Problem 1

Assignment:
Write a method that solves the selection problem using a priority queue (you must write your own PQ), and conduct a series of experiments that indicate that the time complexity of your method is $O(N\ log(N))$.

Solution:
The implementation can be found in the file: PriorityQueue.java, with example driver code in the main class.
To indicate that the priority queue runs with a time complexity of $O(N\ log(N))$ i have conducted a series of experiments.
The experiments was designed in the following way:

100 different test cases was run. Every test case was run 10 times to extract an average count. The initial value of N elements started as 10.000. After every test case the N elements was increased with 10.000, giving me data from N = 10.000 to N = 1.000.000 in increments of 10.000.

The k element i am looking for in my test cases is the last Kth element, essentially meaning that i look for the biggest element in the list, which should give a worst case of O(n log(n))

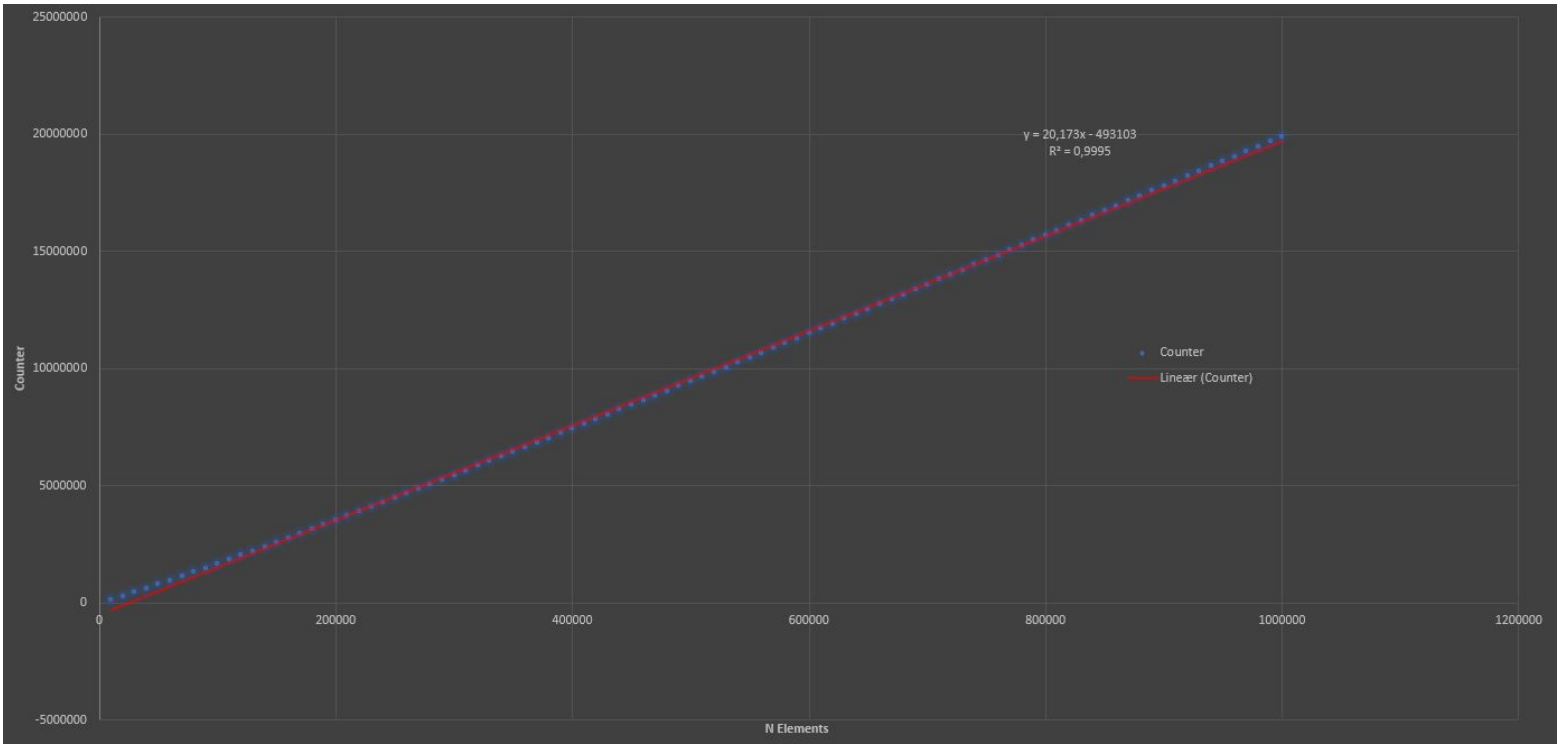

Figure 01: *Plotted data from the binary heap selection problem.*

In figure 01 we see the plotted data, with the counter data points represented as blue dots. On the x-axis we have the amount of elements, and on the y-axis we have the counter value.
If we do linear regression on the data we get a model that has an R^2 value of 0,9995 meaning that the model is a very good fit. The plotted data appears linear, however if we look closer into the partial dataset in table 01, the counter values matches surprisingly well with the actual O(n log(n)) values, making it clear that the algorithm implemented runs with a time complexity of O(n log(n)). To visually see that the graph is not in fact linear, the dataset would have to be extremely large. The entire dataset can be found in the file named Priority_Data_Regression.xlsx

