# Project One: Testing Running Times

Jean-Paul Nguyen

April 29, 2019

## 1 Introduction

### 1.1 Purpose

The purpose of this project is to examine and analyze different sorting algorithms with regards to different factors and properties, such as input size, input permutation, and treatment of data within the algorithm. This analysis will work with how each sort in question deals with the data, interesting behaviors, and overall what makes the fastest algorithm in sorting and why.

### 1.2 Experimental Setup

The way I implemented the sorting algorithms was by creating timing functions for each sort that took in the input size $n$, the number of repetitions $reps$, and a boolean to indicate if the input was fully randomized $fullyRandom$. These timing functions would return the given input size and the averaged time in seconds, which is calculated by running the algorithm on $reps$ many randomized permutations, then taking the average of those times. This straight average allowed all the timings to be consistently measured.

During the project, I focused on doing multiples of 10 for the input size, and doing 3 reps for all the sorts to keep them consistent. The maximum input size was kept between 10,000 and 100,000; this allowed enough times to spline for the data-oblivious sorts while not lasting too long for the data-sensitive sorts. From here, I graphed the times of each sort individually with the fully randomized and the nearly sorted inputs on a log-log graph; since most of these algorithms are greater than or approximately $O(n \log n)$ in time complexity, a log-log graph made sense to see the splines of the data. Each graph for the individual sorts has the timings of the different inputs with input size as the $x$-axis and timing in seconds as the $y$-axis.

### 1.3 Outline

The paper will delve into the following five sorts:

1. Bubble Sort
2. Insertion Sort
3. Shell Sort
4. Spin the Bottle Sort
5. Annealing Sort

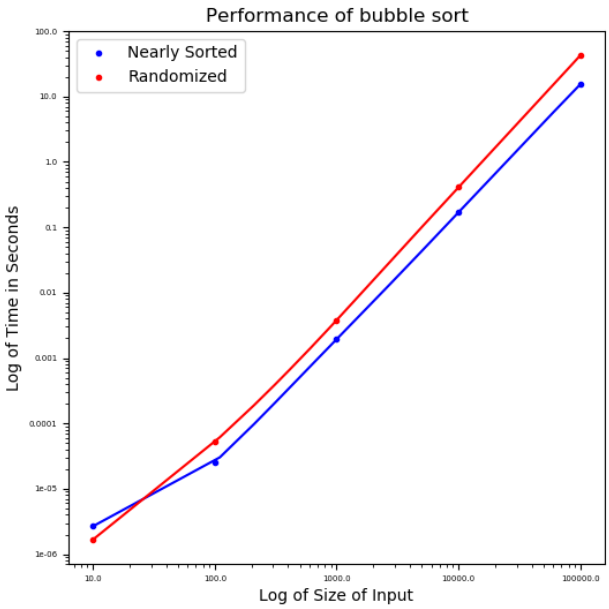
Then, the paper will go into comparing the sorts' performances. First, the data-sensitive sorts (Bubble, Insertion, and Shell) are compared. Then, the data-oblivious sorts (Spin the Bottle and Annealing) are compared. Finally, all of them are analyzed together, ending with an answer as to which algorithm happens to be the best.

# 2 Bubble Sort

## 2.1 Implementation

Bubble sort is the classic sorting algorithm where adjacent elements are repeatedly swapped if they are in the wrong order. For my particular implementation in C++, I had a short-circuited bubble sort; this optimization made it so that the algorithm would stop if there were no swaps done (i.e. the input vector is already sorted). Otherwise, the algorithm itself was the usual implementation.

## 2.2 Performance




| input | n | seconds |
|---|---|---|
| nearly sorted | 10 | 2.66667e-06 |
| nearly sorted | 100 | 2.56667e-05 |
| nearly sorted | 1000 | 0.001945 |
| nearly sorted | 10000 | 0.169301 |
| nearly sorted | 100000 | 15.4216 |
| fully randomized | 10 | 1.66667e-06 |
| fully randomized | 100 | 5.23333e-05 |
| fully randomized | 1000 | 0.003798 |
| fully randomized | 10000 | 0.409131 |
| fully randomized | 100000 | 42.78 |

Figure 1: Graph and Data of Bubble Sort

Here is a graph of bubble sort being run on varying input sizes, with two different types of input permutation: Nearly Sorted and Fully Randomized. The input goes from 10 integers to 100,000 integers. Along with that is a table of the data from timing bubble sort on the different input types and sizes.

