Introduction to Algorithms: 6.006
Massachusetts Institute of Technology October 7, 2008
Professors Ronald L. Rivest and Sivan Toledo Handout 3

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, October 21** at **11:59PM**.

Part B questions are due **Thursday, October 23** 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, October 21

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

(b) **(30 points)** Improve the `detect_collisions` method. Come up with an asymptotically faster algorithm than the one from part (a), then implement it. Put your code in `detection.py`, and uncomment the line in `gas.py` that imports your new code (just below the `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.

Submit `detection.py` to the class website.

(c) **(12 points)** Argue that your part (b) algorithm is asymptotically faster. You do not need to give a formal proof; be concise, but convincing.

(d) **(4 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, October 23

1. **(25 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.)

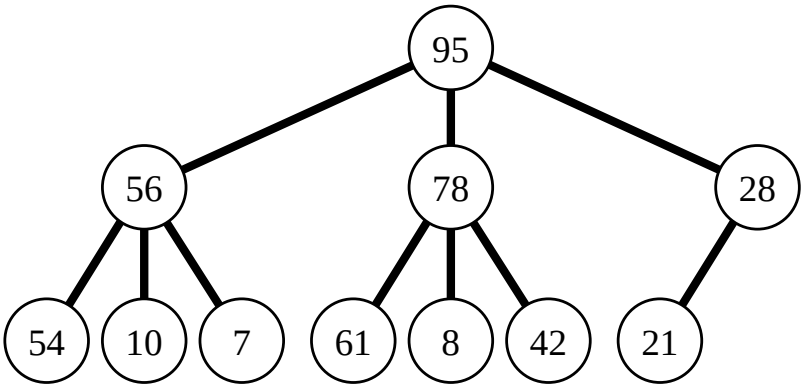
(a) **(5 points)** One idea is to sort the input array in descending order, and then list the first i elements of the array. Analyze the worst-case running time of this algorithm.

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

(c) **(10 points)** Now suppose that the elements of the array are drawn, without replacement, from the set $\{1, 2, ..., 2n\}$. Given this additional constraint, can you improve upon your part (b) running time? If so, describe your algorithm and analyze its running time in terms of n and i. If not, why not?

2. **(25 points)** d-ary Heaps

In class, we've seen binary heaps, where each node has at most two children. A d-ary heap is a heap in which each node has at most d children. For example, this is a 3-ary heap:



(a) **(2 points)** Suppose that we implement a d-ary heap using an array A, similarly to how we implement binary heaps. That is, the root is contained in $A[0]$, its children are contained in $A[1 \ldots d]$, and so on. How do we implement the PARENT(i) function, which computes the index of the parent of the ith node, for a d-ary heap?

(b) **(2 points)** Now that there might be more than two children, LEFT and RIGHT are no longer sufficient. How do we implement the CHILD(i, k) function, which computes the index of the kth child of the ith node? ($0 \leq k < d$)

(c) **(5 points)** Express, in asymptotic notation, the height of a d-ary heap containing n elements in terms of n and d.

(d) **(5 points)** Give the asymptotic running times (in terms of n and d) of the HEAPIFY and INCREASE-KEY operations for a d-ary heap containing n elements.

(e) **(8 points)** Let's suppose that when we build our d-ary heap, we choose d based on the size of the input array, n. What is the height of the resulting heap (in terms of n) if we choose $d = \Theta(1)$? What if $d = \Theta(\log(n))$? What about $d = \Theta(n)$?

(HINT: remember that $\log_d(n) = \frac{\log(n)}{\log(d)}$. This might simplify your expressions a little.)

(f) **(3 points)** How does the choice of d affect the running times of HEAPIFY and INCREASE-KEY?