Tuan

Question 1:

a) If the array is not increasing, then the procedure is going to compare `x` with the middle element and if `x` is larger, then it will continue to search in the second half of the array, otherwise it will search in the first half. The program is going to do that until it finds `i` such that `a(i) < x <= a(i+1)`. Generally it will not produce the same result as it would if the array was sorted.

b) If `N = 0`, then the program will return 0.
`i = 0`, `j = 0`, so the test for the while loop will be false. Therefore, it will return 0.

c) It will not work properly if the array is larger `Int.MaxValue`. So to fix that I can just make the function `search` return a `BigInt`.

```
object BinarySearch{
  /** Find index i s.t. a[0..i) < x <= a[i..N).
  * Pre: a is sorted. */
  def search(a: Array[Int], x: Int) : BigInt = {
    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)/2 // 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
  }

  def main(args : Array[String]) = {
    var a = new Array[Int](args.size - 1)
    var i = 0
    while(i < args.size - 1){
      a(i) = args(i).toInt
      i += 1
    }
    val input = args(args.size - 1).toInt

    println(search(a, input))

    val testArr1 = Array(1, 2, 3, 4, 5, 6, 7, 8, 9)
    val testArr2 = Array(1, 2, 3, 4, 4, 6, 7, 8, 9)
    assert(search(testArr1, 4) == 3)
```

```
    assert(search(testArr2, 4) == 3)
    assert(search(testArr2, 9) == 8)
  }
}
```

Question 2:

a)

```
object TernarySearch{
  /* Calculates the integer square root
   * Pre: n >= 0
   * Post: a s.t. a^2 <= n */
  def intSqrt(n : Int) : Int = {
    require(n >= 0)
    // Invariant : a^2 <= n < b^2 && 0 <= a < b
    // Variant : b - a
    var a = 0; var b = n + 1
    while(a + 1 < b){
      val m1 = (b-a) / 3 + a // one third of the interval
      val m2 = 2 * (b-a) / 3 + a // two thirds of the interval
      if(m1*m1 <= n){
        if(m2*m2 <= n){ a = m2 }
        else{ a = m1; b = m2 }
      }
      else{ b = m1 }
    }
    // a+1 = b, so a^2 <= n < (a+1) ^ 2
    a
  }

  // Copied from the lecture notes
  /* Find a s.t. a^2 <= y < (a+1)^2
   * Pre: y >= 0 */
  def sqrt(y : Int) : Int = {
    // Invariant : a^2 <= y < b^2 && a < b
     assert(y>=0)
    // Invariant I: a^2 <= y < b^2 and a<b
    var a = 0; var b = y+1
    while(a+1<b){
      val m = (a+b)/2 // a < m < b
      if(m*m<=y) a=m else b=m
    }
    // I and a+1=b, so a^2 <= y < (a+1)^2
```

```
    a
  }

  def main(args : Array[String]) = {
    if(args.size != 1){ println("Wrong number of arguments") }
    else{
      val input = args(0).toInt
      if(input < 0 && input < Int.MaxValue){
        println("Please input a non-negative number")
      }
      else{ println(intSqrt(input)) }
    }

    var i = 0
    while(i < 46000){
      if(intSqrt(i) == sqrt(i)){i += 1}
      else{println("This does not work " + i); i += 1}
    }
  }
}
```

b) This definition of ternary search cannot produce an empty interval. In order to do that the difference between a and b has to be 1, in which case the test for the while loop will be false.

Question 3:

a)

```
object Game{
  /** If the guess is bigger than the unknown number */
  def tooBig(y : BigInt) : Boolean = {
    val x = 568924555 // value x for testing purposes
    val isBigger = (y > x)
    isBigger
  }

  /* Returns the unknown number
   * Pre: 0 < X */
  def find(a : Int, b : Int) : Int = {
    // a - lower bound, b - upper bound to do binary search
    // Invariant : guess1 <= X <= guess2
    var total = 0 // number of calls of tooBig

    var guess1 = a; var guess2 = b
```

```
    while(guess1 + 1 < guess2){
      val med = (guess1 + guess2) / 2
      if(tooBig(med)) {guess2 = med; total += 1 }
      else { guess1 = med; total += 1 }
    }
    // guess1 + 1 = guess2, so guess1 = x
    total
  }

  def main(args : Array[String]) = {
    //println(find(0, 1000))

    var i = 1; var total = 0
    // Invariant : i <= X && 1 <= i && i = 2 ^ k (k - integer)
    while(!tooBig(i * 2)){
      i *= 2
    }
    // The loop stops on the first number i s.t. i <= X < i * 2
    // So to find X we can do binary search for the interval [i..i * 2)

    // If we want to do it in (1+e)log(2)X + r steps
    // then we can just multiply i by 2^1/e(if e > 1)
    // Otherwise we can just do the normal procedure

    println(find(i, i * 2)) // total number of calls of tooBig
  }
}
```

