

Data Structures & Algorithms

Ricard Solé Casas

June 7, 2017

Foreword

Some of the code samples in this hand-in are in Haskell, as opposed to pseudocode. This is because it made for easier testing and data modeling for the problems. It happens to be a very expressive language that closely resembles mathematical functions.

Declaration

I confirm that the submitted coursework is my own work and that all material attributed to others (whether published or unpublished) has been clearly identified and fully acknowledged and referred to original sources. I agree that the College has the right to submit my work to the plagiarism detection service. TurnitinUK for originality checks.

Acknowledgements

I'd like to thank my partner Shannon for her continued support and challenges that help me grow, both professionally and personally. I would also like to thank all of you who also helped me get here.

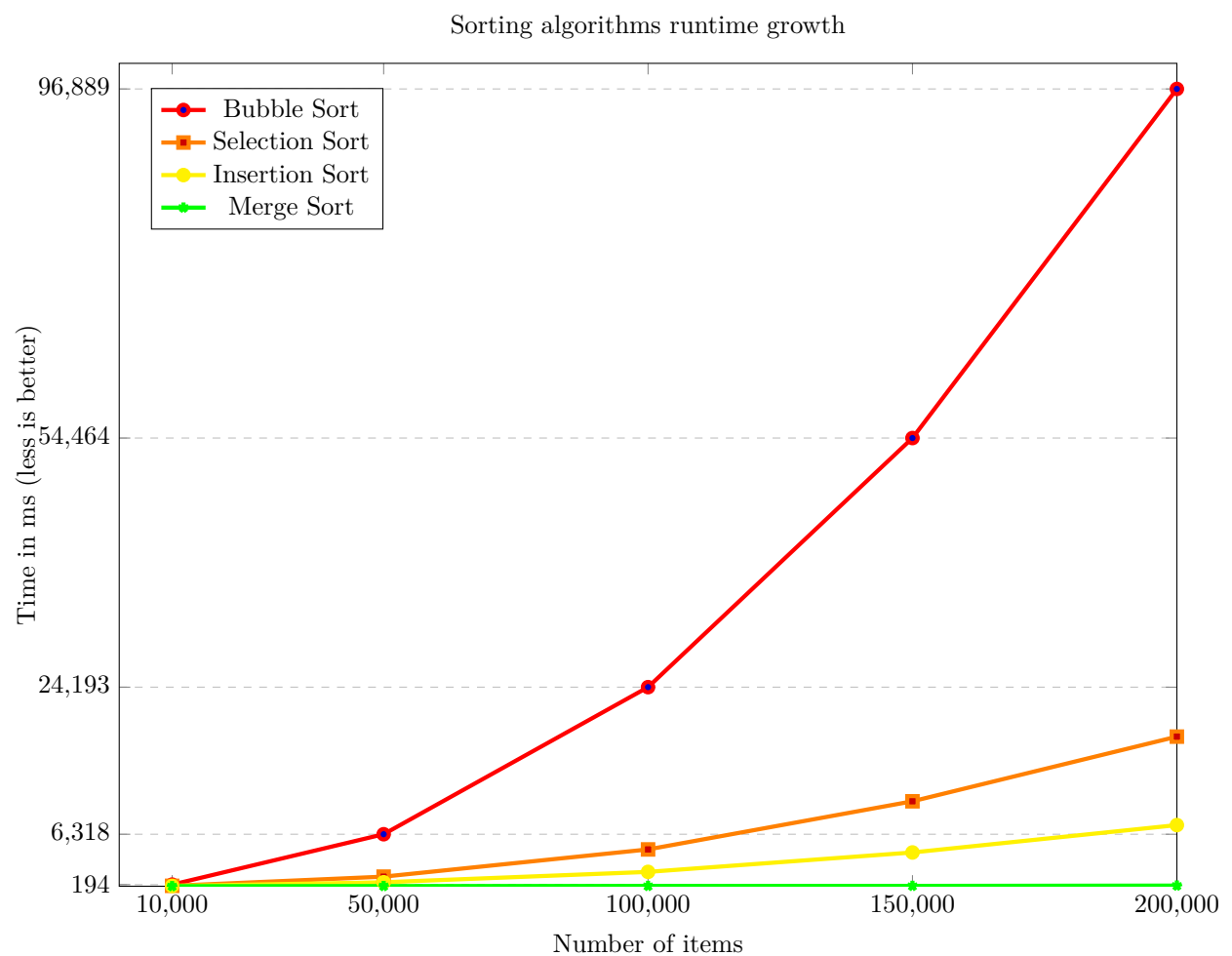
Contents

1	Analysis of Sorting Algorithms	3
1.1	Sort time growth	3
1.2	Runtime Comparison	4
1.3	Reflection	4
2	Segregate Even and Odd numbers	5
2.1	Task	5
2.2	Implementation	5
2.3	Complexity analysis	5
3	Recursion	6
3.1	Pizza cutting	6
3.1.1	Problem definition	6
3.1.2	Implementation	6
3.2	Parking	6
3.2.1	Problem definition	6
3.2.2	Implementation	6
3.2.3	Possible Arrangements $n = 6$	7
4	Appendix A	8
4.1	Benchmark.java	8
4.2	SelectionSort.java	9
5	Appendix B	10
5.1	Isolate.hs	10
6	Appendix C	11
6.1	Pizza.hs	11
6.2	Parking.hs	11

