# COMP20010 Lab Six: Algorithm Analysis

Kashif Hussain

November 12, 2013

## 1 Part 1 Algorithm

### 1.1 Asymptotic run-time analysis

My algorithm runs in asymptotic time $O(nLogn)$. The argument for this follows:

The storing of values requires n comparisons of i to n. (i ¡ n for loop) i++ is executed n-1 times (increments and stores -¿ 2n-2 (for loop) scanf executes n times indexing an array occurs n times storing a value occurs n times.

This means the for loop is $O(n+2(n-1)+n+n+n) = O(6n-2) = O(cn)$ $= O(n)$

The quicksort algorithm has best/average case of O(nlogn) but the worst case is $O(n^2)$. This is because: The partition operation takes O(n). In the most unbalanced case, each time we perform a partition we divide the list into two sublists of size 0 and n-1 (for example, if all elements of the array are equal). This means each recursive call processes a list of size one less than the previous list. Consequently, we can make n-1 nested calls before we reach a list of size 1. This means that the call tree is a linear chain of n-1 nested calls. The ith call does O(n-i) work to do the partition, and sum from i=0 to n of (n-i) = $O(n^2)$, so in that case Quicksort takes $O(n^2)$ time. That is the worst case

So the quicksort takes $O(n^2)$ (Worst Case)

While loop: Worst case involves it searching from the 90th percentile value to the last value in the array (if the numbers in the range of the 90th percentile to the final are the same value including the final number) to produce a -1.

This means while (found == 0) is checked 0.1*n times (0.1* = final 10% of values) the (numbers[nextInteger] ¿ numbers[ninetyPercentileArrayNo]) comparison takes place 0.1*n times nextInteger = nextInteger +1 is 2 operations (summing and storing) and so occurs 2*0.1*n times

Worst case is: $O((0.1*n)+(0.1*n)+(2*0.1*n)) = O(0.4n) = O(n) =$ $O(n)$

This gives a final asymptotic time of:

$O(n^2+n+n) = O(n^2+2n) = O(n^2)$ (Worst Case) $O(nlogn+n+n) =$ $O(n(logn+2)) = O(nlogn)$ (Average Case)

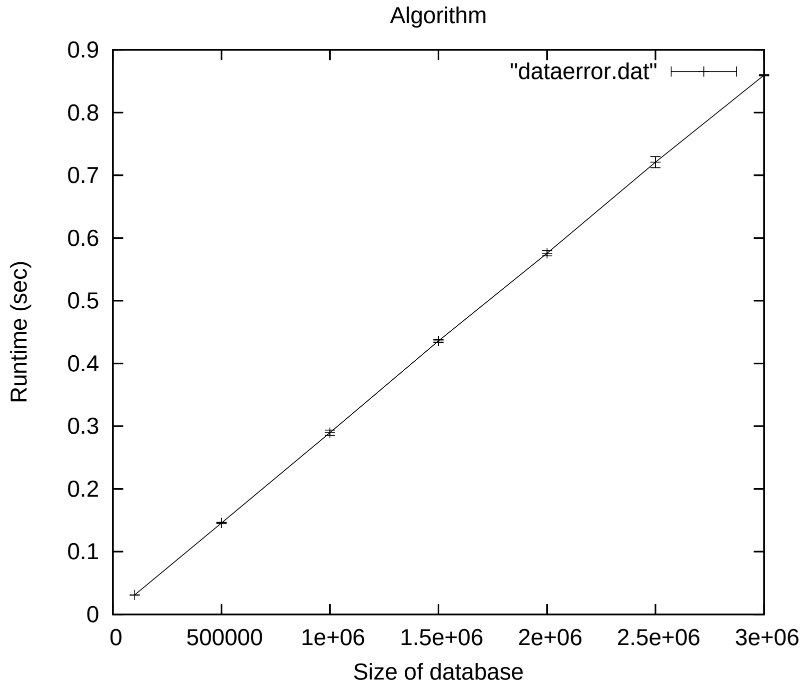


Figure 1: Error plot of different inputs for Algorithm 1

### 1.2 Experiments

I will be using real time for my experiments.

