UNBC Laboratory Assignment
Computer Science 200—Fall 2021

Dijkstra's Dutch Flag Problem

Goals:

The goals of this laboratory exercise are three-fold:

- To practise using loop-invariants.
- To better understand the partition algorithm.
- To gain confidence in interfacing with libraries supplied by someone else.

Due Date:

This assignment is in two parts.

- Part I is due *Wednesday, 03 November* by the beginning of class.
- Part II is due *Wednesday, 10 November* by the beginning of class.

Utilities

Your goal for this part of the assignment is to write a `Utilities` class that contains several small algorithms for working with a generic array. The interface is shown in Figure 1 on page 5.

With the exception of the `swap` method, each of these algorithms contains one or more loops. Write your code to contain a comment that describes the loop invariant, as shown in the sample `shuffle` algorithm.

There are four `isSorted` algorithms for determining if a section $[\ell, r)$ of an array `data` is sorted with respect to a comparator `c`. If $[\ell, r)$ is not supplied, it should be taken to be $[0, \texttt{data.length})$. If the comparator `c` is not supplied, it should be taken to be `Comparator.naturalOrder()`. Three of these algorithms should just call a different flavour of the algorithm. The fourth should document its loop invariant(s).

Finally, there is a `partition` algorithm. The intent is as follows: after executing

```
int m = partition(data, ell, arr, p);
```

the following should be true:

1. for $i \in [0, \ell)$, `data[i]` should be unchanged.
2. for $i \in [\ell, m)$, `p.test(data[i])` should be `false`.
3. for $i \in [m, r)$, `p.test(data[i])` should be `true`.

4. for $i \in [r, n)$, `data[i]` should be unchanged.

where ℓ = `ell`, r = `arr`, and n = `data.length`.

⇒ Explain why this `partition` algorithm is slightly different from the quick sort partition algorithm.

⇒ Submit your `Utilities.java` file via blackboard.

⇒ This part is due by Wednesday, 03 November.

The Dutch Flag problem

The Dutch flag problem is due to Edsgar Dijkstra, and illustrates the value of carefully thinking through loop invariants.

The problem goes as follows. You are given an array filled with n red, white, and blue objects; and you are to sort the array so that the red objects are to the left, and the blue ones to the right.[1] In general, sorting requires $\Omega(n \log n)$ average time, but in this particular case, because there are only three possible kinds of objects, you can achieve $\Theta(n)$ worst-case time.

Write two algorithms to solve the Dutch flag problem.

1. The first algorithm should use a single `for`-loop. Before coding this algorithm, draw a loop invariant that show where four regions—the red region, the white region, the blue region, and the unknown region–lie, and what variables describe the edges of the regions. Include the loop invariant, either directly in the code, or in separate documentation.
2. The second variant should contain no loops, and two calls to `Utilities.partition` (See the note below about λ-expression for help with using `partition` on `RedWhiteBlue` objects.)

Write your algorithms to be compatible with the `RedWhiteBlue` class shown in Figure 2 on page 6. An implmentation of this class will be given to you as a `.jar` file on the blackboard site (see "Using `.jar` files" below).

Test both of your algorithms by

1. showing that they correctly sort a random array (using `Utilities.isSorted`), and
2. showing that the running time is linearly proportional to the problem size (by plotting, as in previous labs).

⇒ Submit your `*.java` for your `Utilities` class, your `Stopwatch` class, your two algorithms for sorting, and your testing code.

⇒ Submit your test data.

⇒ This part is due by Wednesday, 10 November.

[1] To be truly Dutch, the red ones should be at the top, and the blue ones at the bottom. Red on the outside and blue on the inside is the French flag.

Using `.jar` files

Exactly how to use `.jar` files depends on how you compile and run Java.

