Introduction to Algorithms: 6.006
Massachusetts Institute of Technology March 3rd, 2009
Professors Sivan Toledo and Alan Edelman Handout 4

Problem Set 3

This problem set is divided into two parts: Part A problems are programming tasks, and Part B problems are theory questions.

Part A questions are due **Tuesday, March 17th** at **11:59PM**.

Part B questions are due **Thursday, March 19th** at **11:59PM**.

Solutions should be turned in through the course website in PDF form using LaTeX or scanned handwritten solutions.

A template for writing up solutions in LaTeX is available on the course website.

Remember, your goal is to communicate. Full credit will be given only to the correct solution which is described clearly. Convoluted and obtuse descriptions might receive low marks, even when they are correct. Also, aim for concise solutions, as it will save you time spent on write-ups, and also help you conceptualize the key idea of the problem.

Exercises are for extra practice and should not be turned in.

Exercises:

- CLRS 6.1-3 (page 130)
- CLRS 6.2-1 (page 132)
- CLRS 6.3-1 (page 135)
- CLRS 6.4-1 (page 136)
- CLRS 6.4-3 (page 136)
- CLRS 6.5-4 (page 141)
- CLRS 8.2-2 (page 170)
- CLRS 8.4-1 (page 177)

Part A: Due Tuesday, March 17th

1. **(50 points)** Gas Simulation

   In this problem, we consider a simulation of n bouncing balls in two dimensions inside a square box. Each ball has a mass and radius, as well as a position (x, y) and velocity vector, which they follow until they collide with another ball or a wall. Collisions between balls conserve energy and momentum. This model can be used to simulate how the

molecules of a gas behave, for example. The world is $400\sqrt{n}$ by $400\sqrt{n}$ units wide, so the area is proportional to the number of balls. (These values are available in the code via the `gas.get_world_max_x()` and similarly-named methods.) Each ball has a minimum radius of 16 units and a maximum radius of 128 units.

Download `ps3_gas.zip` from the class website. For the graphical interface to work, you will need to have pygame or tkinter installed. They currently run slightly different interfaces. Feedback is appreciated.

Run the simulation with `python gas.py`. You may notice that performance, indicated by the rate of simulation steps per second, is highly dependent on the number of balls. If you run the test suite using `python test_detection.py`, you may find that the 2000-ball test cases run for a very long time, or at least longer than you care to wait.

Your goal is to improve the running time of the `detect_collisions` function. This function computes which pairs of balls collide (two balls are said to *collide* if they overlap) and returns a set of `ball_pair` objects for collision handling. You should not need to modify the rest of the simulation. (If you think something else should be modified, e-mail `6.006-staff` with your feedback.)

Full credit will be awarded only for an optimal solution, with partial credit given for solutions with slower asymptotic running times.

(a) **(5 points)** What is the running time of the original `detect_collisions` method in terms of n, the number of balls?

(b) **(15 points)** Improve the `detect_collisions` method. What is the running time of your algorithm? Argue that it is asymptotically faster than the original. You do not need to give a formal proof; be concise, but convincing.

If you wish, you may assume in your analysis that the balls are uniformly distributed throughout the world.

(c) **(25 points)** Implement your algorithm. Put your code in `detection.py` (replacing the original `detect_collisions` method).

Your new code must still find all the same collisions found by the old code (any pair of balls for which `colliding` returns true). To check that you are detecting the same collisions, run your code and the original code with the same parameters, and make sure that they detect the same number of collisions. The test code in `test_detection.py` will also check this for you.

Submit `detection.py` to the class website.

(d) **(5 points)** Using your improved code, after 1000 timesteps with 200 balls, how many collisions did you get? How many simulation steps per second did you run? How many simulation steps per second could you run with the original code and the same number of balls?

Part B: Due Thursday, March 19th

1. **(30 points)** Find the Largest i Elements in Sorted Order

   Given an array of n numbers, we want to find the i largest elements in sorted order. That is, we want to produce a list containing, in order, the largest element of the array, followed by the 2nd largest, etc., until the ith largest. Assume that i is fixed beforehand, and all inputs have $n > i$. (That is, i is chosen beforehand, so that i does not depend on n. In particular, this means that $i = o(n)$.)

   (a) **(5 points)** One idea is to mergesort the input array in descending order, and then list the first i elements of the array. Analyze the running time of this algorithm in terms of n and i.

   (b) **(10 points)** Describe an algorithm that achieves a faster asymptotic time bound than the one in Part (a). Analyze the running time of this algorithm in terms of n and i.

   (c) **(15 points)** Now suppose that the elements of the array are drawn, without replacement, from the set $\{1, 2, ..., 2n\}$. Give an algorithm that solves the problem, with this additional constraint, and analyze its running time in terms of n and i.

   For parts (b) and (c), full credit will be awarded only for an optimal solution, with partial credit given for solutions with slower asymptotic running times.

2. **(20 points)** Dynamic Medians

   Marianne Midling needs a data structure "DM" for maintaining a set S of numbers, supporting the following operations:

   - CREATE( ): Create an empty set S
   - INSERT(x): Add a new given number x to S
   - MEDIAN( ): Return a median of S. (Note that if S has even size, there are two medians; either may be returned. A median of a set of n distinct elements is larger than or equal to exactly $\lfloor (n+1)/2 \rfloor$ or $\lceil (n+1)/2 \rceil$ elements.)

   (Assume no duplicates are added to S.)

   Marianne proposes to implement this "dynamic median" data structure DM using a max-heap A and a min-heap B, such that the following two properties always hold:

   1. Every element in A is less than every element in B, and
   2. the size of A equals the size of B, or is one less.

   To return a median of S, she proposes to return the minimum element of B.

   (a) **(5 points)** Argue that this is correct (i.e., that a median is returned).

(b) **(15 points)** Explain how to implement INSERT(x), while maintaining the relevant properties. Analyze the running time of your INSERT algorithm in terms of n, the number of elements in S.