Selection problem (k^{th} largest element):

Input: Array, List or stream, integer k

Output: k^{th} largest element in collection, or

k largest elements.

Internal version

- Array/List.. fits in memory.
- study later – algorithm based on ideas from QuickSort
  $RT = O(n)$, expected time.

External version.

- Data is too big to fit in memory.

Solved using Priority Queues if k fits in memory.

$RT = O(n \log k)$.

Extend version of Selection problem:

Naive ideas: (a) Sort data and take k^{th} largest – infeasible.
(requires multiple passes over the data)

(b) Put the data into a Priority Queue → remove() k times.
(max heap)
– Not enough memory.

Idea: Scan the stream – Keep track of the best k seen so far. – store k largest elements in a min heap (Priority queue with natural ordering)

Algorithm: klargest (Iterator <Integer> iter, int k)

```
PriorityQueue <Integer> q = new PriorityQueue<>();

k       for (i=0; i<k; i++) {
1           if (! iter.hasNext()) { throw new Exception("Not enough elements")
                                    or return null? } ?
logk        q.add(iter.next());
        }
n-k     while (iter.hasNext()) {   //LI: q has the k largest elements seen so far.
1           x = iter.next();
            ~~if x > q.peek~~
1           if (x.compareTo(q.peek()) > 0) {
logk            q.remove();
logk            q.add(x);
            }
        }
        return  -> q.peek()              <- only the k^th largest
                -> q.toArray()           <- all k elements in no particular order
                -> Collections.sort(q)   <- output in sorted order.
```

RT Analysis: Heap has at most k elements at any time.

Each operation of PQ is $O(\log k)$.

Total number of operations: $k(1+\log k) + (n-k)(2+2\log k)$

$= O(n \log k)$.

Another application of PQ: Heap Sort:

Create a binary heap with the elements of the given array.

```
i = 0;
while (! q.isEmpty()) { arr[i++] = q.remove(); }
```

buildHeap() (sometimes called heapify()).

given an array of elements - place them in heap order.

top down

```
for (x : arr) { q.add(x); }
```

RT = $\log 1 + \log 2 + \dots + \log n$
$= O(n \log n)$.

X Too big.

bottom up:

Place elements into q all at once.

array in priority queue = pq

```
for (i = parent(size-1); i>=0; i--){
    percolateDown(i);
}
```

✓ RT $= O(n)$.

RT Analysis of bottom-up buildHeap:

In a complete binary tree, there are $\frac{n}{2^{h+1}}$ nodes at height h.

Worst RT of percolateDown(i) = height of node at index i.

$$RT = \sum_{h=0}^{\log n} \frac{n}{2^{h+1}} \cdot h = n \cdot \sum_{h=1}^{\log n} \frac{h}{2^{h+1}} \leq 2n = O(n)$$

Let $x<1$ $\quad \frac{1}{1-x} = 1 + x + x^2 + \dots = f(x)$.

$$x \cdot \frac{d}{dx}(f(x)) = \frac{x}{(1-x)^2} = \underbrace{x + 2x^2 + 3x^3 + \dots}_{=2}$$

Choose $x = \frac{1}{2}$

Sorting algorithms

Overview:

$O(n^2)$ algorithms:
- Selection sort
- Bubble sort
- Insertion sort

] - Don't use these unless ??

Questionable algorithms - shellsort - RT = ?? - No
- Religion.

O(n log n) algorithms:

1. Heap sort - $O(1)$ extra space, $O(n \log n)$ time.
   - not used because MergeSort is better
2. Merge sort - $O(n)$ extra space, $O(n \log n)$ time.
   - best algorithm for sorting.
3. Quick sort - $O(\log n)$ extra space (for recursion),
   (Randomized algorithm)
   $O(n \log n)$ expected time.

   Version: Dual pivot QuickSort
   - best algorithm known up to some n
     $n \sim 10M$.

O(n) algorithms:

Special algorithms that apply under special situations.

1. Counting sort: elements are integers $1..10n$.
2. Radix sort: elements are composed of digits
   $RT = O(d(n+k))$. k = # of values per digit, d = no. of digits
   if $d = O(1)$, $k = O(n)$: $RT = O(n)$.
3. Bucket sort.

Merge Sort : Divide and conquer algorithm to sort an array. — Recursive.

Idea: split array into 2 equal halves.
Sort each subarray.
Merge them into one sorted sequence

$T \approx 16$?

```
mergeSort ( arr ) :
    tmp = new array same
          size as arr
T[] tmp = (T[])new Comparable(
                      arr.length)
    mergeSort(arr, tmp, 0,
              arr.length)
```

```
merge (arr, tmp, leftStart,
       rightStart, rightEnd) :
// Merge arr[leftStart ... rightStart-1]
// and arr[rightStart .. rightEnd - 1]
// into arr[leftStart .. rightEnd - 1]
// in sorted order.
```

Next class

```
mergeSort (arr, tmp, left, n) :
// Sort n elements starting at arr[left]
   if n < T then
       insertionSort (arr, left, n)
   else
     Ln ← n/2
     mergeSort (arr, tmp, left, Ln)
     mergeSort (arr, tmp, left+Ln,
                n-Ln)
     merge (arr, tmp, left,
            left + Ln, left + n)
```