Data Structure and Algorithms coursework

Joe Halloran

Part 1: Analysis of sorting algorithms

To compare the efficacy of different sorting algorithms I have generated arrays of differing lengths (starting at 10,000 all the way to 200,000, incrementing by 1000 each time). Each array contains integers, randomly generated between 1 and 50.

Each randomly generated array was copied 4 times (using the Java) `Array.clone()` method and then passed to 4 different sorting algorithms.

- Bubble sort
- Insertion sort
- Merge sort
- Quick sort

The speed by which these algorithms sorted the data was recorded using Java's `System.nanoTime()`. Results were compared to induce the efficiency of each algorithm.

Fig 1 Shows how each algorithm performed as the array length increases. It clearly shows bubble sort performs worse that insertions sort, which in turn performs worse than merge sort and quick sort.

Incidentally, it appears as if merge sort is not on the graph. This is because it is so close in performance to quick sort, that the graphic cannot render both lines separately.

For this reason, a focused comparison of merge sort and quick sort is supplied in *fig 2*. It shows that quick sort narrowly outperforms merge sort. The fluctuations in the graph relate to variations in the distribution (tending towards worse or best case scenarios) of the randomly generated arrays.

Fig 3 focuses specifically on a randomly generated array of length 100,000, again comprised of integer values 1-50. It further supports the findings outlined above.

