Logbook 1 (Weeks 1-7)

https://github.com/ZMRamsey/APDCoursework.git

The full package, including a copy of this logbook, is stored in this git repository.

Week 1/2 - Randomiser

The randomiser package creates an array of non-repeating integers, which are used by the searcher methods to find the kth element within them.

I created three versions of the CleverSearcher class that have slightly different features.

CleverSearcher was the first to be made, and uses a sub array of size k, which stores the k largest elements seen at that point in the array. For the first k ints, this simply copies each value it comes across. From this point, it runs across the rest of the main array testing each int. If an element is found that is larger than the smallest element in the sub array, the new int overwrites it. Then, the sub array is sorted using the standard Arrays.sort() method.

Full Clever Searcher (First Half)

```
public int findElement() throws IndexingError {
    int[] fullArray = getArray();
    int k = getIndex();

    if (k <= 0 || k > fullArray.length) {
        throw new IndexingError();
    }
    int testingElement;

    if (k <= (fullArray.length / 2))
    {
        Integer[] testingArray = new Integer[k];

        for (int i = 0; i < k; i++) {
            //For first k elements, it isn't tested and just fills the test array
            testingArray[i] = fullArray[i];
        }

        Arrays.sort(testingArray);
        //find kth largest
        for (int i = k; i < fullArray.length; i++) {
            testingElement = fullArray[i];
            for (int j = 0; j < testingArray.length; j++) {
                if (testingElement > testingArray[j]) {
                    if (j != 0) {
                        testingArray[j - 1] = testingArray[j];
                    }
                    testingArray[j] = testingElement;
                }
            }
        }
        return testingArray[0];
    }
```

This class isn't the most efficient way to find the kth element, since it relies on fully sorting the sub array. In extreme cases with vast arrays, this can severely delay the program compared to alternatives. Despite this, it works well, passing every test.
FullCleverSearcher improves on the idea of CleverSearcher by more efficiently sorting the sub array. Instead of sorting the sub array every time a new large element is found, the element is instead compared to the next largest element in the sub array. If it is larger, they are swapped. This

means that the sub array stays perfectly ordered, and any sorting can be more focused on just the elements that need to be changed.

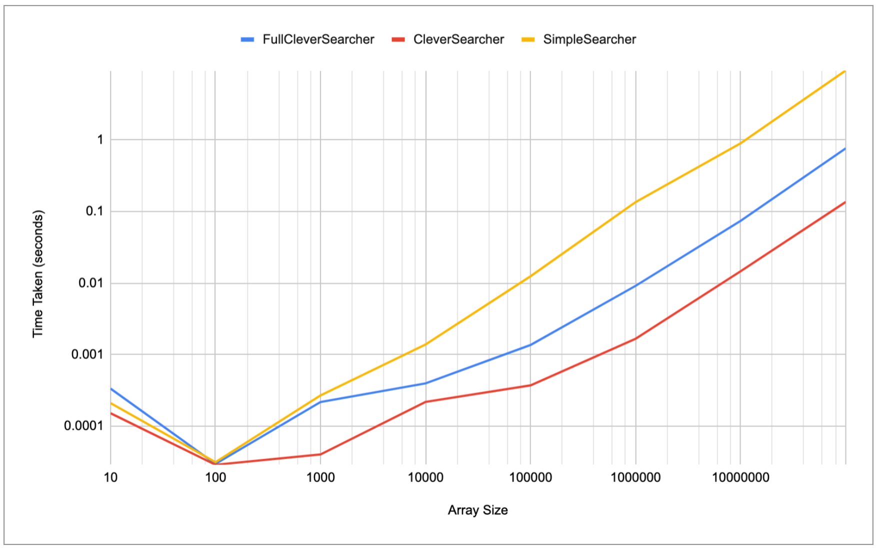
FullCleverSearcher also introduces another improvement. If k is greater than half the size of the full array, it will instead find the length - k smallest. For example, it is quicker to find the 10th smallest of 100 rather than the 90th largest, since it requires a smaller sub array and so less sorting. In the code sample above only the first half, when k is less than half the size of the full array.