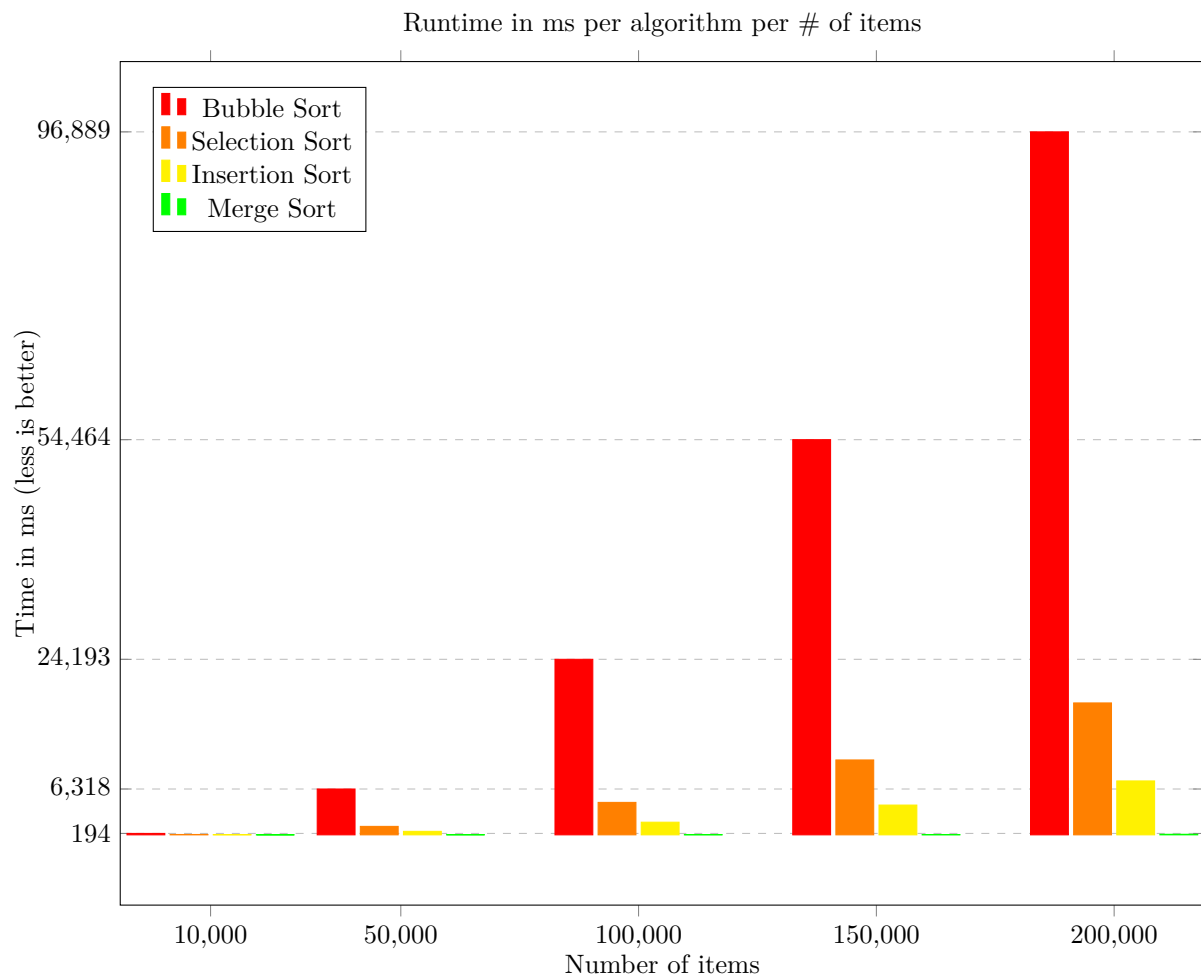
Chapter 1

Analysis of Sorting Algorithms

1.1 Sort time growth



1.2 Runtime Comparison



1.3 Reflection¹

We know that the complexity for the algorithms benchmarked above — $O(n^2)$, $O(n^2)$, $O(n^2)$ and $O(n \log_2(n))$ respectively—, and we can see that mapped accurately in the plots above.

The first plot, the graph displayed on section 1.1, shows the increment in time for the *bubble*, *selection*, and *insertion* sort algorithms as exponential. Their curves vary greatly because Big O is not meant to give an accurate description of the time, just of how it'll scale. The point being, they all scale in a similar manner. The green line represents the *merge* sort algorithm which is, along with *quicksort*, considered to be the best algorithm overall. We can barely see it scale because the plot is drawn next to the quadratic plot of *bubble* sort. If we look closely, though, we can see it almost resembles a linear time complexity, but not quite.