## 2.3 Analysis

As seen here, as the log of the input's size increases, the log of time (in seconds) also increases in a linear fashion for both Nearly Sorted and Fully Randomized. Since bubble sort has a time complexity of $O(n^2)$ in the worst case, this is to be expected for a log-log graph.

Bubble sort is shown to be faster on nearly sorted input compared to fully randomized input by a good margin,

with a different of 27.35 seconds on the largest size, 100,000 integers. Bubble sort works by swapping out inversions, and in a nearly sorted input, there are less inversions, so less swapping happens. Note, though, that bubble sort still has to go through most of the input to check until there are no swaps done, hence why even though there is a sizable difference in the data, the log-log scale of the graph shows that the slopes of the two input types are similar.

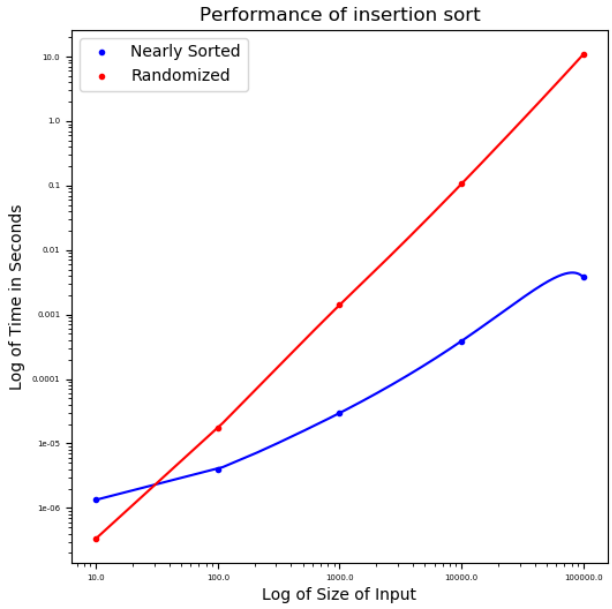
Overall, bubble sort is good enough for smaller inputs but may have trouble when the inputs become larger, such as after 100,000 items.

# 3 Insertion Sort

## 3.1 Implementation

Insertion sort is another classic sorting algorithm where the sorted list is built one element at a time. Each element is inserted to an already sorted list, and this happens until every element is added. The implementation of this algorithm is also the usual implementation, with no other optimizations made.

## 3.2 Performance




| input | n | seconds |
|---|---|---|
| nearly sorted | 10 | 1.33333e-06 |
| nearly sorted | 100 | 4e-06 |
| nearly sorted | 1000 | 2.96667e-05 |
| nearly sorted | 10000 | 0.000388 |
| nearly sorted | 100000 | 0.00376867 |
| fully randomized | 10 | 3.33333e-07 |
| fully randomized | 100 | 1.73333e-05 |
| fully randomized | 1000 | 0.00141033 |
| fully randomized | 10000 | 0.106451 |
| fully randomized | 100000 | 10.8947 |

Figure 2: Graph and Data of Insertion Sort

Here is a graph of insertion sort being run on varying input sizes, with two different types of input permutation: Nearly Sorted and Fully Randomized. Like bubble sort, the input goes from 10 integers to 100,000

integers. Along with that is a table of the data from timing bubble sort on the different input types and sizes.

### 3.3 Analysis

Similar to bubble sort, insertion sort has a time complexity of $O(n^2)$, but the algorithm does better than bubble sort with both the nearly sorted and randomized input. Compared to bubble sort, which took 15.4216 seconds on average with nearly sorted input, insertion sort took 0.00377 seconds on average, which is a significant improvement. Similarly, on fully randomized input, insertion sort beats bubble sort's 42.78 seconds on average with a time of 10.8947 seconds on average – this even beats bubble sort's performance on nearly sorted input!

