Introduction to Algorithms: 6.006
Massachusetts Institute of Technology
Professors Ron Rivest and Srini Devadas

October 23, 2007
Handout 13

Problem Set 4

This problem set is due **November 6** at **11:59PM**.
Solutions should be turned in through the course website in PS or PDF form using LaTeX or scanned handwritten solutions, or they may be handwritten and turned in to a member of the 6.006 course staff on or before the due date. Hand-drawn diagrams may also be referenced in your LaTeX writeup and turned in at the next day's recitation.
A template for writing up solutions in LaTeX is available on the course website.

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

- Exercise 6.1-3 from CLRS.
- Exercise 6.2-1 from CLRS.
- Exercise 6.3-1 from CLRS.
- Exercise 6.4-1 from CLRS.
- Exercise 6.4-3 from CLRS.
- Exercise 6.5-4 from CLRS.
- Exercise 8.2-2 from CLRS.
- Exercise 8.4-1 from CLRS.

---

1. **(12 points)** Heap Delete

   The operation HEAP-DELETE(A, i) deletes the item in node i from heap A. Give a pseudocode implementation of HEAP-DELETE that runs in $O(\lg n)$ time for an n-element max-heap, using notation similar to p. 140 of CLRS; you may choose to use Python syntax.

2. Monotone Priority Queues

   A "monotone priority queue" (MPQ) is a data structure that supports the following operations:

   - MAX(Q) - Returns the maximum element in Q. The maximum of a new, empty MPQ is initially ∞. Otherwise, the maximum of an empty MPQ is the last element to have been deleted. Note that MAX(Q) leaves the elements of Q unchanged.

- DELETE-MAX(Q) - If Q is empty, returns MAX(Q). Otherwise, removes and returns MAX(Q). If the queue is empty after the operation, the last deleted value remains the maximum. In other words, MAX(Q) is monotonically decreasing and does not reset when the MPQ is empty.
- INSERT(Q, x) - Inserts x into Q given that $x \leq$ MAX(Q). If $x >$ MAX(Q), then the MPQ is not modified.

For this problem, assume that x is an integer in the range $[0, k]$ for some fixed integer value k.

(a) **(9 points)** Give an implementation of a monotone priority queue that takes $O(m \log m)$ time to perform m operations starting with an empty data structure.

(b) **(9 points)** Give an implementation of a monotone priority queue that takes $O(m + k)$ time to perform m total operations. Hint: Use an idea from COUNTING-SORT.

3. 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 box is 8192 by 8192 units wide, and each ball has a maximum radius of 128 units.

The initial code, featuring an interactive graphical simulation, is given to you at http://courses.csail.mit.edu/6.006/fall07/source/gas.py. You may need to install the pygame module (available from http://pygame.org) for graphics if you don't already have it.

You may notice that performance, indicated by the simulation steps per second rate, slows down significantly as you increase the number of balls. Your goal is to improve the running time of the `detect_collisions` function, which computes whether pairs of balls collide (two balls are said to collide if they overlap) and dispatches to the `handle_collision` function to compute the new ball velocities. You do not need to worry about `handle_collision`.

For this problem, there is no single right answer. We'd like you to explore the techniques we've introduced in class to improve the running time of the simulation.

(a) **(3 points)** What is the running time of `detect_collisions` in terms of n, the number of balls? Do not include the time used by `handle_collision`.

(b) **(15 points)** Write a more efficient `detect_collisions` routine. To be correct, it must still detect any collisions. The code provided does correctly calculate whether balls collide, so you can use it to compare against your results.

Submit your version of gas.py, containing an improved `detect_collisions` routine. You may find the option to automatically pause after a certain number of timesteps useful, as well as the count of total collisions to spot if something is wrong.

(c) **(9 points)** Explain how your `detect_collisions` algorithm is asymptotically faster than the original implementation. We do not expect a formal proof here, but give justifications where you can. Again, do not include the time used by `handle_collision`.

(d) **(3 points)** After 2048 timesteps, what is the ratio of the increase in simulation steps per second of your version compared to the given code for $n = 100$? $n = 200$? How many total collisions are counted in each case? (For this problem, use the interface at the starting screen to set the number of balls and to pause at 2048 steps.)