Precept Topics

- Quicksort
- Heaps and Priority Queues

Relevant Material

- Book chapters: 2.3, 2.4 and 2.5.

A. RECAP: Quicksort + Priority Queues

Your preceptor will briefly review key points of this week's lectures,

B. EXERCISE: Runtime Analysis

Suppose we have the following code which uses a binary-heap based ***minimum priority queue*** (MinPQ). Assume that $n > k$, and that `a[]` is an array containing random integers.

```
1  void foo(int k, int[] a) {
2        MinPQ<Integer> pq = new MinPQ<Integer>();
3        int n = a.length;
4
5        for (int i = 0; i < n; i++) {
6              pq.insert(a[i]);
7              if (pq.size() > k) pq.delMin();
8        }
9
10       for (int i = 0; i < k; i++)
11             System.out.println(pq.delMin());
12 }
```

(a) Describe what the code outputs in terms of the array `a[]` and the parameter k.

(b) What is the order of growth of the running time of the code as a function of both n and k?

(c) Suppose we were to remove line **7**. What would the code's output and order of growth be?

B. EXERCISE: Streaming Median (Spring '18 Midterm)

Suppose that your application receives a stream of data and it should, at any point, report the *median* of the data received so far. Our goal is to design a data type that can support such median queries on data streams efficiently.

```
1 public class StreamingMedian<Item extends
2 Comparable<Item>> {
3       public StreamingMedian() {...}
4       public void insert(Item key) {...}
5       public Item median() {...}
  }
```

Note. The median of a set of elements is the middle element when the elements are considered in sorted order. If there is an even number of elements, the median is the smaller of the two middle elements. For example, the median of both `[1, 2, 3]` and `[1, 2, 3, 4]` is 2.

(a) Describe an implementation that can support the `insert()` and `median()` operations in $O(n)$ time, where n is the number of elements seen so far. Mention whether your analysis of the running time is *worst-case*, *amortized* or *probabilistic*.

(b) Describe an implementation that can support the `insert()` and `median()` operations in $O(\log n)$ time, where n is the number of elements seen so far. An amortized bound is fine.
Hint: use two priority queues.

EXERCISE (optional): Hacking Quicksort

You have seen, in lecture, that quicksort has $\Theta(n^2)$ worst-case runtime – but this is very (very!) unlikely to happen if we run `StdRandom.shuffle()` beforehand. Let's calculate how unlikely that really is.

Let `a` be an array of length n. Recall that the worst case occurs when `StdRandom.shuffle(a)` turns out to sort the array (either in increasing or decreasing order). Assume in this problem that all keys are distinct, and feel free to use a calculator.

(a) What is the probability `StdRandom.shuffle(a)` sorts `a` when $n = 10$? How about $n = 100$?
How do these compare against the number of nanoseconds since the Big Bang?
Do the same with any other unfathomably large number of your choice.

(a) Now let's try our hand at a probabilistic argument that shows quicksort usually makes significant progress. Call a pivot choice *bad* if it divides a subarray very unevenly: less than 10% of its size on one side, and more than 90% on the other. Call the pivot *good* otherwise.

What is the probability that the first 10 pivot choices are all bad? How about the probability that the first 100 choices are bad?

(b) Let's try to attack the quicksort implementation shown in lecture. Our goal is to carefully craft worst-case inputs that force quicksort to always take $\Theta(n^2)$ time.

Suppose you've managed to get your hands on the *seed* used in `StdRandom.shuffle()`. This means you can reset the seed (of your system) then run `StdRandom.shuffle()` to obtain the exact same shuffle as the quicksort method would.

How can you use this knowledge to design worst-case inputs for any given array length n?