ESO207A: Data Structures and Algorithms	
Notes:	*Selection Problem*

The *selection* problem takes an array $A[1,\ldots,n]$ and an parameter $1 \leq k \leq n$ and returns the kth smallest element. More generally, it takes an array segment $A[p,\ldots,r]$ and returns the kth smallest element for $1 \leq k \leq r-p+1$. For example, if $k=1$, then $Select(A,p,r,1)$ returns the smallest element in the segment $A[p,\ldots,r]$ (ties are broken arbitrarily).

One popular solution to the *selection* problem is via the *Partition* procedure. The procedure $Partition(A,p,r,v)$ takes a value v and returns a pair of indices (l,m) that splits the array $A[p,\ldots,r]$ into three parts as follows.

1. All values in $A_L = A[p,\ldots,i]$ are $< v$ (left partition)
2. All values in $A_M = A[i+1,\ldots,j]$ equal v (middle partition).
3. All values in $A_R = A[j+1,\ldots,r]$ are $> v$ (right partition).

Suppose we wish to solve $Select(A,p,r,k)$ to find the kth smallest element in $A[p\ldots r]$. First we call $Partition(A,p,r,v)$, where, we choose v to be some value of $A[p]\ldots A[r]$ (any value will do, for now). Assume $1 \leq k \leq r-p+1$. Suppose $Partition(A,p,r,v)$ returns (l,m). We then have the following recursive equation.

$$Select(A,p,r,k) = \begin{cases} Select(A,p,l,k) & \text{if } l-p+1 \leq k \\ v & \text{if } l-p+1 < k \leq m-p+1 \\ Select(A,m+1,r,k-(m-p+1)) & \text{otherwise.} \end{cases}$$

We can write this in pseudo-code and add a statement for the exit condition of recursion.

```
SELECT(A,p,r,k) // Finds kth smallest element in A[p...r]. Assumes 1 ≤ k ≤ r-p+1
1.  if p == r return A[p]
2.  Choose a pivot element v from A[p]...A[r]
3.  (l,m) = Partition(A,p,r,v)
4.  if l-p+1 ≤ k
5.      return SELECT(A,p,l,k)
6.  elseif l-p+1 < k ≤ m-p+1
7.      return v
8.  else
9.      return SELECT(A,m+1,r,k-(m-p+1))
```

Efficiency Analysis. The running time of the algorithm basically depends on the choice of v. Notice that we have chosen v to be lie in $A[p,\ldots,r]$. This means that the middle segment A_M has size at least 1 and is therefore non-empty. However, if we choose v to be the always the largest element of the input argument to *Partition* of an array of initial size n, then, after the first call, the partition A_L would be of size $n-1$ and $A_M = 1$ (A_R = empty). Thus, the array shrinks by

only one. If this is repeated, the subsequent array would be of size $n-2$, and so on. If we were to find the median using such a partition, then the selection algorithm will take time

$$\Theta(n) + \Theta(n-1) + \ldots + \Theta(n/2) = \Theta(n^2) \ .$$

Now suppose we choose an index s randomly from $[p, \ldots, r]$ and with equal probability $1/(r-p+1)$ and use $v = A[s]$ as the pivot element. The above example then would be extremely unlikely (can you calculate the probability of this happening?). In view of this, one would expect that while the worst case time still can be $\Omega(n^2)$, the *expected* time should probably be much less. Fortunately, it lies very close to the best case running time.

To distinguish between lucky and not so lucky choices of v, call v good if it lies within 25th to 75th percentile of the array segment it is chosen from. Why are these pivots good? Because they ensure that the size of A_L and A_R is at most three fourths of the size of the original array segment A. (you see why?) But this means that half of the elements of A are "good" choices.

Given that a randomly chosen v has 50% chance of being good, how many v's do we need to pick on average before we get a good v? An equivalent re-formulation is the following.

Fact. On expectation, a fair coin has to be tossed two times for a "heads" to show up.

Proof. 1. By definition of expectation, the probability that a heads appears for the first time after k tosses is $\frac{1}{2^k}$. So the expectation is

$$\sum_{k=1}^{\infty} k \cdot \frac{1}{2^k} = 2 \ .$$

2. Let E be the expected number of coin tosses when a heads is first seen. We toss the coin once. With probability 1/2, it is heads and we are done. Otherwise, with probability 1/2, it is tails, and we are exactly in the same situation as we started and so we repeat. This gives

$$E = 1 + \frac{1}{2} \cdot E$$

which simplifies to $E = 2$.

Let us consider this *Randomized_Partition* procedure, which is the same as the earlier Partition procedure, except that the choice of the pivot element is made randomly and with equal probability among all the elements of the array segment on which it is being called. So, we see that after two calls to Partition, the array will shrink to at most three-fourths of its size. We can argue this as follows. Let $T(n)$ be the *expected* running time of the *Select* procedure on an array segment of size n. Then, we can write

$$\begin{aligned} &\text{Time taken on an array of size } n \\ \leq &(\text{Time taken on array of size } 3n/4) + (\text{time to reduce array size to} \leq 3n/4) \end{aligned}$$

Now take *Expectation* of both sides and using the fact that *the expectation of a sum is the sum of the expectations*, we obtain

$$T(n) \leq T(3n/4) + O(n) \ .$$

This is because two (successive) calls to Partition take time $O(n)$. Solving the recurrence, we get $T(n) = O(n)$.