Merge Sort Algorithm

Akash Kumar (16193) ; Akshay Dutt (16195) & Gautam Saini (16211)

Department of ECE Dronacharya college Of Engineering Khentawas ,

Farrukh Nagar-123506 Gurgaon, Haryana

ABSTRACT

Given an array with n elements, we want to rearrange them in ascending order. Sorting algorithms such as the Bubble, Insertion and Selection Sort all have a quadratic time complexity that limits their use when the number of elements is very big. In this paper, we introduce Merge Sort, a divide-and overcome algorithm to sort an N element array. We evaluate the O(NlogN)time complexity of merge sort theoretically and empirically. Our results show a large improvement in efficiency over other algorithms.

INTRODUCTION

Search engine is basically using sorting algorithm. When you search some key word online, the feedback 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 algorithms such as Bubble, Selection and Insertion Sort limits their performance when array size increases. In this paper we introduce Merge Sort which is able to rearrange elements of a list in ascending order. Merge sort works as a divide-and-conquer algorithm. It recursively divide the list into two halves until one element left, and merge the already sorted two halves into a sorted one. Our main contribution is the introduction of Merge Sort, an efficient algorithm can sort a list of array elements in O(NlogN) time. We evaluate the O(NlogN) time complexity theoretically and empirically. The next section describes some current sorting algorithms: Bubble Sort, Insertion Sort and Selection Sort. Section 3 provides a details explanation of our Merge Sort algorithm. Section 4 and 5 discusses empirical and theoretical evaluation based on efficiency. Section 6 summarizes our study and gives a conclusion. Note: Arrays we mentioned in this article have the size of N.

RELATED WORK

Selection sort [1] works as follows: At each iteration, 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 elements in sorted region will increase by each repetition. Repeat this on the rest of the unsorted region until it is exhausted. 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 iteration, 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 elements in sorted region will increase by 1 each iteration. 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 insertion sort and 40% slower than selection sort which shows that Insertion is the fastest among the three. We will evaluate insertion sort compared with merge sort in empirical evaluation. Bubble sort works as follows: keep passing through the list, exchanging adjacent element, if the list is out of order; when no exchanges are required on some pass, the list is sorted. In Bubble sort, Selection sort and Insertion sort, the O (N2) time complexity limits the performance when N gets very big. We will introduce a "divide and conquer" algorithm to lower the time complexity.

APPROACH

Merge sort uses a divide-and-conquer approach:
1) Divide the array repeatedly into two halves
2) Stop dividing when there is single element left. By fact, single element is already sorted.
3) Merges two already sorted sub arrays into one.

Pseudo Code:

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

A[p] is the beginning element and A[r] is the ending element

Output:

Array 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, and r)
```

Figure 1. The merge sort algorithm (Part 1)

```
MERGE (A, p, q, and 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 doL [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 controls when to stop dividing – when there is single element left. Line 2-4 divides array A [p…r] into two halves. Line 3', by fact, 2, 1, 4, and 3 are sorted element, so we stop dividing. Line 5 merge the sorted elements into an array. In figure 2, Line 6-7, N1, N2 calculate numbers of elements of the 1st and 2nd half. Line 8, two blank arrays L and Rare 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, pointer i, j is pointing to the first elements of L and R by default (See Figure 3, a); Figure 4, c)). Line 17, after k times comparison (Figure 3, 2 times; Figure 4, 4 times), the array is sorted in ascending order. Line 18-22, compare the elements at which i and j is "pointing". Append the smaller one to array A [p…r]. (Figure 3, a) Figure 4 a)). After k time's comparison, we will have k elements sorted.

