Practical 1

Last updated: September 28th 2018, at 4.27pm

Introduction

These are the practical exercises for weeks 1 and 2. There will be another single set of exercises for weeks 3 and 4. Thereafter there will be a set of exercises for each week of lectures. In each set of exercises one or more questions will be identified as *logbook exercises*. Your answers to these exercises should go into your logbook — see the assignment specification available on Brightspace. You may also wish to include answers to some of the other exercises, or to include other relevant work that you have developed independently. Inclusion of such work can earn you up to 10% compensation for any shortcomings in other aspects of your logbook. Please note that in most exercise sets, one or more questions will be identified as *model exercises*. Model answers to these questions will be provided, and there is therefore little point in including answers to these questions in your logbook, unless your answer is better than the model answer.

1 The Code

The code bundle contains some source code, in the `src` folder, some test cases, in the `tests` folder, and some timer methods, in the `timer` directory.

1.1 Source Code

The source code consists of two packages — an `arrayGenerator` package that implements various `int` array generators, and a `search` package, which contains implementations of the "find kth largest" problem from the lecture.

1.1.1 `searcher`

The `searcher` package contains the following interfaces and classes:

- abstract class `Searcher`

  Defines an abstract class for the "find the kth largest element" problem from the lecture. Implements the constructor and access methods, but not the `findElement` method.

- *concrete* **class** `SimpleSearcher` **extends** `Searcher`

  Implements the `findElement` method from the `Searcher` abstract class, using a simple "sort, then index" algorithm.

- **class** `IndexingError` **extends** `Exception`

  An `Exception` class for catching indexing errors. The index should be between 1 and the size of the array, inclusive. 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.

1.1.2 `arrayGenerator`

When testing `Searcher`s it is useful to have the access to "random listings", as seen in the week 0 exercises. A listing is (an object that generates) a special type of array of **int**s. A listing of size n will contain each of the values $0, \ldots, n-1$ exactly once. A random listing has these values in a random order. Lisitngs have the useful property that the k^{th} largest element in a lsiting of size n is $n-k$. For example, in a listing of size 10 the 1^{st} largest is 9, which is $10-1$, the 2^{nd} largest is 8 $(10-2)$, the 3^{rd} largest is 7 $(10-3)$, and so on.

The `arrayGenerator` package contains the following interfaces and classes:

- **interface** `ArrayGenerator`

  Defines an interface for array generator classes.

- **interface** `ListingGenerator` **extends** `ArrayGenerator`

  `ListingGenerator` is simply a wrapper for the `ArrayGenerator` class. However, a `ListingGenerator` object should have the propoerty that in a `ListingGenerator` of size n each of the values $0 \ldots n-1$ should occur exactly once (though not necessarily in the correct order). This property is tested in the corresponding tester class — see section 1.2.1.

- *concrete* **class** `SortedListingGenerator` **implements** `ListingGenerator`

  An implementation of `ListingGenerator` in which the entries in the generated array are sorted.

- **abstract class** `RandomListingGenerator` **extends** `SortedListingGenerator`

  Extends the `SortedListingGenerator` class towards an implementation of a random array generator, but without defining *how* the array can be randomised.

- *concrete* **class** `SimpleRandomListingGenerator` **extends** `SortedListingGenerator`

  A naïve algorithm for randomising an ordered array such as that generated by `SortedListingGenerator` 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 `SimpleRandomListingGenerator` class contains an implementation of the abstract class `RandomListingGenerator`, using this algorithm.

- `CleverRandomListingGenerator` **extends** `SortedListingGnerator`

  Another implementation of a random listing generator, that uses a more efficient way of randomising the array.

```
1  for each element in the array {
2     pick another, random element of the array;
3     swap the two elements;
4  }
5
```

Note that when picking the random element on line 2 it doesn't matter if you happen to pick the same element as the one you are currently looking at.

1.2 Test Cases

The test cases are also arranged in two packages. One package provides test cases for array generators, and the other for searchers.

1.2.1 `arrayGenerator`

- **abstract** (tester) **class** `ArrayGeneratorTest`

  Defines a general method for checking that a generator generates arrays of the required size. Also defines an abstract method, `createArrayGenerator`. Concrete implementations of this abstract class will implement `createArrayGenerator` to return array generators of the required type (and size).

  The class further defines a (non-exhaustive) set of test methods for testing specific sizes of array generators. This test suite could do with expanding to test boundary conditions, and error situations.

- **abstract** (tester) **class** `ListingGeneratorTest` **extends** `ArrayGeneratorTest`

  Defines a general method to test the contents of a listing generator — i.e. to check that it *is* a listing. Also overrides the `createArrayGenerator` method to ensure that it returns a `ListingGenerator`, rather than the more general `ArrayGenerator`.

The class further defines a (non-exhaustive) set of test methods. This set could probably do with expansion.

- *concrete* (tester) **class** `SimpleRandomListingGeneratorTest` **extends** `ListingGeneratorTest` Implements `createArrayGenerator` to return a `SimpleRandomListingGenerator`. The test methods inherited from `ArrayGeneratorTest` and `ListingGeneratorTest` will therefore test these `SimpleRandomListingGenerator`s.

- *concrete* (tester) **class** `CleverRandomListingGeneratorTest` **extends** `ListingGeneratorTest` Implements `createArrayGenerator` to return a `CleverRandomListingGenerator`. The test methods inherited from `ArrayGeneratorTest` and `ListingGeneratorTest` will therefore test these `CleverRandomListingGenerator`s.

1.2.2 `searcher`

The searcher test suite contains the following classes:

- **abstract class** `SearcherTest` Defines a general method for testing whether a searcher returns the correct value. Defines an abstract `createSearcher` method that implementing classes will implement to return a searcher of the correct type (and size).

  The class further defines a (non-exhaustive) set of test cases. This set could do with expansion.

- *concrete* **class** SimpleSearcherTest **extends** `SearcherTest`

  Implements `createSearcher` to return a `SimpleSearcher`, so that the tests inherited from `SearcherTest` will test these `SimpleSearcher`s.

1.3 Timers

Similarly, there are two packages in the timer folder, in addition to the `timer` package, in which the `Timer` interface is defined.

1.3.1 `arrayGenerator`

This package contains two classes that time simple, and clever random listing generators.

1.3.2 `timer`

This package currently only contains one class, which times simple searchers.

Questions

There are no model questions this week. However, the `CleverRandomListingGenerator`, and the corresponding tester and timer classes serve as a model answer to the week 0 exercises, which are effectively part of this week's exercises.

The logbook exercise for this week is:

1. Implement the `Searcher` interface, using the more efficient approach (with a small helper array) outlined in the lecture. Call this class `CleverSearcher`.
2. Create a test class to test the functionality of your implementation.
3. Also create a timer class to time the execution of the `findElement` method in your `CleverSearcher` implementation. Compare this with the time taken by the `SimpleSearcher` implementation when performing searches of the same size.

In addition to the functionality of your implementation, your work will be assessed on:

- documentation
- structure
- naming
- testing

Don't forget to include your self-evaluation in your logbook!