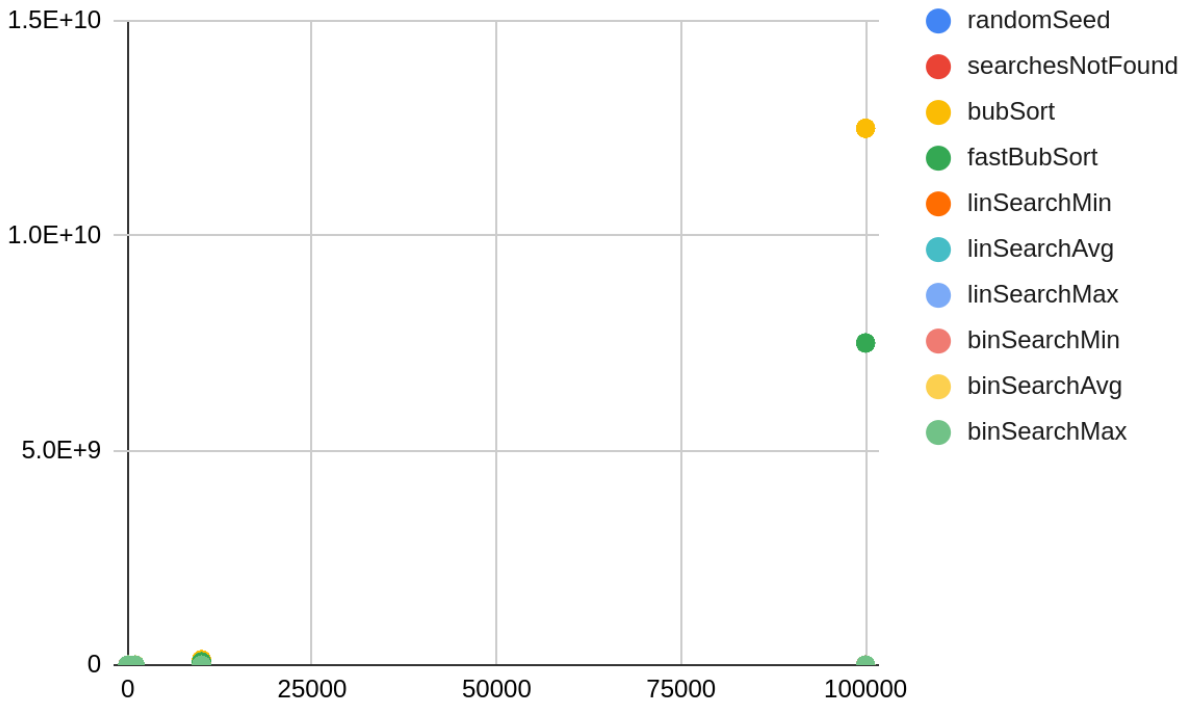
Analyzing the algorithmic complexity of some really bad algorithms

This is a short report that talks about how an algorithm's complexity varies with respect to the size of their inputs. A table of the data produced by our algorithms can be found in the same folder as this, in a file called "Results.pdf": it is not included in this PDF for readability reasons.

