CS 584/684 Spring 2017 Homework 1 – due noon, Wednesday, April 12 2017

Your solutions to problems 1-3 should be type-set in LaTeX and submitted in both `.tex` and `.pdf` form, with file names `hw1.tex` and `hw1.pdf`. These two files, plus any additional source files invoked from your `.tex` file (such as pictures), should be bundled together into a single `.zip` file named *your-last-name*`-tex-hw1.zip`. Your code for problem 4 should be submitted in a single, separate `.zip` file named *your-last-name*`-code-hw1.zip` with contents as described below in the description of problem 5.

Submit by emailing to `hamialex@pdx.edu` including the two zip files as separate attachments and including "`CS584 HW1`" in the subject line.

Latex guidelines: You *must* use the latex `template.tex` file provided on the course web page as the basis for your submission. Note that this file also imports the `framework.tex` file. Please use the (somewhat verbose) `clrscode3e` package (available from the CLRS book web-site and in many tex installations) for writing CRLS-like pseudo-code. If you need to include a pdf graphic, use the `graphicx` package. You can look at the `.tex` file for the first set of lecture notes to get some ideas.

1. In a "Secret Santa" gift exchange, each of the n participants obtains the name of another participant to whom they will secretly give a gift. Suppose we try to perform a name assignment as follows: first each participant writes their own name on a piece of paper and puts it in a hat, then each participant draws a piece of paper blindly from the hat. The assignment attempt fails if anyone draws their own name.

(a) Suppose you participate in a name assignment attempt. What is the probability p that you draw your own name?

(b) What is the *expected* number of people that draw their own name? (Hint: Use indicator variables.)

Now suppose we would like to calculate the probability that *no* participant draws their own name (and hence the assignment attempt succeeds). We might be tempted to make the following argument:

i. The probability that any given person draws their own name is p.

ii. Hence the probability that any given person does not draw their own name is $1 - p$.

iii. So we claim that the probability that no person draws their own name is $(1 - p)^n$.

(c) Show that this claim is false by giving a concrete counterexample, i.e., choose a (small!) value of n, compute the probability explicitly, and show that it does not equal the claimed probability.

(d) Explain which step in the argument given above was unjustified.

(Incidentally, establishing and proving a *correct* formula for the probability that assignment succeeds is non-trivial, and the techniques involved are beyond the scope of this course.)

2. As presented, the MOM (median-of-medians) selection algorithm divides the input into groups of 5. Using a group of odd size helps keep things a little simpler (because otherwise the group medians are messier to define), but why the choice of 5?

(a) Show that the same argument for linear worst-case time continues to work if we use groups of 7 instead.

(b) Show that the argument fails if we try to use groups of 3 instead. (Note: This is *not* asking you to show that the worst-case time of the algorithm is supra-linear if groups of 3 are used (although that is also true) — merely that this particular proof won't work.)

3. Consider the HOARE-PARTITION procedure given in in CLRS problem 7-1. The goal of this problem (parts (b),(c), and (d)) is to prove that this procedure is correct. To give a careful proof, you need to write down explicit invariants for all loops, and then prove that they hold at loop entry, are maintained by the loop bodies, and are sufficient to prove the procedure's post-conditions. More precisely, if I is the invariant of the loop **while** c **do** s, then: I should be true before the loop is executed the first time (regardless of whether c is true or not), and it should remain true after s executes; $I \wedge c$ can then be assumed to be true after the loop finishes executing.

Although the basic invariant for the outer loop is fairly straightforward, there are some complications at the boundaries. It is therefore easier to prove correctness by *unrolling* each loop once. Specifically, each statement of the form **repeat** s **until** c can be unrolled to $(s;$ **while**$(\neg c)$ **do** $s)$. Moreover, the top-level **while** loop can also be viewed as a kind of **repeat** $-$ **until** loop and then unrolled. The resulting procedure is shown below, and it is this procedure that you should prove correct with respect to the stated pre- and post-conditions. These are shown using the same notational conventions as in the lecture notes: ret is the return value, A' is the original contents of array A, and $\mathrm{PermOn}(A, A', p, r)$ means that $A[p\,..\,r]$ is a permutation of $A'[p..r]$ and $\forall k \notin [p\,..\,r], A[k] = A'[k]$. (Notice that the post-condition here is weaker than the post-condition for Lomuto's PARTITION, in that the value j returned is *not* necessarily the position of the pivot value x in the final array. Consequently, Hoare's algorithm is not directly suitable for use by QUICK-SELECT.)

HOARE-PARTITION-UNROLLED(A, p, r)

```
    // Requires on call: 1 ≤ p < r ≤ A.length.
    // Ensures at return:
    //   p ≤ ret < r
    //   ∧∀j ∈ [p..ret], ∀k ∈ [ret+1..r], A[j] ≤ A[k]
    //   ∧PermOn(A, A', p, r)
 1  x = A[p]
 2  j = r
 3  while A[j] > x
 4      j = j − 1
 5  i = p
 6  while A[i] < x // notice that the body of this loop never executes!
 7      i = i + 1
 8  while i < j
 9      exchange A[i] with A[j]
10      j = j − 1
11      while A[j] > x
12          j = j − 1
13      i = i + 1
14      while A[i] < x
15          i = i + 1
16  return j
```

4. [programming problem] This problem asks you to design an algorithm for the following problem: Given an n-element *sorted* sequence A of distinct integers and an m-element sequence B of integers (not necessarily sorted or distinct), report how many elements of B appear in A. Implement your solution as a runnable program in one of the following languages: Java, Python, C, or Haskell. (If you wish to use a different language, consult the instructor first.) Note: you may *not* use library functions to do the substance of the work (e.g. sorting, searching, hashing) in your implementation. But it is fine to use library functions for IO, memory allocation, etc. Your submission will be scored first on correctness and then on speed, so choosing an algorithm with good asymptotic properties is important, but choosing a correct algorithm and implementing it correctly are even more important.

Your program should take one command line argument, which is the name of an input file. The format of that file will be as follows:

- First line contains a number c = number of test cases in the file.

- Then come c test cases, each of the following form:
  - Line containing the number of elements n in sequence A.
  - Line containing the values in A with exactly one space separation between them. (Note that these values are distinct and are in sorted order.)
  - Line containing the number of elements m in sequence B.
  - Line containing the values in B with exactly one space separation between them. (Note that these values are not necessarily in sorted order and are not necessarily distinct.)

Note: If either n or m is 0, the corresponding line of values should still exist, but be empty. You can assume that c, n, and m are integers in the range $[0..10^6]$ and the values in A and B are in the range $[-10^9 .. 10^9]$.

Your program should output (to `stdout`) one line for each test case, of the form "`Case i: p`" where p is the number of elements of B that appear in A.

Example Input:

```
2
5
-3 -1 0 2 4
4
2 -2 2 4
0

4
1000 100 10 10000
```

Corresponding Example Output:

```
Case 1: 3
Case 2: 0
```

Warning: If your output format is not correct (even spacing), you will get no credit; this problem will be graded by doing a `diff` against a standard output file.

Place your code (one or more source files) together with a `Makefile` in a fresh subdirectory with the name *your-last-name*`-code-hw1`. Then create a single `zip` archive with the name *your-last-name*`-code-hw1.zip` containing just that directory and its contents. It should be possible to build an executable file called `hw1` and test it on an input file `/path/to/foo` by the following steps:

1. `unzip` *your-last-name*`-code-hw1.zip`
2. `cd` *your-last-name*`-code-hw1`
3. `make`
4. `./hw1 /path/to/foo`