Introduction to Algorithms: 6.006
Massachusetts Institute of Technology — March 7, 2008
Professors Srini Devadas and Erik Demaine — Handout 5

Problem Set 3

This problem set is due **Thursday March 20** 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)

---

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

   In this problem you will implement an additional operation in a min heap. Namely, `heap.delete(i)` deletes the item at index i in the array.

   Download `ps3_heap.zip`. In `heap_delete.py`, which inherits from the code in `heap.py`, implement `delete(i)`. Your code should run in $O(\log n)$ time where n is the number of nodes currently in the heap.

   The added code doesn't need to be more that a couple lines, but be sure to add comments explaining why it works.

   Run `test_heap_delete.py` to help determine if your new delete method works, and submit `heap_delete.py` to the class website.

2. Monotone Priority Queues

A "monotone priority queue" (MPQ) is a (max) priority queue that only allows monotonically decreasing elements to be extracted. It supports the following operations:

- `Max`(Q): Returns the key of the most recently extracted node. If no nodes have been extracted, returns ∞. This does not modify the MPQ.
- `Extract_Max`(Q): Removes and returns the maximum node currently in Q, and updates `Max(`Q`)`. If Q is empty, returns `Max(`Q`)`.
- `Insert`(Q, x): Inserts x into Q given that $x \leq$ `Max`(Q). If $x >$ `Max`(Q), then the MPQ is not modified.

When asked to "describe an implementation", you may start with something already proven in class or in the book, and simply describe modifications to that.

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

Solution:

Simply start with an ordinary max_heap priority queue, and modify it for the new functionality. Keep track of `Max`(Q) separately. On insert, do an extra check to see if the new element is bigger than `Max`(Q). On `Extract_Max`(Q), update the `Max`(Q). All operations still take the same $O(\log m)$ time, because the heap has at most m elements, so in total, at most $O(m \log m)$ work is done.

(b) **(9 points)** Now suppose that every inserted key x is an integer in the range $[0, k]$ for some fixed integer value k. Describe an implementation of such a monotone priority queue that takes $O(m + k)$ time to perform m total operations.

Hint: Use an idea from `Counting_Sort`.

Warning: Be careful about the case when the queue becomes empty.

Solution:

Use an array A, where $A[i]$ stores the number of elements with $key = i$ currently in the queue. Again, the max is stored separately. Inserting takes $O(1)$ time (if the key is less than the max, increment $A[key]$). Extracting the max is done by starting at the previous max in the array, and trying every lower slot until one is found that isn't empty. Over the lifetime of the queue, we will only walk down each slot once, and the rest of the work done is constant time.

We need to be careful that when the queue is empty, if we try to extract max then we don't waste time walking down slots in the array, so instead we separately keep track of the size of the array. This way we can figure out in $O(1)$ time whether the queue is empty.

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 world is $400\sqrt{n}$ by $400\sqrt{n}$ units wide, so the area is proportional to the number of balls. 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.

   You may notice that performance, indicated by the rate of simulation steps per second, is highly dependent on the number of balls. 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.)

   (a) **(2 points)** What is the running time of `detect_collisions` in terms of n, the number of balls?

   Solution: $\Theta(n^2)$

   (b) **(6 points)** Argue that the following algorithm is asymptotically faster:
   Divide the world into square bins of width 256. For each ball, put it in its appropriate bin. Then for each bin, check for collisions where either both balls are in the bin, or one ball is in the bin, and the other ball is in an adjacent bin.

   Solution: Each ball can be put in a bin in constant time, so binning the balls takes time $O(n)$.

   Each bin can hold at most a constant number of balls (because every ball has a minimum radius, and each bin has a finite radius). So checking a ball for collisions within a bin, and within neighboring bins, requires checking against at most $O(1)$ other balls. Thus, the total time required to check every ball is still $O(n)$.

   (c) **(20 points)** Implement the `detect_collisions` algorithm described in part (b). Put your code in `detection.py`, and uncomment the line in `gas.py` just below `detect_collisions` that imports your new code.

   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.

   In your documentation, do not assume that the reader has read this problem set.

   Submit `detection.py` to the class website.

(d) **(2 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?