Sorting Algorithm

Monika Tulsiram

Computer Science

Indiana State University

mtulsiam@sycamores.indstate.edu

December 3, 2014

Abstract

In general, Sorting means rearrangement of data according to a defined pattern. The task of sorting algorithm is to transform the original unsorted sequence to the sorted sequence. While there are a large number of sorting algorithms, in practical implementations a few algorithms predominate. The most classical approach is comparison based sorting which include heap sort, quick sort, merge sort, etc.

In this paper, we study about two sorting algorithms, i.e., quick sort and merge sort, how it is implemented, its algorithm, examples. We also look at the efficiency and complexity of these algorithms in detail with comparison to other sorts. This is an attempt to help to understand how some of the most famous sorting algorithms work.

1 Introduction

In computer science, sorting is one of the most extensively researched subjects because of the need to speed up the operation on thousands or millions of records during a search operation. When retrieving data, which happens often in computer science, it's important for the data to be sorted in some way. It allows for faster more efficient retrieval. For example, think about how much chaos it'd be if the songs on your iPod weren't sorted. Another thing is, sorting is a very elementary problem in computer science, somewhat like addition and subtraction, so it's a good introduction to algorithms. And it's a great vehicle for teaching about complexity and classification of algorithms. Sorted data has good properties that allow algorithms to work on it quickly. For instance, searching random data takes $O(n)$ time, whereas searching a sorted list takes $O(log(n))$ time, an enormous improvement. Sorted lists can also be easily merged with each other while maintaining the sorted property.

Sorting algorithms are usually judged by their efficiency. Here, efficiency refers to the algorithmic efficiency as the size of the input grows large and is generally based on the number of elements to sort. Many algorithms that have the same efficiency do not have the same speed on the same input. First, algorithms must be judged based on their average case, best case, and worst case efficiency. Hence, efficient sorting is important for optimizing the use of other algorithms (such as

search and merge algorithms) which require input data to be in sorted lists; it is also often useful for canonicalizing data and for producing human-readable output. More formally, the output must satisfy two conditions:

1. The output is in non-decreasing order (each element is no smaller than the previous element according to the desired total order)
2. The output is a permutation (reordering) of the input.

Quick Sort: Even if you know just a little about sorting algorithms, then chances are you know Quick Sort. And of course an algorithm can't just call itself Quick Sort without having some merit to it. Quick Sort is one of the faster sort algorithms. It is in-place sorting since it uses only a small auxiliary stack and doesn't use a lot of memory. Quick Sort is best for most sets of data. Quicksort is a Divide and Conquer algorithm. It first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays. The steps are:

1. Pick an element, called a pivot, from the array.
2. Reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
3. Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

Merge Sort: Merge sort is the first of the faster and more complex sorting algorithms to be discussed. It runs on the same order as the other fast sorting algorithms. It is much more stable, but has drawbacks in memory usage. Since Merge Sort doesn't sort in place, it has to allocate memory for another list of the same size as the input. Merge sort uses recursion to simplify sorting down to a minimum number of comparisons, then merge together with other comparisons. In this way, Merge Sort is a divide and conquer algorithm. Most implementations produce a stable sort, which means that the implementation preserves the input order of equal elements in the sorted output. Conceptually, a merge sort works as follows:

1. Divide the unsorted list into n sub-lists, each containing 1 element (a list of 1 element is considered sorted).
2. Repeatedly merge sub-lists to produce new sorted sub-lists until there is only 1 sub-list remaining. This will be the sorted list.

1.1 Our Results

Quick sort is typically faster than merge sort when the data is stored in memory. However, when the data set is huge and is stored on external devices such as a hard drive, merge sort is the clear winner in terms of speed. It minimizes the expensive reads of the external drive and also lends itself well to parallel computing.