Fig 1: Run time of various sorting algorithms

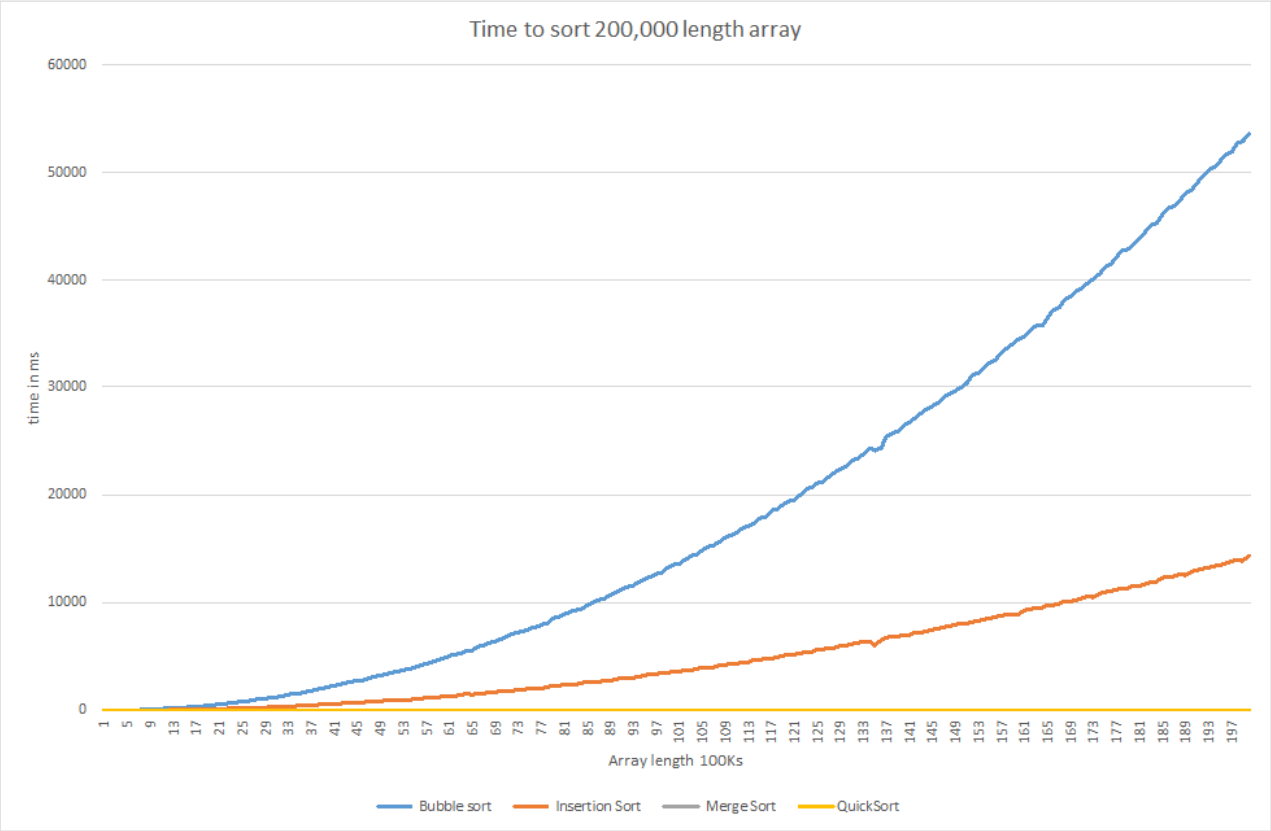


Fig 2: Run time of merge sort and quick sort algorithms only

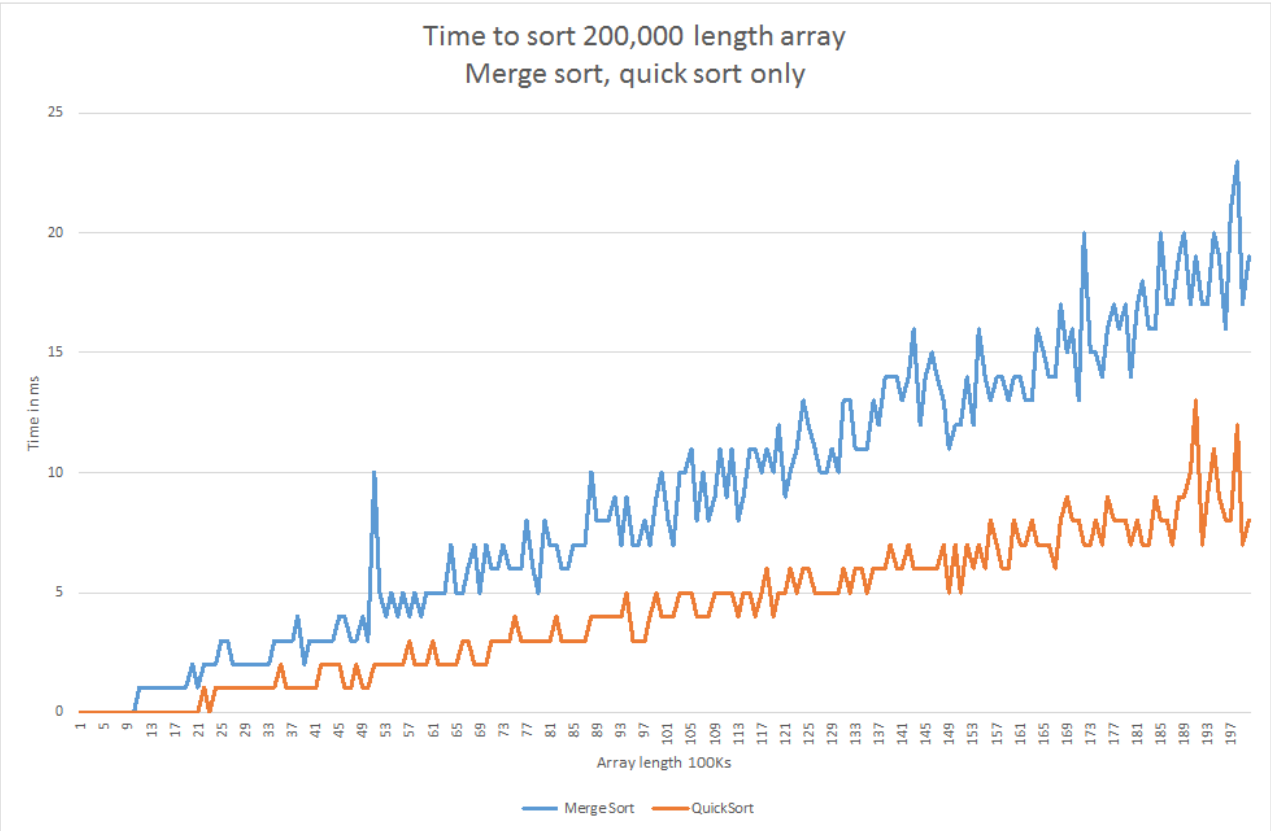
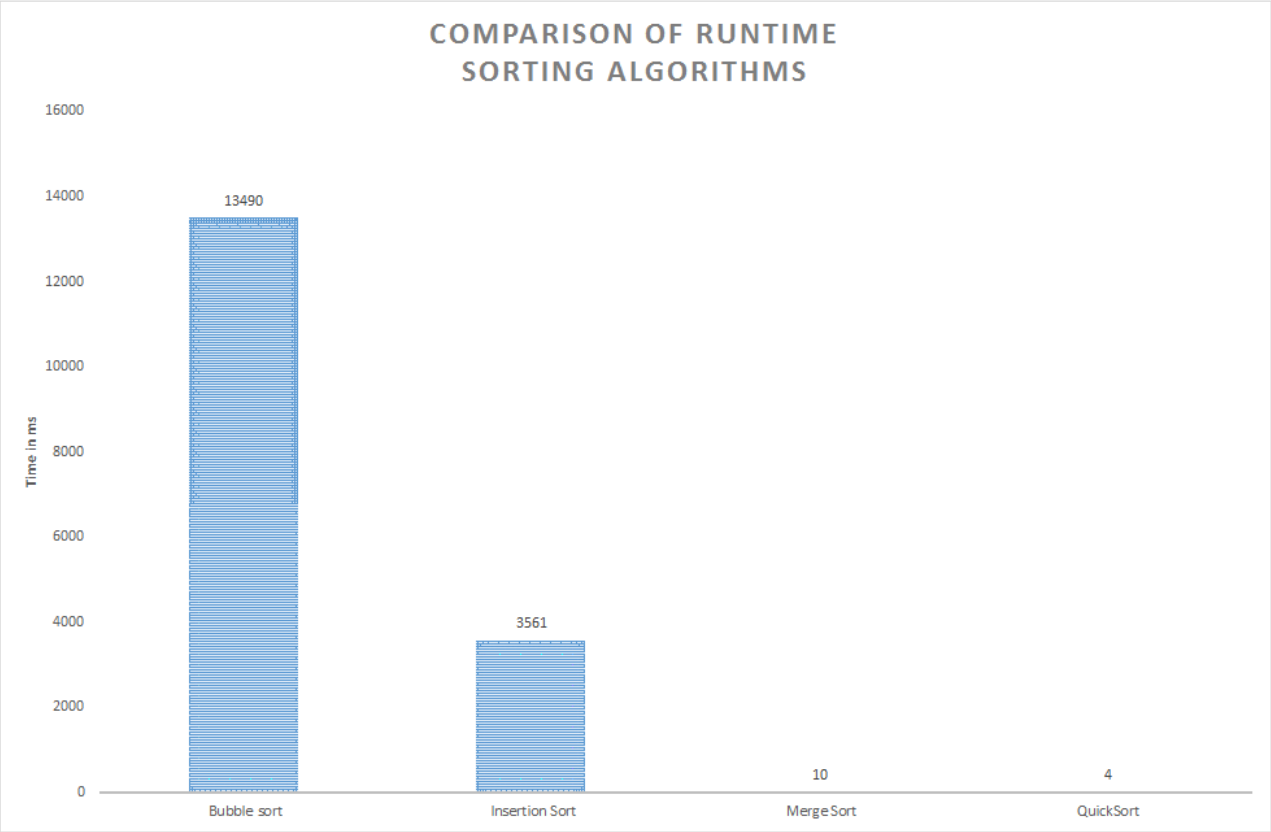