GenericSearcher, the final searcher, uses generic typing to allow it to find the kth element in any comparable array of objects. The algorithm is the same as FullCleverSearcher, using a k size sub array and swapping until the elements are in the right positions. This searcher had to be tested with Integers instead of ints due to the latter's incompatibility with generics.

testMinusNumberError and testOutOfBounds

```
@Test
void testMinusNumberError() throws IndexingError {
    try{
        testSearcher(50, -1);
         fail("Expected indexing error didn't occur");
    } catch (IndexingError indexingError)
    {
        //Successful test
    }
}

@Test
void testOutOfBounds() throws IndexingError {
    try{
        testSearcher(100, 103);
        fail("Expected indexing error didn't occur");
    } catch (IndexingError indexingError)
    {
        //Successful test
    }
}
```

Each searcher has a relevant jUnit test which extends the standard SearcherTest. I added additional tests to the SearcherTest class to make sure the searchers were being checked for everything. This included two methods, testMinusNumberError and testOutOfBounds, to make sure that error handling was correctly implemented.



I also created timer classes for the different searchers to see which was most effective at different array sizes. I plotted this data on a graph, using logarithmic axes due to the range of the results. The simple searcher was the least efficient, and drastically rose on the last value. Interestingly, FullCleverSearcher was less efficient than the standard clever searcher. I believe that part of is because FullCleverSearcher takes into account the size of k, but in the timer classes k is always 5, which nullifies the advantage. All searchers performed best at an array size of 100, which seems to be in the perfect spot that it is large enough for searching algorithms are necessary, but not so large that it takes a long time to parse through the array.

Week 3/4 - Generic Swap

The swap function is relatively simple, utilising the standard technique of having a temporary variable to store one of the values.

Generic Swap Function

```
public static <T> T[] swap(T[] array, int var1, int var2)
{
    T handler;
    if (var1 < array.length && var2 < array.length) {
        handler = array[var1];
        array[var1] = array[var2];
        array[var2] = handler;
    }
    return array;
}
```

I also created two unary predicates. The first, IsOdd is incredibly simple at only one line long. The second, IsPrime, is more interesting. First it removes 0 and negative numbers which can’t be prime, then integers 2 and 1 which wouldn’t work with the main test and are checked separately. From there, it removes even numbers before running through every number from 3 to the square root of the test number. This is because the highest possible factor a number can have is its square root. If a factor is found, it cannot be prime, and it returns false.

IsPrime Unary Predicate

```
public boolean test(Integer n) {
    if (n > 0) {
        if (n <= 2) {
            //if n is either 2 or 1, return true for 2, false for 1
            return n == 2;
        } else {
            //Check if even, if it is return false
            if (n % 2 != 0) {
                //Run through ints from 3 to Sqrt of number to find factors
                for (int i = 3; i < java.lang.Math.sqrt(n); i++)
                {
                    //Check if i is a factor of n
                    if (n % i == 0)
                    {
                        return false;
                    }
                }
                //If no factors are found, it's prime
                return true;
            } else {
                //Return false if even
                return false;
            }
        }
    }
    //Returns false if less than or equal to 0
    else { return false; }
}
```

Week 5 - Sorting

Selection sort works by parsing through the array, finding the next smallest value, and placing it within a sorted array at the left of the main array. This sorting algorithm is generic to make it applicable to arrays of any comparable types. I tested it with both characters and integers to get the data for the speed graphs.

Generic Selection Sort

```
public T[] sort(T[] array) {

    for (int i = 0; i < array.length - 1; i++)
    {
        int index = i;
        for (int j = i + 1; j < array.length; j++){
            if (array[j].compareTo(array[index]) < 0){
                index = j;
            }
        }
        T smaller = array[index];
        array[index] = array[i];
        array[i] = smaller;
    }
    return array;
}
```

