Data Structures and Algorithms Coursework

By Shannon Sullivan

Part I: Analysis of Sorting Algorithms

To compare the efficiency of different sorting algorithms, I have generated a large array of random numbers. The array was then "copied" using the Java `Array.clone()` method, and passed to the following four sorting algorithms:

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

To capture the time necessary to complete the sorting methods I utilized the Java `Date()` object. I applied the `getTime()` method at the start and end of each sorting method call and calculated the time elapsed.

Figure 1

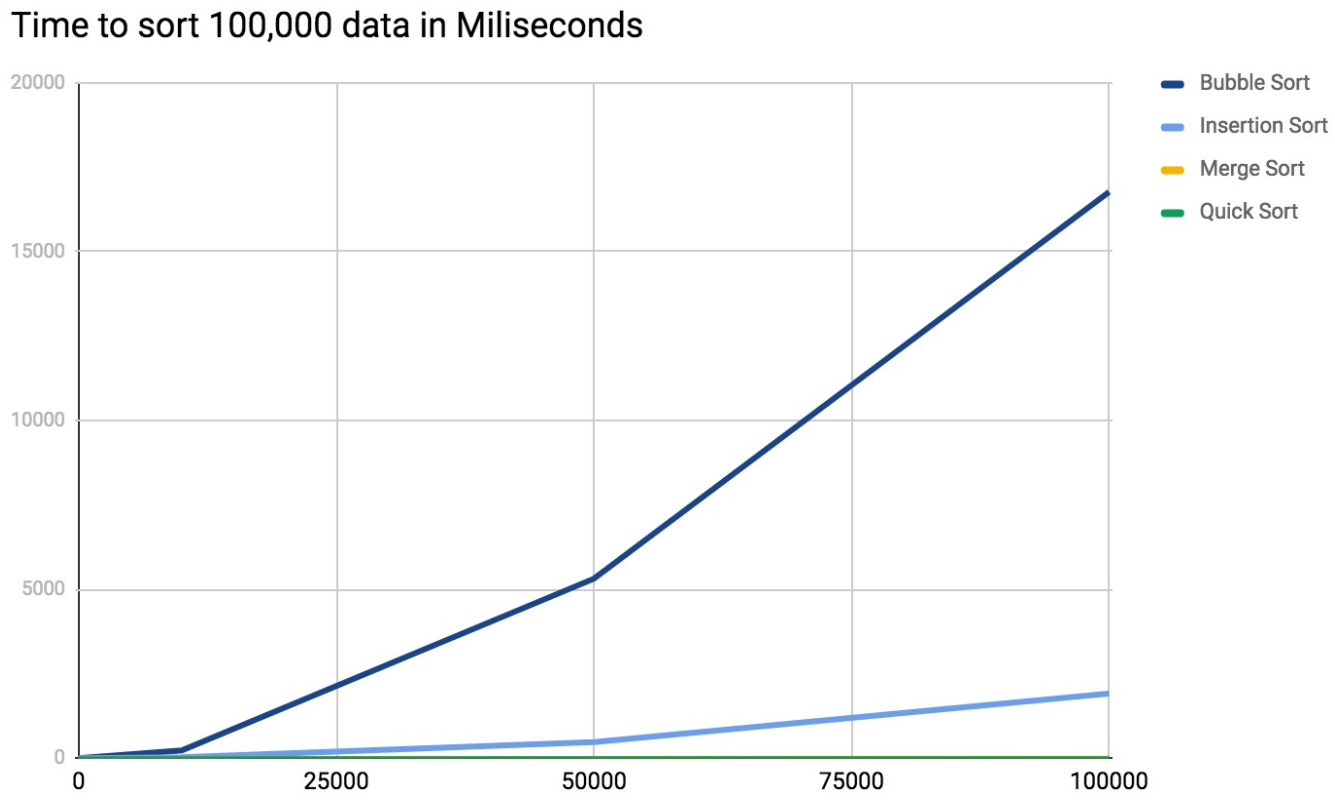

Figure 1 illustrates a comparison of the amount of time elapsed for each of the three algorithms.