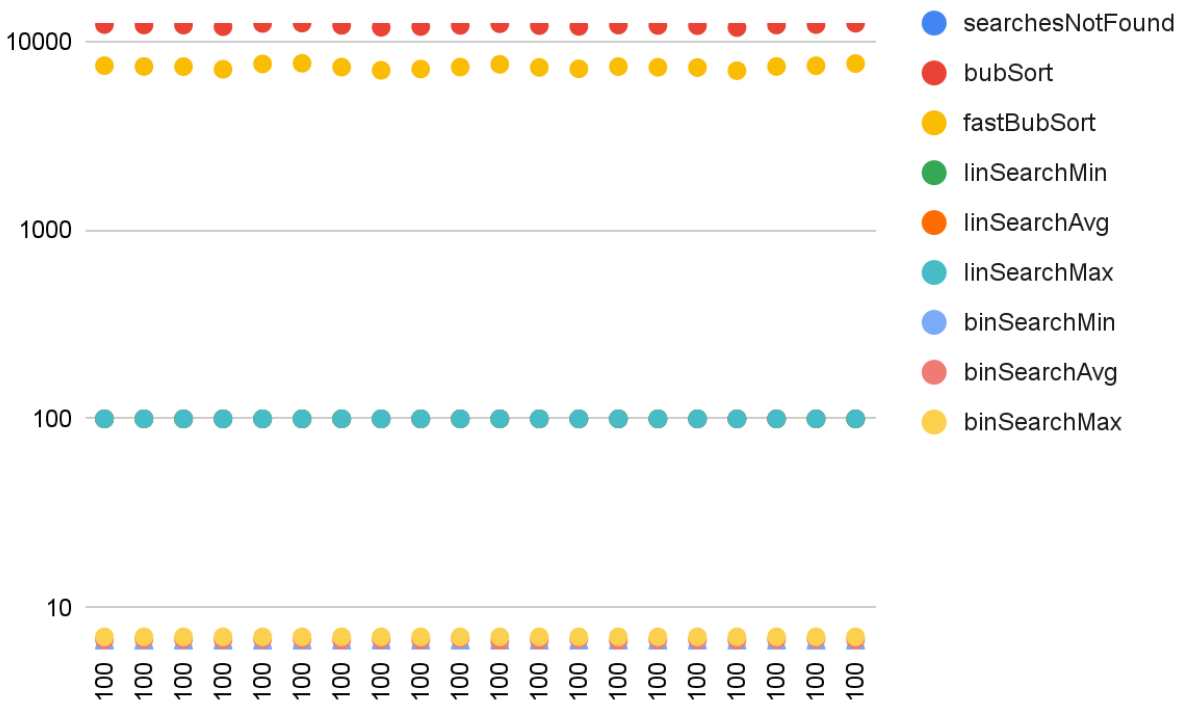
The algorithms, with a few exceptions, performed very poorly. Despite some significant optimizations to runtime speed, such as enabling multithreading, the algorithms still took quite a long time to process arrays with 100,000 elements. A trial involving 1 million elements is being run as we speak, but may not complete before this lab is done. Nonetheless, the trends in the data are extremely clear: the algorithms are really bad for big values.

For the sake of readability, I will be displaying most graphs in this document with one or both axes on a logarithmic scale. This somewhat obscures the trends from the naked eye, since exponential growth appears as simply linear. To justify this, please see the following chart, which shows all the data collected using no logarithmic scaling.



This chart clearly shows that both bubble sorts are really, really slow for a large array. However, almost no other information can be gained from this graph, since the entire rest of the dataset is crammed into a tiny area in the bottom left corner.

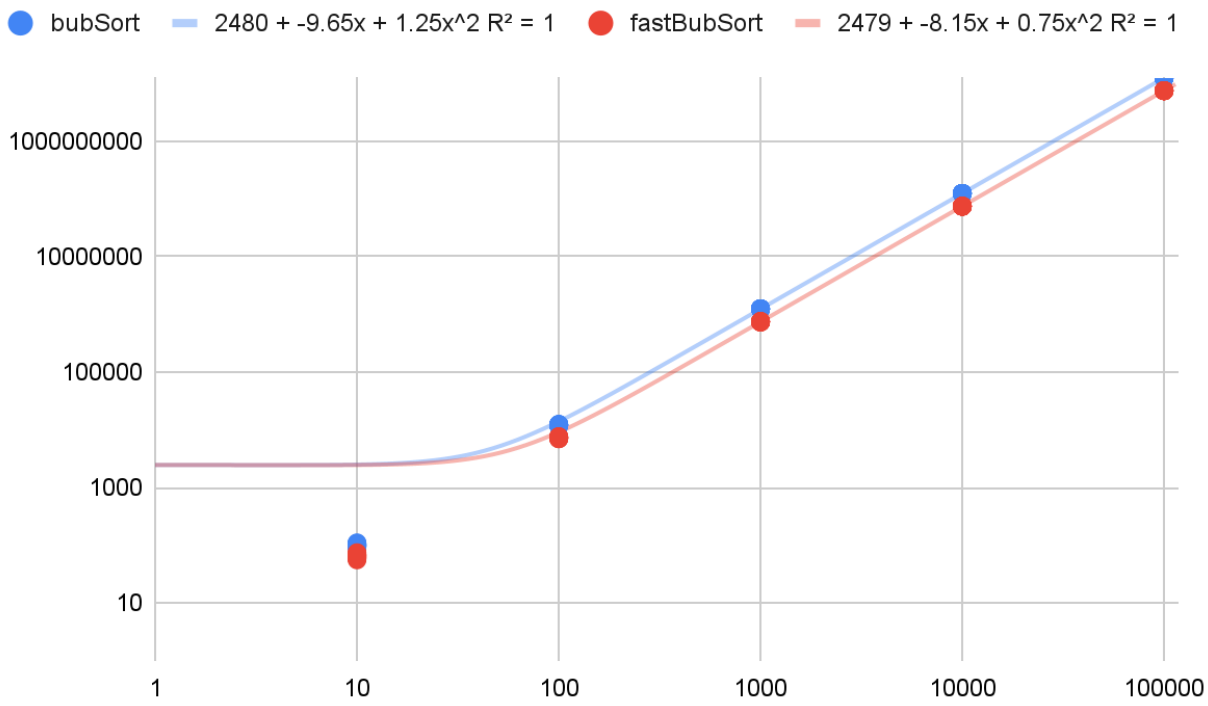
Now, lets briefly discuss how the algorithms performed for different random seeds. The following chart graphs all the values we examined against the random seed that produced them, for the 100-member array (chosen since it has all values in reasonable ranges)



