1. Simple Calculator

The approached used is similar to *Dijkstra's Two-Stack Algorithm for Expression Evaluation* (p. 129). I created a stack for operands and one for values. Then I loop through an equation as a string and push/pop according to which is the current character of the string in the loop.

To make sure the code works for equations containing integers > 9, I used a while loop with an if test which concatenates the next element to a temporary string as long as the next element in the equation string is an integer.

2. Triplicates in four lists

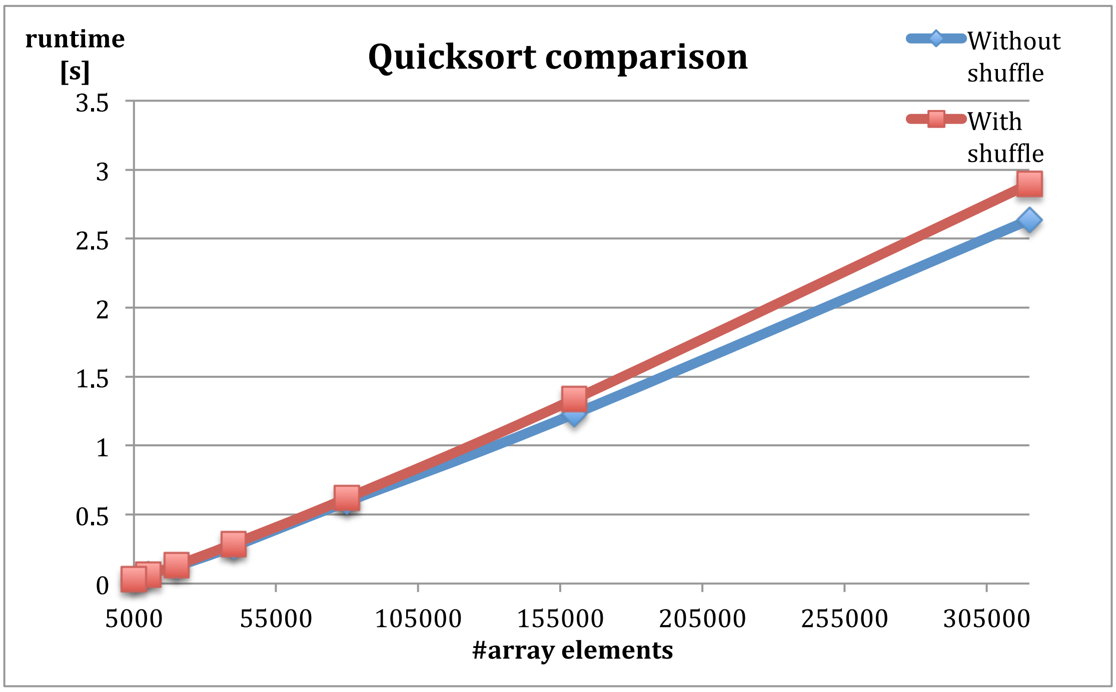
The approach used will Mergesort all four lists. Then do a for loop through the first list and binary search the remaining three lists for each name. The first time a name occurs 3 times the loop is broken. The procedure is repeated with a for loop for the seconds list. This is to consider instances where 3 similar names occur in lists 2-4 but not in list 1.

