CIS2344: Algorithms, Processes and Data

October 1st 2019

Notes last updated: October 10th 2019, at 4.07pm

Contents

1 What is This Module About?

1.1 Contact Information

CIS2344 Algorithms, Processes & Data

Hugh Osborne
SJ1/13
(47)3768
h.r.osborne@hud.ac.uk
scom.hud.ac.uk/scomhro

1.2 It's about...

There are two major themes to the module:

- Algorithms and data

  Looking at some common algorithms and data structures. Some of this material may be familiar from the Computing Science & Maths module in the first year, but we will be going into it in more depth.

- Algorithms and processes

  Looking at concurrent processes — how to programme them, the specific difficulties inherent in working with concurrent processes and how to solve these diffculties.

We will be looking at both practical and theoretical aspects of these themes.

1.3 To Note

- Java is the teaching language that shall be used on this course. Everyone on the course should have encountered Java, or a similar object-oriented language prior to starting this module.

  All program based material will use this language.

- Copies of all lecture and tutorial material are kept on Brightspace

- This course does not teach Java. Rather it attempts to build on your prior Java (or some other object-oriented) learning.

1.4 Recommended Reading

The recommended text for the Algorithms and Data theme is

- Data Structures and Algorithms in Java, *M.T.Goodrich, R.Tamassia, M.H.Goldwasser*, Wiley, *6th Edition*, 2014

Another relevant text is

- Analysis of Algorithms, *Jeffrey McConnell*, Jones & Bartlett, *2nd Edition*, 2008

If you have trouble loaning these books out of the library, then try looking at many similar titled books based on Java and other languages such as C and C++. Note that I will not be following the books above *closely*.

The recommended text for the Algorithms and Processes theme is

- Jeff Magee & Jeff Kramer, *Concurrency: State Models & Java Programs*, Wiley, *2006*

For the quantum computing section of this theme I *strongly* recommend

- Noson S. Yanofsky & Mirco A. Mannucci, *Quantum Computing for Computer Scientists*, Cambridge University Press, *2008*

I also generally highly recommend these books:

- Logicomix: An Epic Search for Truth, *Apostolos Doxiadis & Christos H. Papadimitriou*, Bloomsbury, *2009*
- The Thrilling Adventures of Lovelace and Babbage, *Sydney Padua*, Penguin, *2016*

1.5 Knowledge

As programmers we do not want to be constantly re-inventing the wheel. There is a wealth of knowledge about solutions to common programming problems. This module will develop your knowledge about differing (standard) solutions and how good they are and when it is appropriate to deploy them.

In learning these standard solutions we add to our personal programming tool kit. We also, by example, learn what makes a good solution, what constitutes elegant and efficient code and something about the limits of what is possible in relation to devising new programming solutions.

1.6 Skills

This course aims at developing skills to:

- design and implement solutions to problems
- analyse your potential solutions
- determine what/which program fragments should be optimised

Motto

Just because a solution works, it does not mean that it is a good solution.

1.7 Assessment

- Similar to SD&D (Programming): Logbook of "weekly" exercises weeks 1–22
- Hand-ins (with self-assessment):
  - Weeks 1–7: 17.15, Tuesday 17/12/2019
  - Weeks 8–11: 17.15, Tuesday 21/1/2020
  - Weeks 13–16: 17.15, Tuesday 3/3/2020
  - Weeks 17–22: 17.15, Tuesday 6/5/2020
- Marking
  - 60% for work on identified exercises
  - 20% for quality of self-assessment
  - 20% for overall quality of logbook

  10% "compensation" available through additional work

The assignment specification is available on Brightspace.

1.8 Lecture Plan

Please note: Practicals are timetabled *before* the lecture. Consequently, practicals for week 1's lecture will take place in week 2, for week 2's lecture in week 3, etc., ...

1.8.1 Lecture Plan: Term 1

Week 1	Revision	Week 7	Linked Lists
Week 2	Testing	Week 8	Binary Trees
Week 3	Generics	Week 9	Hashtables
Week 4	Generics	Week 10	Graphs
Week 5	Searching, Sorting	Week 11	Topological Sorts
Week 6	— *Guidance Week* —	Week 12	Recap Hand-in Weeks 1–7 17/12/2019

1.8.2 Lecture Plan: Term 2

Week 13	Concurrent Systems Hand-in Weeks 8–11 21/1/2020	Week 19	Quantum Systems Hand-in Weeks 13–16 3/3/2020
Week 14	Dekker's Algorithm	Week 20	Quantum Computing
Week 15	Semaphores	Week 21	Correctness
Week 16	Monitors	Week 22	Complexity
Week 17	Modelling Circuits	Week 23	Overspill/Recap
Week 18	— *Guidance Week* —	Week 24	Recap
Hand-in Weeks 17–22, 6/5/2020			

2 Example, and Java Revision

Suppose we have a group (list, array) of n numbers and we wish to find the k^{th} largest number. We wish to develop a class of objects that can be given an array and index, k, and that can find the k^{th} largest entry in the array.