Finally, we will have array A [p…r] sorted. (Figure 3, 4)

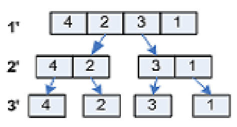

Figure 1-Divide

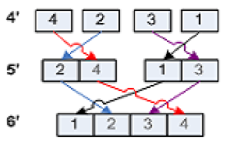

Figure 2- Merge and sort

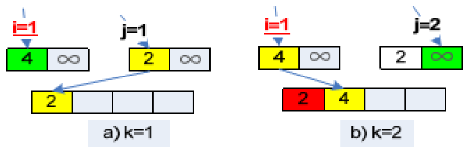

Figure 3 Merge Example

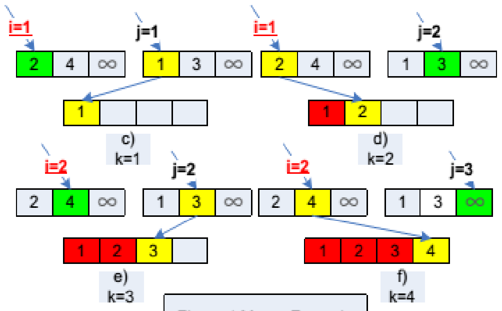

Figure 4 Merge Example

THEORETICAL EVALUATION

Comparison between two array elements is the key operation of bubble sort and merge sort. Because before we sort the array we need to compare between array elements. The worst case for merge sort occurs when we merge two sub arrays into one if the biggest and the second biggest elements are in two separated sub array. Let N=8, we have array {1, 3, 2, 9, 5, 7, 6, 8,}. From figure 1.a, element 3(second biggest element), 9(biggest element) are in separated sub array. # of comparisons is 3 when {1, 3} and {2, 9} were merged into one array. It's the same case merge {5, 7} and {6, 8} into one array. From figure 1.b, 9, 8 are in separated sub array, 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.

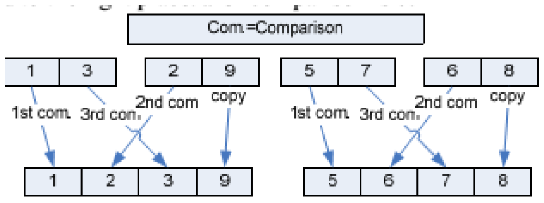

Figure 1.a Example of worst Case of Merge Sort

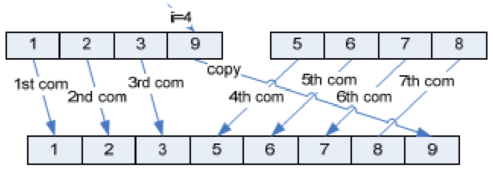

Figure 1.b Example of worst Case of Merge Sort

Let T (N) =# of comparison of merge sort n array element. In the worst case, # of comparison of last merge is N-1. Before we merge two N/2 sub arrays into one, we need to sort them. It took 2T (N/2). We have

$T(N)=N-1+2T(N/2)$ [1]

One element is already sorted.

$T(1)=0$ [2]

$T(N/2)=N/2-1+2T(N/4)$ [3]

We use substitution technique to yield [5]

$T(N)=N-1+N-2+4T(N/4)$ [4]

$=N-1+N-2+N-4+8T(N/8)$

…

$=N-1+N-2+N-2^{K-1}+2^{K}T(N/2^{k})$ [5]

If k approach infinity, $T(N/2^{K})$ approaches T(1).

We use $K=\log_{2}^{N}$ replacing k in equation [5] and equation [2] replacing T(1) yields

$T(N)=N\log 2N-N+1$ [6]

Thus $T= O(N\log 2N)$

the best case for merge sort occurs when we merge two sub arrays into one. The last element of one sub array is smaller than the first element of the other array. Let N=8, we have array {1, 2, 3, 4, 7, 8, 9, 10}. From figure

2.a, it takes 2 comparisons to merge {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 array size)

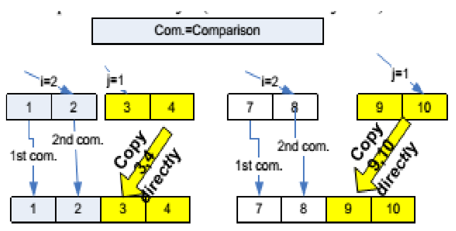

Figure 2.a Example of best Case of Merge Sort

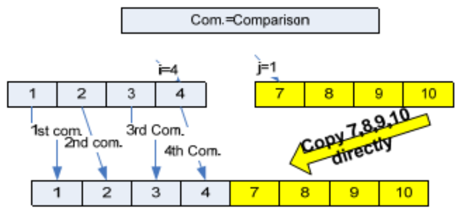

Figure 2.b Example of best Case of Merge Sort

T(N)=# of comparison of merge sort n array element. In the worst case, # of comparison of last merge 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]

