DD2459: Software Reliability, sofRel22

Lab 2: Black-box and Requirements-Based Testing:
Sorting and Searching
Answer all 4 questions.

Introduction: The aim of lab exercise 2 is to give you experience of black-box and requirements-based testing methods, as taught in the lectures. Specifically, you will gain experience of: (1) writing precise functional code requirements (using JML pre- and postconditions) for simple and well-known algorithmic problems, (2) writing a random test case generator, (3) writing a pairwise test case generator, (4) using a JML postcondition as an automated test oracle, (5) benchmarking test suites against one another (in this case random and pairwise test suites).

You can solve this lab using any programming language known to you, as **no code needs to be handed in**.

Sorting, *searching* and *membership* are three classical problems in computer science for which dozens of algorithms exist. Each of these problems take as input an array of values chosen from some ordered set (e.g. integers, floating points). In *sorting*, we return an output array of the same length, consisting of the input array elements reordered into ascending order. In *searching*, we take an additional input value, which is a value that might occur in the array. This value is known as the *key*. Searching returns an integer value that should be *n* if the key occurs in the input array, in position *n* and -1 if it does not occur anywhere in the array. *Membership* returns a Boolean value that should be *1* if the key occurs in the input array, and 0 if it does not occur anywhere in the array.

It is well known that if we need to do a large number of searches and/or membership queries, then it is more efficient to pre-sort the array and then use a so-called *binary search*, which can run in $\log_2$ N time, where N is the length of the input array.

Pseudocode for a binary search algorithm can be given as follows.

```
method BinarySearch( A : array of integer, key : integer ) :
integer
begin
      var x, l, r : integer;

      l=1; r = A.length();

      repeat
            x = (l+r) div 2;
            if key < A[x] then r=x-1 else l=x+1
      until ( key==A[x]) or (l>r)

      if key==A[x] then return x else return -1
end
```

Algorithm 2: Binary Search

Exercises.

1. Draw a condensation graph for Algorithm 2.

2. Write appropriate pre and postconditions using the JML language (ie. write appropriate *requires-ensures* conditions) for:

(i) sorting,

(ii) searching, hint assume `key` is a native data type e.g. `int key` (otherwise must check `key` is also non-null)

(iii) membership,

(iv) binary searching.

3. Implement three programs (in your favorite programming language) to perform:

(i) sorting of integer arrays of arbitrary length,

(ii) membership queries on sorted arrays of arbitrary length using binary search, and

(iii) membership queries on unsorted arrays of arbitrary length, by combining program (i) with program (ii).

4. Build a random and a pairwise testing framework for program (iii), as defined in the course notes. This requires:

(1) some extra coding for random number generation, and generating combinations of pairs,
(2) writing these to a file as complete test cases.
(3) you will need to be able to call program (iii) repeatedly on both files of test cases (random and pairwise).
(4) You will need some experimentation to see how large each file needs to be (i.e. the number of test cases) in order to find a specific mutation error. See further instructions below.

Before you write any code, you should consider: is it better to treat an array input variable of length *N* (such as `A` in Algorithm 2) as one variable or as *N* individual variables? What are the consequences for your test results of making either choice?

Choose a modest array size for sorting and searching, e.g. $N <= 20$. Keep this fixed throughout all testing experiments. Rewrite program (iii) in at least 6 different ways to inject coding errors into it. (This is called *mutation*, and will be studied in lecture 7.) Your choices should inject errors into program (iii) either by injecting into program (i) or into program (ii). You could also experiment with injecting integration errors between programs (i) and (ii).

For each injected error, compare the performance of random testing and pairwise testing to find that error. For this comparison you can measure in each case (random or pairwise) the number of test cases from file that need to be executed until the error is found. **Hint**: you can work much faster if you can find a way to use your solution to Question 2, to write an oracle that automatically judges each test outcome (pass/fail). Otherwise you will have to stare at the test results and judge them manually (time consuming!)

(i) Write up your results in a 6 X 2 table, that clearly labels each testing method across the 2 columns, and each injected error down the 6 rows. In each table entry $x_{i,j}$ for row i and column j, you should write the *average number* (for random) and *minimum number* (for pairwise) of test cases executed before the injected error is found. If the error is not found, or an infinite loop (timeout) occurs you can write this.

(ii) Briefly describe each of your 6 injected errors, with reference to the original code.

(iii) Do your results show any variation in testing effort (i.e. number of test cases executed) between injected errors, or injected locations (programs (i) or (ii))? Try to explain any phenomena that you observe.

(iv) Can you repeat the above experiment for a much larger integer array size *N*, say $N = 100$ or $N=500$? What changes do you observe?

Bibliography:

1. www.pairwise.org more info on pairwise testing.
2. D.E. Knuth, Art of Computer Programming Vol 3: Sorting and Searching, Addison Wesley, 1998.
3. www.cs.ucf.edu/~leavens/JML/ more info on JML.