,5,7,6,8,}. From figure 1.a, element 3(second biggest element),9(biggest element) are in separated sub assortment. # of comparisons is 3 when {1,3} and {2,9} were unifyd into one assortment. It's the same case unify {5,7} and {6,8} into one assortment.

From figure 1.b, 9,8 are in separated sub assortment, we can see after 3 comparisons element 1,2,3 are in the right place. Then after 4 comparisons, element 5,6,7,8 are in the right place. Then 9 is copied to the right place. # of comparison is 7.

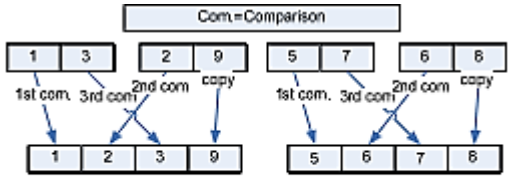


Figure 1.a Example of worst Case of Merge Sort

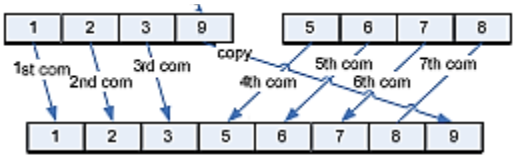


Figure 1.b Example of worst Case of Merge Sort

Let T(N)=# of comparison of unify sort n assortment element. In the worst case, # of comparison of last unify is N-1. Before we unify two N/2 sub assortments into one, we need to sort them. It took 2T(N/2). We have TNN-12TN/2 1 One element is already sorted. T10 2 TN/2N/2-12TN/4 3 We use substitution technique to yield 5 TNN-1N-24TN/4 4 N-1N-2N-48TN/8 … N-1N-2N-2K-12K T N/2k 5 If k approach infinity, T(N/2K) approaches T(1). We use K=log2N replacing k in equation [5] and equation [2] replacing T(1) yields

T (N)=Nlog2N-N+1 [6]

Thus T= O(Nlog2N) The best case for unify sort occurs when we unify two sub assortments into one. The last element of one sub assortment is smaller than the first element of the other assortment.

Let N=8, we have assortment {1,2,3,4,7,8,9,10}. From figure 2.a, it takes 2 comparisons to unify {1,2} and {3,4} into one. It is the same case with merging {7,8} and {9,10} into one.

From figure 2.b, we can see after 4 comparisons, element 1,2,3,4 are in the right place. The last comparison occurs when i=4, j=1. Then 7,8,9,10 are copied to the right place directly. # of comparisons is only 4(half of the assortment size)

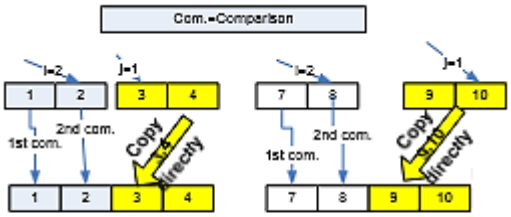


Figure 2.a Example of best Case of Merge Sort

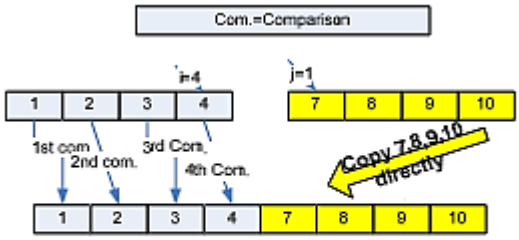


Figure 2.b Example of best Case of Merge Sort

Let T(N)=# of comparison of unify sort n assortment element. In the worst case, # of comparison of last unify is N/2. We have T(N)=N/2-sT(N/2) [1]

One element is already sorted. T(1)=0 [2]

T(N/2)=N/4+2T(N/4) [3] Equation [3] replacing T(N/2) in equation [1] yields T(N)=N/2+N/2+4T(N/4) [4]

T (N) =kN/2+2kT(N/2k) [5] Equation [5] was proved through mathematical induction but not listed here. We use k=log2N replacing k in equation [5] and equation [2] replacing T(1) yields

T (N) =Nlog2N/2 [6] Thus T=O (Nlog2N)