This chart shows that small variances exist in the runtime for all the visible cateogories. This is due to the different contents of the lists and the searched values, which would change the degree to which the list was already sorted (which impacts the sorting algorithms due to the fact that I used a slightly different technique to annotate them), and whether the elements that were being searched for were generally larger or smaller than the list (impacting the direction that the BinarySearch would travel, and thus it's runtime. The only completely unaffected values are those for the Linear sort, because it was designed to be naive and not exit early if it passed the point in the list where the value should be, and the number of failed searches, because the value of 100 was too small for any duplicates to appear. Duplicates did appear for larger values of N, thanks to the Birthday Paradox.

While variance does exist in all the numbers, that variance is not terribly large: it is clear that the data involved doesn't significantly impact the runtime

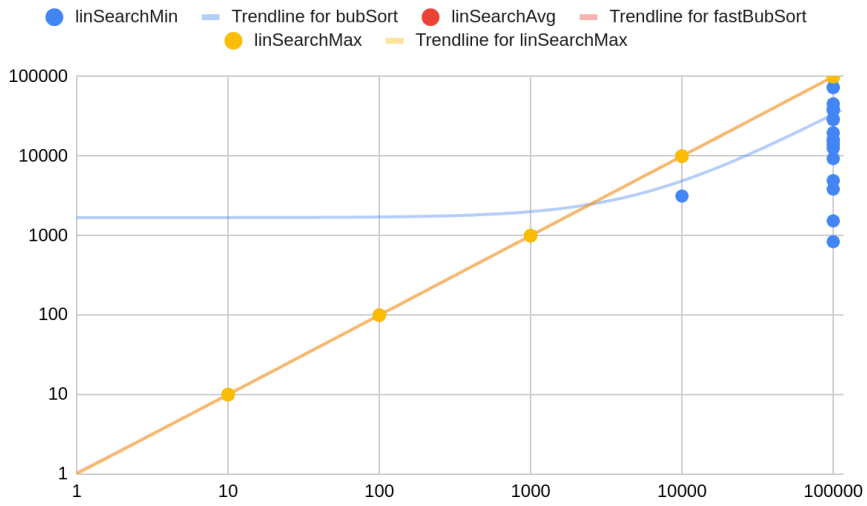
Now to look at how values change as we reach large values of N. This is the primary point of this lab: to investigate the impacts of large values of N have on runtime. First, the sorting algorithms are examined:



This graph shows how the varying sizes of array impact the number of iterations. While there is a distinct departure from the trendline for small values of N, the R^2 is one, indicating that there is a perfect fit. This departure likely reflects the limitations of Google Doc's trendline tools for data with such a large range. When the graph is fit against a third-degree polynomial instead, the coefficient of the cubic term is vanishingly small, indicating that the data isn't actually experiencing a cubic growth.

From this graph, we can conclude that both of the sorting algorithms, while differing greatly in actual runtime speed, have the same complexity class of $O(N^2)$. This makes them grossly unsuitable for any application involving a large number of elements: anything more than a few thousand elements leads to massive runtimes.

The next graph examines the linear sort, graphing the minimum, average, and maximum number of executed iterations against the number of elements.



This shows how the linear search algorithm runs in linear time, in the worst and the average cases. Notably, the best case (eg, the minimum number of iterations across the sampled data) experiences significant improvements for a large value of N. This is due to the birthday paradox. As N increases, the probability of any given target search element actually lying within the array increases greatly. This causes the algorithm to experience runtimes other than the worst case: which is otherwise the only runtimes that occur. Notably, the average case is not yet substantially affected in these trials: because the range of numbers is approximately 0-2 billion, the probability of these best-case occurrences remains low, which prevents them from having a significant impact on the average complexity.

