UNBC Laboratory Assignment
Computer Science 200—Winter 2024

Sorting Algorithms

Due Date:

This assignment uses algorithms written in the previous programming assignment. It is due Friday, 08 March

The Dutch Flag problem

Goals:

The goals of this laboratory exercise are two-fold:

- To practise using loop-invariants.
- To better understand the partition algorithm.

The 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 1 on the following page. Note carefully that objects of the `RedWhiteBlue` are fairly opaque. You can compare objects with either `.equals` or `.compareTo`. You can ask an object whether it `isRed()`,

[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.

```
package ca.unbc.casper ;

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

  /*
   * various comparisons.  equality is by value, not address
   * cmpareTo throws NullPointErexeption if given a null argument
   *   Red < White < Blue
   */
  public boolean equals(Object o)         { /*...*/ }
  public boolean equals(RedWhiteBlue oo)  { /*...*/ }
  public int compareTo(RedWhiteBlue that) { /*...*/ }


  // convenience mthod
  @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

  /* The sole means of creating RedWhiteBlue objects:
   * makeRWBArray(int r, int w, int b)
   * returna an array with r red values, w white values, and b blue values
   */
  public static RedWhiteBlue[] makeRWBArray(int r, int w, int b)
                                        { /*...*/ }
}
```

Figure 1: `RedWhiteBlue` class declaration

`isWhite()`, or `isBlue()`, and you can convert it to a `String` representation. That's it. The sole way to create new objects, is the `makeRWBArray` method. For testing, you'll want to use this method and the `.shuffle` method from the `Utilities` class that you wrote last assignment.

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.

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

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:

- 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

Insertion Sort

⇒ Implement insertion sort in a manner suitable for inclusion in a library of routines to be supplied to another user. In particular:

- Make sure that your algorithm works when the array supplied by the user is very small, even 0 or 1 elements.
- Code your algorithm with generic public static function interfaces. Minimally, provide a version that takes an array of arbitrary elements and a `Comparator` on those elements.
- Ensure that your implementation of insertion sort is stable.
- Write a version of insertion sort that uses sentinel elements.
- Write two versions of each algorithm: one that has a signature like

```
public static <E extends Comparable<E>> void sort(E [] data) ...
```

and one that has a signature like

```
public static <E> void sort(E [] data, Comparator<E> pred) ...
```

Testing and Measuring Insertion Sort

⇒ Test your sorting algorithms for *correctness*. That is, write code that verifies that your sorting algorithm produces (a) a sorted array, and (b) an array that is a permutation of the original. You may want to code some test routines to generate data other than permutations. For instance, an array randomly filled with 1's and 2's provides a good test for how your algorithms handle equal elements.

⇒ Test insertion sort sort for stability.

⇒ Time your sorting procedures for various different sized data collections.

All of your tests should be automated. That is, they should produce timing data files that are easy to use with your plotting software.

You must hand-in plots of running times versus size of data collection!

⇒ Show that your actual running times are consistent with the $\Theta(n)$-behavior for insertion sort and Shell sort.