IMPERATIVE PROGRAMMING HT2018

SHEET 3

GABRIEL MOISE

Question 1

```
object Question1

{

/** Find index i such that a[0..i) < x <= a[i..N).

* Pre: a is sorted. */

def search(a: Array[Int], x: Int) : Int =

 {

   val N = a.size

   // Invariant I: a[0..i) < x <= a[j..N) && 0 <= i <= j <= N

    var i = 0; var j = N

    while(i < j)

    {

     val m = i + (j-i) / 2 //instead of val m = (i+j)/2 (for part (c)

     // i <= m < j

     if(a(m) < x) i = m+1 else j = m

    }

    // I && i = j, so a[0..i) < x <= a[i..N)

    i

  }

}

/**
```

(a) If the array is not increasing we cannot be sure that we get the right answer whether x is in the array or not. We make comparisons between the element from the middle of the interval we constrained our search to, but that doesn't mean anything if the array is not in increasing order, because the element can be anywhere regardless of the comparisons. So, based on comparisons, we increase the left limit or decrease the right one until we get to a point which can be or not the element we are looking for (the probability to find the good answer is very small).

```
scala> Question1.search(Array(1,2,3,8,5,6,7),6)

res5: Int = 3
```

(b) In N is zero, or in other words, the array is empty, we set i to be 0 and j to be 0 so we don't enter the while loop, therefore we simply return i, which is 0.

(c) The problem appears when (i+j) is greater than or equal to 2^{31}, for arrays with at least 2^{30} elements we can have that, but we can fix this by replacing the line val m = (i+j)/2 with val m = i + (j-i)/2, which is equivalent to that, so that we don't exceed the Int range.

```
*/
```

Question 2

```
object Question2
```

```
{
  def binary (y: Int) : Int =
    {
      // Trivial cases:
      if (y==0) return 0
      if (y==1) return 1
      // Invariant I: a^2 <= y < b^2 and 0 <= a < b
      var a = 0; var b = y // problems with overflow at part (c)
      while(a+1 < b)
      {
        var m = (a+b)/2 // a < m < b
        if  (m <= Math.pow(y,0.5)) a = m
            else b = m
      }
      // a^2 <= y < (a+1)^2
      a
    }
  /** Pre : integer y > 0
      Post : integer a >= 0 such that a^2 <= y < (a+1)^2 */
  def ternary (y : Int) : Int =
    {
      // We first return the special cases y=0 and y=1 so that by starting with
      // right=y we can say that right^2>=y (by setting right=y+1 we have a problem at part (c) with the overflow)
      if (y==0) return 0
      if (y==1) return 1
      var left = 0 ; var right = y
      // Invariant I : left^2 <= y < right^2 && 0 <= left < right <= y
      // Variant (right-left)
      while (left + 2 < right)
      {
        // right - left >= 3 => (right-left) / 3 >= 1
        var midLeft = left + (right-left) / 3
        var midRight = right - (right-left) / 3
        // midLeft < midRight <=> left + (right-left) / 3 < right - (right-left)/3 <=> 2*((right-left)/3) < right-left (which is true since right-left>=3)
        // (b) left < midLeft < midRight < right, so we notice that we only work with intervals that are non-empty!!
        if (Math.pow(y,0.5) < midLeft) right = midLeft // We don't use y<midLeft*midLeft for part (c) where we can overflow
```

```
      else if (Math.pow(y,0.5) < midRight) {left = midLeft ; right = midRight}
      else left = midRight
      // I
    }
    // I && ! (left+2<right) => 1 <= (right-left) <= 2
    var result = 0
    if (right == left + 1) result = left
      else
      {
       // left + 2 = right
       var middle = left + 1
       if (Math.pow(y,0.5) < middle) result = left //left * left <= y < middle * middle
         else result = middle // middle * middle <= y < right * right
      }
    result
  }
 def main (args: Array[String]) =
  {
   // (a) && (c) checked up to Int.MaxValue - 1 (so also for up tp 46000 and up to 10^8)
   for (i <- 0 to 2147483647) assert (binary(i) == ternary(i))
  }
}
```