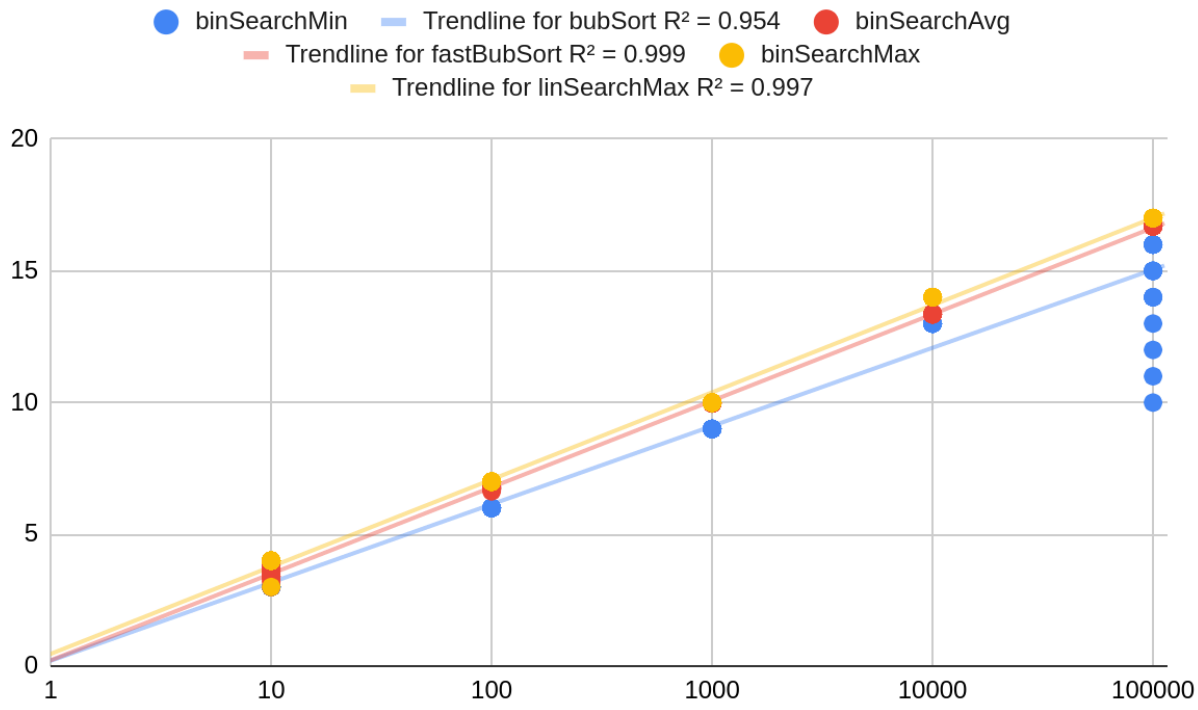
The linear search algorithm used in this lab can be improved substantially, although the improvements would not truly impact the complexity class. The linear search currently will continue through the list to the end, every time: however, because the list is sorted, it can drop out sooner if the current number is larger than its own number. While this would substantially help project runtime, it would not improve the complexity classification of the algorithm.

What would help overall runtime even more would be to plan ahead. Rather than scanning the entire list N times, needing to start from the beginning each time, it would be far more effective to only run through the list once, but check for every target element on that way. By sorting the list of targets, one can then iterate through both arrays in parallel, checking if the current element in the array of keys is greater than the element in the array of values. As the iteration passes the point in the values array where the current key would be found, it goes on to the next key. This reduces the complexity of the linear algorithm assessment from $O(N^2)$

(running an O(N) task N times) to O(N) (not including the costs of sorting the target array. Furthermore, it would make better use of the CPU caches, since the data would have much better locality: rather than needing a line last accessed during the previous run of the search, it would load each line once, work with it, and then never use it again.

Linear search is the best algorithm available to find a single element in an unsorted list. However, because we are looking in a sorted list, we can use a different algorithm for significantly better results.

The following graph shows the binary search algorithm against the size of the input. Notably, while the x-axis is expressed in a logarithmic scale, the y-axis is not: that is because of the efficiency of the algorithm, which never reaches the obscene runtimes of the other algorithms.



The fact that this algorithm uniquely scales in a non-polynomial fashion makes it far more effective than the rest. As before, with large values of N, the probability of being able to actually find the target element in the list greatly increases: depending on where the target element lies, that can lead to even better best-case runtimes. The fact that this element increases so slowly with respect to the input size makes it well suited for even massive-scale operations.