Problem Sheet 3

Question 1:

a) If the list is not sorted, then the program will do the following: Let's say we provide the input of the function to be `(3,11,5,7,13,2,15,18)`, and let's say we're trying to find `2` in the list.

After the first iteration, we will get that `i := (i+1)/2`, and since this invariant will not hold: `a[0..i) < x <= a[i..N)`, we can say for certain that the program will not return the correct post-conditions as specified.

b) if `N = 0`, then the following happens. The test fails before any iteration since `i < j` would initially be false. Since `i = 0`, the returned value happens to be `0`. This also follows our invariant and our postconditions, so this returned value is correct subject to our preconditions.

c) I'm not aware how the size of Arrays could be increases - (perhaps use another implemented data structure?), but I can mention a way for it to be used for large entries.

Suppose that `n > 2^(31) - 1`, which would mean that as an int, $n_0 \equiv n \mod 2^{31} - 1$. With this in mind, instead of using regular inequality checks, we can use a more robust version. From discrete maths:

$$(a-b) \equiv (c-d) \mod 2^{31} - 1$$

This means that instead of trying to compare `a` and `b`, we should compare `a-b` to the equivalent representations modulo $2^{31} - 1$. Since the implementation of Int already does this, we need to use the following cases:

- If $a - b > 0$, the either the absolute value of `a` is in fact bigger than `b`, or `b` is bigger than `a` but `b` has overflowed and has become negative.
- If $a - b < 0$, then either the absolute value of `a` is in fact smaller than `b`, or `a` is bigger than `b`, but has underflowed.

We can calculate $a - b$ beforehand and then we can buy an extra $2^{31} - 1$ more possible values, since $-2^{31} + 1 \mod 2^{31} - 1$ is equivalent to 0, so we can exploit this and increase the size of our domain.

Question 2:

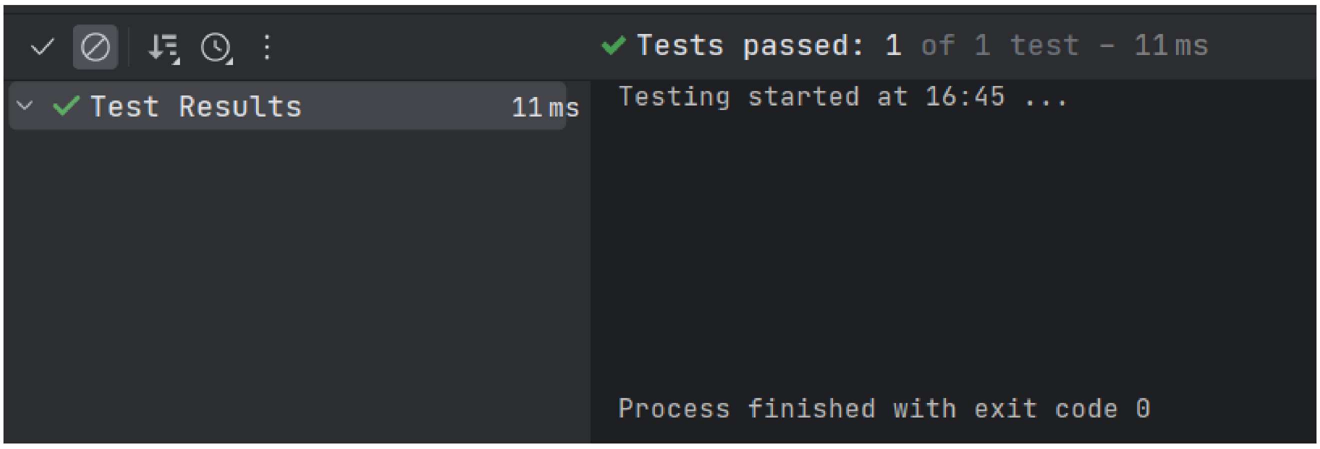
Part a. Here is the algorithm with the test suite alongside it

```
class Ternary {
```

```scala
  /**
   *   * @param y from which we find the integer square root from.
   *     @return low such that x*x <= y < (x+1)*(x+1)
   *   */
  def ternaryRootSearch(y: Int): Int = {
    require(y >= 0)
    var low = 0 ; var high = y;
    while (low <= high) {  // I = low^2 < y < (high+1)^2
      val mid1 = low + (high - low) / 3
      val mid2 = high - (high - low ) / 3

      if (mid1*mid1 <= y && (mid1+1)*(mid1+1) > y)
        return mid1
      if (mid2*mid2 <= y && (mid2+1)*(mid2+1) > y)
        high = mid1 - 1
      if (mid1*mid1 > y)
        high = mid1 - 1
      else if (mid2*mid2 <= y)
        low = mid2 + 1
      else
        low = mid1 + 1 ; high = mid2 - 1
    }
    low // I && ¬test = low^2 < y < (high+1)^2 && ¬(low <= high)
    // therefore the invariant and not the test do give the post-conditions  }
}
```