If X is at most 1000, then maximum number of calls to `tooBig` is log2 1000, which is 9, because at each iteration the search space is reduced by half.

b) With the above program Bill can find X in 2log2 X. First it would take log2 X time to find `i`, such that `i <= X < i * 2`. It would take additional log2 X time to search from `i` to `i * 2`.

c) If the constant is bigger or equal to 1, then we can just use the previous program. If the constant is smaller than 1, then we can write the constant as 1/k, such that x is bigger than 1. So in the main function, instead of multiplying `i` by 2, we can multiply it by 2^x.
Hence, to find that `i` it would take $\log_{2\hat{}k} X = \frac{1}{k}\log_2 X = \varepsilon \log_2 X$.
We have the interval (`i, i * 2^k`). So it would take at most $\log_2(X * 2^k) = \log_2 X + \log_2 2\hat{}k = \log_2 X + k$.

Question 4:

```
object BinaryInsertionSort{
  /** Find index i s.t. a[start..i) < elem <= a[i..finish].
    * Pre: a is sorted. */
  def search(a: Array[Int], elem: Int, start : Int, finish : Int) : Int
= {
    val N = a.size; var i = start; var j = finish
    // Checking the ends of the array
    if(elem <= a(i)) j = i else
    if(elem == a(j)) i = j else
    if(elem > a(j)){j += 1; i = j} else

    // invariant I: a[0..i] < elem <= a[j..finish]
    //           && 0 <= i < j <= finish
    while(i < j && j - i != 1){
      val m = (i+j)/2 // i <= m < j
      if(a(m) < elem) i = m else j = m
    }
    // I && i = j - 1, so a[0..i) < elem <= a[i..finish)
    j
  }

  /* Insertion sort using binary search
   * Pre arr.size >= 0 */
  def insertionSort(a : Array[Int]) : Array[Int] = {
    // Invariant : a[0..i) is sorted
    //          && a[0..i) is a permutation of a0[0..i)
    //          && a[i..N) = a0[i..N) && 1 <= i <= a.size
    // Variant : a.size - i
    val N = a.size; var i = 1
    while(i < N){
      val index = search(a, a(i), 0, i - 1)
      val swap = a(i)

      // Moving all the elements that are bigger
      // than a(i) one index to the right
      // Invariant : a[j..i-1] = a[j+1..i] && index <= j <= i - 1
      // Variant : j - index
      var j = i - 1
      while(j >= index){
        a(j + 1) = a(j)
        j -= 1
```

```
      }

      a(index) = swap
      i += 1
    }
    // i = N, so a[0..N) is sorted
    a
  }

  def main(args : Array[String]) = {
    var a = args.map(_.toInt)
    println(insertionSort(a).deep)

    val testArray1 = Array(6, 4, 8, 10, 2, 1)
    val testArray2 = Array(10, 3, 1, 2, 1, 9)
    assert(insertionSort(testArray1).deep ==
                         Array(1, 2, 4, 6, 8, 10).deep)
    assert(insertionSort(testArray2).deep ==
                         Array(1, 1, 2, 3, 9, 10). deep)
  }
}
```

The order of the comparisons of elements of a is $\log_2 N$, because at each iteration the search space is halved. The overall running time of the function is $O(n\log n)$, because we do log$n$ searches for each element of the array.

Question 5:

Choosing the leftmost element as a pivot might be a bad idea when the array is sorted or nearly sorted. Then at each iteration, the function produces a very small segment and a segment that is almost as large as the original. In this case the program will take $O(n^2)$ time. If the pivot is chosen at random, the probability of choosing such pivot, that is either smaller or bigger than the rest, is very small. If the pivot is chosen at random, the expected length of the larger segment is going to be ¾ of the original, so the expected behavior of the program is $O(n\log n)$.