Fig 3: Comparison of runtime for various sorting algorithms



Time complexity discussion

The time taken to complete bubble sort and insertion sort grows exponentially as the array lengths increase. Both have time complexity of 0(n^2) (in the *average* case), meaning the time to execute increases at an accelerated rate, where the array length increases at a fixed rate. This makes them entirely undesirable for sorting data on any serious scale.

As observed above, merge sort and quick sort, are much faster. They have a time 0(n log(n)) (in the 'average' case). This means the rate of increase in execution time decelerates as the array length increases at a fixed rate. This is not to say it get *quicker* as the array length increases (that would be truly miraculous). It means that the rate of growth slows down, much like it does in human beings: babies grow most rapidly in the first months of their life. This is a highly desirable property for sorting algorithms, as it means the additional time cost decreases every time you add one to the array length. This means both arrays are well suited to sorting large data sets.

Part 2: Segregate even and odd numbers

Pseudo code

```
// Pseudo code
input array[]
low ← 0
high ← array.length
while low < high do
    while low < array.length and array[low] % 2 = 0 do  // While in range and not od
        low ← low + 1           // Increment
    end while
    while high >= 0 and array[high] % 2 = 1 d    // While in range and not even
        high ← high - 1         // Decrement
    end while
    if low < high:        // Only swap if odd number is lower in list than even numb
        // Swap values
        temo ← array[low]
        array[low] ← array[high]
        array[high] ← temp
end while
```

Time complexity discussion

The above algorithm has time complexity 0(n). This means the number operations required to complete the algorithm has a linear relationship to the length of the input array. This can be understood if we look in detail at the operation of the algorithm.

Take a *average case*, which we can hypothesise to be where odd and even numbers alternate in the array. The algorithm includes an incrementing `while` loop and a decrementing `while` loop. These loops work their way from the top and bottom, swapping odd numbers for even where necessary along the way, eventually meeting in the middle. At this point the first top-level while loop, `while low < high`, kicks in and terminates the algorithm with the array now sorted. This prevents the incrementer and decrementer passing each other and running needlessly through the sorted portions of the array. This process shows the work required to complete the algorithm is directly and proportionally dependant on the array length, justifying the conclusion that is has time complexity O(n).