Question 3

```
object Question3
{
 // 1 <= X <= 1000
 val X = …
 def tooBig (y : BigInt) : Boolean =
  {
   y>X
  }
 def find (l : BigInt) : BigInt =
  {
   // We will call find(1) to find X at part (b)
   var left = l
```

```
    // Invariant I : left/2 <= X
    while (tooBig(left) == false) left = left * 2
    // I holds => left/2 <= X
    // tooBig(left) == true => left > X
    // From these two, we set (left,right) = (left/2,left)
    var right = left
    left = left / 2
    search(left,right)
  }
 def search (l : BigInt, r : BigInt) : BigInt =
  {
    // We will call search(1,1001) to find X at part (a)
    var left = l ; var right = r
    // left <= X < right
    while (left + 1  < right)
    {
      var mid = (left+right)/2
      // left < mid < right
      if (tooBig(mid)) right = mid
        else left = mid
      // I
    }
    // left + 1 = right => X = left
    left
  }
}
```

/**

(a)

Let D = right-left. We terminate the loop only when D = 1.

Initially, D = 1000. After each iteration, if D is even, D becomes D/2, and if it's odd, it can become D/2 or D/2+1. So, the sequence of possible D's is:

1000 -> 500 -> 250 -> 125 -> 62/63 -> 31/32 -> 15/16 -> 7/8 -> 3/4 -> 1/2 (the "/" means or, not division)

Here, we made 9 calls to "tooBig". In the case that D is 1 after this, we stop, otherwise we make another call to get D = 1. So in total, we can have either 9 or 10 calls to "tooBig".

Because we halving D at each step, we can deduct that the search function needs $\log_2 D$ or $\log_2 D+1$ operations.

(b)

1. We start by finding lower and upper bounds for X: from k = 0, we search for k such that $2^k <= X < 2^{k+1}$, so we will get tooBig(2^k) = false and tooBig(2^{k+1}) = true => this is done in O($\log_2 X$) as k= $\log_2 X$ and we perform (k+1) calls of "tooBig").

2. We apply search (2^k, 2^{k+1}) which needs at most $\log_2 D+1$, as we have seen at part (a), where here $D=2^{k+1}-2^k=2^k$, so we again need k operations, so $O(\log_2 X)$.

In the end we have time complexity of $O(\log_2 X)$.

(c)

Let $t = 2^{1/\varepsilon}$.

1. We first find an upper bound for X, by finding k such that $t^k <= X < t^{k+1}$, so we start from left=1.0 and at each step we multiply left by t. This needs $\log_t X$ steps.

2. We can find X with a binary search from t^k to t^{k+1}, which needs

$\log_2(t^{k+1}- t^k) = \log_2(t^k) + \log_2(t-1) = \log_2 X + \log_2(2^{1/\varepsilon} -1)$ steps(rounded upwards for non-integer numbers).

So, the number of steps we need in total is:

$\log_t X + \log_2 X + \log_2(2^{1/\varepsilon} -1) = \varepsilon* \log_2 X + \log_2 X + 1/\varepsilon$

We approximated $\log_2(2^{1/\varepsilon} -1)$ with $\log_2(2^{1/\varepsilon}) = 1/\varepsilon$, as we only treat here the cases where $0< \varepsilon<1$

(for $\varepsilon> 1$ we already have a method that finds X in $2*\log_2 X$ steps which is less than $(1+\varepsilon)\log_2 X+r$). Therefore, $1/\varepsilon$ is bigger than 1, so the (-1) from the logarithm is negligible.

Therefore, we obtained a total of $(1+\varepsilon)\log_2 X + r(\varepsilon)$ steps, where $r(\varepsilon) = 1/\varepsilon$.