2.1 Interfaces

An **interface** class would typically be used in the context of a large software development project to define a minimum set of methods that any implementation of that interface *must* implement. This makes it possible, for example, for one group of developers (the blue team, say) to be given the task of implementing the interface while another group (let's call them the red team) can develop another part of the system that uses methods from the interface. Once the blue team has finished their implementation can be "plugged into" the red team's code without, in theory at least, producing any nasty surprises.

We will be using interfaces to define the methods that you are required to implement in the practical exercises.

2.1.1 An Interface for the Example Problem

```
public interface Search {
    // define method signatures, but not implementations. E.g....
}
```

```
/**
* Find the kth largest element in an array of ints
* @return — kth largest int in array
* @throws IndexingError — if k is not a valid index
*/
```

```
public int findElement() throws IndexingError;

public int[] getArray();

public int getIndex();
```

2.1.2 Another Example

In order to test any code that we write to solve the example problem, it might be useful to have (a) class(es) that generate suitable arrays for us. An interface for such (a) class(es) could be:

```
public interface ArrayGenerator {

/**
* @return the array of ints generated by this ArrayGenerator
*/

    public int[] getArray();
}
```

2.2 Exceptions

Exceptions are usually used to deal with exceptional circumstances — e.g. trying to access an element that is not in an array (think of interrupts in low level programming).

2.2.1 How it used to be done

Consider a method:

```
public void aMethod() {
    int element;
    int[] array = new int[maxElements];
    int index;
    ...// do something
    element = array[index]; // or some other access
    ...// do something else
}
```

What if the index is out of range?

```
public boolean aMethod() {
    ...// do something
    if (y < 0 || y >= maxElements) {// or whatever the array size is
        return false;
    }  else {
        element = array[index]; // or some other access
    }
    ...// do something else
    return true;
}
```

Called elsewhere in the code:

```
if (!aMethod()) {
    // deal with the problem
}  else {
    // carry on
}
```

2.2.2 Problems

- Need to remember to test result
  - Deal with it there?
  - Pass it back up the chain?
- What if it's not simple **true**/**false** situation?
- What if we need to test many different method "results"? Complicated code!)
- What if we want to return a real result from our method?

2.2.3 Exceptions

Use exceptions.

```
public void aMethod() throws Exception {
    ...// do something
    if (y < 0 || y >= maxElements) {// or whatever the array size is
        throw new Exception("Index out of bounds!");
    }  else {
        element = array[index]; // or some other access
    }
    ...// do something else
}
```

If exception is encountered execution of this method halts, and the calling method reinstated. If the calling method doesn't deal with the exception (see later) it in turn stops, and *its* calling method called. If no method deals with the exception execution of the whole programme will terminate and java will report the exception.

Called elsewhere in the code

Either pass it further up the hierarchy:

```
public void caller() throws Exception {
    ...; aMethod(); ...;
}
```

or deal with it:

```
public void caller() {
    ...;
    try {
        aMethod();
    }  catch (Exception e) {// don't do this!
        ...; // deal with error
    }
    ...;
}
```

Catching `Exceptions` as in the code above is *very* bad programming practice. You should define your own exception classes (see below) so that you have some control over which exceptions are, and are not caught. If you catch a general `Exception` it may be something caused by an event completely unrelated to the type of exception you expect to be possible.

2.2.4 Defining Exception Classes

It is better to define your own exception classes, rather than using the generic `Exception` class.

```
public class IndexingError extends Exception {
    public IndexingError() {
        super("Index out of bounds!");
    }
}
```

```
public void aMethod() throws IndexingError {
    ...;
    if (y < 0 || y >= maxElements) {// or whatever the array size is
```

```
        throw new IndexingError();
    }  else {
        element = array[index]; // or some other access
    }
    ...; // carry on
}


...;
try {
    aMethod();
}  catch (IndexingError indexError) {
    System.out.println("There was an indexing error accessing an array element!");
}
...
```

2.3 Abstract Classes

2.3.1 Partial Implementations of Interfaces

Often it is possible to implement part, but not all of an **interface** class, or of a set of **interface** classes. For example, given the `Timer` and `Search` interfaces it is straightforward to define `Timer`'s `timedMethod` method, *assuming* that `Search's findelement method will be implemented`. However, the actual implementation of the `findelement` method depends on the algorithm chosen. The implementation of the `timedMethod` method does *not* depend on the implementation of the `solution` method. In this case we can define a class in which the `timedMethod` method is implemented but the `findelement` method is not. This class (here called `TimedSearch`) must be **abstract** because it does not implement all the classes required by the **interface**s it "implements".

```
public abstract class TimedSearch implements Search, Timer {

    public void timedMethod() throws IndexingError {
        try {
            findElement();
        }  catch (IndexingError error) {
            // just ignore it
        }
    }

}
```