N_Elements	Counter	NlogN
10000	132702	132877
20000	285385	285754
30000	445337	446180
40000	610747	611508
50000	779914	780482
60000	950690	952360
70000	1124250	1126654
80000	1301492	1303016
90000	1480161	1481187
100000	1659740	1660964
110000	1840334	1842185
120000	2021353	2024720
130000	2202892	2208459
140000	2388480	2393309
150000	2575323	2579190
160000	2762970	2766033
170000	2951385	2953779
180000	3140215	3142374
190000	3329713	3331771
200000	3519612	3521928

Table 01: *Partial dataset from the binary heap selection problem*

Problem 2

Assignment

Implement the quickSelect method and conduct a series of experiments that indicate that the time complexity of the method is O(N). When you conduct your experiments, you should do it on a variety of numbers and a relatively large amount of numbers, i.e. 1000 or more, and you must find a way to `count' the number of instructions that the methods perform, so that you can relate them to N, thus being able to estimate the time complexity. It is not sufficient to simply time your program.

Solution

The implementation can be found in QuickSelect.java file, with example driver code located in the Main.java file.

The tests in this assignment was done in a similar way as in problem 1. In this problem however, i chose to do each test run 100 times, in order to get a more representational average count. This was done in order to counter bad pivot selections that would create large fluctuations in the dataset.

This time i chose to look for an element in the middle of the array, in order to see if the algorithm was able to do the selection in O(n) time complexity.

