Practical 1

Last updated: October 1st 2015, at 1.29pm

1 The Code

This week the project contains the following interfaces and classes:

- Timed searches
  - **interface** `TimedSearch`
    Defines an interface for the "find the k$^{\text{th}}$ largest element" problem from the lecture.
  - **abstract class** `SearchTimer`
    Implements the `time` method from the `TimedSearch` interface, but not the `findKthElement` method.
  - *Concrete* **class** `SimpleSearchTimer`
    Implements the `findKthElement` method from the `TimedSearch`
- **class** `IndexingError` **extends** `Exception`
  An `Exception` class for catching indexing errors. The index should be between 1 and the size of the array. Note that the index is not the index of an element in the array, but is the k in "k^{th} largest element. For the largest element in the array, k is one.
- Array generation
  - **interface** `ArrayGenerator`
    Defines an interface for array generator classes.
  - *Concrete* **class** `SortedCount`
    A simple array generator class that generates arrays of the form $[1, 2, \ldots, n]$.
  - **abstract class** `RandomClass` **extends** `SortedCount`
    Extends the `RandomCount` class towards an implementation of a random array generator, but without defining *how* the array can be randomised.

– *Concrete* **class** `SimpleRandomCount`

A naïve algorithm for randomising an ordered array such as that generated by `SortedCount` is to repeatedly pick elements from the sorted array and place them in sequential positions in a new array (obviously, we need to keep track of whether an element has already been picked). Since we want to randomise the original array we then copy the result back to the original array:

```
create a new array of the right size;
while (the new array is not full) {
   pick a random element from the original array;
   if (this element is not already in the new array) {
      add it to the new array;
   }
}
overwrite the original array with the new array;
```

The `SimpleRandomCount` class contains an implementation of the abstract class `RandomCount`, using this algorithm.

2 Questions

1. `SimpleRandomCount` and `SimpleSearchTimer`

   (a) The `SimpleRandomCount` class contains some syntactic and semantic errors.
   
   Correct these errors.

   (b) There are insufficient comments in `SimpleRandomCount`.
   Add appropriate comments to the file.

   (c) Now create a test class to test the functionality of `SimpleSearchTimer`. Use `SimpleRandomCount` to generate test arrays. Because arrays generated by `SimpleRandomCount` contain the numbers $0, \ldots,$ `size` -1 (with `size` the size of the array) the k^{th} largest element will always be `size` $- k$ — you can use this to check that you are getting the correct result. For example, if `array` is an array of size 25 generated by `SimpleRandomCount` a call of

   ```
   findKthElement(array,4)
   ```

   should return 21.

2. `CleverSearchTimer`

   (a) Implement the `TimedSearch` interface using the "clever" solution from the lecture. Call this class `CleverSearchTimer`.
   

(b) Now create a test class to test the functionality of your implementation. Use `SimpleRandomCount` to generate test arrays. test the functionality of your implementation.

(c) Now use `SimpleRandomCount` and the `time` methods of the `SimpleSearchTimer` and `CleverSearchTimer` to compare the efficiency of the two implementations of the `TimedSearch` class.

3. `CleverRandomCount` *Logbook exercise*

(a) `SimpleRandomCount` is not an efficient implementation of the abstract `RandomCount` class. Design and implement a better solution. Call this class `CleverRandomClass`.

(b) Add a timing method to the `RandomCount` class, and use this to compare the efficiency of the two extensions of this class.

This is this (and next) week's *logbook exercise*. You should include details of your solution to this exercise in your logbook. Your work will be assessed on:

- documentation
- structure
- naming
- testing

D a v i d T h a x t e r 2 1 / 0 1 / 2 0 1 6