A Comparison of Sorts of 'Sorts'

by: Patrick Korth

9/8/2017

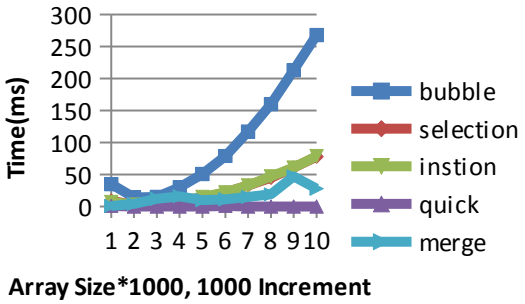
Data can be organized in a multitude of ways. In this specific study the arrangements are arrays of Java wrapper class 'Integers'. The arrays are tested with four different arrangements to see how each sort performs on each one. The arrays are to be sorted from the smallest element to the largest. The first type of array is an array already arranged in increasing order, the second is an array arranged in decreasing order, the third is an array of randomly generated numbers, and finally the fourth is an array generated to test the worst case scenario of the quickSort method. Two series of test arrays were used. One consisting of smaller arrays: 1000 through 10,000, incremented by 1000 (10 arrays total) and another series of larger arrays: 10,000 through 100,000 incremented by 10,000. The random array was generated by first generating an increasing array and then looping through it n times, each time selecting a random element, then performing a series of exlusive-or operations with the ith iteration to randomize it. The worst case array was generated by creating an array of all one number. Why this is the worst case will be discussed in detail later. The arrays are timed in milliseconds and integer-rounded to the nearest whole millisecond.

The first method tested on each array was the bubbleSort method. This method loops through the array, moving linearly from one element to the next, comparing it to the element in the next index. If The element in the next index is smaller, the elements exchange positions.

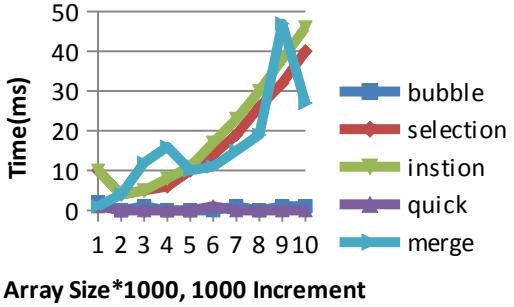
for (loop through the array until all elements are in order);
    for (loop through the array swapping elements);
        if(the ith element is larger then the (i+1)th element, swap their positions);
Do this until the array does not require a single swap as it's passed through.

In a worst case scenario bubble sort has an order of magnitude of O(n^2). A bubble sort will quickly approach this worst case scenario as the size of the array grows, assuming an array that is not already ordered (see figure 1a.). The only time a bubble sort outperforms other sorts studied in the paper is when the list is sorted in increasing order or the list consists of all the same element (see figure 1b). This happens because no swaps need to be made, it loops the list once, seeing that the list is in order and breaks. There are very few real world applications of a bubble sort it would seem. The only time a bubble sort would outperform the other sort methods studied is for a nearly sorted list; perhaps if it were known that only a few elements out of place by a few indices it could be useful. I could also be used to simply confirm that it is sorted.

1a. Random Array (small)



1b. Increasing Array (small)



The next two sort methods studied in this paper will be discussed together as the outcomes are virtually identical. Both have worst case scenarios of O(n^2). Selection sort and insertion sort both work by moving through the array looking for the smallest value. Selection sort does this by logically breaking the array into two parts, sorted and not sorted. It moves through the not sorted portion of the array, searching for the smallest value and, once found, placing that value at the front of the already sorted sub-array.