And here is the test suite:



```
Tests passed: 1 of 1 test – 11 ms
Test Results                11 ms
Testing started at 16:45 ...

Process finished with exit code 0
```

```scala
class TernaryTest extends AnyFunSuite {

  test("Testing the boundary value of y = 2^31") {
    // since sqrt(2^31-1) is equal to 46340.95, we should
    // expect the answer to be 46340     val ternary = new Ternary()
    val n = 65537
    assert(ternary.ternaryRootSearch(n) == 256)

  }
}
```

Unfortunately, my program doesn't seem to work for higher numbers. There's a problem with my implementation.

Part b)

With the implementation I have, as well as the fact that my loop invariant was defined without the use of the boundaries, I think it may be true that my program may run into a place where it might deal with an empty subset. However, since the other two third-points are implicitly lower than the `high` value, I don't think it's possible.

Question 3.

Part a and b:

```scala
class Question3 {
  /**
   *   * @param maxValue max value as the bigInt
   *     @param tooBig   the boolean function that tells whether the number is
too big.
   *      @return low, where the number is the true guess.
   * */
   def findX(maxValue: BigInt, tooBig: BigInt => Boolean): BigInt = {
    var low: BigInt = 1
    var high: BigInt = maxValue
    while (low <= high) {
      val mid = low + (high - low) / 2
      if (tooBig(mid)) {
        // If mid is too big, then X is in the lower half
        high = mid - 1
      } else {
        // If mid is not too big, then X is at least mid
        if (!tooBig(mid + 1)) {
          low = mid + 1
        } else {
          // If mid + 1 is too big, then we've found X
          return mid
        }
      }
    }
    low
  }
}
```

The code above solves part a if the `maxValue` parameter is set to 1000. In that case, the maximum number of calls to `tooBig` would only be 10 calls.

For the second part, the parameters can be changed so that `maxValue` can be changed to the the max limit of X.

Part C:

- Our approach can be to start with an exponential search to find an upper bound for X. This involves checking `X > 2^k` for k = 0,1,2,..., until finding the smallest k such that `tooBig(2^k)` is true. This will take at most $\log_2 X + 1$ calls of `tooBig`.
- Then, we can adjust for ϵ, we can change the intervals of the binary search phase based on ϵ. Instead of dividing the interval into two equal parts, we divide it into parts that are determined by ϵ.
- Then, run that binary search, which will take $\epsilon \log_2 X$ time. Therefore, the total time, by adding up the first search, will take $(1 + \epsilon) \log_2 X + r$ time, where r is the extra calls of `tooBig` that the partition of the partition-search that our searching algorithm may take.

