CPSC 320 Assignment #2

January 12, 2015

Due date: Monday, 2015/01/19 at 5PM

Staple your solution behind the CPSC 320 cover page and submit in our handin box.

We will mark at least one part of each of the 4 problems.

Problems 1–3 ask about this algorithm from the previous assignment for determining whether an array of numbers contains two identical numbers (duplicates or "dups"):

```
HAS_DUPS(array):
  For each index i from 1 to the length of the array:
    For each index j from (i+1) to the length of the array:
      If array[i] is equal to array[j]:
        Halt and return true
  Halt and return false
```

1. We could describe the running time of `HAS_DUPS` in the worst case (when the array contains no duplicates) as $T(n) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} 1$.

   (a) Simplify this sum to a polynomial (i.e., a sum of terms each of the form $c \cdot n^k$, where each c and k are constants and no two terms have the same values of k). Show your work!

   (b) Prove using the definition of Θ (either the limit-based definition or the logical definition in the textbook) that the polynomial you found is in $\Theta(n^2)$.

   (c) Imagine we implemented the "array" using a linked list instead. Each computation of the length n of the "array" then takes $\Theta(n)$ time, and accessing element `array[i]` requires $\Theta(i)$ time. Give and **briefly** justify a good Θ-bound on the performance of the algorithm in this case.

2. A better solution would be to sort the array first and then check for duplicates. This is, effectively, a reduction to sorting, where the first algorithm in the reduction is trivial (the instance of the duplicate-checking problem is the same as the instance of the sorting problem) and checking for duplicates is part of the second algorithm in the reduction.

   (a) Give an efficient second algorithm for the reduction to convert a sorted array (solution to an instance of the sorting problem) into a boolean indicating whether there are duplicates (solution to an instance of the duplicate-checking problem).

   (b) Give and **briefly** justify a good Θ-bound on the worst-case runtime of your algorithm.

   (c) Name two fast sorts you could use to solve the sorting problem.

   (d) Give and **briefly** justify a good Θ-bound on the worst-case runtime of a solution to this problem using your reduction and a fast sorting algorithm of your choice.

3. An even better solution might be to use a hash table.

   (a) Write out an efficient algorithm for checking for duplicates in a list of numbers assuming you have available an initially empty hash table `H` supporting the operations `INSERT(H, key, value)`, `FIND(H, key)`, and `CONTAINS?(H, key)`, all of which run in expected $\Theta(1)$ time.

   (b) Give and **briefly** justify bounds on the expected performance of your algorithm from the previous part for an input without duplicates.

   (c) Give and **briefly** justify bounds on the expected memory usage of your algorithm for an input without duplicates. (For memory usage, ignore the memory used by the instance and count only memory used by your code.)

4. Consider the following algorithm that takes n as an input:

```
While n > 2:
  If n is prime:
    Decrement n
  Let f be the smallest prime factor of n
  Set n to (n / f)
```

(a) Give and **briefly** justify a good O-bound on the number of iterations of the loop in terms of n. Hint: find a particular type of number that is easy to analyze and briefly argue that no other type of number could take any **more** time asymptotically.

(b) BONUS (not required, might be worth a small amount of bonus points, will not generally receive much partial credit for wrong answers): Give and **briefly** justify a good Ω-bound on the number of iterations of the loop in terms of n.