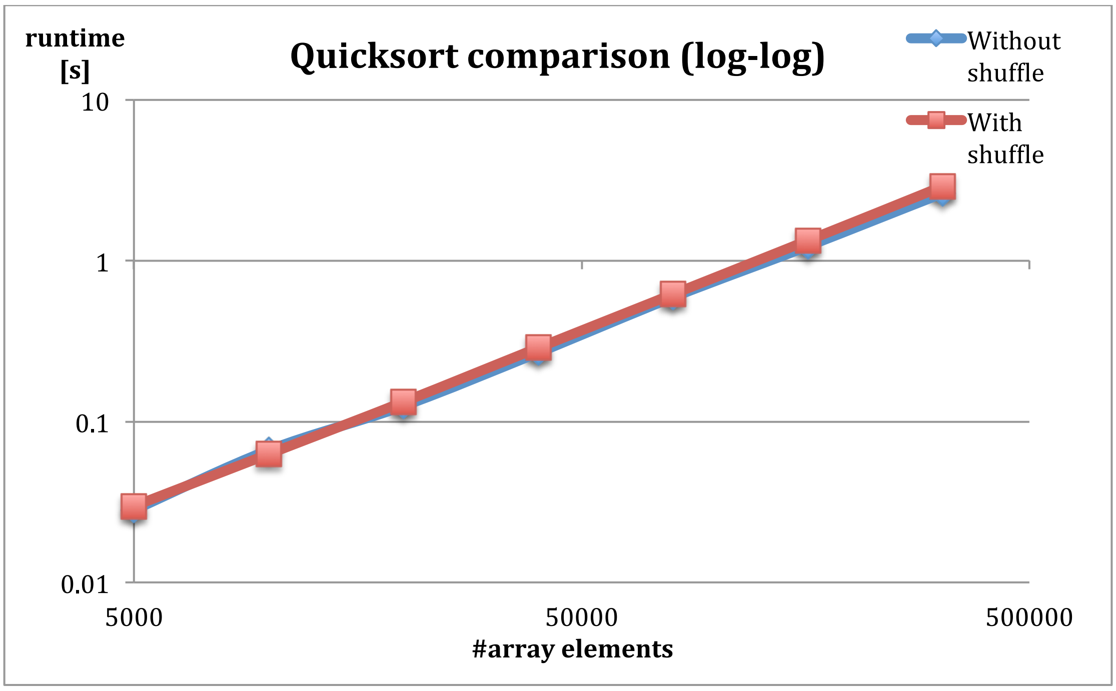
Analysis:
Mergesort guarantees max O(N*lg(N)). With Binary search we are guaranteed to have a maximum lg(N) search time. For each of the two for loops we do 3x binary searches. Each of the for loops will at worst loop through all N elements (names). Worst case scenario will then be poroportional to 2*N*3*log(N) which gives O(N*lg(N))

3. Quicksort, empirically

I used the doubling interval as suggested (5K, 10K, 20K, 40K, 80K, 160K, 320K) and ran normal Quicksort and Quicksort where the initial shuffle was removed. I first ran the code multiple times for each number of array elements (5k, 10k etc) but found that running them consequentially all in the same run was equally good but more elegant and thus I chose the latter approach. I did 40 runs for each different *N* and disregarded the first 10 runs since they consistently took longer time and skewed the results. The runtime for each *N* was takes as an average of 30 runs. I also turned of java compiler optimization because it caused unreliable results.

My hypothesis was that the initial shuffle is redundant and should make the algorithm run slower because the arrays are already randomized. The results supported this hypothesis:





Feeding the results into *LinearRegression.java* gave the following results:

Linear regression Quicksort	Slope (a)	Intercept (b)
with shuffle	1.102	−5.610
without shuffle	1.082	−5.541

We then can estimate the Quicksort runtime using $Y = aX - b$.

$$Y_{without\ shuffle} = 1.082 \cdot \mathrm{X} + 5.541$$

$$Y_{with\ shuffle} = 1.102 \cdot \mathrm{X} + 5.610$$

Here X is the logarithm of number of elements in the array. We want to express it with N instead. We need to find a function $f(N)$ such that

$$\log\big(f(N)\big) = a \cdot \log(N) - b:$$

$$\log\big(f(N)\big) = a \cdot \log(N) - b \Leftrightarrow$$
$$10^{\log(f(N))} = 10^{a \cdot \log(N) - b} \Leftrightarrow$$
$$f(N) = 10^{\log(N^a)} \cdot 10^{-b} \Leftrightarrow$$
$$f(N) = N^a \cdot 10^{-b}$$

Inserting our values from linear regression we get:

$$f(N)_{with\ shuffle} = N^{1.102} \cdot 10^{5.61}$$

$$f(N)_{without\ shuffle} = N^{1.082} \cdot 10^{5.541}$$

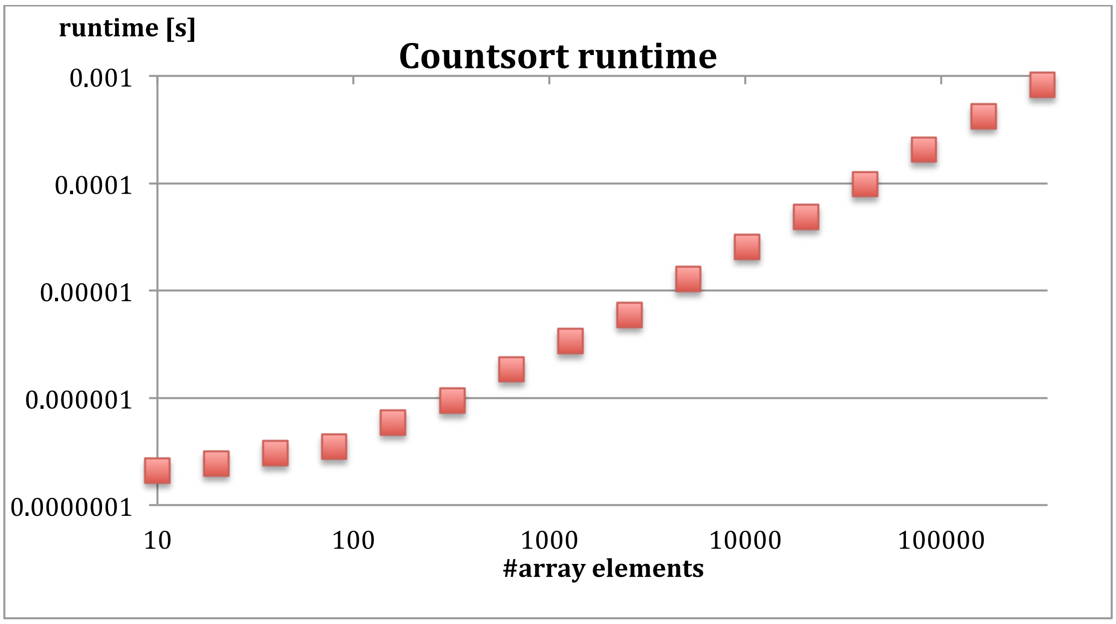
In the $f(N)$ function the a represents the slope of the running time vs. number of array elements in a log-log plot. **The constant b is related to how fast the particular computer runs the algorithm.** We see that the runtime is not too far from $O(N^{1.2})$ (the slope of a linearithmic algorithm in a log-log plot is ~1.2). The Quicksort with shuffle takes only slightly more time than without shuffle. We know that shuffling takes linear time and thus shouldn't make much difference, although we see from the linear-scale plot that the difference increases for very large N, which makes sense due to the exponential nature.

What does this mean?

4. Countsort

For Countsort I do the same as in the previous exercise. The algorithm was readily available on the world wide web: Countsort.

The difference in approach in this task compared to the previous is that I chose to do separate runs for each N. This is because the code ran very fast for low N and I had to make large adjustments to the number of runs for each N to get a good average.



The run time of Countsort is linear and proportional to the number of array elements *n* and *k* (the difference of the *max* (99) and *min* (0) array values).

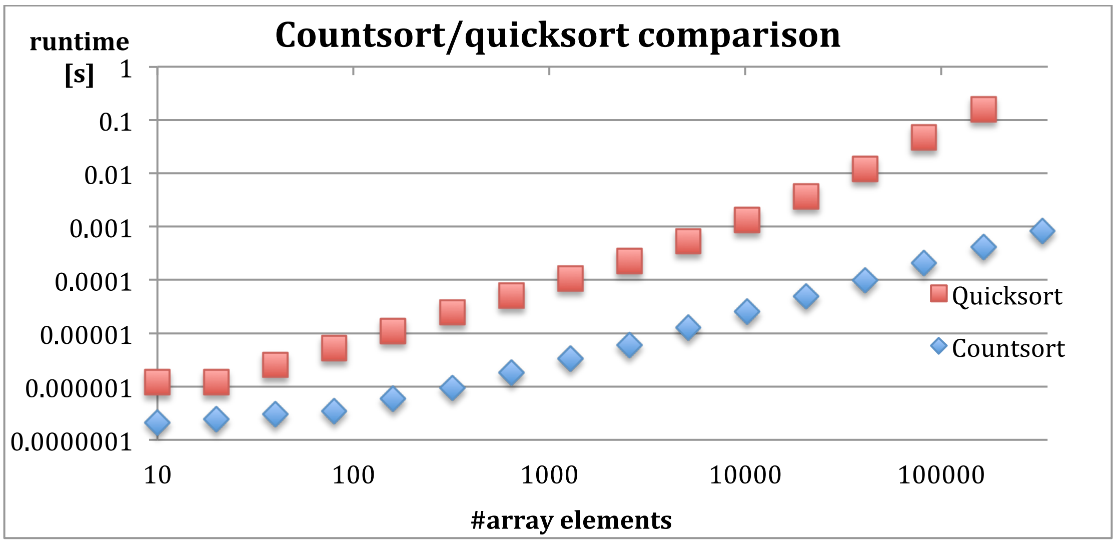
The count array will take *O(k)* time since it has to iterate through at most *k+1* elements. Populating the auxiliary array takes *O(n)* time. The total runtime is *O(n+k)*, and when *n>>k* we can omit k resulting in a linear runtime dependent on *n*. Consequently I would assume that the slope should have been less steep for *N<100*, but the opposite is the case. I don't have a good explanation for this.

