Based on algorithms and discussion from Weiss, *Data Structures and Algorithm Analysis in JAVA*, Third Edition. Pearson.

The selection problem

Given a group of N numbers, determine the *k*-th largest.

Example, for the numbers 2, 15, 6, 25, 10, 5, 9
If k = 1, the kth largest is 25.
If k = 2, the kth largest is 15. Etc.

For the examples below, use the list above and assume k = 3.

Solution 1 (Chapter 1)– read numbers into an array, sort the numbers into *decreasing* order, then return the element in position *k* (which would be at index k-1 in a Java array). Efficiency O(n^2), assuming a simple sort like selection sort (or O(n log n) if you use mergesort or quicksort, Chapter 7).

Ex. Sort numbers into the order: 25, 15, 10, 9, 6, 5, 2, return 10 as answer (3rd largest).

Solution 2 (Chapter 1)– read the first *k* numbers into an array and sort them into *decreasing* order. Read the remaining elements one by one, ignoring all numbers smaller than the *k*-th element. For numbers larger than the *k*-th element, place it in its correct position in the array, bumping one of the other elements out of the array. When the algorithm ends, the element in the *k*-th position is returned as the answer. Efficiency O(n*k*), if *k* = n/2 (the median), its O(n^2).

Ex. Sort first 3 numbers: 15, 6, 2, then consider others: 25 put in correct position in list, knocking 2 out. List is now: 25, 15, 6. Next, 10 put in list, resulting in 25, 15, 10. Next two values (5 & 9) ignored because they are less than 10. Done: return 10 as answer.

Weiss: a simulation involving a random file of 30 million elements and k = 15,000,000 will show that either algorithm requires several days of computation to complete. (You should verify this.)

Solution 3 (Chapter 6) – This algorithm uses a max priority queue, implemented as a binary heap. Read the n elements into an array and then apply **buildHeap**. Then perform *k* deleteMaxs, the last element deleted is the answer. Efficiency: O(n) to build heap, followed by *k* deleteMins, each being O(log n) for a total of O(k log n). The total efficiency is described by O(n + *k* log n). For small values of *k*, the buildHeap dominates given O(N), for larger values of *k*, the deleteMins dominate; if k = n/2 (the median), then the efficiency of O(n log n).

Ex. The heap built from the array above using BuildHeap would be the following (in its array storage): 25, 15, 9, 1, 10, 5, 6, the third deleteMax would return 10, and the remaining heap looks like this: 9, 6, 5, 2

Based on algorithms and discussion from Weiss, *Data Structures and Algorithm Analysis in JAVA*, Third Edition. Pearson.

Solution 4 (Chapter 6) – Similar to Solution 2 & 3 but uses a min binary heap, rather than array or max heap, to store the data. The first *k* elements are put into the array and then BuildHeap is applied, at a cost of O(*k*). (The *k*th largest, of those seen so far, is at the root – it is the minimum element in the set, S, stored in the heap.) We then go through the rest of the elements, and for those that are smaller than the root, we ignore them; those that are larger than the root, we have to remove the element at the root and insert this new element. The cost is O(1) to determine if the element is to be ignored or inserted, for each remaining element, plus O(log k), to delete the root and insert the new element if necessary. The total time is $O(k + (n - k)\log k) = O(n \log n)$.

__**Ex.**__ Binary heap of size three created with first three elements: 2, 15, 6 (2 is the 3rd largest element seen thus far).
Next 25 is read, 2 is removed and 25 is added and the min-heap rebuilt, 6, 15, 25.
Next 10 is read, 6 is removed and 10 added, 10, 15, 25
Next two numbers 5 & 9 ignored, since they are less than 10. 10 returned as answer.

Solution 5 (Chapter 7) - based on the QuickSort partitioning strategy. Partition the list around some pivot value *v* into S_1 (elements less than *v*) and S_2 (elements greater than *v*).

- If $k < |S_1|$, then the kth smallest is in S_1. Repeat the partitioning on S_1, looking for the kth smallest in S_1.
- If $k = |S_1| + 1$, then *v* is the kth smallest element. Stop.
- Otherwise, the kth smallest element is in S_2. Repeat the partitioning on S_2, looking for the $(k - = |S_1| - 1)$-st element in S_2.

This algorithm, called QuickSelect, does less work than QuickSort, as it makes only one recursive call. The expected running time is O(n) – worst case would be $O(n^2)$, like quicksort.

__Ex.__ Assume we use the median of three technique for chosing pivot (look at first last and middle element and choose the one with median value, in this case it is 9)
First partition results in: 2 5 6 **9** 10 15 25, since pivot (9) in position 4, not 3, we do a partion on those element to the right of 9, now looking for the newK = 3-3+1 = 1st element.
Second partition, pivot is now 15 results in: 10 **15** 25, pivot in location 2, partition again on left hand side, but since it is of size 1, no partition necessary, returns 10.