CSE 332 Data Abstractions, Winter 2013
Homework 4

Due: **Friday, Feb 15, 2013** at the BEGINNING of lecture. Your work should be readable as well as correct. You should refer to the written homework guidelines on the course website for a reminder about what is acceptable pseudocode. You will notice that homework Four has FOUR fun questions!! Please write your **section** at the top of your homework.

Problem 1: Algorithm Analysis

The methods below implement recursive algorithms that return the first index in an unsorted array to hold 17, or -1 if no such index exists.

```
int first17_a(int[] array, int i) {
  if (i >= array.length)
      return -1;
  if (array[i]==17)
      return 0;
  if (first17_a(array,i+1) == -1)
      return -1;
  return 1 + first17_a(array,i+1);
}
```

```
int first17_b(int[] array, int i) {
  if (i >= array.length)
      return -1;
  if (array[i]==17)
      return 0;
  int x = first17_b(array,i+1);
  if (x == -1)
      return -1;
  return x + 1;
}
```

(a) What kind of input produces the worst-case running time in an absolute "number of operations" sense, not a big-O sense, for **`first17_a (arr,0)?`**

(b) For **`first17_a`**, give a recurrence relation, including a base case, describing the worst-case running time, where n is the length of the array. You may use whatever constants you wish for constant-time work.

(c) Give a tight asymptotic ("big-Oh") upper bound for the running time of **`first17_a(arr,0)`** given your answer to the previous question. That is, find a closed form for your recurrence relation. Show how you got your answer.

(d) What kind of input produces the worst case running time in an absolute "number of operations" sense, not a big-O sense, for **`first17_b(arr,0)?`**

(e) For **`first17_b`**, give a recurrence relation, including a base case, describing the worst-case running time, where n is the length of the array. You may use whatever constants you wish for constant-time work.

(f) Give a tight asymptotic ("big-Oh") upper bound for the running time of **`first17_b(arr,0)`** given your answer to the previous question. That is, find a closed form for your recurrence relation. Show how you got your answer.

(g) Give a tight asymptotic ("big-Omega") worst-case lower bound for the *problem* of finding the first 17 in an array (not a specific algorithm). Briefly justify your answer.

(See back of this page for remaining problems)

Problem 2: Deletion in Hashing

This problem is to get you to think about how lazy deletion is handled in hash tables using open addressing.

(a) Suppose a hash table is accessed by open addressing and contains a cell X marked as "deleted". Suppose that the next successful find hits and moves past cell X and finds the key in cell Y. Suppose we move the found key to cell X, mark cell X as "active" and mark cell Y as "open". Suppose this policy is used for every find. Would you expect this to work better or worse compared to not modifying the table? Explain your answer.

(b) Suppose that instead of marking cell Y as "open" in the previous question, you mark it as "deleted" (it contains no value, but we treat it as a collision). Suppose this policy is used for every find. Would you expect this to work better or worse compared to not modifying the table? Explain your answer.

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*. No phone number appears in the input more than once but there is no limit on the size of the input. Write precise (preferable Java-like) 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.) 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*. (If enough >= hi-lo, then the whole range must be sorted as usual.) While one correct solution is to ignore the enough parameter, 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!