This difference in performance is most likely due to how the algorithms work: On each pass, bubble sort is going through all the remaining unsorted values, while insertion sort is adding to an already sorted list, which is faster to do. As seen in practice, this different in passes make insertion sort work more efficiently than bubble sort in sorting the inputs.

In addition, from the graph, insertion sort does a really good job with nearly sorted input compared to randomized input, being faster on 100,000 items by a multiple of 1,000. This also shows on the graph, with the spline of the nearly sorted input having a more horizontal slope than the spline of the randomized sorted input. As such, insertion sort would be a good choice for nearly sorted input.

Overall, insertion sort is really good in practice for sorting, and in the search for good shell sort gaps, this algorithm is used as the baseline for whether the shell sort is good or not.

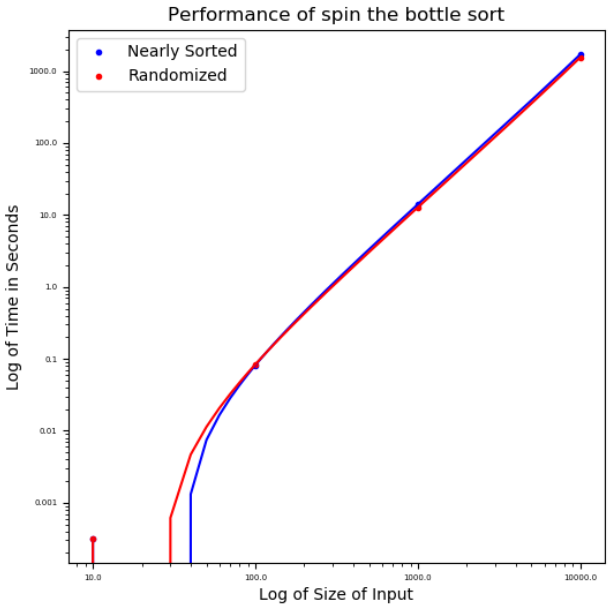
## 4 Spin the Bottle Sort

### 4.1 Implementation

Spin the bottle sort is an interesting sort that can be implemented as a data-oblivious input, i.e. the algorithm does not know about the given data that it is applied to save for the end. This means that as the algorithm is running, no information about the input can be confirmed (e.g. what parts of the input has been sorted and what has not). This is a result of the algorithm being very probabilistic and random in implementation.

To make sure that this algorithm was sorting correctly, I decided to check on each iteration if the input is already sorted; if not, then the algorithm continues to randomly choose and check for each element.

## 4.2 Performance




| input | n | seconds |
|---|---|---|
| nearly sorted | 10 | 0.000317333 |
| nearly sorted | 100 | 0.0818967 |
| nearly sorted | 1000 | 14.092 |
| nearly sorted | 10000 | 1712.16 |
| fully randomized | 10 | 0.000317 |
| fully randomized | 100 | 0.0840107 |
| fully randomized | 1000 | 12.7884 |
| fully randomized | 10000 | 1554.8 |

Figure 3: Graph and Data of Spin the Bottle Sort

Here is a graph of spin the bottle sort being run on varying input sizes, with two different types of input permutation: Nearly Sorted and Fully Randomized. The input goes from 10 integers to 10,000 integers. Along with that is a table of the data from timing bubble sort on the different input types and sizes.

## 4.3 Analysis

Compared to the other two sorting algorithms, spin the bottle sort does a very poor job. Theoretically, spin the bottle sort has a time complexity of $O(n^2 \log n)$, which is way worse than $O(n^2)$ of bubble sort and insertion sort. This is clearly shown in practice; the previous two algorithms took less than a second each (on average) for 10,000 integers, but spin the bottle took more than 1,500 seconds on average, which is worse by a huge amount!

In addition, the algorithm seemed to not have much of a difference in its timings based on the type of input. Both nearly sorted and fully randomized seemed to converge towards similar values in graphs and in the data. This may be a result of the data-obliviousness of the algorithm; since spin the bottle does not really know how far off it is from being sorted at any point except the end, the type of input would not affect its performance.

Overall, spin the bottle sort has poor performance compared to the previously-covered algorithms, but its property of data-obliviousness is shared with annealing sort. This means that spin the bottle can act as a baseline for whether the chosen temps and reps are good for annealing sort.