Question 6:

```
object ImprovedPartition{
  /** Partition the segment a[l..r)
    * @return k s.t. a[l..k) < a[k..r) and l <= k < r */
  def partition(a : Array[Int], l: Int, r: 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 }
    println(a.deep)
  }
  // swap pivot into position
  a(l) = a(i-1); a(i-1) = x
  println(a.deep)
  i-1 // position of the pivot
}

/** Partition the segment a[l..r)
  * @return k s.t. a[l..k) < a[k..r) and l <= k < r */
def improvedPartition(a : Array[Int], l : Int, r : 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) = a0[r..N)
  //          && a[l..r) is a permutation of a0[l..r)
  var i = l + 1; var j = r
  while(i < j){
    if(a(i) < x) i += 1
    else{
      // Invariant : x = a(l) < a[t..j) && i < t < j
      //          && a[t..j) is a permutation of a0[t..j)
      var t = j - 1
      while(a(t) > x && t > i){ t -= 1 }
      // a(t) is the last number in the array s.t. a(t) <= x,
      // so for all z in [t+1..j), a(z) > x
      if(i < t){
        var swap = a(i)
        a(i) = a(t)
        a(t) = swap
        i += 1
      }
      j = t
    }
    println(a.deep)
  }
  // swap pivot into position
  a(l) = a(i-1); a(i-1) = x
```

```
    println(a.deep)
    i-1 // position of the pivot
  }

  def main(args : Array[String]) = {
    var a1 = Array(5, 3, 10, 0, 2, 1, 9, 7, 4, 6)
    var a = Array(10, 1, 0, 20 , 13, 4, 16, 16, 13, 6, 10)
    //println(partition(a, 0, 8))
    // 9 moves two times. First it swaps with 6, then it swaps with 7.

    println(improvedPartition(a1, 0, 10))
  }
}
```

Let us take array a1. If the original program is used, then 9 would be moved two times. First it will be swapped with 6, and then with 7.

Question 7:

a)

```
object TailRecursive{
  /** Partition the segment a[l..r]
    * @return k s.t. a[l..k) < a[k..r) and l <= k < r
    * This is a modified version of improvedPartition
    * The difference is the array is a[l..r] instead of a[l..r) */
  def improvedPartition(a : Array[Int], l : Int, r : 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 + 1
    while(i < j){
      //println(i + " " + j)
      if(a(i) < x) i += 1
      else{
        // Invariant : x = a(l) < a[t..j) && i < t < j
        //          && a[t..j) is a permutation of a0[t..j)
        var t = j - 1
        while(a(t) > x && t > i){ t -= 1 }
        // a(t) is the last number in the array s.t. a(t) <= x,
        // so for all z in [t+1..j), a(z) > x
        if(i < t){
          var swap = a(i)
```

```
          a(i) = a(t)
          a(t) = swap
          i += 1
        }
        j = t
      }
      //println(a.deep)
    }
    // swap pivot into position
    a(l) = a(i-1); a(i-1) = x
    //println(a.deep)
    i-1 // position of the pivot
  }

  /* Sorts an array with quicksort method
   * Pre: a.size >= 0
   * Post: a is sorted && a is a permutation of a0 */
  def QSort(a : Array[Int], l: Int, r: Int) : Unit = {
    if(r-l > 1){ // nothing to do if segment empty or singleton
      var k = improvedPartition(a, l, r)
      QSort(a, l, k)
      var k1 = improvedPartition(a, k + 1, r)
      while(k1 < r){
        QSort(a, k + 1, k1)
        k1 = improvedPartition(a, k1 + 1, r)
      }
    }
    println(a.deep)
  }


  def main(args : Array[String]) = {
    var a = args.map(_.toInt)
    val testArray = Array(5, 3, 10, 3, 2, 10, 9, 7, 4, 6)
    println(QSort(a, 0, args.size))
    assert(QSort(a, 0, 10).deep ==
                Array(3, 3, 4, 5, 5, 6, 8, 8, 10, 10).deep)
  }
}
```

b) The stack can be called `N` times if the array is sorted. Then the QSort call would have to sort a singleton, whereas the while- loop would have to sort an array with length `N-1`.

c)

```
object TailRecursive{
  /** Partition the segment a[l..r]
    * @return k s.t. a[l..k) < a[k..r) and l <= k < r
    * This is a modified version of improvedPartition
    * The difference is the array is a[l..r] instead of a[l..r) */
  def improvedPartition(a : Array[Int], l : Int, r : 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 + 1
    while(i < j){
      //println(i + " " + j)
      if(a(i) < x) i += 1
      else{
        // Invariant : x = a(l) < a[t..j) && i < t < j
        //          && a[t..j) is a permutation of a0[t..j)
        var t = j - 1
        while(a(t) > x && t > i){ t -= 1 }
        // a(t) is the last number in the array s.t. a(t) <= x,
        // so for all z in [t+1..j), a(z) > x
        if(i < t){
          var swap = a(i)
          a(i) = a(t)
          a(t) = swap
          i += 1
        }
        j = t
      }
      //println(a.deep)
    }
    // swap pivot into position
    a(l) = a(i-1); a(i-1) = x
    //println(a.deep)
    i-1 // position of the pivot
  }

  /* Sorts an array with quicksort method
   * Pre: a.size >= 0
   * Post: a is sorted && a is a permutation of a0 */
  def QSort(a : Array[Int], l: Int, r: Int) : Unit = {
```

```
    // Invariant : a[l..i) and a[j..r) are sorted
    //          && a[l..r) is a permutation of a0[l..r)
    //          && l <= i < j <= r
    // Variant : j - i
    var i = l; var j = r
    while(i < j){
      var k = improvedPartition(a, i, r)
      if(k - i < j - k){
        QSort(a, i, k - 1)
        i = k + 1
      }
      else{
        QSort(a, k + 1, j)
        j = k - 1
      }
    }
    println(a.deep)
  }


  def main(args : Array[String]) = {
    var a = args.map(_.toInt)
    println(QSort(a, 0, args.size))
    val testArray1 = Array(5, 3, 10, 8, 2, 1, 9, 7, 4, 6)
    val testArray2 = Array(5, 3, 10, 5, 8, 3, 10, 6, 4, 8)
    assert(QSort(testArray1, 0, 10).deep ==
                Array(3, 3, 4, 5, 5, 6, 8, 8, 10, 10).deep)
  }
}
```