As you can see, as n gets larger, the shape of the graph approximates to nlogn The gradient from x = 100000 to 3000000 is 1 This tells me that the power of n is 1. Using fitscript i get: a = 1.96323e-08 for f(N) = 1.96323e-08 * N * log(N)

| data size | run time | $t = 1.96323e - 08 * N * log(N)$ |
|---|---|---|
| 100000 | 0.031 | 0.0293 |
| 500000 | 0.14583 | 0.1466 |
| 1000000 | 0.28967 | 0.2933 |
| 1500000 | 0.43586 | 0.4400 |
| 2000000 | 0.57573 | 0.5866 |
| 2500000 | 0.72089 | 0.7333 |
| 3000000 | 0.8597 | 0.8800 |

the difference in run time in steps of 500000 gradually decreases 500000 to 1000000 -¿ Approx 0.145 increase 1500000 to 2000000 -¿ Approx 0.140 increase 2500000 to 3000000 -¿ Approx 0.135 increase

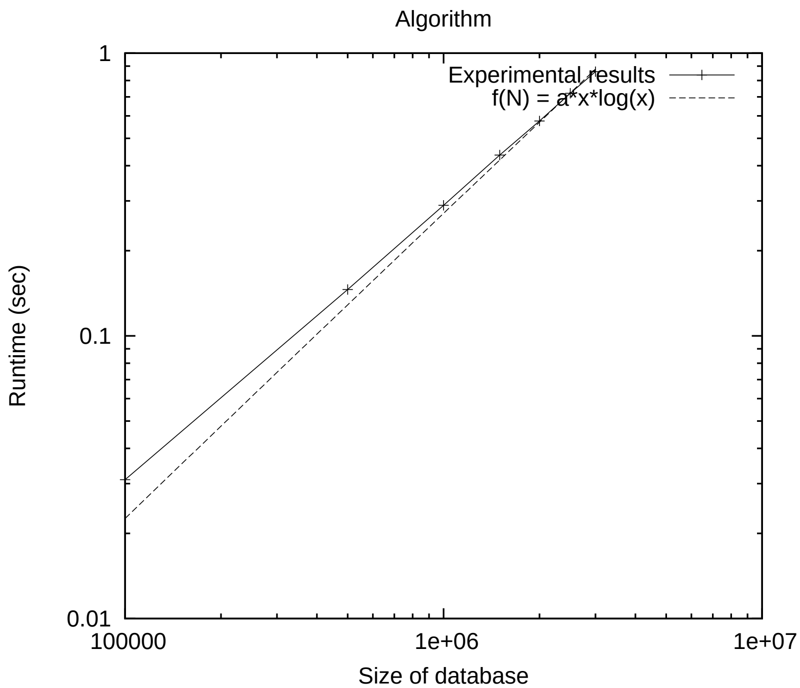


Figure 2: Fitplot of different inputs for Algorithm 1

The graph of nlogn tends to curve down for high values of n and shows a pattern of doing so. Every step of 500000 has a smaller and smaller increase in runtime

### 1.3 Prediction

My estimate of the equation for the run-time of the algorithm is:

$$t(N) = 1.96323e - 08 * N * log(N) \tag{1}$$

Using this, the estimated time to find the ninetieth percentile of a file containing 60 million numbers is 30.44 seconds.

## 2 Part 2 Algorithm

### 2.1 Asymptotic run-time analysis

My algorithm runs in asymptotic time $O(N)$.

The argument for this follows:

The for loop which scans a file for integers does the following: The storing of values requires n comparisons of i to n. (i ¡ n for loop) i++ is executed n-1 times (increments and stores) (for loop) scanf executes n times storing a value in variable value occurs n times checking if value is greater than or equal to 1000000 occurs n times storing the values and indexing an array occurs k times (where k = n-x) This means the for loop is $O(n + 2(n - 1) + n + n + n + k) = O(6n - 2 + k) = O(cn) = O(n)$

The least significant radix sort has a worst case perfomance of O(kN)

Indexing and storing variables in the 2nd array does the following: k = the number of values ¿= 1000000 in N. (k=n-x)

makes k comparisons (i¡k) makes k-1 incrmements and storage operations (i++) index's and stores a value k times (number2[i] = numbers[i])

$O(n - x + 2(n - x - 1) + 2(n - x)) = O(5n - 4x - 2) = O(cn) = O(n)$

In total this gives:

$O(n + kn + n) = O(n)$

### 2.2 Experiments

I will be using real time for my experiments

As you can see, as n gets larger, the shape of the graph approximates to n The gradient from x = 1 to 1000000 is 1 This tells me that the power of n is 1. Using fitscript i get: a = 1.2563e-07 for f(N) = 1.2563e-07 * N

