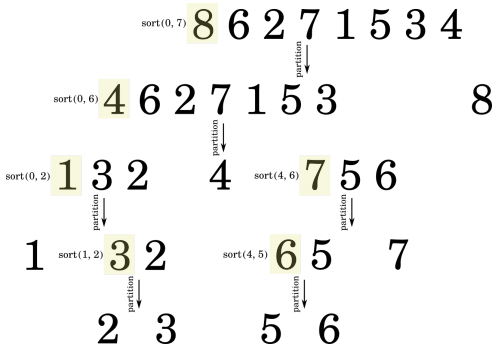
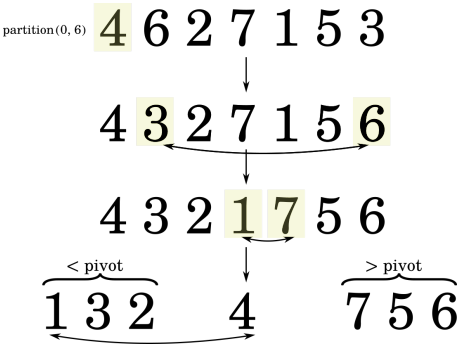
Precept Outline
- Review of Lectures 7 and 8:
  - Quicksort
  - Heaps and Priority Queues

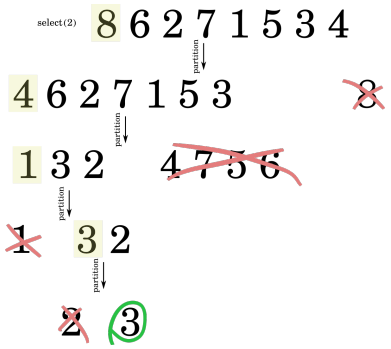
Relevant Book Sections
- Book chapters: 2.3, 2.4 and 2.5

A. Review: Quicksort + Heaps

Your preceptor will briefly review key points of this week's lectures. Here are some images representing examples they will show you, for partition, quicksort and quickselect.







B. Priority Queues

Part 1: Runtime

Consider the following code which uses a binary-heap based *minimum priority queue* (MinPQ). Assume that $n \geq k$, and that `a[]` is an **immutable** array containing arbitrary integers.

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

Give a succinct description (that could, e.g., be a comment before the first line) of what the method `foo` prints in terms of the array `a[]` and the parameter k.

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

Suppose we were to remove line 5. What is the code's output? And what is the order of growth?

Part 2: Data Structure Design (Fall'19 Midterm)

Design a data type to implement a *double-ended priority queue*. The data type must support inserting a key, deleting a smallest key, and deleting a largest key. (If there are ties for the smallest or largest key, you may choose among them arbitrarily.)

To do so, create a `MinMaxPQ` data type that implements the following API:

```
public class MinMaxPQ<Key extends Comparable<Key>> {
    MinMaxPQ()                                  // create an empty priority queue
    void insert(Key x)                          // add x to the priority queue
    Key min()                                   // return a smallest key
```

```
    Key max()                                   // return a largest key
    Key delMin()                                // return and remove a smallest key
    Key delMax()                                // return and remove a largest key
}
```

Here are the performance requirements:

- The `insert()`, `delMin()`, and `delMax()` must take time proportional to $\log n$ or better in the worst case, where n is the number of keys in the priority queue.
- The `min()` and `max()` methods must take constant time in the worst case.

In your answer mention: the instance variables you'll use, your implementation of `min()`/`max()`, your implementation of `insert(x)` and your implementation of `delMin()`/`delMax()`.

Notes: To describe your solution, use either English prose or Java code (or a combination of the two). If your solution uses an algorithm or data structure from the course, do not reinvent it; simply describe how you are applying it.

C. Algorithm Design: The Hotel Problem

You are the manager of a (tall and narrow) hotel with one room in each of n floors. Early in the morning (before you arrived), n guests showed up at the same time, told the front desk their preference – a floor that is either "high" or "low" – and were assigned rooms arbitrarily (without consideration for their preferences).

Your task is to correct this assignment by swapping pairs of guests, relocating each guest at most once. By the end of the process, all guests who prefer low floors should be on floors below those who prefer high floors. Moreover, no guest may end up in a room they prefer less than their original assignment ("high"-preferring guests may only be moved up, and "low"-preferring may only be moved down). Your solution should take time $O(n)$.