# 5 Shell Sort

## 5.1 Implementation

Shell sort is similar to insertion sort except that instead of checking each and every element, there are defined gaps that the algorithm checks in every pass. The idea is to improve the sorted list as the gaps get smaller and smaller, leading to less swaps being needed.

For this, I tried out two paths towards improving the gaps sequencing: dividing based on ratio and tweaking existing gaps in the research world.

## 5.2 Performance

In lieu of graphing out the plots, I compared different combinations of temps and gaps based on the times:

| sort | n | seconds |
|---|---|---|
| shell 0 | 10 | 3.33333e-06 |
| shell 0 | 100 | 1.93333e-05 |
| shell 0 | 1000 | 0.000450333 |
| shell 0 | 10000 | 0.0313853 |
| shell 0 | 100000 | 2.16697 |
| shell 1 | 10 | 1.66667e-06 |
| shell 1 | 100 | 9e-06 |
| shell 1 | 1000 | 0.000445333 |
| shell 1 | 10000 | 0.027163 |
| shell 1 | 100000 | 2.86163 |
| shell 2 | 10 | 1.66667e-06 |
| shell 2 | 100 | 2.53333e-05 |
| shell 2 | 1000 | 0.000690333 |
| shell 2 | 10000 | 0.0401673 |
| shell 2 | 100000 | 3.70179 |
| shell 3 | 10 | 2.66667e-06 |
| shell 3 | 100 | 1.7e-05 |
| shell 3 | 1000 | 0.000517667 |
| shell 3 | 10000 | 0.018404 |
| shell 3 | 100000 | 2.57416 |
| shell 4 | 10 | 2.33333e-06 |
| shell 4 | 100 | 1.33333e-05 |
| shell 4 | 1000 | 0.000429333 |
| shell 4 | 10000 | 0.0237763 |
| shell 4 | 100000 | 4.39538 |
| shell 5 | 10 | 3e-06 |
| shell 5 | 100 | 1.6e-05 |
| shell 5 | 1000 | 0.000823333 |
| shell 5 | 10000 | 0.0521097 |
| shell 5 | 100000 | 2.30307 |

| sort | n | seconds |
|---|---|---|
| shell 0 | 10 | 1.33333e-06 |
| shell 0 | 100 | 2.6e-05 |
| shell 0 | 1000 | 0.001391 |
| shell 0 | 10000 | 0.123547 |
| shell 0 | 100000 | 15.9025 |
| shell 1 | 10 | 7e-06 |
| shell 1 | 100 | 4.26667e-05 |
| shell 1 | 1000 | 0.001874 |
| shell 1 | 10000 | 0.125472 |
| shell 1 | 100000 | 12.2305 |
| shell 2 | 10 | 2e-06 |
| shell 2 | 100 | 4.03333e-05 |
| shell 2 | 1000 | 0.001695 |
| shell 2 | 10000 | 0.106047 |
| shell 2 | 100000 | 12.0835 |
| shell 3 | 10 | 3e-06 |
| shell 3 | 100 | 4.23333e-05 |
| shell 3 | 1000 | 0.002092 |
| shell 3 | 10000 | 0.155442 |
| shell 3 | 100000 | 13.6163 |
| shell 4 | 10 | 2.33333e-06 |
| shell 4 | 100 | 2.63333e-05 |
| shell 4 | 1000 | 0.00191167 |
| shell 4 | 10000 | 0.116026 |
| shell 4 | 100000 | 12.6638 |
| shell 5 | 10 | 2.33333e-06 |
| shell 5 | 100 | 3.73333e-05 |
| shell 5 | 1000 | 0.00201667 |
| shell 5 | 10000 | 0.129221 |
| shell 5 | 100000 | 11.5812 |

Figure 4: Data of Annealing Sorts

Shown in the figure is two tables of data; on the left is different annealing sorts on nearly sorted input, and on the right is the sorts on fully randomized input.

### 5.3 Analysis

The process on getting the gaps is the following:

1. The first iteration started out with $n/2$ and dividing by 2 for each gap. This is Shell's original gap sequence, and the performance was better than bubble sort by around 27 seconds. As such, I tried to improve on this.

