Algorithms Assignment 1

Jeremy Kwok
100341977

Question 1

1.1

This function will return true if any element in the list is 0, and false otherwise.

1.2

T1(n) = T(=) + T(==) + T(==) + T(==) + T(=) + T(/) + T(=) + T(*) T(/) + T1(n/3) + T1(n/3) + T1(n/3)

1.3

T1(n) = O(1) + T1(n/3) + T1(n/3) + T1(n/3)
T1(n) = 3T1(n/3) + O(1)

Initially, my intuition tells me that T1(n) is Theta($\log_3$(n)), because the function returns true when an element is equal to 0. Since the function splits the list into thirds, and the function does not complete the rest of the OR statements when any of the statements returns true, then the asymptotic complexity should be log(n). Despite this logic, the proofs showed another asymptotic complexity. I derived the complexity using two methods:

First, we can use the Master Method from CLRS, Chapter 4.
a = 3, b = 3.
f(n) = 1. Set E = 1.
f(n) = O(n($\log_b$(a)-E) = 1. Thus rule 1 applies, and we can say that T1(n) = Theta(n^$\log_b$(a)) = Theta (n^1) = Theta(n).

Alternatively, we can use the substitution method.

T1(n) = 3T1(n/3) + O(1)
Consider Theta(n)
n + b = 3((n/3) + b) + c
n + b = n + 3b + c
3b = -c
b = -c/3

T1(n) = n - c/3
Therefore we can state that T1(n) = Theta(n)

a*log(n) + b = c + 3b + a*(log(n/3)+log(n/3)+log(n/3))
a*log(n) + b = c + 3b + a*(log(n^3/27))
b = -c/2

Since we have a constant equals a constant, the solution exists.

therefore T1(n) = Theta(log(3)(n))

