CPSC 320 Assignment #5

February 22, 2015

Due date: Monday, 2015/03/02 at 5PM

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

We guarantee that we will mark at least one sub-part of each question.

1. You're designing a divide-and-conquer algorithm for a problem. An existing algorithm that solves the problem (**without** using divide-and-conquer) runs in $\Theta(n^2)$ time.

   You've started work on a divide-and-conquer approach that breaks a problem instance of size n down into 5 subproblems each of size $n/3$ in constant time, but you're not quite sure what the best algorithm is to put the solutions to the subproblems together into a solution to the original problem instance.

   Use the Master Theorem to give clear advice on what the critical possible complexities are for the algorithm to combine the solutions.

   (E.g., you might say something of a similar form to this **incorrect** statement: "Any algorithm to combine the solutions in $o(n)$ time will result in a solution to the problem that is in $\Theta(n^{1.5})$. Anything in $\Theta(n)$ will result in a solution that is in $O(\frac{n^2}{\lg n})$, but larger functions will result in runtimes no better than the existing algorithm.")

2. Some textbook problems:

   (a) Problem 5.6.

   (b) Problem 5.7.

   Note: We'll post some suggestions on Piazza for this one that should help!

(c) **Just for fun** (don't submit): "Steepest gradient descent" is a good way to find a local minimum. Analyse its worst-case performance on Problem 5.7, assuming you start at the upper-left grid point and always look at the (at most) 4 grid points orthogonally surrounding a particular point. (Be sure to describe a—scalable!—worst case input!)

3. A friend describes the following algorithm, saying perhaps it can be used to sort lists of numbers:

```
NOTE: A is an array of numbers of length n. Sorting "in place"
  means mutating the existing array. (So, if we sort the array
  [1, 3, 2] in place, we swap the last two elements so the same
  array now holds [1, 2, 3].) A[3..5] is the three-element
  subarray of A composed of A[3], A[4], and A[5].

DrawAndQuarter(A, lo, hi):
  if hi - lo + 1 < 4:
    Sort the elements of A[lo..hi] in O(1) time in place
  else:
    Let mid <- lo + (hi - lo + 1)/2
    Let q1  <- lo + (mid - lo)/2
    Let q3  <- mid + (hi - mid + 1)/2

    DrawAndQuarter(A, q1, q3 - 1)
    DrawAndQuarter(A, lo, mid - 1)
    DrawAndQuarter(A, mid, hi)
```

(a) Prove or disprove that this is a correct sorting algorithm.

(b) Analyse the runtime of this algorithm. (Note: we can analyse the algorithm's runtime whether or not it is correct!) You may use any method you wish for the analysis, but you should produce good Θ bounds on the best-case and worst-case runtimes in terms of n, and you may want to try multiple approaches on your own for practice!

(c) Imagine we alter the algorithm to replace the line `DrawAndQuarter(A, mid, hi)` with:

```
Let sorted be true
For i from mid to hi - 1:
```

```
  If A[i] > A[i+1]:
    Change sorted to false
If sorted is false:
  DrawAndQuarter(A, mid, hi)
```

Reanalyse the algorithm's runtime, giving good Θ-bounds on the best- and worst-case runtimes. Again, you may use any technique you wish.

(d) **Just for fun** (don't submit): Could "Drawing-and-Thirding" work as a sorting algorithm? (Sort the first two-thirds, sort the second two-thirds, and then sort the first two-thirds.) Analyse.

4. **Midterm Review**

(a) Looking back at Question 2 on the midterm (about MSTs), we'll consider this commonly proposed but incorrect reduction:

```
A1: Contract all negative-weighted edges (discarding
the higher cost edge when two edges coincide).

A2: Union all negative-weighted edges with the set of
edges in the underlying instance's solution.
```

Use an example problem instance to illustrate the distinct flaw with this algorithm suggested by each of the following parts:

i. Edges of one particular weight exercise a **minor** flaw.

ii. Another flaw can keep the reduction's result from even being a **solution**, much less an optimal solution.

(b) In Question 3 (the "Lazy" Interval Scheduling Problem), Part 4 builds **toward**—without actually finishing—a proof of optimality of the unspecified "new" greedy algorithm. We'll consider a common confusion people had between an "instance" of a problem, a "solution", and an "optimal solution". We'll also consider how similar or disimilar $\mathcal{O}$ can be from the "new" greedy algorithm's solution.

Reminder: in a particular instance of the problem, the *conflict set* of a job j is the set of all jobs that conflict with j, **including** j itself. (So, every job has **at least** one job in its conflict set: itself.)

i. Which part(s) establish that $\mathcal{O} - \{c\}$ are a **solution** to the new instance?

ii. Which part(s) establish $\mathcal{O} - \{c\}$'s optimality?

iii. It turns out that at **no** stage of Part 4 do any of the three questions refer to the "new" greedy algorithm's solution. Indeed the algorithm's solution can be **very** different from $\mathcal{O}$. Draw an example instance in which two solutions—both optimal—share exactly one job in common but each contain at least three other jobs.