If you use basic command line compilation, something like

Unix

```
javac -cp .:RBW.jar *.java
java  -cp .:RBW.jar Main
```

or[2]

Windows

```
javac -cp .;RBW.jar *.java
java  -cp .;RBW.jar Main
```

should work (assuming that `public static void main(...)` is contained in `Main.java`).

λ-expressions, or how to make a `Predicate`

In order to separate the red objects from the non-red objects in an array of `RedWhiteBlue` objects using `Utilities.partition`, we need to create an object that satisfies the interface `java.util.function.Predicate`. Here is a sequence of decreasingly painful ways to do so:

1. create a named class implementing `Predicate`, then create an object:

```
1  private static class RedPredicate implements Predicate<RedWhiteBlue>
2  {
3    public boolean test(RedWhiteBlue obj) { return obj.isRed() ; }
4  }
5  // ...
6  Utilities.partition(data, new RedPredicate()) ;
```

2. create an object of an anonymous class implementing `Predicate`:

```
1  var redPredicate = new Predicate<RedWhiteBlue> ()
2      {
3        public boolean test(RedWhiteBlue obj) { return obj.isRed() ; }
4      } ;
5  Utilities.partition(data, redPredicate) ;
```

3. create an explicit λ-expression as follows

```
1  Utilities.partition(data, (obj)->{ return obj.isRed() ; }) ;
```

4. create an explicit λ-expression, but use some shortcuts

```
1  Utilities.partition(data, obj->obj.isRed() ) ;
```

5. Use a `::` expression:

```
1  Utilities.partition(data, RedWhiteBlue::isRed ) ;
```

More information about Java λ expressions can be found at:

[2] Thaks to KT for reminding me that Windows has a different path syntax

- https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
- http://tutorials.jenkov.com/java/lambda-expressions.html#var-parameter-types-java-11
- https://www.baeldung.com/java-8-lambda-expressions-tips

```
import java.util.function.Predicate ;
import java.util.Comparator ;

public class Utilities
{
  // a classic
  public static <E> void swap(E [] data, int i, int j)       {...}

  // tests for sortedness
  public static <E extends Comparable<? super E>>
  boolean isSorted(E [] data)                                {...}
  public static <E>
  boolean isSorted(E [] data, Comparator<E> c)               {...}
  public static <E extends Comparable<? super E>>
  boolean isSorted(E [] data, int i, int j)                  {...}
  public static <E>
  boolean isSorted(E [] data, int i, int j, Comparator<E> c) {...}


  // a general purpose partition algorithm
  public static <E>
  int partition(E [] data, int ell, int arr, Predicate<E> p) {...}

  // a sample algorithm, with loop invariant
  public static <E> void shuffle(java.util.Random rnd, E [] data)
      {
      final int n = data.length ;
      for(int i=n;i>1;--i)
          {
          // Loop Invariant:
          //    The n-i rightmost elements are randomly selected
          //    with equal probabilities from the initial array.
          //    The range [0,i) contains the elements that have not
          //    yet been selected.
          swap(data,i-1,rnd.nextInt(i)) ;
          }
      }
}
```

Figure 1: `Utilities` class declaration

```
public class RedWhiteBlue implements Comparable<RedWhiteBlue>
{
  // ... implementation details

  public int compareTo(RedWhiteBlue that) { ... }
  // Red < White < Blue

  @override
  public String toString () { ... }

  // Queries to determine the color of an object.
  public boolean isRed  () { /* ... */; }
  public boolean isWhite() { /* ... */; }
  public boolean isBlue () { /* ... */; }
  // for a given object, exactly one of isRed, isWhite or
  // is isBlue is true


  /* ============================
  * static methods to get RedWhiteBlue objects
  */
  public static RedWhiteBlue getRed   () { /* ... */ }
  public static RedWhiteBlue getWhite () { /* ... */ }
  public static RedWhiteBlue getBlue  () { /* ... */ }
  // getWhite().isWhite() is true, and so on ...

}
```

Figure 2: `RedWhiteBlue` class declaration