Question 4:

Here is the procedure for the insertion sort algorithm:

```
class InsertionSort {

  /**
   *   * @param a : list of unsorted integers to be sorted
   *   * */  def sort(a : Array[Int]) : Array[Int] = {

    for (n <- 1 until a.length) {
      val key = a(n)
      var left = 0
      var right = n -1

      // Binary search to find the correct insertion point for a(n)
      while (left <= right) {
        val mid = left + (right - left) / 2
        if (key < a(mid)) right = mid - 1
        else left = mid + 1
      }

      // Shift elements to make room for the key
      for (j <- (n-1) to left by -1) {
        a(j + 1) = a(j)
      }
      a(left) = key
    }
    a
  }

}
```

with a test suite as well:

```scala
class InsertionSortTest extends AnyFunSuite {

  test("Regular Test") {
    var a = Array(1,3,6,11,23,134,5,42,23,231,43,7)
    val sorted = Array(1, 3, 5, 6, 7, 11, 23, 23, 42, 43, 134, 231)
    val insertionSort = new InsertionSort()
    a = insertionSort.sort(a)

    assert(sorted.sameElements(a))

  }

}
```

Optimally, this algorithm should only take $O(n \log n)$ time to complete, but since there are two loops required in order to shift the elements after the key is found, this algorithm happens to be $O(n^2)$.

Question 5:

Picking the first element is not optimal if the array is already sorted. The algorithm will end up pivoting over the whole list, and do that again but without the first element. It will end up taking quadratic time to complete.

Picking a random element as the pivot better increases the odds that we pick the element closest to the median. Since we are dealing with randomness, we can model this with some probability techniques.

$$\text{Let } X_i \text{ define the pivot picked}$$

Now, since we pick randomly, and as the list grows, we can use the central limit theorem to suggest that X_i is distributed normally, meaning that it is likely that it will be close to the mean, which luckily happens to be the median. We can approximate the expected value for it too.

$$\mathbb{E}[X_i] = \sum_{i=0}^{n-1} (Z_i)$$

where Z_i are Bernoulli Trails of the random value being the i-th statistic.

Now, the expected value for this function ends up being $\mathbb{E}[X_i] = n \cdot p,\ p \in (0, 1)$, meaning that picking a random element as the pivot makes it more likely that it is closer to the median.

And with that, if the time taken for Quicksort to run ends up being $T(n)$, we can definitively say that on average, $T(n) \approx 2T(n/2) + O(n)$, which happens to be $T(n) = O(n \log n)$ by the master theorem.

This essentially makes Quicksort more optimal.

Question 6:

Here is the version which each element moves only once:

```
def improvedPartition(a: Array[Int], l: Int, r: Int): Int = {
  val pivot = a(l)
  var i = l + 1
  var j = r - 1
// Invariant: For all indices k,
  // if l < k < i then a[k] <= pivot,
  // if j < k < r then a[k] > pivot,
  // and a[l] is the pivot.
  while (i <= j) {
    while (i <= j && a(i) <= pivot) i += 1
    while (i <= j && a(j) > pivot) j -= 1
    if (i < j) {
      val temp = a(i)
      a(i) = a(j)
      a(j) = temp
      i += 1
      j -= 1
    }
  }

  a(l) = a(j)
  a(j) = pivot
  j
```

Question 7:

Part a) Here is the altered version of the quicksort algorithm

```
def QSort(a: Array[Int], l: Int, r: Int): Unit = {
  var left = l
  var right = r

  while(right - left > 1) { // Continue while there is more than one element to
sort
    val k = partition(a, left, right) // Use the provided partition function

    if (k - left < right - (k + 1)) {
      QSort(a, left, k) // Recursively sort the smaller segment to reduce stack
depth
      left = k + 1 // Prepare to iteratively sort the larger segment
    } else {
      QSort(a, k + 1, right) // Recursively sort the smaller segment to reduce
stack depth
      right = k // Prepare to iteratively sort the larger segment
    }
  }
}
```

part b)

The dept of the call stack in quicksort is determined by the number of recursive calls before reaching the base case. In the worst-case scenario, the dept of the recursion can reach N (where the array could be sorted to begin with).

To mitigate this, the pivot can be selected randomly which could help make the quicksort algorithm more optimised, and also avoid reaching the max call stack.

part c)

To ensure that the algorithm doesn't exceed $\lceil \log_2 n \rceil$, we can choose to always recurse on the smaller half, we can cause the stack to not exceed unnecessarily. We can also do the equivalent of "loop unrolling" and introduce some iteration in the algorithm in order to reduce the chances that the recursive calls do not exceed $\lceil \log_2 n \rceil$.