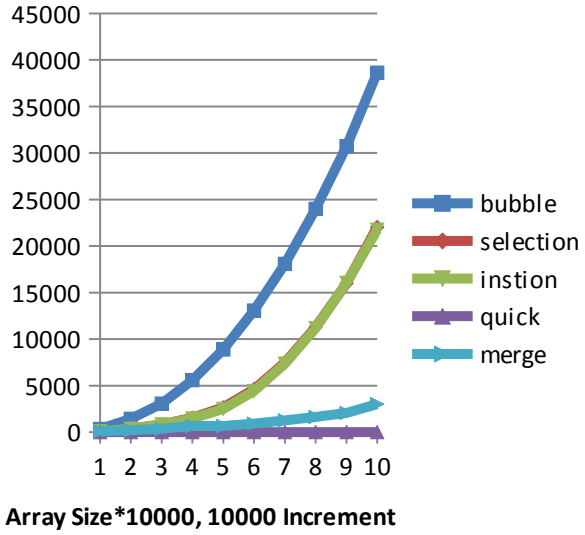
```
for (loop through the array until the sub-array size is the same size as the whole array)
    for (loop through unsorted portion of array until smallest element is found)
        if (if the ith element of the unsorted portion of the array is smaller than)
            set the variable tracking the position of the front of the of the unsorted array to
            the ith value;
    if (if the ith value is not the last value in the unsorted array)
        swap the ith value in the unsorted array into the front position of the sorted sub-array;
```

An insertion sort works very similarly. Insertion sort chooses the first element of the array as its sorted sub-array then moves onto the next element of the unsorted portion of the array. It takes this next unsorted element and compares it to the sorted sub-array. When it finds the spot in the sub-array where the previous element is smaller than the element being sorted and the next element is larger, it ‘inserts’ the value into this location in the sub-array. This continues until there are no values left in the unsorted array.

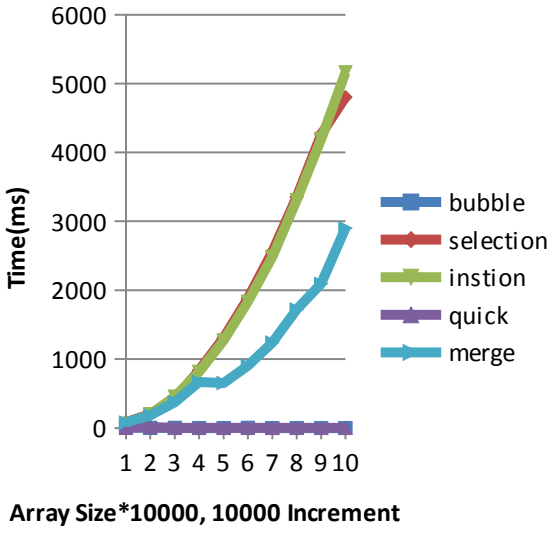
```
For (loop through the array adding 1 to the index counter until the entire array has  been parsed)
    create variable to hold new position;
    while (while the next element in the unsorted sub-array is larger than the previous element,
        increment the new position variable)
        if (if there are still elements left to be sorted)
            break;
    if (if the new position has not been incremented to the end of the array, e.g. sort is not done)
        for (decrement position until new position is found)
            make room for insertion;
        insert element pulled from unsorted sub-array into new position found in sorted sub-
        array;
```

Insertion sort and selection sort both perform virtually identically on smaller arrays as well as larger. They both take exponentially more time to compute as the array grows but don’t seem to have any special cases. They are faster than bubblesort on random and decreasing data sets but still will eventually reach their worst case of O(n^2) (see 2a. And 2b on next page). One possible advantage to these algorithms over the ultimately faster divide and conquer algorithms (to be discussed) is their low overhead for sorting smaller data sets (<1000 or so elements) where computation time is negligible.