Question 8:

a) With normal Quicksort implementation, if all elements are equal, then the running time would be $O(n^2)$. The program would not do any swaps but it will call Quicksort n times.

b) If several identical entries are scattered throughout the array, then after being sorted there will be a subsegments, which are formed by identical elements.

c) The primary way of doing quicksort is by using the function `partition`. The `else` clause is the only way to swap elements. In order to sort the array, quicksort has to execute the `else` many more times than the `then` clause to actually swap elements. `Partition` would execute more the `then` clause only if the array is nearly or entirely sorted.

d)

```
object EqualPartition{
```

```
  def partition(a : Array[Int], l : Int, r : Int) : (Int, Int) = {
    // Invariant : a[l..i) < pivot && a[i..j) = pivot && a[k..r) > pivot
    //          && a[l..r) is a permutation of a0[l..r)
    //          && l <= i < j <= k <= r
    // Variant : k - j
    var i = l; var j = l + 1; var k = r
    val pivot = a(l)
    while(k - j > 0){
      if(a(j) < pivot){
        var swap = a(i)
        a(i) = a(j)
        a(j) = swap
        i += 1; j += 1
      }
      else if(a(j) == pivot){
        j += 1
      }
      else{
        var swap = a(k - 1)
        a(k - 1) = a(j)
        a(j) = swap
        k -= 1
      }
    }
    // j = k, so a[i..j) = pivot
    //println(a.deep) // to test the array is partitioned properly
    (i, j)
  }

  def main(args : Array[String]) = {
    var a = args.map(_.toInt)
    println(partition(a, 0, args.size))

    var testArray1 = Array(5, 3, 1, 5, 6, 5, 10, 2, 5, 5, 4)
    var testArray2 = Array(5, 3, 4, 0, 2, 1, 9, 7, 10, 6)
    assert(partition(testArray1, 0, 11) == (4, 9))
    assert(partition(testArray2, 0, 10) == (5, 6))
  }
}
```

e)

```
object EqualPartition{
```

```
def partition(a : Array[Int], l : Int, r : Int) : (Int, Int) = {
  // Invariant : a[l..i) < pivot && a[i..j) = pivot && a[k..r) > pivot
  //          && a[l..r) is a permutation of a0[l..r)
  //          && l <= i < j <= k <= r
  // Variant : k - j
  var i = l; var j = l + 1; var k = r
  val pivot = a(l)
  while(k - j > 0){
    if(a(j) < pivot){
      var swap = a(i)
      a(i) = a(j)
      a(j) = swap
      i += 1; j += 1
    }
    else if(a(j) == pivot){
      j += 1
    }
    else{
      var swap = a(k - 1)
      a(k - 1) = a(j)
      a(j) = swap
      k -= 1
    }
  }
  // j = k, so a[i..j) = pivot
  //println(a.deep) // to test the array is partitioned properly
  (i, j)
}

def QSort(a : Array[Int], l : Int, r : Int) : Unit = {
  if(r - l > 1){
    var k1 = partition(a, l, r)._1
    var k2 = partition(a, l, r)._2
    QSort(a, l, k1 - 1); QSort(a, k2, r)
  }
}

def main(args : Array[String]) = {
  //var a = args.map(_.toInt)
  //println(partition(a, 0, args.size))

  var testArray1 = Array(5, 3, 1, 5, 6, 5, 10, 2, 5, 5, 4)
```

```
    var testArray2 = Array(5, 3, 4, 0, 2, 1, 9, 7, 10, 6)
    assert(partition(testArray1, 0, 11) == (4, 9))
    assert(partition(testArray2, 0, 10) == (5, 6))
  }
}
```

The improvement in this case is that when all the entries of the array are identical, the running time is going to be *O(n)*.