Figure 2

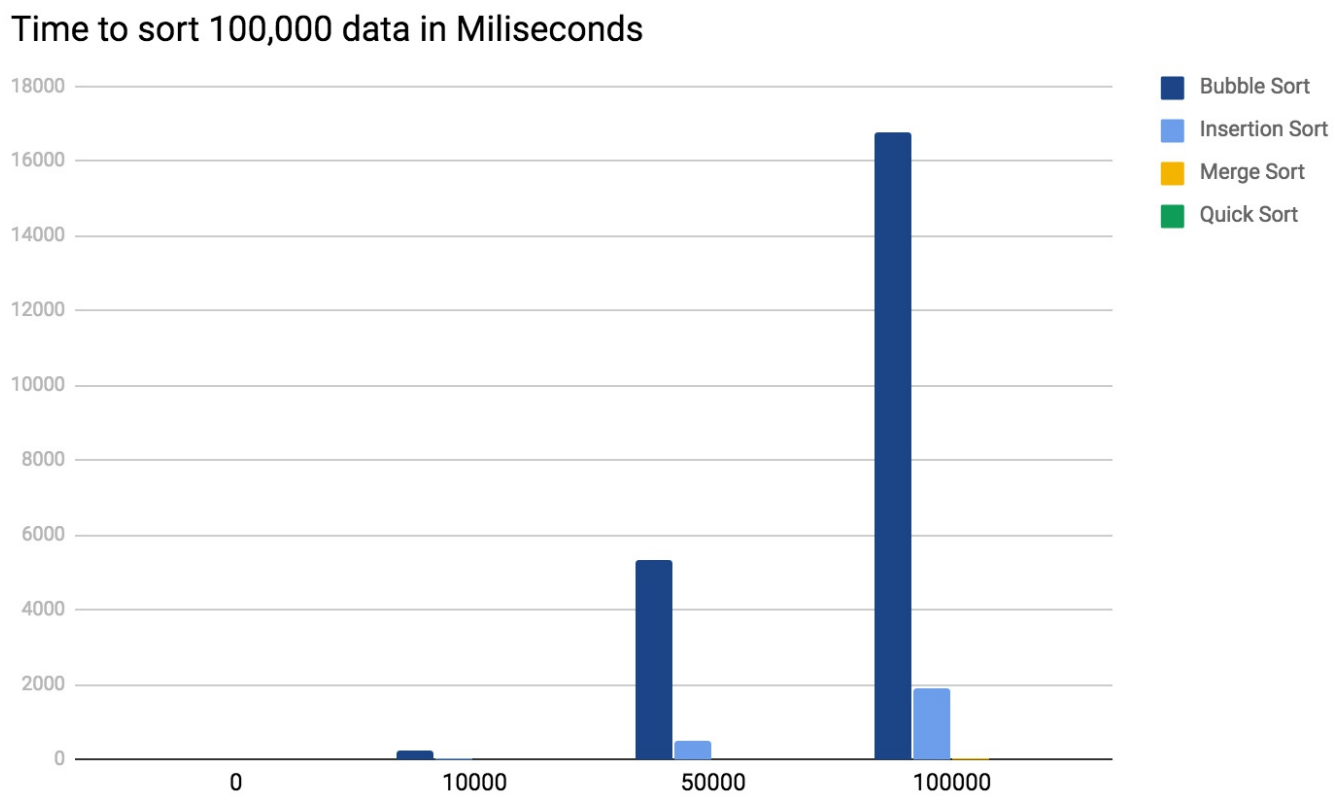

Figure 2 illustrates the same comparison of sorting times as a bar graph.

Reflection

Running the four sorting algorithms with varying quantities of data demonstrated that for certain algorithms, the amount of time required to complete the sort increased as the amount of data inputs increased. For other algorithms, the time required to complete the sort remained relatively linear regardless of the number of data inputs.

In particular, the time required to complete the Bubble Sort algorithm increased exponentially as the amount of data inputs increased, which is in line with its complexity of `O(n^2)`. The Insertion Sort algorithm performed better, with very little time required to complete the sort while inputs remained low, but the time increased drastically as the inputs increased. For this reason, Insertion Sort also shares a complexity of `O(n^2)`.

In contrast, the Merge Sort and Quick Sort algorithms remained relatively linear as the data inputs increased, to the point where their plot points are indistinguishable from each other on both charts. Both of these algorithms boast a complexity of `O(n log n)`.

Part II: Segregate Even and Odd numbers

Given an array A[ ], write an algorithm in pseudocode that segregates even and odd numbers. The algorithm should put all even numbers first, and then odd numbers.

Example:

```
Input  = {12, 34, 45, 9, 8, 90, 3}
Output = {12, 34, 8, 90, 45, 9, 3}
```

In the output, order of numbers can be changed, i.e., in the above example 34 can come before 12 and 3 can come before 9.

Pseudocode:

```
input = A[12, 34, 45, 9, 8, 90, 3]
lowIndex = 0
highIndex = input.length - 1

while lowIndex != highIndex do
    if input[lowIndex] % 2 != 0 do
        while input[highIndex] % 2 != 0 do
            highIndex ← highIndex - 1
        end while

        temp ← input[lowIndex] // odd number gets stored in temporary variable
        input[lowIndex] ← input[highIndex]
        input[highIndex] ← temp
    end if

   lowIndex ← lowIndex + 1
end while
```

Reflection

In terms of complexity, the number of operations required to complete this program is directly proportional to the number of input data, resulting in a linear complexity or `O(n)`. If we consider `n` to be the number of elements in the input array, then the maximum number of operations this algorithm can have is `n`. In the best case scenario, the loop will still check every element in the array and thus it also still has a complexity of `0(n)`.

Part III: Recursion

Design a recursive method for each of the following problems:

Problem 1

When you cut a pizza, you cut along a diameter of the pizza. Let Pizza(n) be the number of slices of pizza that exist after you have made n cuts, where n ≥ 1. For example, Pizza(2) = 4 because there are four slices after two diagonal cuts.

Pseudocode:

```
slicePizza(n) do
    if n = 1 do
        return 2 // base case
    end if

    return 2 * slicePizza(n-1) // recursive call
end
```

Example call:

```
slicePizza(4)
2 * slicePizza(3)
2 * 2 * slicePizza(2)
2 * 2 * 2 * 2 = 16
```

Problem 2

A bunch of motorcycles and cars want to parallel park on a street. The street can fit n motorcycles, but one car take up three motorcycle spaces. Let A(n) be the number of arrangements of cars and motorcycles on a street that fits n motorcycles.

Pseudocode:

```
A(n) do
    if n = 1 or n = 2 do
        return 1
    end if

    if n = 3 do
        return 2
    end if

    return A(n-1) + A(n-3)  // recursive call
end
```

Example Call:

```
A(6)
A(5) + A(3)
(A(4) + A(2)) + 2
((A(3) + A(1)) + 1) + 2
((2 + 1) + 1) + 2 = 6
```