1.4

See rewrite.java

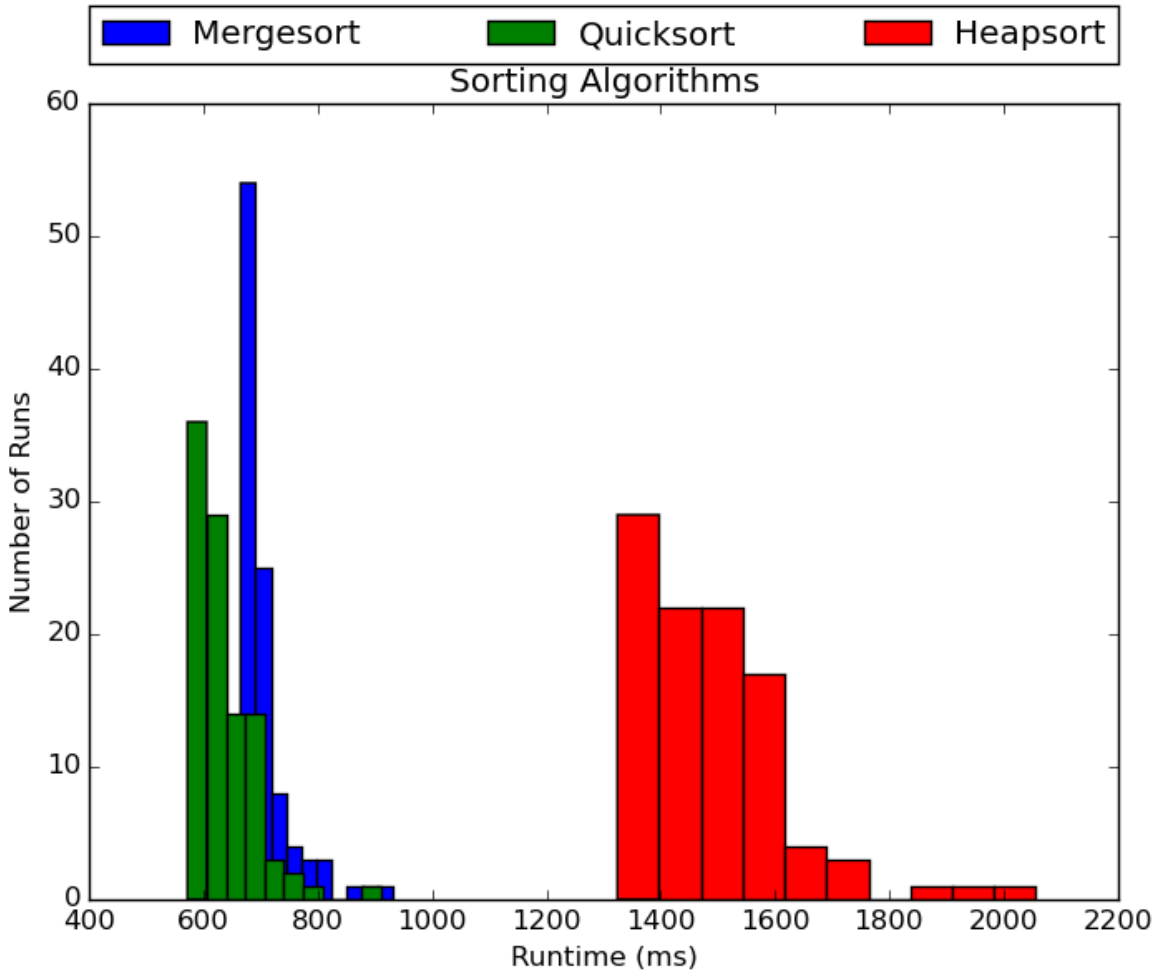
1.5

rewrite.java uses a for loop to iterate through every element of the list. Thus, the asymptotic run-time of rewrite.java is Theta(n). Since this is the same as the asymptotic run time complexity of the given function, then we can say they are the same.

Question 2

See attached files.

Graph:



From this graph we can see that Mergesort is slightly slower than Quicksort, and Heapsort is much slower than both. While all of these sorts have the same asymptotic complexity, Heapsort is still slower. This is because asymptotic complexity is only concerned about the growth of the function, rather than the absolute speed of the sort. It is possible that this implementation of Heapsort has much more runtime overhead than the other two sorts.

Question 3

3.1

Assume: 1 byte = 1 character
Therefore: 100 characters + 1 newline character = 1 string
Therefore: 100 million strings = 10,100,000,000 bytes
We know: 1 GB = 2^30 bytes = 1,073,741,824 bytes

Therefore the space required to store 100 million strings of 100 characters long is approximately 10 GB.

3.2

See attached files.

3.3

Splitting and Sorting:
Load the file with 100 Million strings into memory 1 million strings at a time.
For each set of 1 million strings, perform mergesort and write to an auxiliary data file, ending with 100 data files containing 1 million strings.

Selection and Writing:
From each file, load the first string from the file into memory. These strings are the lowest strings in each cluster of 1 million strings.
Using a simple loop, determine which of these strings is the lowest.
Write that string into sorted.txt, and replace that string in memory from the file it came from.

Repeat 100 million times until we have written each file into sorted.txt

3.4

Complexity Analysis:

Splitting and Sorting:
Load each element => O(n)
mergesort 100 times => O(100*nlog(n))
Write to files => O(n)

Selection and Writing
Load first string into memory => O(n/100)
Determine which string is lowest => O(100)
Write lowest into sorted => O(1)

Replace lowest in memory => O(1)

Thus we have O(n) + O(100*nlog(n)) + O(n) + O(n/100) + n*O(100) + 2n(O(1))

We can simplify this statement to our highest complexity bound, which is O(nlog(n)). Thus we can say that our algorithm runs in O(nlog(n)) time.

Data Passes:

Splitting and Sorting:
Load each element => 1 Data Pass
mergesort 100 times => Each mergesort is n/100 of the data. Each mergesort does log(n/100) reads of each datum. Thus we have 100*log(n/100) Data Passes
Write to files => 1 Data Pass

Selection and Writing
Load first string into memory => 1/100 Data Passes
Determine which string is lowest=> (100) Data Passes
Write lowest into sorted => 1 Data Pass
Replace lowest in memory => 1 Data Pass

Total for n = 100 000 000 => 1 + 100*log(1000000) + 1 + 1/100 + 100 + 1 + 1 = 124 Data Passes.

In terms of time, we can break this down. We know from the statistics gained from prior questions that the average runtime of mergesort over 1000000 elements is 673 ms. Since we do this 100 times, sorting will take 6730 ms for Mergesort, which is roughly 20 Data passes.

3.5

See attached file.

Runtime: 983778 ms or 16.3963 minutes.