# How Long is this Gonna Take?

Undergraduate students are surprised to learn that as much intellectual energy has been invested in sorting and searching as almost any other part of Computer Science. Think of Duke Energy's customer database—it’s huge. New customers have to be added, former ones deleted, bills must be sent out, customers send in their payments and inquire about their accounts. An efficient data organization is required for Duke to function at all. The first attack on organizing data involves sorting data elements into some order, and exploiting that order when trying to retrieve a particular element.

Hundreds of sorting algorithms have been developed, and like all sorting algorithms, Selection Sort accomplishes its task by making comparisons and data movements. We often compare algorithms by counting the number of comparisons and movements required—the fewer the better. This begs the question, how many comparisons and movements does the Selection Sort make? And, are these actions affected by the initial arrangement of data values in the array? This is the focus of this lab.

Start with the *SelectionSort* class in the zip file attached to this item. Keep the name *SelectionSort*, and add a *main* method to it.

* Modify the *selectionSort* method to have two counters, one for the number of comparisons, and one for the number of data swaps. Each time two data elements are compared (regardless of whether the items are in the correct order—we're interested in that a comparison is being done at all), increment the comparison counter. Each time two data items are actually swapped, increment the data swap counter.
* At the end of the *selectionSort* method, print the size of the sorted array, and the counters. (Be sure to identify which counter is which in your print message
* In your *main* method,
  + Declare a *final int*, NUM\_ELEMENTS. Initially set NUM\_ELEMENTS to 10 to debug your program.
  + Declare and create three *double* arrays of NUM\_ELEMENTS length, *lo2Hi, hi2Lo, random.*
  + Initialize the first array, *lo2Hi*, with values 1.0, 2.0, …, NUM\_ELEMENTS
  + Initialize the second array, *hi2Lo* with values NUM\_ELEMENTS, 24.0,…, 1.0
  + Initialize the third array, *random*, with random *double* values between 0.0 and less than 1.0
  + call the *selectionSort* method using each array. (Note: you might want to print the array elements themselves for debugging purposes when NUM\_ELEMENTS is small, but you’ll not want to print them with larger values for NUM\_ELEMENTS.)
* Run your program three times with different values for NUM\_ELEMENTS: 1000, 2000 and 4000.

In your submission write some text describing the relationship between the number of comparisons of the various values of NUM\_ELEMENTS. For example, what do we find if we divide the number of comparisons for 2000 elements by the number of comparisons for 1000 elements? What do we find if we divide the number of comparisons for 4000 elements by the number of comparisons for 2000 elements?

Epilog: As you can tell, Selection sort doesn’t scale very well. The number comparisons increase quadradically as a function of number of elements. There comes a point that, because of array size, it’s impractical to use Selection sort. The good news is there are hundreds of sorting algorithms. Some suffer from the same performance shortcomings as Selection sort, but others that are almost “magical” in that increasing the number of elements has minor impact on performance. If you’re interested, take a look at chapter 23 Sorting.

## Grading Elements

* use class *SelectionSort* with the *selectionSort* method
* instrument *selectionSort* with *comparisonCnt* and *swapCnt*, and print the number of array elements and the *comparisonCnt* and *swapCnt* values
* write a *main* method with *final int NUM\_ELEMENTS*
* declare and create three *double* arrays of length NUM\_ELEMENTS
* initialize the arrays as specified
* call *selectionSort* with each array
* Run the *SelectionSort* with different values for NUM\_ELEMENTS: 1000, 2000 and 4000.
* Document the ratio of *comparisonCnt* values for 2000 elements and 1000 elements.
* Document the ratio of *comparisonCnt* values for 1000 elements and 2000 elements