*/

Question 4

```
object Question4
{
  def insert (a : Array [Int]) : Array[Int] =
    {
      val N = a.size
      var pos = 1
      // Invariant I : a[0..pos) sorted increasingly && 1 <= pos <= N
      // Variant (N-pos)
      while (pos < N)
      {
        // I && 1 <= pos < N
        var el = a(pos)
        var k = binary (a,pos)
        // We need to move the elements a(k),a(k+1)...a(pos-1) to the right by one position
        var i = pos
        // Invariant J : a[i..pos) are shifted to the right && there is an empty cell in a(i) && k <= i <= pos
        // Variant (i-k)
        while (i > k) { a(i) = a(i-1); i -= 1 }
        a(k) = 0
        // J && i==k => a(k) has no element in it, so we put there a(pos)
```

```
      a(k) = el
      // a [0..(pos+1)) sorted increasingly
      pos += 1
      // I
    }
    // pos = N && I => a[0..N) is sorted increasingly, so we return a
    a
  }
 // Finding the place in a[0..pos) where we should put a(pos), binarily.
 def binary (a : Array[Int], position : Int) : Int =
  {
     var X = a(position)
     // First, we'll treat the limit cases :
     if (a(0) >= X) return 0
     if (X >= a(position-1)) return position
     var left = 0 ; var right = position-1
     // Invariant I : a(left) < X <= a(right)
     while (left + 1 < right)
     {
       var mid = (left+right) / 2 // left < mid < right
       if (X > a(mid)) left = mid
          else right = mid
       // I
     }
     // I holds and left + 1 = right => a(left) < X <= a(left + 1), so we have to put X on the (left+1)th position
     (left + 1)
  }
 /**
```

We have N calls of the function binary. Each call does around O(log2(position)) operations as we make 2 comparisons initially and then in the while loop where the difference between right and left halves after each step (each step is one comparison) and we start with the difference being position-1. As position takes each value from 1 to N, we have

log2(1) + log2(2) + ... + log2(N) = log2(N*(N+1)/2), which is O(N*log2(N)).

First of all, the binary function takes O(log2(N)) time. In each step of the loop from the function insert we have O(log2(N)) time complexity from binary, and then in the worst case scenario, when the initial array is sorted decreasingly, we need pos operations to insert the element because we need to shift the array to the right every time, so the complexity is linear, and overall, as the while loop is executed N times, the time complexity is O(N^2).

```
 */
 // Some tests:
```

```
  def main (args: Array[String]) =
    {
      val a1 = Array (1,1,1,1,1)
      assert (insert(a1).sameElements(a1))
      val a2 = Array (1,2,3,4,5)
      assert (insert(a2).sameElements(a2.sorted))
      val a3 = Array (5,4,3,2,1)
      assert (insert(a3).sameElements(a3.sorted))
      val a4 = Array (1,2,3,3,1,2,3,2,2,1,1,3)
      assert (insert(a4).sameElements(a4.sorted))
      val a5 = Array (2)
      assert (insert(a5).sameElements(a5.sorted))
    }
}
```

Question 5

```
/**
```

If the array is sorted decreasingly, the partition function will need linear time and will partition each array into a singleton (the first element) and the N-1 elements remaining. So, the complexity of QSort will be:

T(N) = T(N-1) + O(N), which means that T(N) = O(N^2).

By splitting the array randomly, the large array will be on average of size 3*N/4, as the larger sub-array is always of length between N/2 and N, so by uniformity, on average we get 3*N/4. Thus, the complexity of the QSort will be:

T(N) = T(N/4) + T(3*N/4) + O(N), which runs in O(N*log(N)) (this can be proven with a recurrence tree).

```
*/
```