2 History

Hollerith's sorting machine was developed in 1901-1904 which used radix sort. His later machines mechanized the card-feeding process, added numbers, and sorted cards, in addition to merely counting data. Later sorting problem emerged as a great subject of research. Optimal algorithms have been known since the mid-20th century such as quick sort, heap sort, merge sort, etc. The Quicksort algorithm developed by Hoare, is one of the most efficient internal sorting algorithms and is the method of choice for many applications. The algorithm is easy to implement, works very well for different types of input data, and is known to use fewer resources than any other sorting algorithm. Merge sort is a divide and conquer algorithm that was invented by John von Neumann in 1945. A detailed description and analysis of bottom-up merge sort appeared in a report by Goldstine and Neumann as early as 1948. There are many useful new algorithms are still being invented, with the now widely used Tim sort dating to 2002, and the library sort being first published in 2006.

3 Algorithm

3.1 Quick Sort

```
Quicksort(A, p, q)
  if p < q
      then r = partition(A, p, q)
      Quicksort(A, p, r-1)
      Quicksort(A, r+1, q)
  Partition(A, p, q)
  x = A[p]
  i = p
  for j = p+1 to q
  if A[j] x
       then i = i+1
       Swap A[i] with A[j]
   Swap A[p] with A[i]
return i
```

3.2 Merge Sort

```
function mergesort(m)
   var list left, right, result
  if length(m) <= 1
      return m
  else
      var middle = length(m) / 2
  for each x in m up to middle - 1
      add x to left
  for each x in m at and after middle
```

```
      add x to right
  left = mergesort(left)
  right = mergesort(right)
     if last(left) <= first(right)
        append right to left
      return left
      result = merge(left, right)
   return result

function merge(left,right)
   var list result
  while length(left) > 0 and length(right) > 0
     if{first(left) <= first(right)
         append first(left) to result
         left = rest(left)
     else
         append first(right) to result
         right = rest(right)
     if length(left) > 0
         append rest(left) to result
     if length(right) > 0
         append rest(right) to result
   return result
```

4 Implementation

4.1 Quick Sort

Our goal is to choose a pivot and partition all remaining elements based on whether they are smaller than or greater than the pivot.

STEP 1: Choosing the pivot
Depending on pivot, the algorithm may run very fast, or in quadric time. Some fixed element: e.g. the first, the last, the one in the middle. This is a bad choice - the pivot may turn to be the smallest or the largest element, then one of the partitions will be empty. Randomly chosen (by random generator) - still a bad choice. The median of the array (if the array has N numbers, the median is the [N/2] largest number. This is difficult to compute - increases the complexity. The median-of-three choice: take the first, the last and the middle element. Choose the median of these three elements.
Example: 8, 3, 25, 6, 10, 17, 1, 2, 18, 5
The first element is 8, the middle is 10, last is 5. The three elements are sorted: [5, 8, 10] and 8 is the median. After finding the pivot, the array will look like this.
5, 3, 25, 6, 8, 17, 1, 2, 18, 10

STEP 2: Partitioning

1. While partitioning, we move larger elements to the right side and smaller elements to the left side. Two pointers (L, R) are used to scan the elements from left to right and from right to left. Let's do the partitioning on the above example. L=5 and R=10
While L is to the left of R, we move L right, skipping all the elements less than the pivot. If an element is found greater than the pivot, L stops. While R is to the right of L, we move R left, skipping all the elements greater than the pivot. If an element is found less than the pivot, R stops. When both L and R have stopped, the elements are swapped. When L and R have crossed, no swap is performed, scanning stops, and the element pointed to by L is swapped with the pivot. In the example, the first swapping will be between 25 and 2, the second between 18 and 1.
2. Restore the pivot.
After restoring the pivot we obtain the following partitioning into three groups:
[5, 3, 2, 6, 1] [8] [18, 25, 17, 10]

STEP 3: Recursively quicksort the left and the right parts.

4.2 Merge Sort

Our goal is to divide the array and merge it.

STEP 1: Divide
Divide the array into two sub-arrays. The array is recursively divided into two halves till the array size becomes 1.