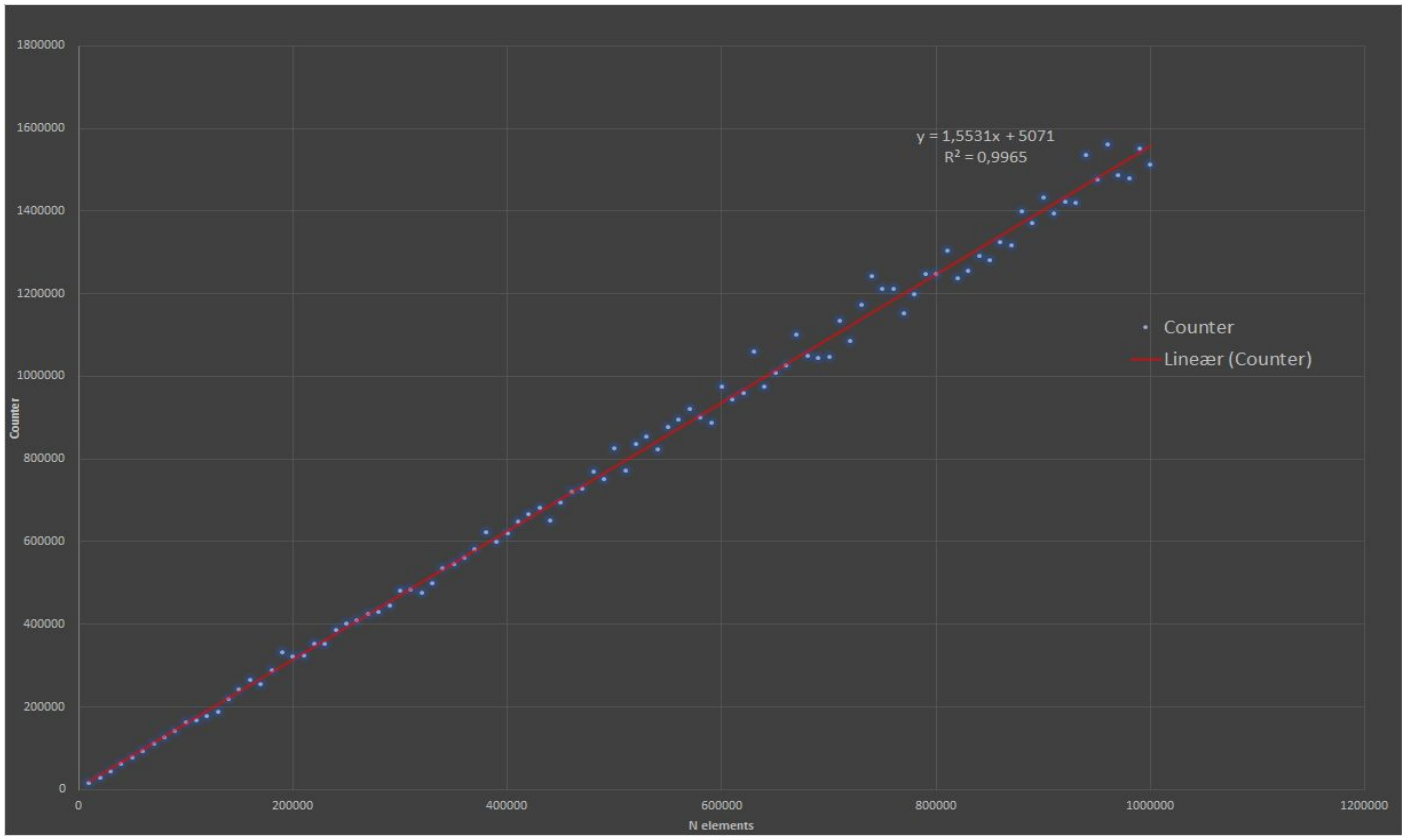

Figure 02: *Plotted data from the quickselect problem.*

In figure 02 we see the plotted data from quickselect problem. Again we have the counter value represented as blue dots. The x-axis shows the amount of elements, and the y-axis the counter. Doing linear regression gives a model with an R^2 value of 0.9965 which again is a very good fit.

Looking at table 02, we can see that 10.000 elements have a time complexity of 14.962 - if we double the amount a of elements to 20.000 we get a time complexity of 29.406, which is very close to being double the time complexity as well. We can do the same with 100.000 elements and 200.000 elements, and get a similar result. This suggests, that the algorithm is linear and runs with a time complexity of O(n). The equation on the figure suggests the constant 1,5531 that we can multiply any amount of N elements with in order to get the time complexity.
The entire dataset for quickselect can be found in the file QuickSelect_Data_Regression.xlsx.

N_Elements	Counter
10000	14962
20000	29406
30000	45463
40000	63201
50000	77124
60000	94198
70000	111203
80000	127330
90000	141847
100000	163049
110000	169066
120000	178607
130000	188686
140000	219570
150000	242659
160000	266232
170000	254534
180000	288286
190000	331344
200000	321618

***Table 02**:* Partial dataset from the quickselect problem

Problem 3

Assignment

Solve the average case recurrence equation for quickSelect.

Solution

$$T(N) = \frac{1}{N}\sum_{i=1}^{N-1} T(i) + cN$$

Multiplies with N

$$N \cdot T(N) = \sum_{i=1}^{N-1} T(i) + cN^2$$

recurrence

$$(N-1)\cdot T(N-1) = \sum_{i=1}^{N-2} T(i) + c(N-1)^2$$

Subtract the two summations

$$N\cdot T(N) - (N-1)\cdot T(N-1) = \sum_{i=1}^{N-1}(T(i)) + cN^2 - \sum_{i=1}^{N-2}(T(i)) + c(N-1)^2$$

$$N\cdot T(N) - (N-1)\cdot T(N-1) = T(N-1) + cN^2 - 2cN - cN^2 \cdot c - cN + c$$

Reduce

$$N\cdot T(N) - (N-1)\cdot T(N-1) = T(N-1) + 2cN$$

Isolate N * T(N)

$$N\cdot T(N) = T(N-1) + 2cN - (N-1)\cdot T(N-1)$$

$$N\cdot T(N) = T(N-1) + 2cN - (N\cdot T(N-1) - 1)\cdot T(N-1)$$

$$N\cdot T(N) = T(N-1) + 2cN$$

Divide by N

$$\frac{N\cdot T(N)}{N} = \frac{T(N-1)}{N}\,\frac{2cN}{N}$$

Telescoping start

$$T(N) = T(N-1) + 2c$$
$$T(N-1) = T(N-2) + 2c$$
$$T(N-2) = T(N-3) + 2c$$

.

.

$$T(2) = T(1) + 2c$$

Telescoping end

$$T(2) = T(1) + 2c$$
$$= T(1) + \sum_{i=2}^{N} 2c$$
$$= T(1) + 2c \sum_{i=2}^{N} 1$$
$$= T(1) + 2c(N-1)$$
$$= T(1) + 2cN - 2c$$

Remove constants

$$T(N) = 2cN$$
$$2cN = O(N)$$