Question 6

```
object Question6
{
/** Partition the segment a[l..r)
* return k s.t. a[l..k) < a[k..r) and l <= k < r */
def partition(l: Int, r: Int, a:Array[Int]) : Int =
  {
    val x = a(l)
    // pivot
    // Invariant a[l+1..i) < x = a(l) <= a[j..r) && l < i <= j <= r
    //          && a[0..l) = a_0[0..l) && a[r..N) = a_0[r..N)
    //          && a[l..r) is a permutation of a_0[l..r)
    var i = l+1; var j = r
    while(i < j)
```

```
  {
    if(a(i) < x) i += 1
      else {val t = a(i); a(i) = a(j-1); a(j-1) = t; j -= 1 }
  }
  // swap pivot into position
  a(l) = a(i-1); a(i-1) = x
  i-1 // position of the pivot
 }
/**
An example to illustrate the fact that an element that is bigger than the pivot can be moved twice before finding its place is:  4 3 5 6, with the pivot a(0) = 4, i = 1, j = 4. First, we skip a(1) = 3 (i=2), then we see that 5>=4, so we swap 5 with 6 to get  4 3 6 5 (at this point i=2, j=3). Then, we see that 6 >= 4, so we swap again 6 with 5 to get 4 3 5 6 (i=j=2, and we stop, the position of the pivot is 1). So, for 6 we needed 2 comparisons to find its place and we swapped it twice, although we didn't need to swap it at all.
*/
 def partition2 (l: Int, r: Int, a:Array[Int]) : Int =
  {
    val x = a(l)
    // pivot
    // Invariant a[l+1..i) < x = a(l) <= a[j..r) && l < i <= j <= r
    //          && a[0..l) = a_0[0..l) && a[r..N) = a_0[r..N)
    //          && a[l..r) is a permutation of a_0[l..r) (same invariant as before)
    var i = l+1; var j = r
    while(i < j)
    {
     if(a(i) < x) i += 1
       else
         {
          // Invariant J : x <= a[j..j_0), where j_0 is the value of j before getting in the while-loop
          // Also i+1<j so that we don't get out of the segment we want to partition
          while ((a(j-1) >= x) && (i+1<j)) j -= 1
          // This way, we only swap a(i) with a(j-1) if a(j-1) is smaller than x, so that we don't have to swap it again next time
          val t = a(i); a(i) = a(j-1); a(j-1) = t; j -= 1
         }
    }
    // swap pivot into position
    a(l) = a(i-1); a(i-1) = x
    i-1 // position of the pivot
```

```
  }
  def main (args: Array[String]) =
  {
    val a1 = Array(1,2,3,4,5,6)
    val b1 = Array(1,2,3,4,5,6)
    assert(partition(0,6,a1) == partition2(0,6,b1))
    val a2 = Array(6,5,4,3,2,1)
    val b2 = Array(6,5,4,3,2,1)
    assert(partition(0,6,a2) == partition2(0,6,b2))
    val a3 = Array(4,1,2,1,4,5,6,1)
    val b3 = Array(4,1,2,1,4,5,6,1)
    assert(partition(0,8,a3) == partition2(0,8,b3))
    val a4 = Array(1,1,1,1)
    val b4 = Array(1,1,1,1)
    assert(partition(0,4,a4) == partition2(0,4,b4))
    // Used two arrays for the assertions as each call to any partition function changes the array.
  }
}
```

Question 7

```
object Question7
{
  val a = ...
  def partition(l: Int, r: Int) : Int =
  {
    val x = a(l)
    var i = l+1; var j = r
    while(i < j)
    {
      if(a(i) < x) i += 1
        else {val t = a(i); a(i) = a(j-1); a(j-1) = t; j -= 1 }
    }
    a(l) = a(i-1); a(i-1) = x
    i-1
  }
  /**
```

(a) We use a while-loop instead of the second call to QSort because after we are done with the first QSort call in the initial algorithm, we only need to modify the value of l to (k+1) for the second QSort. So, we keep l in a variable aux, which can be modified.

(b) In the worst case-scenario, the stack might be O(N), as we might QSort over the bigger partition of the sub-array we are currently sorting(the left sub-array), so in the case when the calls are for sorting an array of size N, then (N-1), then (N-2) ... and finally 1, the call stack reaches depth N.