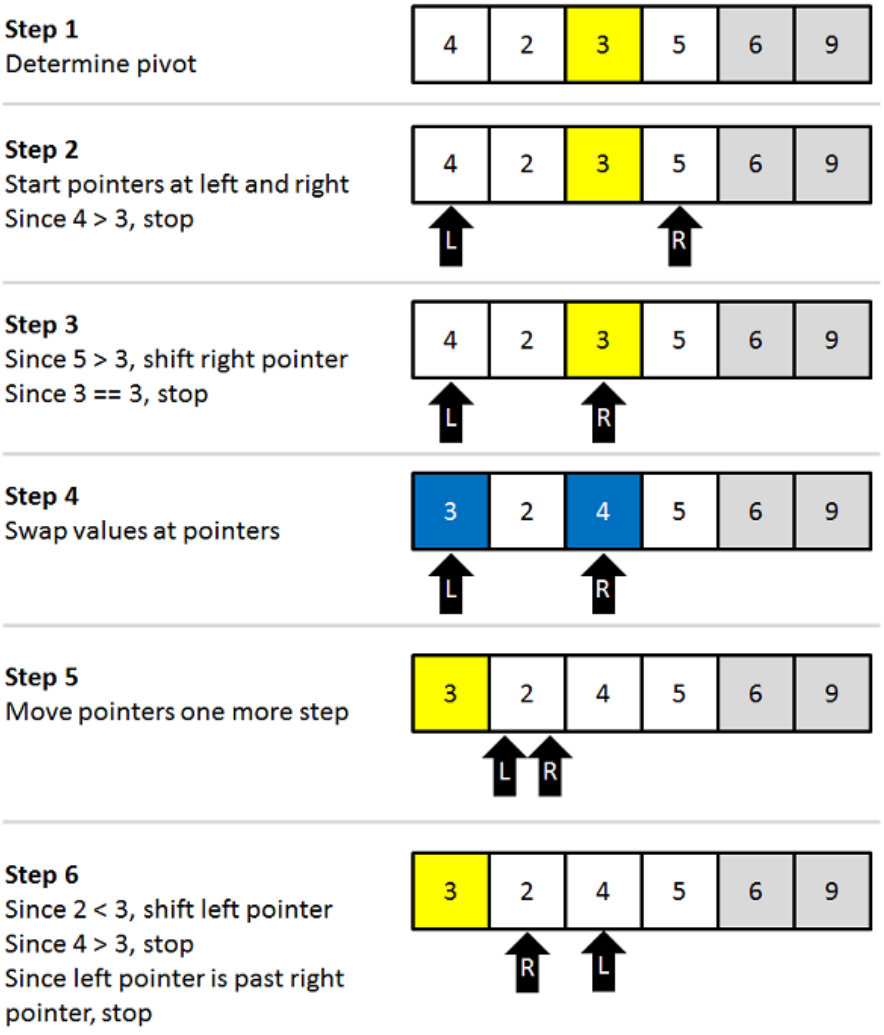
STEP 2: Conquer
Conquer by recursively sorting the two sub-arrays.

STEP 3: Combine
Repeatedly merge divided units to produce new sub-lists until there is only 1 sub-list remaining. This will be the sorted list at the end.

