Investigating the Relative Efficiency of Different Sorting Algorithms

Quinn Stevens

June 23, 2017

Executive Summary

This report compares and contrasts the time efficiency of the Bubble Sort, Insertion Sort and Merge Sort algorithms, and contains an appendix with pseudocode for a function to segregate even and odd numbers in an array.

Contents

Chapter 1

Setting up Sort_Tester.java

In order to set up the testing, I first added the following code to allow the user to define the size of the random array to generate.

```java
import java.util.Scanner;
...
Scanner scanner = new Scanner(System.in);
System.out.println("How big is your array?");
int arraySize = scanner.nextInt();
```

Listing 1: Setting up Sort_Tester.java

Then I generated a random array of that size using:

```java
import java.util.Random;
...
int[] arr = new int[arraySize];
Random random = new Random();
for (int i = 0; i < arrayLength; i++) {
    arr[i] = random.nextInt(50);
}
```

Listing 2: Generating the Randomised Array

Before running each test, I copied the generated randomised array into a second array, `toBeSorted`, so I could sort that array without affecting the original array. This meant that each algorithm was operating on the same dataset, which meant the test was fairer.

I then noted the start time, called the algorithm on `toBeSorted`, noted the finish time and worked out the time elapsed in nanoseconds using `System.nanoTime()`.

Chapter 2

Implementing Quicksort

I decided that the fourth sorting algorithm I would implement would be Quick Sort. The code I used to implement that sorting algorithm is listed on the next page in Listing 3.

```java
public static void quickSort(int[] inputArray, int low, int high) {
    if (inputArray == null || inputArray.length == 0) {
        return;
    }
    if (low >= high) {
        return;
    }

    //pick pivot
    int middle = low + (high - low) / 2;
    int pivot = inputArray[middle];

    // make left < pivot and right > pivot
    int i = low, j = high;
    while (i <= j) {
        while(inputArray[i] < pivot) {
            i++;
        }
        while(inputArray[j] > pivot) {
            j--;
        }

        if (i <= j) {
            int temp = inputArray[i];
            inputArray[i] = inputArray[j];
            inputArray[j] = temp;
            i++;
            j--;
        }
    }

    // recursively sort two subarrays
    if (low < j) {
        quickSort(inputArray, low, j);
    }
    if (high > i) {
        quickSort(inputArray, i, high);
    }
}
```