2a. Random Arrays (large)



2b. Increasing Arrays (large)

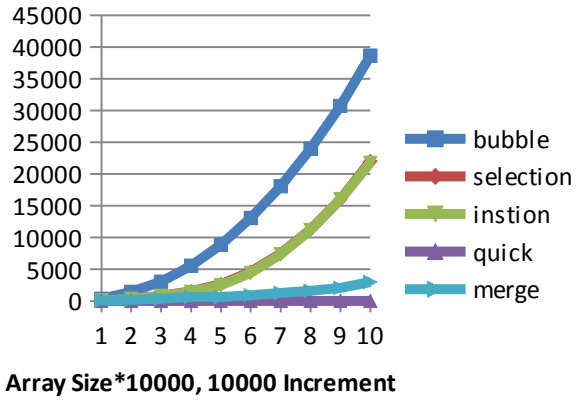


The fourth algorithm studied was the quickSort method. Quicksort is a 'divide and conquer' sorting strategy that starts by choosing a middle element from the array and then organizing the array such that all values smaller than middle element are moved to the left of the middle element and all values larger than the middle element are moved to the left. This process is the 'partitioning' of the array. The array is recursively partitioned in this way until each partition consists of only sorted elements. At this point the array is sorted.

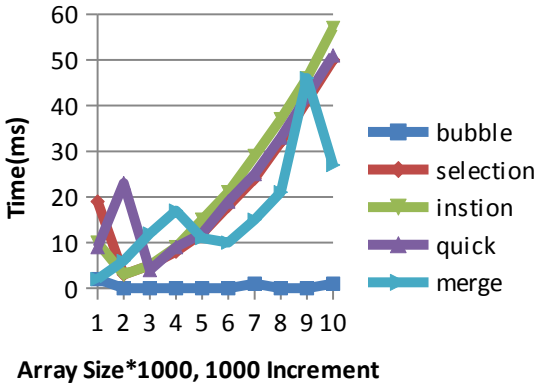
If (smallest element in partition < largest element)
    call partition method to create partition;
    quickSort(recursively call quickSort on elements to the left of partition);
    quickSort(recursively call quickSort method on elements to the right of partition);

Quicksort was the superior sort method studied in this paper. Its recursive nature generally gives it an order of magnitude of O(n log(n)) aside from few very specific cases where it can have a worst case of O(n^2). This worst case happens when all the elements of the array are identical. See 3a. and 3b. The exponential inefficiency of this worst case scenario is so extreme is actually causes a stack overflow When trying to use it with arrays that get much larger than 10,000 elements.

3a. Random Arrays (large)



3b. Quicksort Worst Case (all elements identical)

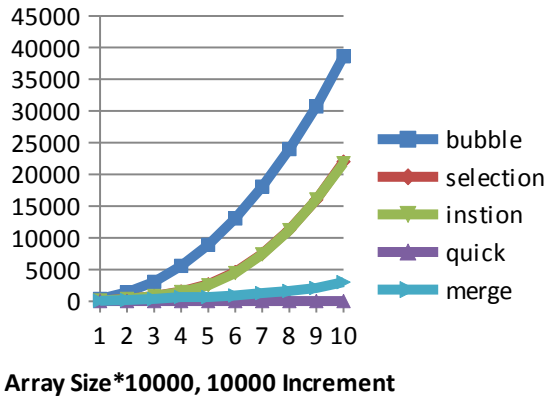


The final sort algorithm studied was the mergeSort method. Mergesort is also a divide and conquer algorithm. Unlike quicksort, mergesort begins by dividing the array into n sub-arrays (one element per sub-array). Next it compares each sub-array's first elements to the first element of the one next to it and merges the two arrays in sorted order. This occurs recursively until the all sub-arrays are gone and the full sorted array remains.