5 Example

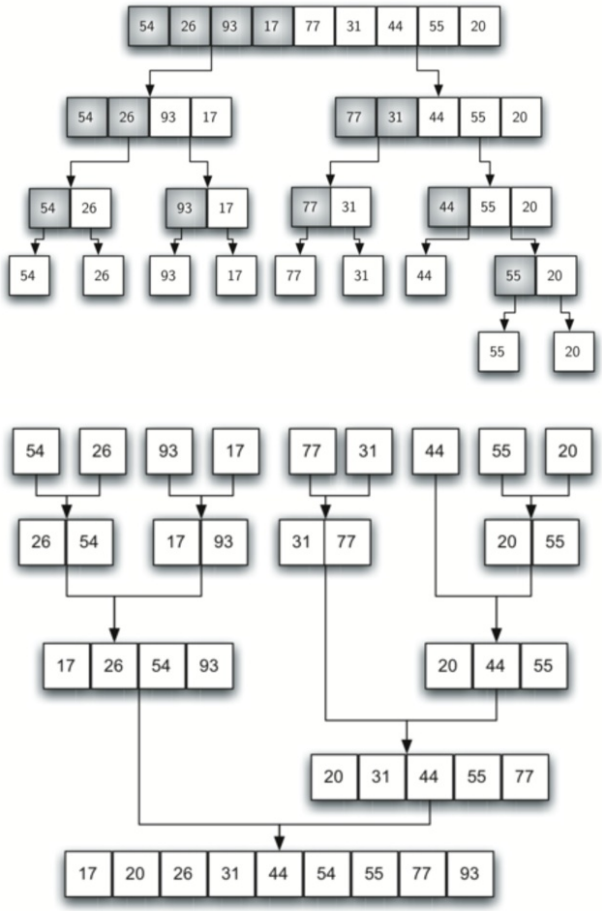
5.1 Quick Sort

Figure 1: Quick Sort Example



5.2 Merge Sort

Figure 2: Merge Sort Example



6 Complexity Analysis

6.1 Complexity of Quick Sort

Worst-case: $O(|N|^2)$ This happens when the pivot is the smallest or the largest element. Then one of the partitions is empty, and we repeat recursively the procedure for $N-1$ elements.

Best-Case: $O(NlogN)$ The best case is when the pivot is the median of the array, and then the left and right part will have same size. There are $logN$ partitions, and to obtain each partitions we do N comparisions(and not more than $N/2$ swaps). Hence the complexity is $O(NlogN)$.

Average Case: $O(NlogN)$

6.2 Complexity of Merge Sort

Time complexity of Merge Sort is $O(NlogN)$ in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.

7 Applications

There are a number of applications of sorting.

Searching: Binary search lets you test whether an item is in a dictionary in $O(logn)$ time. Search processing is perhaps the single most important application of sorting.

Closest Pair: Once the numbers are sorted, the closest pair will be next to each other in sorted order.

Element Uniqueness: Sorting helps to check all the adjacent pairs by linear scan. This is a special case of closest pair.

Median and Selection: Once the keys are placed in sorted order in an array, from the number of instances of k, the kth largest can be found in constant time by simply looking the kth position of the array.

7.1 Applications of quick Sort

1. Quick Sort is ideal for use with large data sets and/or when memory is constrained.
2. Generally, commercial applications use quick sort because it is speedy and does not require much additional memory.
3. Used in In-place Sorting.

7.2 Applications of Merge Sort

1. Merge Sort is useful for sorting linked lists in $O(NLogN)$ time.
2. Inversion Count Problem
3. Used in External Sorting

8 Conclusion

Quick sort is empirically faster than merge sort for most random data sets if you choose the pivot well. This is mainly due to its lower memory consumption which usually affects time performance as well. It is in place and doesn.t require any extra space other than the elements to be sorted. That is why most people consider it to be "better". However it is important to understand that the worst case of quick sort is order n^2 whereas merge sort has a worst case runtime of order $nlogn$. Therefore, quick sort is usually faster but sometimes (very rarely given a good implementation) it is a worse.

Acknowledgements

This paper is written with help from different articles published over the internet. The data presented, the statements made, and the views expressed are soley the responsibility of the author.

References

[1] Robert Sedgewick and Kevin Wayne
Algorithms, 4th Edition by Robert Sedgewick and Kevin Wayne, 2011.

[2] Charles E. Leiserson, Clifford Stein, Thomas H. Cormen, and Ronald Rivest
Introduction to Algorithms, 1990.

[3] Stack Overflow
`http://stackoverflow.com/`

[4] Dreamcode Algorithm Reference
`http://www.dreamincode.net/forums/topic/100921-fundamental-sort-algorithm-reference/`

[5] Internet Merge Sort
`http://en.wikipedia.org/wiki/Merge_sort`

[6] Kent Edu
`http://www.personal.kent.edu/~rmuhamma/Algorithms/algorithm.html`

[7] Sorting Algorithm Wiki
`http://en.wikipedia.org/wiki/Sorting_algorithm`