UNBC Laboratory Assignment
Computer Science 200—Fall 2010

Dijkstra's Dutch Flag Problem

Goals:

The goals of this laboratory exercise are three-fold:

- To practise using loop-invariants.
- To practise timing code to check its asymptotic Θ-behaviour.
- To gain confidence in interfacing with libraries supplied by someone else.

Due Date:

This assignment is due *Tuesday, 5 October* at the beginning of class.

The Dutch Flag problem

The Dutch flag problem is due to 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 it so that the red objects are at the top, and the blue ones at the bottom. Now, 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.

Your assignment is to think hard about this problem, preferably by thinking about loop invariants; code your solution; document in your code all loop invariants that are important; and produce graphs of the time behaviour of your algorithm to verify that your algorithm is indeed $\mathrm{O}(n)$.

The `RedWhiteBlue` class

To ensure that you think about red, white, and blue objects abstractly, you are supplied with a `RedWhiteBlue` class (see Figure 1). This class is given to you as a `.jar` file, with which your program should link. The `.jar` file can be found on Dr Casperson's web-site.[1]

The `RedWhiteBlue` class supplied is deliberately quite minimalistic. In fact it does not contain a zero argument constructor. However, you are more likely to want to create a random vector of `RedWhiteBlue` objects, and the static member function `RedWhiteBlue::makeRandomRWBVector` does this. It takes as arguments the number of objects of each colour that you want.

[1] See `http://web.unbc.ca/~casper/Semesters/2010F/200-homework.php`.

```
public class RedWhiteBlue
{
  private ...
  protected static final RedWhiteBlue red   = ... ;
  protected static final RedWhiteBlue white = ... ;
  protected static final RedWhiteBlue blue  = ... ;

  // public interface starts here.

  // void setSeed initializes a random in a known way.
  // this gives replicable results from makeRandomRWBVector
  public static void setSeed(long seed) { ... }

  public String toString () { ... }

  public  RedWhiteBlue(RedWhiteBlue other) { ... }


  // You can (quickly) swap this RWB object with another
  void swap(RedWhiteBlue other)         { ... }


  // To create a random array of RedWhiteBlue objects, call the
  // following static member function.  The parameters r,w,b are the
  // number of red, white, and blue objects respectively to randomly
  // insert in the vector.
  public static RedWhiteBlue[] makeRandomRWBVector(int r, int w, int b)
      { ... }

  // Queries to determine the color of an object.
  public boolean isRed  () { return value.isRed  ()  ; }
  public boolean isWhite() { return value.isWhite()  ; }
  public boolean isBlue () { return value.isBlue ()  ; }

}
```

Figure 1: RedWhiteBlue class declaration

Stop-watches

In order to verify that your algorithm really has $\Theta(n)$ average-case running times, you should create a `StopWatch` class.

You are going to need the `StopWatch` class in later assignments, so make sure that you create a separate `StopWatch.java` file.

The `StopWatch` class should behave like a mechanical three-button stop-watch. That is, it should implement an interface something like:

```
interface StopWatch
{
    StopWatch() ;
    // The three buttons.        // and queries
```

```
    StopWatch start() ;          double elapsed() const ;
    StopWatch stop() ;           boolean is_running() const ;
    StopWatch reset() ;
}
```

Even though you may not need it for this assignment, your stop-watch should function correctly regardless of the order in which the three buttons are pressed.

The most likely library routine to use to build a `StopWatch` is `System.nanoTime`[2]. If you want, you can use the code at `http://people.apache.org/~bayard/commons-lang-3.0-snapshot-api/org/apache/commons/lang/time/StopWatch.html`, but then please demonstrate that you have tested it. I haven't.

There is more than one notion of elapsed time for a multi-process operating system because single processes don't get all of the processor. What you want to measure is the elapsed CPU time, not the elapsed wall-clock time.

Library locations

The `RedWhiteBlue` class can be found in `RedWhiteBlue.java` and a `.jar` file implementing the file can be found on David Casperson's website.

Check List:

Make sure that you have:

⇒ written a / function with signature

```
void sort(vector<RedWhiteBlue>& data) ;
```

that takes a vector of unsorted data and returns a vector of sorted data.

⇒ verified with your stop watch that it runs in $O(n)$-time, and plotted your results. It should only use constant extra storage (or at least $o(n)$ extra storage!), but you do not need to verify this experimentally.

⇒ shown in your output how you have linked your file against the class supplied.

⇒ documented in your code the invariants for any and all loops inside your sorting function.

⇒ handed in nicely graphed data produced by your stop watch measurements.

Further programming Exercises

Solve Problem 2.28 from *Weiss*. Write loop invariants in the comments for each loop in your code. Write estimates of the Θ-running time for each routine. Carefully time your code to show that it has the expected running time.

[2] A more "modern" approach might be to use the profiling support built into many IDEs.
"It's times like these I like to rub my bloated, overweight, slow IDE in the faces of the emacs elitists. :-)"
– James Schek, Oct 8 '08
(found on `http://stackoverflow.com/questions/180158/how-do-i-time-a-methods-execution-in-java`)