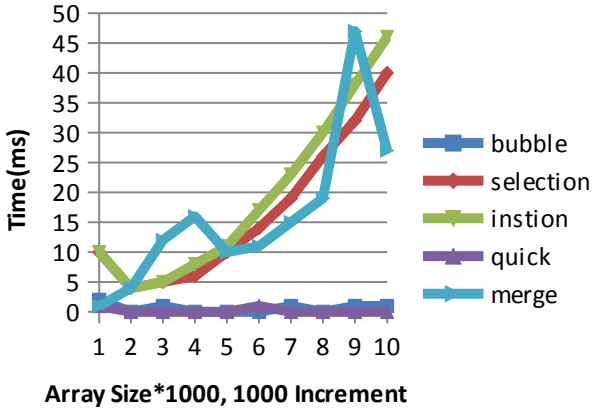
If (if smallest sub-array value < largest sub-array value)
    create a midpoint by averaging the smallest and largest values;
    mergeSort (recursively call mergeSort on all elements left of midpoint until each sub-array consist of one element);
    mergeSort (recursively call mergeSort on all elements right of midpoint consist of one element);
    merge (recursively call merge method to compare first elements of each sub-array and combine them in sorted order);

The mergesort algorithm is much faster than bubble sort, selection sort and insertion sort for most applications (random arrays that need to be sorted, figure 4a.). You can clearly see where these three less efficient (O(n^2)) algorithms exponentially slow down relative to mergesort. However, mergesort is not quite as fast as quicksort for an array of random elements (figure 4a.). One possible reason for this could be that no matter what merge sort has to divide the array down into sub-arrays of single elements no matter the ordering already there. You can see this weakness it mergesort's effectiveness on an already sorted array (figure 4b.). An interesting note regarding mergesort that has not been accounted for is the fluctuation in processing speed of different relatively similarly sized arrays are sorted. It seems that merge sort should have a very consistent curve regardless of array size but we don't see this curve smooth out until sorting much larger arrays (figure 4b. Compared to 4c).

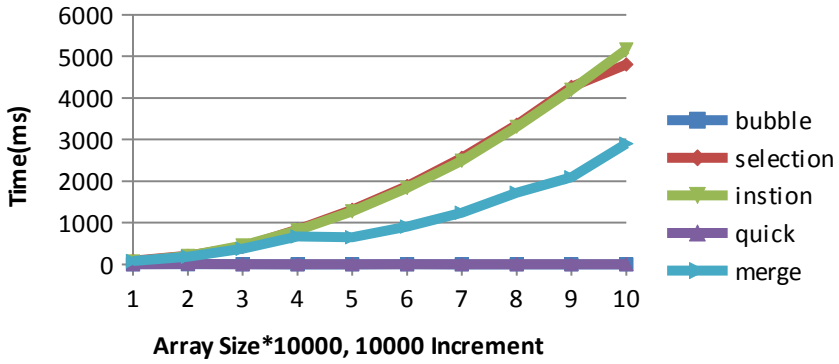
4a. Random Arrays (large)



4b. Increasing arrays (small)



4c. Increasing (large)



Between the five sorting methods discussed it seems that the bubbleSort method is the most inefficient. The only places it outperforms any of the other methods are for arrays that are already sorted or arrays that are comprised of all identical elements. The next two methods studied, selection sort and insertion sort, also come very inefficient for large arrays but not as inefficient as bubble sort. Selection sort and insertion sort could be used in very specific scenarios where much smaller arrays need to be sorted and resources are limited or multiple arrays need to be sorted simultaneously as they require very little overhead. On the other hand the divide and conquer methods must keep track of many variables simultaneously thus additional overhead is required (use of more RAM, cache, pagefile, etc). Merge sort outperforms the first three O(n^2) sorts but has the weakness of always breaking the array down into its single elements every time regardless of order. The clearly superior sorting technique out of the five studied is quicksort. Unlike mergesort, as quicksort divides the sub-arrays out it knows if the sub-array is already in order or not. If it's already sorted it will not divide it further. The only weakness of quicksort in any of the types of arrays generated is when the array consists of all identical elements. This scenario would be rare and also could be easily coded against with one pass of of the array comparing one element to the next. An interesting further study would be to look closely at merge sort and try to determine why the processing time curve is so uneven for certain smaller size arrays.

References:

- www.stackoverflow.com
- www.toptal.com/developers/sorting-algorithms/ (animated explanations of sorts, really cool!)
- www.wikipedia.org
- www.geeksforgeeks.com