V. EMPIRICAL EVALUATION:

The effectiveness of the unify sort procedure will be measured in CPU time which is measured using the system clock on a machine with minimal background processes running, with respect to the size of the input assortment, and compared to the selection sort procedure. The unify sort procedure will be run with the hodgepodge size parameter set to: 10k, 20k, 30k, 40k, 50k and 60k over a range of varying-size assortments. To ensure reproducibility, all datasets and procedures used in this evaluation can be found at "http://cs.fit.edu/~pkc/pub/classes/writing/httpdJan24 .log.zip". The data sets used are synthetic data sets of varying-length groups with random numbers. The tests were run on PC running Windows XP and the following specifications: Intel Core 2 Duo CPU E8400 at 3.00 GHz with 2 GB of RAM. Procedures are run in Java. 5.1 Procedures The procedure is as follows: 1 Store 60,000 records in an collection 2 Choose 10,000 records 3 Sort records using unify sort and insertion sort procedure 4 Record CPU time 5 Increment Assortment size by 10,000 each time until reach 60,000, repeat 3-5 5.2 Results and Analysis

Figure 5 shows Unify Sort procedure is significantly faster than Insertion Sort procedure for great size of assortment. Unify sort is 24 to 241 times faster than Insertion Sort (using N values of 10,000 and 60,000 respectively).

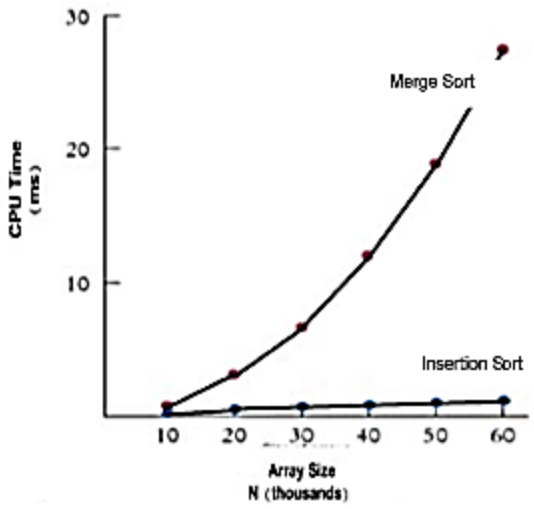


Figure 5. Merge Sort and Insertion Sort comparison

Table 1 shows Unify Sort is slightly faster than Insertion Sort when assortment size N (3000 - 7000) is small. This is because Unify Sort has too many recursive calls and temporary assortment allocation.

Table 1. CPU Time of Merge Sort and Insertion Sort

Data Set	No. of instances	Total tesing time (seconds)	
		Merge Sort	Insertion Sort
dataset1	10000	0.125	12.062
dataset2	15000	0.203	28.093
dataset3	20000	0.281	49.312
dataset4	25000	0.343	76.781
dataset5	35000	0.781	146.4
dataset6	3000	0.031	1.046
dataset7	4000	0.046	1.906
dataset8	5000	0.062	2.984
dataset9	6000	0.078	4.281
dataset10	7000	0.094	5.75

By passing the paired t-test using data in table 1, we found that difference between unify and insertion sort is statistically significant with 95% confident. ($t=2.26$, $d.f.=9$, $p<0.05$)

VI. CONCLUSIONS:

In this paper we introduced Unify Sort procedure, a O(NlongN) time and accurate sorting procedure. Unify sort uses a divide-

and-get the better of method recursively sorts the rudiments of a list while Bubble, Insertion and Selection have a quadratic time complexity that limit its use to small number of rudiments. Unify sort uses divide-and-conquer to speed up the sorting. Our theoretical and empirical analysis showed that Unify sort has a O(NlogN) time density. Unify Sort's efficiency was compared with Insertion sort which is better than Bubble and Selection Sort. Unify sort is slightly faster than insertion sort when N is small but is much faster as N grows.

One of the limitations is the procedure must copy the result placed into Result list back into m list(m list return value of unify sort function each call) on each call of unify . An alternate to this copying is to junior a new field of information with each element in m. This field will be used to link the keys and any accompanying information together in a sorted list (a key and its related information is called a record). Then the merging of the sorted lists proceeds by changing the link values; no records need to be moved at all. A field which contains only a link will generally be smaller than an entire record so less space will also be used.

REFERENCES:

- Colleges Books
- News Papers
- Magazine's
- Internet