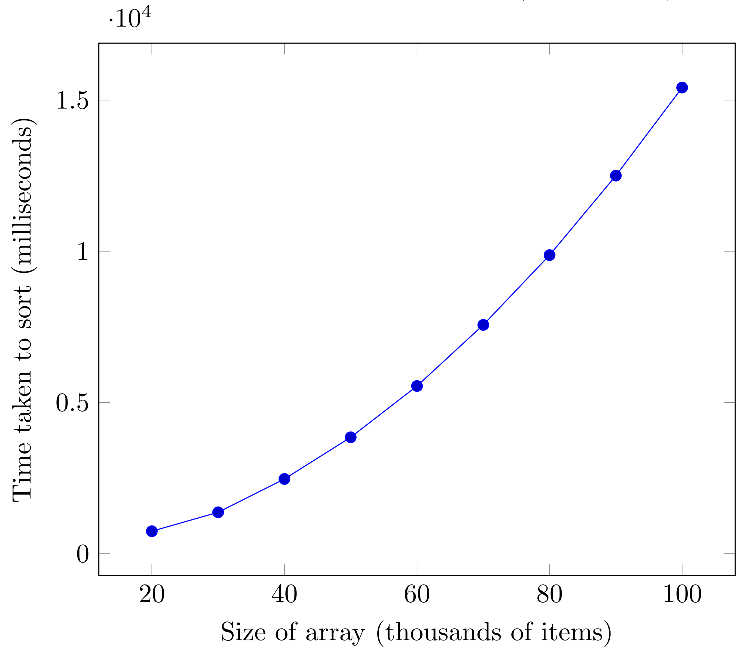
Listing 3: Quicksort Implementation

Chapter 3

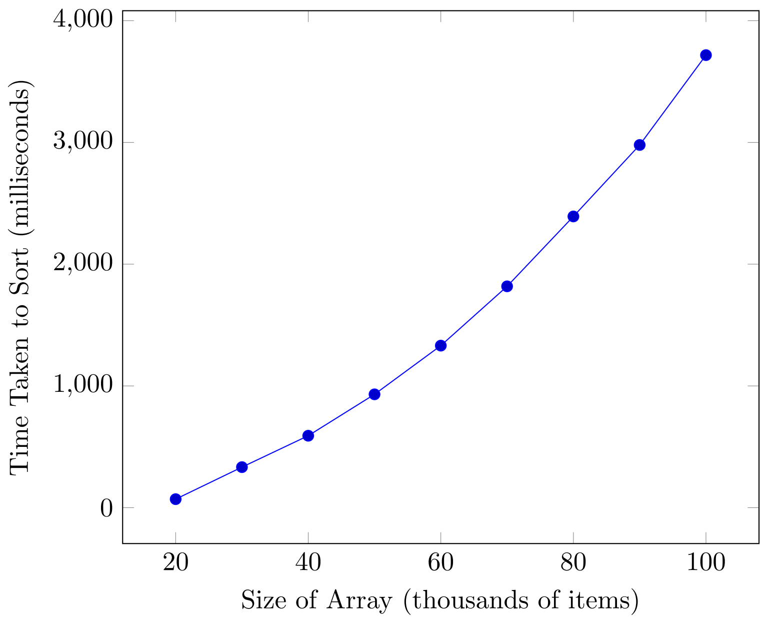
Comparing Time Complexity

In this chapter I will go through the results of running Sort_Tester and discuss their time complexity.

The time to sort 100,000 data (Bubble Sort)

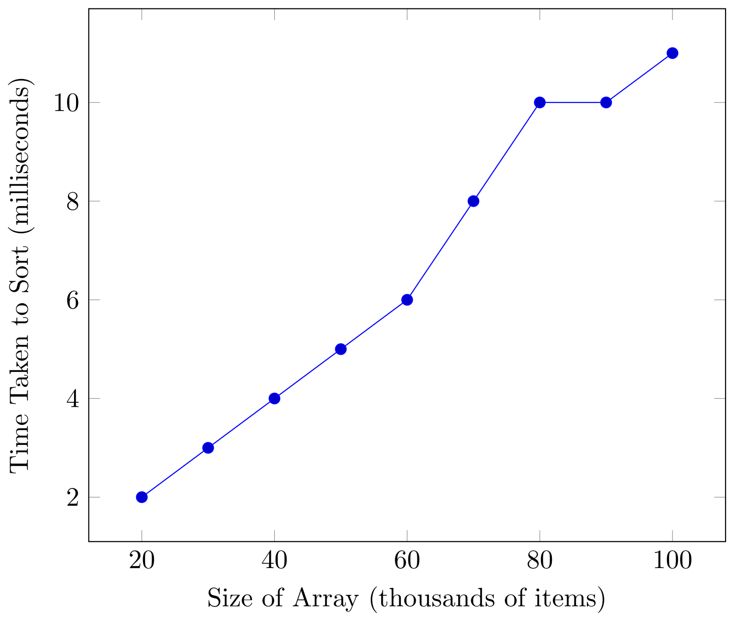


The time to sort 100,000 data (Insertion Sort)

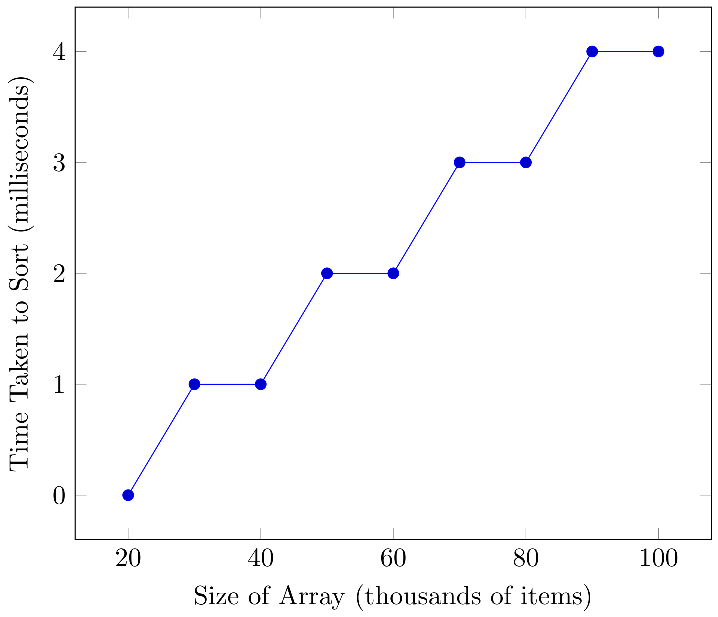


These two graphs are very clearly exponential, although bubble sort's time taken grows a lot faster than insertion sort. Because the graphs are exponential, this means that the time complexity of these algorithms is $O(n^2)$

