CSE 332 Data Abstractions, Winter 2015
Homework 4 – Part B

Due: **Thursday, Feb 12, 2015** at 23:00 (11:00pm) via the catalyst drop box. You should refer to the written homework guidelines on the course website for a reminder about what is acceptable pseudocode. Part B of this homework has TWO terrific questions!! Your combined score from part A & part B will be your total HW4 score.

Submission instructions

Submit an electronic copy to the catalyst dropbox as a PDF file. You can either do the assignment on an electronic word processor (and convert to PDF) or do it on physical paper and scan it (or take a high res photo) and upload a single PDF of the file. It will be much easier to grade if every question starts on a separate page. Don't forget to put your name on the top of the first page.

Problem 3: Sorting Phone Numbers

The input to this problem consists of a sequence of 7-digit phone numbers written as simple integers (e.g. 5551202 represents the phone number 555-1202). The sequence is provided via an Iterator<Integer> - *you do not get an array containing these phone numbers and you cannot go through the iterator more than once*. The sequence is potentially too large to fit in memory so don't try to store the phone numbers themselves in an array. No phone number appears in the input more than once. You may assume that phone numbers will not start with 0, although they may contain zeroes otherwise.

Write **precise** pseudocode for a method that prints out the phone numbers (as integers) in the list in ascending order. **Your solution must not use more than 2MB of memory**. (Note: It cannot use any other storage – hard drive, network, etc.) In your pseudocode you may only declare variables and arrays of these unsigned data types (these are not real Java data types): bit(1 bit), byte(8 bits), short(16 bits), int(32 bits), long(64 bits). **Explain** why your solution is under the 2MB limit.

Problem 4: QuickSort Variation

Consider this pseudocode for quicksort, which leaves pivot selection and partitioning to helper functions not shown:

```
// sort positions lo through hi-1 in array using quicksort (no cut-off)
quicksort(int[] array, int lo, int hi) {
      if (lo >= hi - 1) return;
      pivot = pickPivot(array, lo, hi);
      pivotIndex = partition(array, lo, hi, pivot);
      quicksort(array, lo, pivotIndex);
      quicksort(array, pivotIndex+1, hi);
}
```

Modify this algorithm to take an additional integer argument **enough**:

```
// sort at least enough positions of lo through hi-1 in array using quicksort
// (no cut-off)
quicksort(int[] array, int lo, int hi, int enough) { ... }
```

We change the definition of correctness to require only that *at least* the first 'enough' entries (from left-to right) are sorted *and contain the smallest 'enough' values*. (For example, if you are passed **enough** =5, then the 5 smallest values in the array should be located in locations 0-4 of the array. If **enough** >= hi-lo, then the whole range must be sorted as usual.) While one correct solution is to ignore the **enough** parameter and sort the entire array, come up with a better solution that skips completely unnecessary recursive calls. Assume the initial call to quicksort specifies that 'lo' is 0 and 'hi' is the upper-bound of the array. Watch your off-by-one errors!