Here is the algorithm:

```
def QSort(a: Array[Int], l: Int, r: Int): Unit = {
  var left = l
  var right = r

  while (right - left > 1) {
    val k = partition(a, left, right) // Use the provided partition function
    // Always recurse into the smaller side and iterate on the larger side
    if (k - left < right - (k + 1)) {
      QSort(a, left, k) // Recurse on the smaller half
      left = k + 1 // Use iteration for the larger half
    } else {
      QSort(a, k + 1, right) // Recurse on the smaller half
      right = k // Use iteration for the larger half
    }
  }
}
```

Question 8

Part a) If the partitioning algorithm is not designed in a way in which it doesn't account for duplicates, then each partitioning step will not effectively reduce the problem size as desired.

We will end up with a case where it runs in quadratic time.

Since the array - even with the use of more optimised partitioning algorithms - won't end up partitioning the array in a desirable fashion. Therefore, the algorithm will take $O(n^2)$ time.

b) The presence of multiple identical elements will definitely lead to suboptimal partitioning. Since most partitioning algorithms don't deal with duplicates, we might end up picking identical elements as pivots consecutively.

To answer what will happen, it depends on the partitioning algorithm. If Hoare partitioning is used, then random chance may lead to the same elements being picked as the pivots. A three-way partitioning method may be used instead.

c) I'm going to assume that the same many identical elements array is used here. When sorting an array with several identical entries, the observation that the `else` clause is executed more frequently attributes to how the duplicates are handled during the partitioning process.

In the partition function, the `if` condition checks whether the current element is less than the pivot `(x)`. If many elements are identical and equal to the pivot, this condition will fail for all these elements.

Since the scope of `<=` is larger than just `<`, and also accounts for duplicates, this will lead to the `if` test running its original clause more often than the `else` clause, which can attribute to higher efficiency of the algorithm.

Perhaps changing this condition from `<` to `<=` might be more efficient than changing the partitioning algorithm, but this would require more analysis to confirm or deny.

d) I wasn't able to implement this properly, but here is my design:

```
def partition(a: Array[Int], l: Int, r: Int): (Int, Int) = {
  val pivot = a(l) // Choosing the first element as the pivot for simplicity
  var i = l // Boundary for elements less than pivot
  var j = l // Boundary for elements equal to pivot
  var k = r // Boundary for elements greater than pivot; we adjust 'r' to 'r-1'
if indexing is exclusive

  while (j < k) {
    if (a(j) < pivot) {
      // Swap elements to ensure a[l..i) < pivot
      val temp = a(i)
      a(i) = a(j)
      a(j) = temp
      i += 1
      j += 1
    } else if (a(j) == pivot) {
      // Move to the next element as a[i..j) = pivot
      j += 1
    } else {
      // Swap elements to ensure a[k..r) > pivot
      k -= 1
      val temp = a(j)
      a(j) = a(k)
      a(k) = temp
    }
  }

  (i, j) // Returning the boundaries between each segment
}
```

The pivot is chosen as the first element, but we could pick a random to make it more optimised on an average scale.

The loop invariant: `a[l..i) < pivot && a[i..j) = pivot && a[k..r) > pivot` are maintained.

The array is iterated through with the variable `j`, moving elements into the correct partition based on their comparison with the pivot. This ensures that by the end of the loop, all elements are correctly grouped into less than, equal to, or greater than pivot segments.

e) Some adjustments are required in the quicksort algorithm, but this is what it will look like:

```
def quickSort(a: Array[Int], l: Int, r: Int): Unit = {
  if (l < r - 1) { // Check if the segment has more than one element
    val (i, j) = partition(a, l, r) // Partition the array segment
    quickSort(a, l, i) // Recursively sort the segment with elements less than
the pivot
    quickSort(a, j, r) // Recursively sort the segment with elements greater than
the pivot
  }
}
```