The second plot, a bar graph displayed on section 1.2, uses the same colors and the same data to represent, perhaps more clearly, how each algorithm fares amongst each other using the same dataset.

¹Selection sort and benchmark utility code can be found Appendix A.

Chapter 2

Segregate Even and Odd numbers

2.1 Task

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.

2.2 Implementation¹

```
isolate :: (Integral a) => [a] -> [a]
isolate xs =
    evens xs ++ odds xs

evens :: (Integral a) => [a] -> [a]
evens = filter even

odds :: (Integral a) => [a] -> [a]
odds = filter odd
```

2.3 Complexity analysis

For the algorithm implemented in section 2.2:

- filtering through an array for even or odd numbers has a complexity of $O(n)$, where n is the length of the given list.
- $(++)$ has a complexity of $O(m)^2$, where m is the length of the second list.
 - $m = \text{length}(\text{odds}(xs))$ in our case
- With this we can conclude that the complexity of this algorithm is $O(2n + m)$, which can be simplified to $O(n + m)$.

¹Full program sample can be found on Appendix B

²<https://goo.gl/yZu7Ig>

Chapter 3

Recursion¹

3.1 Pizza cutting

3.1.1 Problem definition

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 $n \geq 1$. For example, `pizza(2) = 4` because there are four slices after two diagonal cuts.

Write a recursive method `pizza(n)` to return the number of slices and verify the correctness of your method when the pizza is cut 4 times (*3 points*).

3.1.2 Implementation

```
pizza :: Int → Int
pizza 0 = 1
pizza 1 = 2
pizza n = 2 * pizza (n - 1)
```

3.2 Parking

3.2.1 Problem definition

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 (*7 points*).

3.2.2 Implementation

```
parking :: Int → Int
parking 0 = 1
parking 1 = 1
parking 2 = 1
parking 3 = 2
parking n = parking (n - 1) + parking (n - 3)
```

¹Full code samples can be found on Appendix C

3.2.3 Possible Arrangements $n = 6$

MMMC, MMCM, MCMM, CMMM, MMMMMM, CC.

Chapter 4

Appendix A

4.1 Benchmark.java

```
package me.rsole.util;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.TimeUnit;

public class Benchmark {
    private List<Thread> threads = new ArrayList<>();

    public void add(String name, Runnable computation) {
        Thread t = new Thread(
            () -> System.out.printf("%s took %s.\n", name, run(computation))
        );
        threads.add(t);
    }

    public void run() {
        for (Thread t : threads) t.run();
    }

    private String run(Runnable computation) {
        long start = System.currentTimeMillis();
        computation.run();
        long end = System.currentTimeMillis();

        return format(end - start);
    }

    private String format(long time) {
        long minutes = TimeUnit.MILLISECONDS.toMinutes(time);
        long seconds =
            TimeUnit.MILLISECONDS.toSeconds(time) -
            TimeUnit.MINUTES.toSeconds(minutes);
        long milliseconds =
            time -
            (TimeUnit.SECONDS.toMillis(seconds) +
            TimeUnit.MINUTES.toMillis(minutes));
    }
}
```

```

    return String.format("%dm %ds %dms", minutes, seconds, milliseconds);
}
}

```

4.2 SelectionSort.java

```

package me.rsole.sort;

import org.jetbrains.annotations.Contract;

public class SelectionSort {
    @Contract(pure = true)
    public static int[] exec(int[] xs) {
        int[] ys = xs.clone();
        int minIdx = 0;

        for (int i = 0; i < ys.length; i++) {
            for (int j = i; j < ys.length; j++) {
                if (ys[j] < ys[minIdx]) minIdx = j;
            }

            int swap = ys[i];
            ys[i] = ys[minIdx];
            ys[minIdx] = swap;
            minIdx = i + 1;
        }

        return ys;
    }
}

```

Chapter 5

Appendix B

5.1 Isolate.hs

```
module Main where

list :: [Int]
list = [12, 34, 45, 9, 8, 90, 3, -1000, -1001]

main :: IO ()
main = print (isolate list)

isolate :: (Integral a) => [a] -> [a]
isolate xs =
    evens xs ++ odds xs

evens :: (Integral a) => [a] -> [a]
evens = filter even

odds :: (Integral a) => [a] -> [a]
odds = filter odd
```

Chapter 6

Appendix C

6.1 Pizza.hs

```
module Main where

main :: IO ()
main = putStrLn $ "After 4 cuts we have " ++ show (pizza 4) ++ " slices."

pizza :: Int → Int
pizza 0 = 1
pizza 1 = 2
pizza n = 2 * pizza (n - 1)
```

6.2 Parking.hs

```
module Main where

main :: IO ()
main = print (parking 5)

parking :: Int → Int
parking 0 = 1
parking 1 = 1
parking 2 = 1
parking 3 = 2
parking n = parking (n - 1) + parking (n - 3)
```