Sorting Project 4

##Placeholder

Sorting

Some of the issues I ran into included not having the one of the iterators equal to the pivot.

In otherwords,

```
while(true){

      while(data[start] < pivotValue){
      // while(data[start] <= pivotValue){
        start += 1;
      }

      while(data[end] > pivotValue){
        end -= 1;
      }

      if(end - start <=  kStop) {
        break partition;

      }else{
        //the pointers have met or crossed.
        data = swap(data, start , end );
        start += 1;
        end -= 1;
      }
    }
```

as above in line 3+4. This created an odd situation where the pivot was of neither set, and if the pivot was the first item in a partition, as in the case of the first pivot, then it would just flip back and forth between start and end.

Yet, many of the examples I looked at did no place bounds on the first inner while loop. This in turn created its own situation. Where now, the pivot was the first value. The first check, checks whether pivot is >= than start. It is, it is equal. So it passes, is on the low. But if this value is the highest value in the set, the start pointer shopuld continue all the way to the end.

Data

To generate our own data, I spent a lot of time. One one hand, some of the options are quite trivial. A completely random string of integers can be obtained by `Math.random()`. Though managing the range can be a bit irksome here. A data set with ascending data (saying nothing about the presence or abscence of duplicates) by simply doing a typical for loop:

```
int min = 1, max = 1000, count = 100;
int range = max - min;
int seed = 1;

  printWriter = new PrintWriter(
        new BufferedWriter(
              new FileWriter(
                    newFile
              )
        )
  );
  count =  (count > max - min) ? max : count;
  for( int i = min ; i < count ; i++ ){
     printWriter.println(i);
  }
```

Yet this has several issues. There is no control over the distribution. For instance, calling for a range of values, 0 - 1000, and callling for 100 values. Using the above method, all you can do is count from 1-100. Maybe you get a bit more clever and choose some arbitrary value that is `initialVal < max - count`. Still, your distribution is not good at all, just in the same distribution. You could get a bit more clever, and make the iterator scaled for the range: such that `(max -min) / count` to find a step value.

Now your distribution is flat, but its very orderly. Realistically, this would be ok for this assignment, but its not very interesting.

When I was trying to come up with a way to generate randomly distributed values, but without dublicating values. I tried a large number of different ways, considering arrays and other things. At some point I came across the `Streams` class, and the extending class `IntStream`. I had come across these before, but they seemed rather opaque to me. I played around with one for a while, eventually finding a way to use it. Initally, I tried to use the `Math.random()` method, and scaling that value. Then, I eventually recalled that I had used another way in the past, though it took some time to recall the class I was thinking of- the `Random` class.

```
Random rng = new Random();
```

The available constructors here allow you to set **min, max**, and **count**, which is perfect for this usage. Determining, the returning class, and how to use it was difficult, given my lack of experiance with Streams. The

`Random` class, is fairly easy to use, simply requiring `<instance>.nextInt()`. That is the easiest way to use the class. Calling:

```
int seed = (int)Math.round( Math.random() * 1000 ); //poss. values 0-1000;
Random rng = new Random(seed);
IntStream intstream = rng.ints(count, min, max);
```

returns an `IntStream` object. I started writing the code to implement the different data types that exist in the files. First I created an enum to represent the types of Data that we have, based on the given data.

```
public enum DataType{
   ASCENDING,
   DESCENDING,
   DUPLICATE,
   RANDOM,
   REVERSE,
   UNSPECIFIED;
}
```

There are a few extra types here, that I initially thought might be useful. Much later than I care to admit, I realized that `DataType.DESCENDING` and `DataType.REVERSE` were the same thing, which is why it exists in here. In practice, it makes the most sense, to have 3 types, `DataType.DESCENDING`, `DataType.ASCENDING`, and `DataType.RANDOM`, and an additional boolean parameter `boolean allowDuplicates`, as in reality that better covers the types of files that might exist, and in some ways this is what I've implemented. As the given data stands however, it seems that files of type `DataType.DUPLICATE` implies `DataType.RANDOM`.

In addition, `DataType.RANDOM` implies that there are no duplicates, as do the other two files: `DataType.REVERSE` and `DataType.ASCENDING`. The type -`DataType.UNSPECIFIED` was added as a sort of catch all. I didn't go back and change these, so we ended up with wuith the break down in the code block below. I set about trying to implement each of these methods, individually (notably, this is when I realized descending and reverse were the same, and realized that duplicate needs to be specified uniquely.) After writing a few of these, I realized the implementation was basically the same for each of the types. So instead of having a method for each type, laboriously written, I just needed an iterator for each. I wrote the method

```
    public static Iterator<Integer> getSpecifiedIntegerIterator( int count, int min, int
                                                        boolean allowDups,
                                                        DataType dataType ){ }
```

There were several caveats as I worked trough creating iterators for each of these. Most of these were easy, notably `DataType.DUPLICATE`, as it has the fewest requirements. Notably, there is no requirement that

duplicates exist. Just that they may exist. If `(max - min) >> count`, then it is entirely possible that there may be no collisions.

Using the `.distinct()` method returns a stream of *unique* integers. Though testing reveals a new issue. Setting the specified size in the `.ints()` constructor for `IntStream`, the returned iterator comes up short. This makes sense, as it is removing duplicates. It is easy enough to leave this open ended for generation, and just close it when we are done.

Using the `.sequential()` method, does not have the expected effect. You might expect that these values be sorted, but they aren't by necessity. The issue is the multithreaded nature of the program. The values are partially in order, in varying intervals, due to the multithreaded nature of the `Streams` class

```
switch( dataType ){
   case DESCENDING:
   case REVERSE:
      if(allowDups) return streamofIntegers.sequential().map( i -> max - i - 1 ).iter
      else return streamofIntegers.distinct().sequential().map( i -> max - i - 1 ).it
   case ASCENDING:
      if (allowDups)return streamofIntegers.sequential().iterator();
      else return streamofIntegers.sequential().distinct().iterator();
   case DUPLICATE:
      return streamofIntegers.iterator();
   case UNSPECIFIED:
   case RANDOM:
   default:
   //implies no Duplicates?
      return streamofIntegers.distinct().iterator();
}
```

For example, evaluating

```
Iterator<Integer> itr  = new Random( seed )
                           .ints( min , max )
                           .boxed()
                           .sequential()
                           .distinct()
                           .iterator();

//returns:

1360 948 3029 1447 3515 1053 4491 4761 3719 2854 1077 2677 2473 4262 1095 3844 84 2875
```

These are not ordered as you can see. I eventually noticed that `.sequential()` meant in terms of threads, presumably as the opposite of `.parallel()`. The method `.sorted()` failed when I tried it the first few times. The program would hang and would not complete. As I was running the program from within the test framework (JUnit 5) determining the error was difficult as my application didn't start until around 25 frames deep and the number of extraneous wrapper classes and reflections that surround each class obfuscate what is going on. I realized, that what seemed to work well earlier- letting the Stream have infinite length.