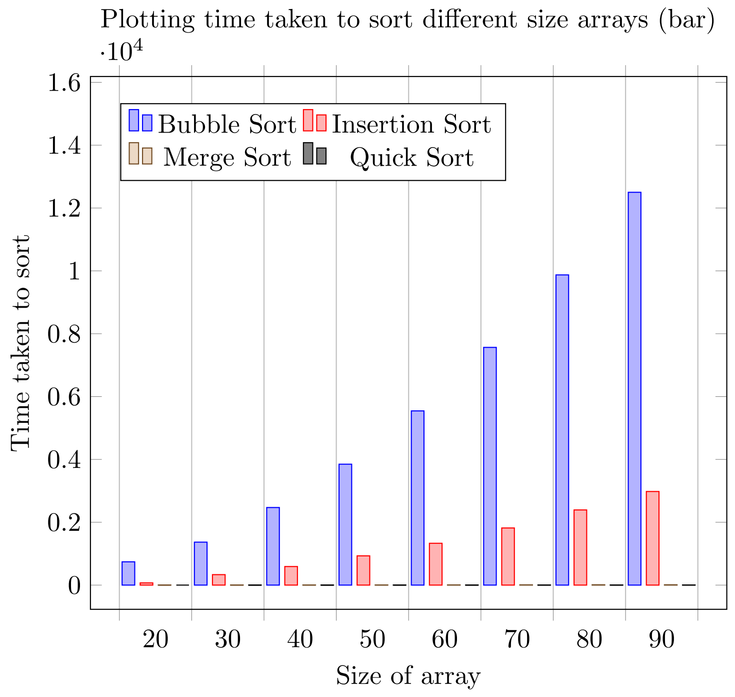
The time to sort 100,000 data (Merge Sort)



The time to sort 100,000 data (Quick Sort)



These two graphs, however, are much harder to determine time complexity from as the times involved are so small. By doing some research online, however, I was able to determine that the time complexity of the two algorithms is $O(n \cdot log(n))$



In this bar graph, we can see that merge- and quicksort are much better than bubble and insertion sort. technically, merge sort is better than quick sort too, because it has a better worst-case time complexity $(O(n \cdot log(n)))$ than quick sort's $(O(n^2)$

Appendix 1: Segregating Odd and Even Numbers

N.B. The 'pseudocode' in these appendices is actually Python as I find that easier to write.

The following pseudocode takes an input array and outputs the array but with all even numbers on the left and all odd numbers on the right.

Here is the pseudocode:

```
def segregate(array):
    # make pointers to the first and last element
    low = 0
    high = len(array)-1

    while low < high:
        # move the lower pointer up until you find an odd number
        while array[low] % 2 == 0 and low < len(array)-1:
            low++
        # move the higher pointer down until you find an even number
        while array[high] % 2 != 0 and high > 0:
            high--
        # swap the two numbers being pointed to
        if low < high:
            temp = array[low]
            array[low] = array[high]
            array[high] = temp
    # output segregated array
    return array
```

This algorithm has a Big-O notation of $O(n)$

Appendix 2: Recursion

Pizza Cutting

Pseudocode

The following pseudocode works out the number of slices of pizza you will have if you make n cuts through the center.

Here is the pseudocode:

```
def pizza(cuts):
    # end condition
    if cuts == 1:
        return 2
    # recurse
    return pizza(cuts-1) + 2
```

Testing

When the function is called for `cuts=4`, it resolves as follows:

```
pizza(4)
v
pizza(3) + 2
v
pizza(2) + 2 + 2
v
pizza(1) + 2 + 2 + 2
v
2 + 2 + 2 + 2
v
4
```

Parking Spaces

Pseudocode

The following function takes the number of motorcycle parking spaces on a road, and returns the number of different ways to park vehicles on that road, assuming a car takes up 3 motorcycle spaces.

```python
# find the number of permutations
def park(spaces):
    # base cases
    if spaces == 0:
        return 0
    if spaces == 1 or spaces == 2:
        return 1
    if spaces == 3:
        return 2
    # recurse
    return park(spaces-1) + park(spaces-3)
```

Testing

For `spaces=5` the code resoves as follows:

```
park(5)
v
park(4) + park(2)
v
park(3) + park(1) + 1
v
2 + 1 + 1
v
4
```

For `park(6)` the function resolves like this:

```
park(6)
v
park(5) + park(3)
v
park(4) + park(2) + 2
v
park(3) + park(1) + 1 + 2
v
2 + 1 + 1 + 2
v
6
```

so `park(6) = 6`

The possible combinations of parking in a street with six spaces are as follows (these were worked out manually as the specification specifically said that the program did not need to work this out):

```
M = motorcycle
C = car
MMMMMM
CMMM
MCMM
MMCM
MMMC
CC
```