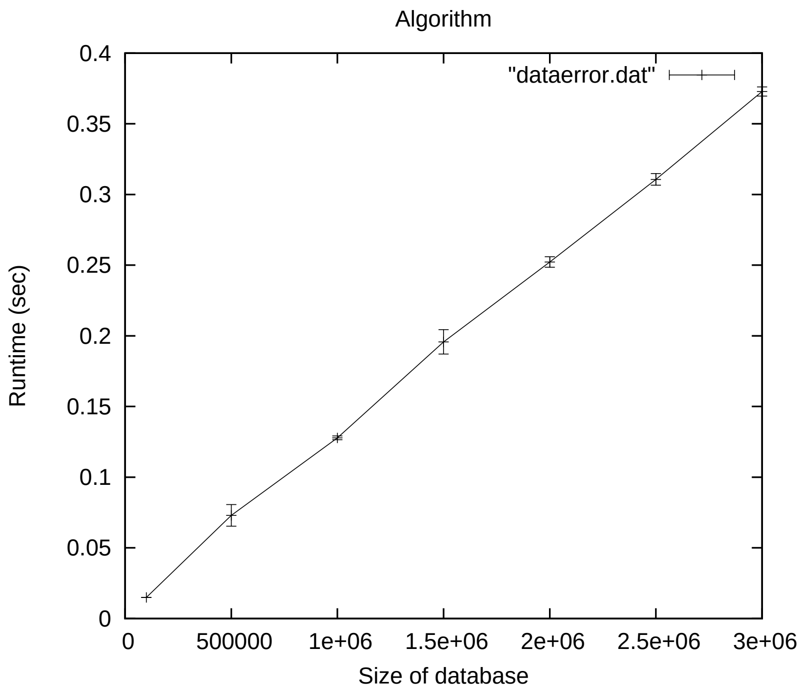


Figure 3: Error plot of different inputs for Algorithm 2

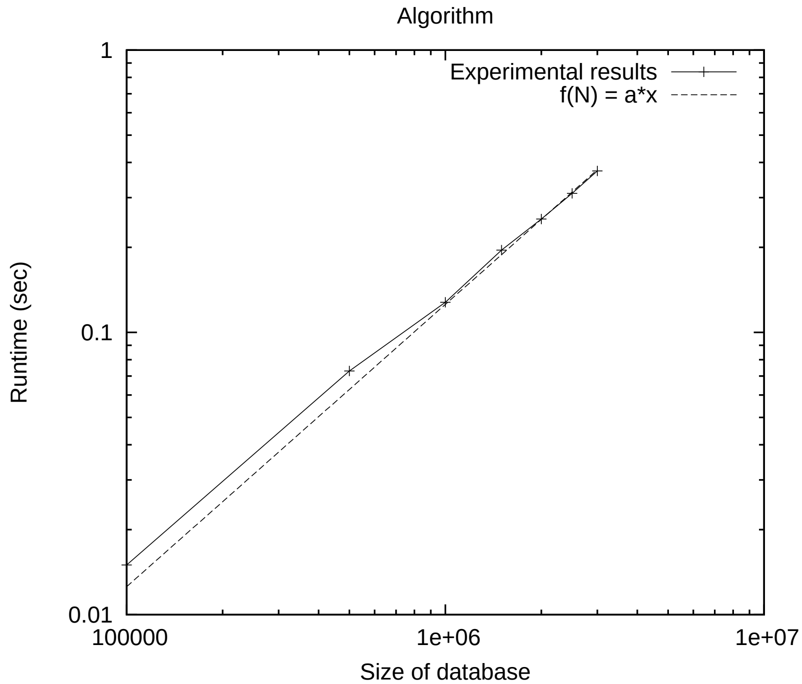


Figure 4: Fit plot of different inputs for Algorithm 2

| data size | run time | $t = 1.2563e - 07 * N$ |
|---|---|---|
| 100000 | 0.0150 | 0.0126 |
| 500000 | 0.0730 | 0.0628 |
| 1000000 | 0.1278 | 0.1256 |
| 1500000 | 0.1958 | 0.1884 |
| 2000000 | 0.2522 | 0.2513 |
| 2500000 | 0.3107 | 0.3141 |
| 3000000 | 0.3729 | 0.3767 |

the difference in run time in steps of 500000 remains approximately constant

500000 to 1000000 -¿ Approx 0.0628 increase 1500000 to 2000000 -¿ Approx 0.0629 increase 2500000 to 3000000 -¿ Approx 0.0626 increase

As the increase in steps of 500000 remains constant, it follows the pattern of f(N) = aN where a remains constant.

### 2.3 Prediction

My estimate of the equation for the run-time of the algorithm is:

t(N) = 1.2563e-07*N(2)Using this, the estimated time to find the ninetieth percentile of a file containing 60 million numbers is 7.54 seconds.