Generic Quick Sort

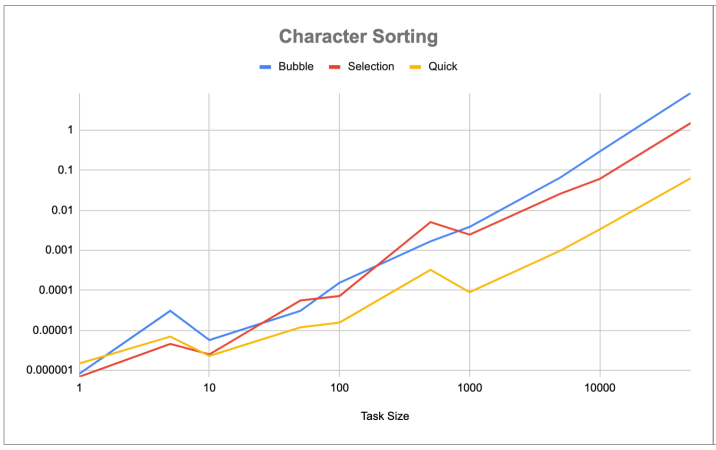
```
private int split;

@Override
public T[] sort(T[] array)
{

    //Initial call to work as override
    return sort(array, 0, array.length - 1);
}

public T[] sort(T[] array, int lower, int upper)
{
    //If there's more than 1 value in the partitioned array,
sort high and low
    if (lower <= upper)
    {
        //sort low
        sort(partition(array, lower, upper), lower, split -1);
        //sort high
        sort(partition(array, lower, upper), split + 1, upper);


private T[] partition(T[] array, int low, int high)
{
    //set pivot as rightmost value
    T pivot = array[high];
    int i = (low - 1);
    for (int j = low; j < high; j++)
    {
        if (array[j].compareTo(pivot) < 0)
        {
            i++;

            T temp = array[i];
            array[i] = array[j];
            array[j] = temp;
        }
    }
    T temp = array[i+1];
    array[i+1] = array[high];
    array[high] = temp;

    split = i+1;
    return array;
```

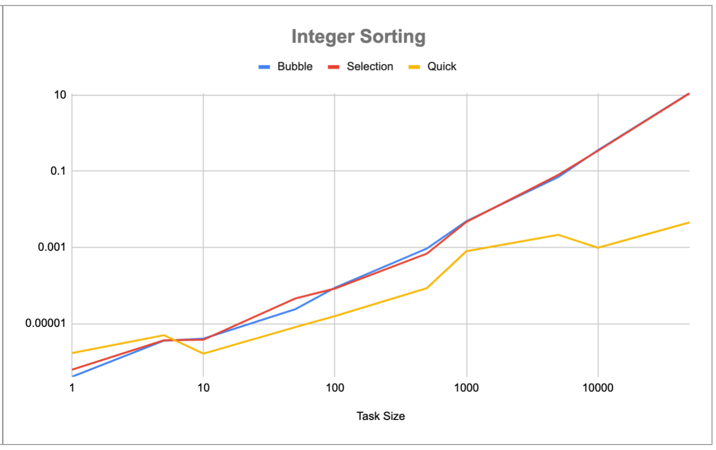
Quick sort was the second sorting algorithm I coded, which is slightly more complex and employs recursion. The sort function that overrides from the standard ArraySort class is used just to call the recursive sort function which includes upper and lower bounds to the sort.

There is also a partition method that actually sorts the array, updates the split, and swaps the pivot to the point where the low and high value meet.

I recorded times for the algorithms to sort arrays ranging in size from 1 to 10,000 values, and plotted them on logarithmic graphs. All the algorithms were faster at sorting characters than integers, and the quick sort was the fastest at both.

Interestingly, all algorithms performed slower at low array sizes, becoming fastest at an array size of around 10, before slowing down. The differences in speed become more clear with the increased task sizes as the inherent inefficiencies are compounded.





Self Assessment

Week	Score	Reasoning
1: Randomiser	4	*There are two different versions of the clever searcher, each with tests to check how effective they are. There are also timers for all searchers and some analysis of the data. However, the documentation is lacking which brings it down to a 4.*
3: Generic Swap	5	*The generic swap works and is tested with full jUnit testing, and it also has fully tested unary predicates.*
5: Sorting	3	*The sorting algorithms work, though Quick sort has a few bugs. I have made appropriate graphs of the data, and made some analysis of it. All the algorithms also have testing classes and timers for both characters and integers.*
7: Linked Lists	0	*I was unable to complete this.*