2. This next iteration changed from dividing by 2 to dividing by 2.2. From what I read, most successful gap sequences had a ratio of 2.2 per gap, so I wanted to see that in action. Sure enough, performance improved by around 3 seconds for 100,000 fully randomized items on average.

3. Similar to the previous version, I tried out doing 2.25 for the divisor. There was some improvement, though it wasn't significant enough to continue going the path of just divisors.

4. This next one was trying out some of the experimentally derived gaps from research. In particular, this one was Ciura's gap sequence from 2001. Curiously, this did not work as well as I thought, doing worse than the two previous versions. As such, I decided to look for another sequence.

5. This sequence was the Incerpi-Sedgewick sequence from 1985, and like the previous version, there was not much improvement over the divisors (though it did fair out better than Ciura's). I decided to build up on this one.

6. Finally, this version was the previous sequence except with some tweeks on the numbers. I read that shell gaps seem to work better as coprimes, so I decided to make the sequence more pronounced with the coprimes. The gaps are 1, 3, 7, 22, 47, 111, 336, 861, 1969, 4592, 13776, 33935, 86961. The reason why coprimes would work better is because it avoids doing any swaps with gaps that have similar factors; this allows the shell sort to not do repeated work when checking and sorting.

While I was not able to beat insertion sort's timings, I did get close with the last version, being within 2 seconds from insertion sort's timing on 100,000 fully randomized inputs. From these, I chose the third and the last versions for my two shell sorts.

## 6 Annealing Sort

### 6.1 Implementation

Annealing sort takes the idea of spin the bottle sort and builds on top of it by making the random choices more limited and restrained. In each pass, for a given temp and number of reps, the algorithm does the same thing as spin the bottle for each integer on both sides.

While the algorithm is still data-oblivious (and arguably, more so than my implemented spin the bottle sort since it doesn't check if the array is sorted), these changes allow for more "educated" guesses and checks. The pool of what numbers can be randomly chosen is constrained compared to sort the bottle, and as the algorithm goes through the input vector, it is very probable for the vector to have been more sorted than before. That is why the sort is an annealing sort; each pass theoretically makes the input vector closer to the goal despite the algorithm not knowing any information as to how far it is.

Without much sources on annealing sort other than the papers of Dr. Goodrich, I implemented the basic annealing sort and made gradual changes on the temps and reps, seeing what parameter changes made for better performance.

## 6.2 Performance

In lieu of graphing out the plots, I compared different combinations of temps and gaps based on the times:

| sort | n | seconds |
|---|---|---|
| annealing 0 | 10 | 0.000615 |
| annealing 0 | 100 | 0.0584553 |
| annealing 0 | 1000 | 5.75689 |
| annealing 1 | 10 | 0.000328667 |
| annealing 1 | 100 | 0.0468077 |
| annealing 1 | 1000 | 4.91776 |
| annealing 2 | 10 | 0.000218 |
| annealing 2 | 100 | 0.0393153 |
| annealing 2 | 1000 | 3.86955 |
| annealing 3 | 10 | 0.000274333 |
| annealing 3 | 100 | 0.0355863 |
| annealing 3 | 1000 | 3.69378 |
| annealing 4 | 10 | 0.000233667 |
| annealing 4 | 100 | 0.0300763 |
| annealing 4 | 1000 | 2.93898 |

| sort | n | seconds |
|---|---|---|
| annealing 0 | 10 | 0.000905 |
| annealing 0 | 100 | 0.0580503 |
| annealing 0 | 1000 | 5.73475 |
| annealing 1 | 10 | 0.000391667 |
| annealing 1 | 100 | 0.051093 |
| annealing 1 | 1000 | 4.82083 |
| annealing 2 | 10 | 0.000223667 |
| annealing 2 | 100 | 0.043769 |
| annealing 2 | 1000 | 3.93344 |
| annealing 3 | 10 | 0.000282333 |
| annealing 3 | 100 | 0.0356583 |
| annealing 3 | 1000 | 3.75272 |
| annealing 4 | 10 | 0.000236333 |
| annealing 4 | 100 | 0.0280483 |
| annealing 4 | 1000 | 2.96095 |

Figure 5: Data of Annealing Sorts

Shown in the figure is two tables of data; on the left is different annealing sorts on nearly sorted input, and on the right is the sorts on fully randomized input.