(c) In the QSort2 version of the algorithm, we choose the smaller partition and we recurse on it(the size of the smaller partition is maximum floor(N/2)), so that in the worst case scenario we have a ceiling(log2(N)) depth of the stack call (worst-case scenario : if at every step the sub-array is partitioned exactly in half) : 2^k, 2^(k-1), ... , 2^1, 1 -> k+1 stack calls, so ceiling(log2(N)).(if N is not a power of 2, we need at most this).

*/

```
def QSort (l : Int, r : Int) : Unit =
 {
  var aux = l
  // Invariant I : a[l..aux) is sorted and all the elements from a[aux..r) are greater than a[l..aux]
  while (r-aux > 1)
  {
   var k = partition(aux,r)
   QSort (aux,k) // We can have QSort(l,k) as mentioned in the task, but it is very inefficient
   aux = k + 1
  }
 }
def QSort2 (l : Int, r : Int) : Unit =
 {
  var left = l ; var right = r
  // Invariant I : a[l..left) is sorted and a[right..r) is sorted and a[l..left) < a[left..right) < a [right..r)
  while (right-left > 1)
  {
   var k = partition(left,right)
   if (k-left<right-1-k) { QSort2(l,k) ; left = k + 1 }
      else               { QSort2(k+1,r) ; right = k }
  }
 }
}
```

Question 8

```
object Question8
{
 val a = …
 def partition(l: Int, r: Int) : Int =
 {
  val x = a(l)
```

```
  var i = l+1; var j = r
  while(i < j)
  {
   if(a(i) < x) i += 1
     else {val t = a(i); a(i) = a(j-1); a(j-1) = t; j -= 1 }
  }
  a(l) = a(i-1); a(i-1) = x
  i-1
 }
def QSort(l: Int, r: Int) : Unit =
 {
  if(r-l > 1)
   {
    val k = partition(l,r)
    QSort(l,k); QSort(k+1,r)
   }
 }
/**
```

(a) If the array we want to sort has all the elements equal, we see that the partition function would return (in O(N) operations) the pivot always to be the l, so the two QSort calls will be QSort(l,l) which will take O(1) and QSort(l+1,r). Therefore, we will need O(N) calls to QSort, so the time complexity will be O(N^2).

(b) If the array has a lot of identical entries in the array the partitioning may return an array that is partitioned with a pivot that is very close to one end, and consequently very far from the other one and thus the complexity is going to be closer to O(N^2) than to O(N*log2(N)). This depends on how big the identical entries are compared to the others (it will be optimal if they are at the middle of the sorted array).

(c) In the partition function, if there are multiple identical entries, then there would be more else clauses executed in the “if” statements because the condition is with "<". This means that we will have more instructions to execute since the else clause has 4, whereas the “if” clause has just 1. So, on average, we do more work by swapping unnecessary elements. Thus, if we replace the "<" in the “if” statement with "<=", in the case where there are more identical entries, we will need less instructions to execute.

```
*/
// (d)
def partition2(l: Int, r: Int) : (Int,Int) =
 {
  val pivot = a(l)
  // pivot
  // Invariant I: a[l..i) < pivot && a[i..j) = pivot && a[k..r) > pivot && l <= i < j  <= k <= r
  //          && a[0..l) = a_0[0..l) && a[r..N) = a_0[r..N) && a[l..r) is a permutation of a_0[l..r)
  var i = l ; var j = l+1 ; var k = r
  while (j < k)
  {
```

```
    if(a(j) == pivot) j += 1
      else if (a(j) < pivot) {val t = a(i); a(i) = a(j); a(j) = t; i += 1 ; j += 1 }
      else {val t = a(j); a(j) = a(k-1); a(k-1) = t; k -= 1 }
   }
   (i,j)
  }
// (e)
  def QSort2(l: Int, r: Int) : Unit =
   {
    if(r-l > 1)
     {
      val (i,j) = partition2(l,r)
      QSort(l,i) ; QSort(j,r) // There is no need to sort the elements a[i..j), since they are all equal from partition2(l,r).
     }
   }
}
```