Given a *worse case*, an array with odd numbers in the first half even numbers in the second half, the incrementer and decrementer behave in the same way (meeting in the middle and stopping). The only difference is the number of swaps that take place. As the number of swaps is dependent on the overall array length, we do not need to adapt our overall assessment that the algorithm is of time complexity O(n).

In the *best case*, a sorted array, again the incrementer and decrementer loops never pass each other and no swaps occur. So again the algorithm is O(n): it loops through the array once; its complexity has a linear relationship to the array length.

Part 3: Recursion

Part 3.1: Pizza

A recursive algorithm to find the number of slices in a pizza, given *n* cuts. Cuts are complete, i.e. they are equal length to the Pizza's diameter and must therefore pass through the centre.

Pizza pseudo code

```
// Pseudo code
pizza(n) {
    if n = 1 then // Base case
        return 2
    else
        return 2 + pizza(n-1)  // Recursion call
    end if
}
```

pizza(4)

```
pizza(4) = 2 + pizza(3)
                  \
                   pizza(3) = 2 + pizza(2)
                                    \
                                    pizza(2) = 2 + pizza(1)
                                                       \
                                                    pizza(1) = 2
                                                     /
                                    pizza(2) = 2 + 2
                                    /
                   pizza(3) = 2 + 4
                   /
        pizza(4) = 2 + 6
        /
Returns 8
```

Part 3.2: Motorbike and car parking

A recursive algorithm to find the number of possible arrangements for parking motorbikes and cars given *n* spaces, where motorbikes take 1 parking space and cars occupy 3.

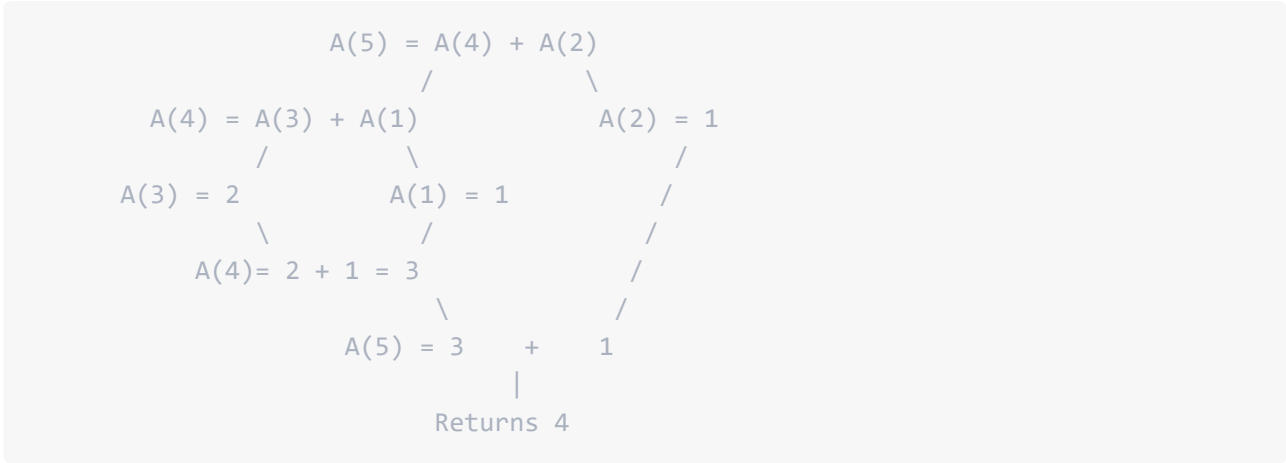
Recursive method for A(n) - pseudo code

This algorithm works on the premise that the problem scope can be reduced in two ways: (1) where the first space is taken up by a motorbike, and (2) where the first 3 spaces are taken by the car. The solution for any value of *n* is the sum of case (1) and case (2).

Therefore the recursion call take the structure `return A(n-1) + A(n-3)`, i.e. all the possibilities with a motorbike first, plus all the possibilities with a car first.

```
// Pseudo code
A(n) {
    if n = 1 or n = 2 then // Base case 1 and 2
        return 1
    else if n = 3 then      // Base case 3
        return 2
    else
        return A(n-1) + A(n-3)  // Recursion call
    end if
}
```

A(5)



All possible arrangements for A(6)

A(6) = 6

- MMMMMM
- CMMM
- MCMM
- MMCM
- MMMC
- CC

Bibliography

- Big O Cheat Sheet. Available at: http://bigocheatsheet.com/ (Accessed: 12 June 2017).
- A Gentle Introduction to Algorithm Complexity Analysis. Available at: http://discrete.gr/complexity/ (Accessed: 12 June 2017).