QUESTION 2

```
object Partition {
    def pivot (a: Array[Int], l: Int, r: Int) = ...

    // (a)
    def partition (a: Array[Int], l: Int, r: Int): (Int, Int) = {
        var p = pivot (a, l, r)
        // invariant i: a[l..i) < p && a[i..j) = p && a[k..r) > p && l ≤ i < j ≤ k ≤ r &&
a[0..l) = a0[0..l) && a[r..N) = a0[r..N) && a[l..r) is a permutation of a0[l..r)
        var i=l ; var j=l+1; var k=r
        while (j < k)
        {
          if (a(j) == p) j+=1
          else if (a(j) < p) {var t= a(i); a(i)=a(j); a(j)= t; i+=1; j+=1 }
          else { var t= a(j); a(j)=a(k-1); a(k-1)=t; k-=1}
        }
        (i,j)
    }
    // (b)
    def find (a: Array[Int], i: Int, l: Int, r: Int): Int ={
        var (m,n)= partition (a, l, r)
        if (i < m) return find (a, i, l, m) // l ≤ i < m, so we search in the left part
        else if (i<n) return a(m) // m ≤ i < n, so it's in the middle part, which is all
                                  equal to a(m)
        else return find (a, i, n, r) // n ≤ i < r, so we search in the right part
    }
    // (c)
    def select (a: Array[Int], i: Int, n: Int): Int = {
        return find(a, i, 0, N)
    }
```

(d) By choosing a pivot which would split the partitioned part into two or three equal sides (as size), the find function will run logarithmically, because at each step, we either find what we need, or we reduce the problem by half or by a third.

In the worst-case scenario, if the pivot is the smallest/biggest element of the sequence we want to partition, we end up with a linear-time "find" as we reduce the problem by 1 at each step.

(e) To get rid of the recursion, we keep track of the left and right bounds where we look after the i^{th} smallest value of the array.

```
def findNonRec(a: Array[Int], i: Int, l: Int, r: Int): Int = {
    var (m,n) = partition(a, l, r)
    var left = l ; var right = r
    while ((m > i) || (n <= i))
    {
      if (i < m) right = m
      else left = n
      var double = (0,0)
      double = partition(a, left, right)
      m = double._1 ; n = double._2
    }
    return a(m)
}

def selectNonRec(a: Array[Int], i: Int, N: Int): Int = {
    return findNonRec(a, i, 0, N)
}
```