For the linear regression I omitted the first 5 points and focused on the part for *N>>k.* This gave:

Slope: 0.852
Intercept: −8.086

To have a slope < 1 seems strange because it would mean the runtime is faster than linear. I had expected a slope closer to 1. The optimization (-Djava.compiler=NONE) was off so I don't have a good explanation to why the slope is < 1.

In general a linearithmic runtime is slightly slower than a linear runtime. To compare this linear profile with a linearithmic one I run the same program substituting Countsort with Quicksort to see how fast it runs.

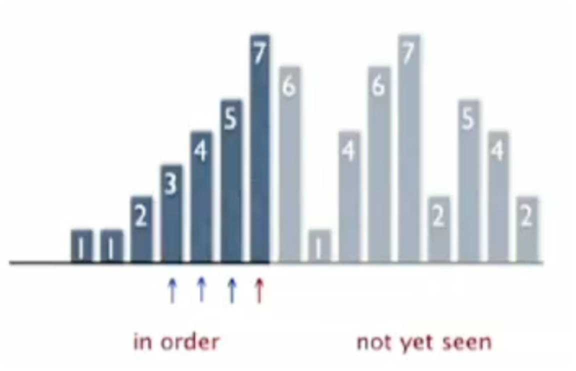


We see that the linear Countsort algorithm is significantly faster than the linearithmic Quicksort algorithm.

5.

a) Explain how insertion sort works and provide a pseudo code

Insertion sort iterates through the unsorted array with a nested for loop, and if the current element is smaller than the one to its left, the current element is exchanged with the one to its left. This makes sure that we will always have a left part of the array that is sorted, while the right part is unseen.



Pseudo code:

```
for(int i = 0; i < A.length; i++)
        for(int j = i; j>0; and A[j] < A[j-1]; j--)
                exchange(A[j], A[j-1] )
```

b) What is insertion sort's worst, best and average case scenarios in terms of number of exchanges and compares?

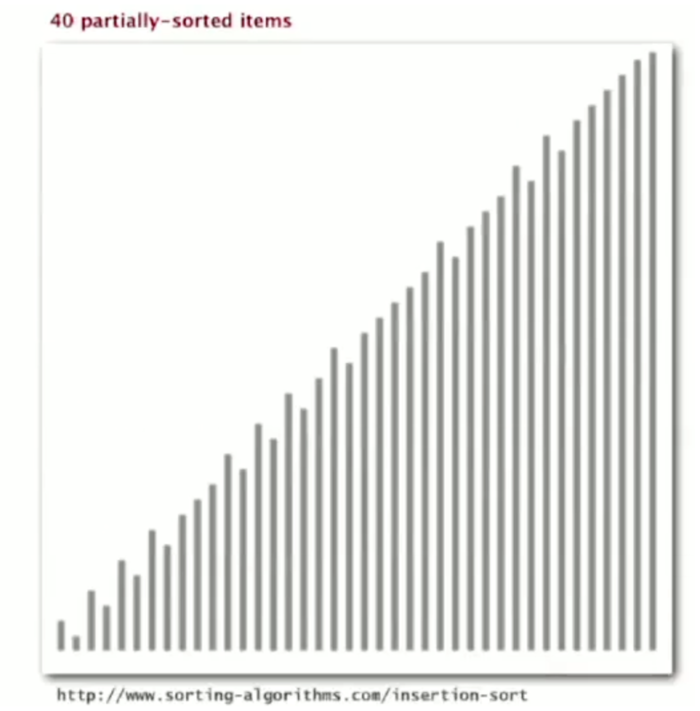
The worst case scenario is if the array is in descending order. For example [10, 9, ... , 0]. In this case the we have $\sim \frac{1}{2} N^2$ compares and exchanges.

The best case scenario is if the array is in ascending order (already sorted). All it does then is to validate that the array is sorted. It will only require *N-1* compares and 0 exchanges.

On average insertion sort will use $\sim \frac{1}{4} N^2$ compares and exchanges.

c) In which cases will insertion sort have linear runtime?

Insertion sort runs in linear time for sorted and partially sorted arrays.



We define an array to be partially sorted if it's number of inversions (pairs of keys that are out of order) is *<= cN* (i.e. linear).

This is because the number of exchanges equals the number of inversions (and compares = exchanges + (N - 1))