Quick Select

The selection problem is as follows:

Given a list of n numbers, find the kth smallest number in the list.

One obvious solution to this problem is as follows:

1) Sort the numbers.
2) Return the value stored in index k-1. (Since arrays are 0-based.)

Based on what we've learned, we know that this algorithm would run in O(nlgn) time.

We might ask if we can do better than this. We know for a fact that in order to gauge what the kth smallest element was, we HAVE to look at each element at least once. Thus, it stands to reason that the best possible run-time we could ever hope to achieve is O(n).

Quick Select, which is based on the Partition function that Quick Sort uses, achieves an average run-time of O(n). (It's worst case run-time is $O(n^2)$ just like Quick Sort, but is very unlikely.)

Basic Idea behind Quick Select

Imagine running Partition on an array of n elements. When the partition is done, it returns an integer, (call this m), which is the index where the partition element has been stored.

Note that the rank of this partition element is m+1.

Thus, if k, the rank of the element we are looking for just happens to equal m+1, we are done!

For example, consider partitioning the following array:

index	0	1	2	3	4	5	6
value	5	2	9	6	1	3	8

The partition produces the following:

index	0	1	2	3	4	5	6
value	1	2	3	5	6	9	8

and returns the value 3, which is the index of the partition element (which was 5).

Consider the situation where we were looking for the 4^{th} smallest element in this array. We know that 5, which is in its correct sorted location IS that element, so we can just return it.

But, this only happens sometimes.

There are two other possibilities:

1) The rank of the element we are looking for is LESS THAN m+1.

2) The rank of the element we are looking for is GREATER THAN m+1.

In the first case, we must only search for our element to the left of the array.

In the second case, we must only search for our element to the right of the array.

Thus, in both cases, we only make ONE recursive call.

Let's go back to our original example, but this time consider searching for the 2nd smallest item in the array. Once again, let's look at the results of our partition:

index	0	1	2	3	4	5	6
value	1	2	3	5	6	9	8

We know that we only need to search in the array from index 0 to index 2, and that within this range we are STILL looking for the second smallest value. This is all the information we need for our recursive call.

Now, consider searching for the 6th smallest value in the array. If we take a look at our original partition:

index	0	1	2	3	4	5	6
value	1	2	3	5	6	9	8

we see that we now want the 2nd smallest value in the array on the right, starting at index 4 and ending at index 6. The reason for this is that there are four values from the old array excluded from our search (these are 1, 2, 3, and 5), so now, instead of looking for the 6th smallest item, we are NOW looking for the 6 – 4 = 2nd smallest item in the array [6 9 8].

This is all the information we need for the other recursive call!

Let's take a look at some code that implements this algorithm:

```
int qsel(int* numbers, int low, int high,
int rank) {

  if (low == high)
    return numbers[low];

  int sp = partition(numbers,low,high);

  if (rank == sp-low+1)
    return numbers[sp];

  else if (rank < sp-low+1)
    return qsel(numbers, low, sp-1, rank);

  else
    return qsel(numbers, sp+1, high,
                         rank-(sp-low+1));

}
```

Quick Select Analysis

In the best case, the partition works the first time around and we find the element in O(n) time.

In the worst case, Quick Select runs identical to the worst case of Quick Sort. The partition element is always the greatest value (or least value) of the ones remaining and the rank of the item for which we are looking doesn't get revealed till the very end. In this situation, we have to run partition n-1 times, the first time comparing n-1 values, then n-2, followed by n-3, etc.

This points to the sum 1+2+3+...+(n-1) which is (n-1)n/2. Thus, the worst case running time is $O(n^2)$.

The analysis of the average case is beyond the scope of this class. Needless to say, the recurrence relation for an arbitrary run of Quick Select looks like:

T(n) = T(k) + O(n), where k represents the size of the new array in which the search has been restricted. The idea for the average case analysis is to go through all possibilities of quick select running, weighting each on the probability of where the partition element will end up. (This process turns out to be even more difficult than the Quick Sort analysis.)