$T(N)=kN/2+2^kT(N/2^k)$ [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

$T(N)=N\log 2N/2$ [6]
Thus $T=O(N\log 2N)$

EMPERICAL EVALUATION

The efficiency of the merge sort algorithm 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 array, and compared to the selection sort algorithm. The merge sort algorithm will be run with the array size parameter set to: 10k, 20k, 30k, 40k, 50k and 60k over a range of varying-size arrays. To ensure reproducibility, all datasets and algorithms used in this evaluation can be found at "http://cs.fit.edu/~plc./pub/classes/writing/httpdJan24.log.zip". The data sets used are synthetic data sets of varying-length arrays 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. Algorithms are run in Java.

PROCEDURES

the procedure is as follows:

1 Store 60,000 records in an array

2 Choose 10,000 records

3 Sort records using merge sort and insertion sort algorithm

4 Record CPU time

5 Increment Array size by 10,000 each time until reach 60,000, repeat 3-5

RESULTS AND ANALYSIS

Figure 5 shows Merge Sort algorithm is expressively faster than Insertion Sort algorithm for great size of array. Merge sort is 24 to 241 times faster than Insertion Sort (using N values of 10,000 and 60,000 respectively)

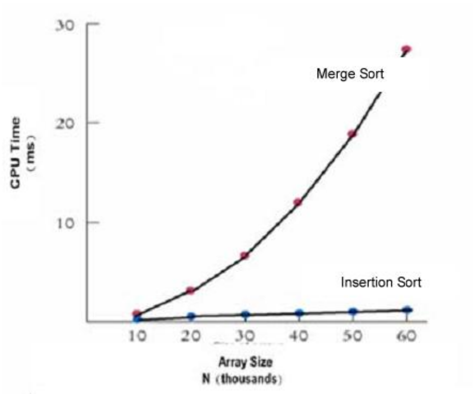

Figure 5. Merge Sort and Insertion Sort comparison

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

Table 1. CPU Time of Merge Sort and Insertion Sort

1.

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

CONCLUSION

In this paper we introduced Merge Sort algorithm, an O (N logN) time and accurate sorting algorithm. Merge sort uses a divide and-conquer method recursively sorts the elements of a list while Bubble, Insertion and Selection have a quadratic time complexity that limit its use to small number of elements. Merge sort uses divide-and-conquer to speed up the sorting. Our theoretical and experimental analysis showed that Merge sort has an O (NlogN)time complexity. Merge Sort's efficiency was compared with Insertion sort which is better than Bubble and Selection Sort. Merge sort is slightly faster than insertion sort One of the limitations is the algorithm must copy the result placed into Result list back into m list (m list return value of merge sort function each call) on each call of merge. An alternative to this copying is to assistant a new field of information with each element in m. This field will be used to link the keys and any associated 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

Sedge wick, Algorithms in C++, pp.96-98, 102, ISBN 0-201-51059-6, Addison-Wesley, 1992

2. Sedge wick, Algorithms in C++, pp.98-100, ISBN 0-201-51059-6, Addison-Wesley, 1992
3. sedge wick, Algorithms in C++, pp.100-104, ISBN 0-201-51059-6 ,Addison-Wesley , 1992
4. Owen Astrakhan, Bubble Sort: An Archaeological Algorithmic Analysis, SIGCSE 2003,
5. Singh, J., & Singh, R. (2014). Merge Sort Algorithm. *International Journal of Research*, *1*(10), 1203-1207.
6. Qin, S. Merge Sort Algorithm. *Department of Computer Sciences, Florida Institute of Technology, Melbourne, FL, 32901*.
7. Cole, R. (1988). Parallel merge sort. *SIAM Journal on Computing*, *17*(4), 770-785.
8. Horstmann, C. S. (2002). *Big Java* (Vol. 3). Wiley.
9. Dittrich, J. P., Seeger, B., Taylor, D. S., & Widmayer, P. (2002, August). Progressive merge join: A generic and non-blocking sort-based join algorithm. In *Proceedings of the 28th international conference on Very Large Data Bases* (pp. 299-310). VLDB Endowment.