## 6.3 Analysis

The process on how each iteration of annealing sorts is the following:

1. This version was just $n$ 0's for the temps and $n$ 1's for the reps. This actually worked incredibly well, already beating spin the bottle's performance on 1,000 elements by half the time. I decided to build off of this version to keep improving the performance in later iterations.

2. This version cut down the number of elements in the temps and reps to half, and I decided to try out using three phases that Dr. Goodrich had described for the $O(n \log n)$ version of annealing sort in his research, except with three phases of my own:

   (a) The first phase was all $n$'s for temps, 2's for reps.

   (b) The second phase was $\log n$'s for temps, 2's for reps.

   (c) The third phase was 0's for temps, 1's for reps.

   Here, I made it so that temps was always decreasing towards 0's, starting with $n$ and then going down to $\log n$. Since the pool of choices for randomization was larger, I increased the reps for them as well. These changes proved to be effective, taking off about a second from the first version's results on 1,000 elements.

3. I decided to keep the temps the same and change only the reps from the second row from 2 to 1, which actually cut the time down further by a second. This showed to me that most of the optimization can be done by doing smaller numbers of reps as long as the temps are enough. I kept this in mind for later versions.

4. I changed how long each phase was; before, the phases were a third of the total number of temps and reps. Now, the first and third phases were cut down to quarters, and the second phase was extended to last half. With the reps from the previous iteration, this again lessens the number of reps done overall while keeping the high probability of the large temps in the first phase to do enough work. This improved the times again, though only by half a second.

5. Finally, the last version had the reps all replaced by 1's again. Since the pattern was that the reps were the main cause for so much of the time being used, I decided to just decrease all the reps back down. This worked pretty well; the first phase and second phase do a good "annealing" with high probability, allowing the final phase to smooth things out in the end. This again improved the times by just a half second.

With these results, I decided to settle on the last two versions of annealing sort as my choices. While not at the same caliber of speed as the data-sensitive algorithms, my annealing sorts are faster than spin the bottle by a good amount, which is what I was aiming for.

# 7 Comparison of Sorts

## 7.1 Data-Sensitive Algorithms

Here is the comparison of bubble, insertion, and my two shell sorts that I chose:

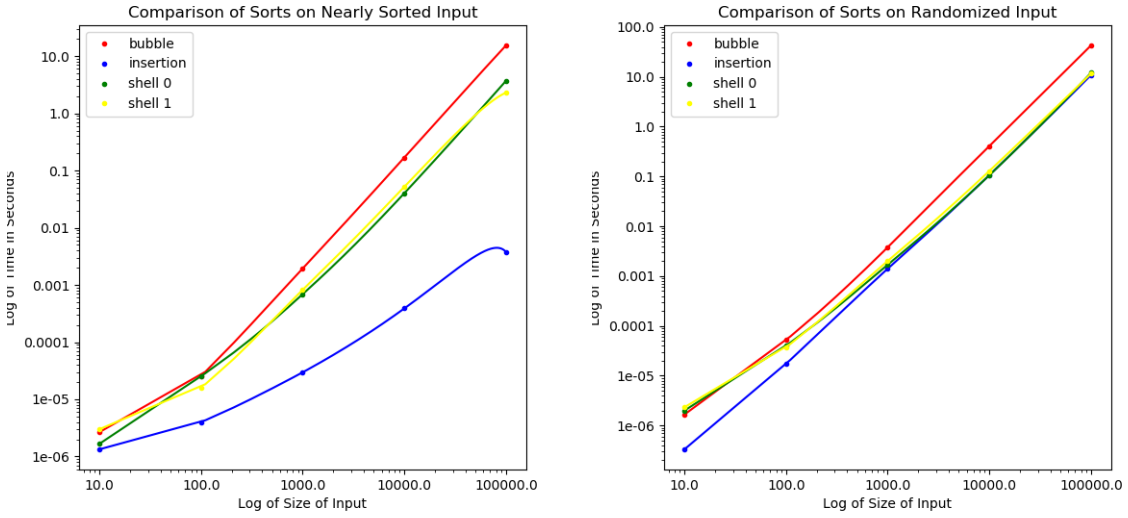



Figure 6: Graph of Bubble Sort, Insertion Sort, and Shell Sorts

Here, insertion sort clearly is the best when it comes to nearly sorted input, and there is no competition there. Bubble sort and the shell sorts will always do extra work on nearly sorted algorithms; the bubble sort will have to check every unsorted element on each pass, and the shell sorts have to go through all the gaps before it reaches one. Since insertion sort is essentially a shell sort with only 1 as the gap, insertion sort skips all of the shell sort work and is more efficient on elements that are almost sorted.

On the fully randomized input, the performances are more similar, with insertion sort only barely beating shell sort 0 and 1. Based on the splines, it is possible that with larger inputs, the shell sorts may do a better job than insertion sort, though I was limited by time from actually getting to see if that was true. Overall,

insertion sort did beat these algorithms in both types of input, with a potential for the shell sorts to do better with larger inputs.

### 7.2 Data-Oblivious Algorithms

Here is the comparison of spin the bottle and my two annealing sorts that I chose:

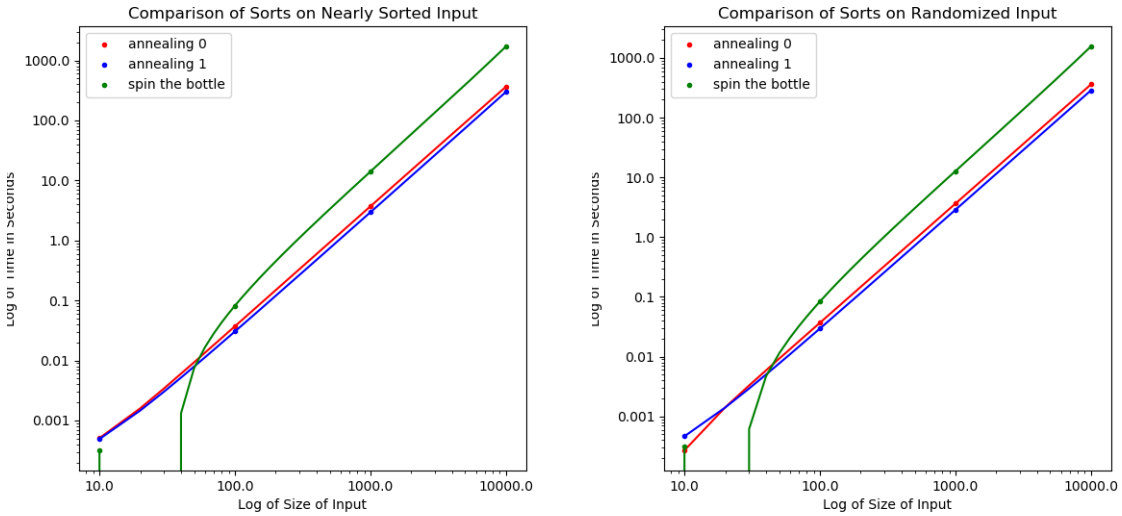



Figure 7: Graph of Annealing Sorts and Spin the Bottle Sort

Here, you can see that the annealing sorts perform better than spin the bottle on both the nearly sorted and the fully randomized input. Due to the data obliviousness, the times for the two types of input are similar, hence why there is not much difference in the graphs. The slope of the spline for spin the bottle is larger than the annealing sorts as well, so we would expect larger inputs to take even more time for spin the bottle compared to the annealing sorts.

## 8 Conclusion

Based on the data alone, it is clear that the data-sensitive algorithms are better than the data-oblivious algorithms. I believe this is because the information of knowing how far the algorithm is in sorting the input can be useful for data-sensitive algorithms to make better assumptions. The data-oblivious algorithms are very probabilistic, which makes them have less of a guarantee to know for sure if the checks and changes are going towards the right solution.

In the end, insertion sort still ended up performing the best among all the sorts, though shell sort did have the potential to be better. If I had better used my time on investing in looking for better shell sorts, it may have been possible to develop a gaps sequence that could beat even insertion sort at least on the fully randomized input.