2.3.2 Partially Implemented Extensions

We could define an implementation of `ArrayGenerator` that creates an array containing $0..n$, in that order:

```
public class SortedListingGenerator implements ArrayGenerator {
   private int[] array; // to store the array generated

   public SortedListingGenerator(int size) {
      array = new int[size];
      for (int index = 0; index < array.length; index++) {
         array[index] = index;
      }
   }
}
```

However, this might not be particularly useful for testing out `findElement` method, due to the array being sorted.

We can extend this class, so that arrays generated by it will be randomised, without actually specifying how they are to be randomised, by defining an abstract class which declares that there must be a `randomise()` method without specifying how it is implemented.

We can also use the class to implement any auxilliary methods that may be useful. So the class signature might be:

```
public abstract class RandomListingGenerator extends
SortedListingGenerator {

// random number generator
private Random random = new Random();


/**
* Generate an array containing elements in a random order
* @param size the size of the array to be generated
*/


public RandomListingGenerator(int size) {
   super(size); // generates an ordered array
   randomise(); // randomises it
}



/**
* A utility to provide a random index into the array
```

```
* @return an integer index between 0 and length-1 inclusive.
*/

protected int randomIndex() {
   return random.nextInt(array().length);
}

/**
* Randomise the order of the elements in the array
*/

protected abstract void randomise();
```

2.4 Non-abstract classes

We've got this far without even thinking about a solution to the problem of finding the k^{th} largest element of the array. We are now ready to do so, and to implement the one remaining method. This means that we can now write a non-abstract class — i.e. a class that implements all the methods required by the **interface** that it is implementing. Non-abstract classes can also be used if they are independent of an interface.

2.4.1 Obvious Solution

There are (at least) rwo possible solutions to our problem — an "obvious" solution and a "clever" solution. We will look at the "obvious" solution here. The "clever" solution is left as an exercise for the practical.

The Algorithm

- **Input:** An array of **int**s and an array index
- **Output:** k^{th} largest element of the array
- read the numbers into an array
- sort the array (in some way!)
- pick out and return the correct element of the array

The Code

```
import java.util.Arrays; // in order to be able to use Arrays.sort(array)
public class SimpleTimedSearch extends TimedSearch {

/**
* Find the kth largest element using an "obvious" solution
* @return kth largest element of array
* @throws IndexingError if k is an invalid index
*/



public int findElement() throws IndexingError {
   int[] array = getArray();
   int k = getIndex();
   if (k <= 0 || k > array.length) {// check for indexing error
      throw new IndexingError();
   }
   Arrays.sort(array); // sort the whole array
   return array[array.length - k]; // desired element is kth from the end
}  // end of obvious solution method
```

2.4.2 Clever Solution

Algorithm

- **Input:** An array of ints and an array index
- **Output:** k^{th} largest element of the array
- read the first k elements into an auxilliary array (these will be "the k largest found so far")
- sort the k-element array (in some way!)
- then...

```
for each remaining element {
   if (it is smaller than the smallest element of
        the aux. array) {
      throw it away;
   }  else {
      remove the current smallest element of
        the aux. array;
      place the element into the correct position in
        the aux. array;
```

```
    }
}
```

. . . and return the smallest (the k^{th}) element of the auxilliary array

Which is the better solution? Why is it better? Code up these solutions, and test them! Is there a difference? Later in the course we will analyse the sorting problem again to obtain truly efficient solutions.

3 For Each Loops

What's wrong with this?

```
public int max(List<Integer> values) {
   int max = values.get(0);
   for (int index = 1; index < values.size(); index++) {
      if (values.get(index) > max) {
         max = values.get(index);
      }
   }
   return max;
}
```

Is this better?

```
public int max(List<Integer> values) {
   int max = values.get(0);
   Iterator<Integer> iterator = values.iterator();
   while (iterator.hasNext()) {
      int value = iterator.next();
      if (value > max) {
         max = value;
      }
   }
   return max;
}
```

Use a "for each" loop

```
public int max(List<Integer> values) {
   int max = values.get(0);
   for (int value:  values) {
      if (value > max) {
         max = value;
```

```
        }
    }
    return max;
}
```

- Only use indices if you need that information (i.e. the index). E.g. you need to know *where* the largest value is.
- Only use iterators if:
  - You (might) want to stop part way through.
  - You (might) want to change the collection (e.g. delete an element)

End of lecture 1