Tuan Date:

Revision paper

2.

a) First, I will write a function partition which partitions an array, with the first element as a pivot.

```
                                                    Int
def partition(arr: Array[Int], l: Int, r: Int): ~~Array[Int]~~ = {
  var i = l+1; var j = r; val pivot = arr(l)
  // Inv: arr[l+1..i) ≤ pivot ≤ arr[j..r) ∧ l ≤ i ≤ j ≤ r
  // ∧ arr[0..l) = arr0[0..l) ∧ arr[r..N) = arr0[r..N)
  // ∧ arr[l..r) is a permutation of arr0[l..r)
  while (i < j) {
    if (arr(i) ≤ pivot) i += 1
    else {
      val swap = arr(j-1)
      arr(j-1) = arr(i)
      arr(i) = swap
      j -= 1
    }
  }
  val swap = arr(i-1)
  arr(i-1) = pivot
  arr(l) = swap
  arr i-1
}

def sort(a: Array[Int], u: Int): Unit = {
  ~~val k = partition(a,~~
  q sort(a, 0, u)
}
```

NOTEBOOK

Date:

```
def qSort(a: Array[Int], l: Int, r: Int): Unit = {
  if (r - l > 1) {
    val k = partition(a, l, r)
    qSort(a, l, k); qSort(a, k+1, r)
  }
}
```

b) This behaviour can be inefficient because some elements may be swapped two times. Suppose we have the program from the previous question. If arr(j-1) is bigger than the pivot and arr(i) is bigger than the pivot, then arr(j-1) and arr(i) would swap. At the next iteration, we would have to swap the element at arr(i) again. Another way this might be inefficient is if we have many equal elements.

c)

```
def partition(arr: Array[Int], l: Int, r: Int): Int = {
  var i = l+1; var j = r; var pivot = arr(l)
  // Inv: arr[l+1..i) ≤ pivot ≤ arr[j..r) ∧
  // l < i ≤ j ≤ r ∧ arr[0..l) = arr0[0..l) ∧ arr[r..N) =
  // = arr0[r..N) ∧ arr[l..r) is a perm. of arr0[l..r)
  while (i < j) {
    while (i < j && arr(i) < pivot) i += 1
    while (i < j && arr(j) > pivot) j -= 1
    val swap = arr(i)
    arr(i) = arr(j)
    arr(j) = swap } }
```

Date:

```
val swap arr (i,l)
    arr(i-1) = pivot
    a(l) = swap
}
```

The sort function will be the same. This solves only one of the problems - every element will be moved at most once. But if the array contains many equal elements, the partition function would still be slightly inefficient.

3.

```
a) trait IntSet {
    state = {Int}    //set of Ints
    init = {}

    bdd
    // Post: set = set ⊕ {elem}  (⊕ is union of sets)
    def add (elem: Int)

    // Post: returns elem ∈ set
    def isIn (elem: Int)

    // Pre: elem is in set
    // Post: set = set ⊖ {elem}
    def remove(elem: Int)

    // Post: returns #set
    def size
```

NOTEBOOK Date:

b) We need to rewrite only the add and remove functions:

```
// Pre: elem ∈ [0..N)
// Post: set = set + {elem} and return true
//   v set = set0 and return false if elem ∈ set
def add(elem: Int): Boolean

// Pre: elem ∈ [0..N)
// Post: set = set - {elem} and return true
//   v set = set0 and return false if elem ∉ set
def remove(elem: Int): Boolean
```

c)

```
class BitMapSet extends IntSet {
  // Abs: set = {x | a(x) == true}
  // DTI: count = #{x | a(x) == true}

  var a = new Array[Boolean](N)
  var count = 0

  def add(elem: Int): Boolean = {
    val oldValue = a(elem)   // assert(0 ≤ elem && elem < N)
    a(elem) = true
    !oldValue
  }

  def isIn(elem: Int): Boolean = {
    if (elem < 0 || elem ≥ N) false
    a(elem)
  }
```

Date:

```
def remove (elem : Int) : Boolean = {
  val oldValue = a (elem)   assert (0 ≤ elem && elem < N)
  a (elem) = false
  oldValue
}
def size = count
```

d)

```
def sort (xs : Array [Int]) : Array [Int] = {
  var newArray = new Array [Int] (xs.length)
  var bitMap = new BitMap(Set)
  for (i ← 0 until N) bitMap.add (xs(i))
  // The assertion whether elem ∈ {0..N) is done in the
  // function

  var i = 0; var j = 0
  // Inv: [0..j) are in newArray ∧ i ≤ xs.length
  while (i < newArray.length) {
    while (!bitMap.a(j)) j += 1
    newArray (i) = j
    i += 1
  }
}
```

4.

a)

```
class Tree (data : Int; left : Tree, right : Tree)
```

b)

```
def inorder (t : Tree) : Unit = {
  if (t.left != null) inorder (t.left)
```

```
print (t.datum + " ")
if (t.right != null) inorder (t.right)
```

c)

```
def makeTree (u: Array[Int], a: Int, b: Int) : Tree = {
  if (b - a == 1) new Tree (u(a), null, null)
  else {
    ~~val mid = u ((b-a)/2) // assume it won't overflow~~
    val midIndex = (b-a)/2
    new Tree (u(midIndex), makeTree (u, a, midIndex),
              makeTree (u, midIndex+1, b))
  }
}
```

d)

```
def inorder (t: Tree) : Unit = {
  var current = t  // current points to the root
  // Invariant: T(current) is not printed yet.
  while (current != null) {
    if (current.left == null) {
      print (current.datum + " ")
      current = current.right }
    else { // make current the right child of the rightmost
// node in the left subtree
      var node = current.left
      while (node.right != null) node = node.right
      node.right = current
      current = current.left
    }
  }
}
```

Date:

e) The code in the previous questions traverses a node at most twice, and because it is a tree there will be no cycles. Therefore, the code runs in time proportional to the size of the tree.

5.

a)
```
word::String,
class Node (Vcount : Int, next : Node)
```

b)
```
def add (w: String) = {
  val i = hash(w)  // the cell in which w needs to go
  var current = table(i).next // header cell
  //Inv: w is not in L(table(i).next, current), where
  //L(a,b) = [] , if a = b
  // L(a,b) = a :: L(a.next, b) , if a ≠ b
  while (current.next != null || current.word != w) {
    current = current.next
  }
  if (current.word == w) current.count += 1
  else current.next = new Node (w, 1, null)
}
```

c)
```
def output (bucket : Node) : Node = {
  var newList = new Node (null, 0, null), var last = newList
  var current = bucket
  //Inv: L(current, null) have not been added to newList
  //  ∧ L(newList.next, null) is ordered
```

NOTEBOOK Date:

```
while (current.next != null) {
  //Invar: (newList, last).count < current.count
  while (last.next != null && last.next.count <
                                  current.count) {
    last = last.next }
  if (last.next == null) last.next = new Node (current.word,
                                  current.count, null)
  else {
    last.next = new Node (current.word, current.count,
                                  last.next)
  }
  last = newList
  current = current.next
}
newList
}

d)
def merge (l1: Node, l2: Node): Node = {
  var curr1 = l1.next; var curr2 = l2.next
  var newList = new Node (0, 0, null); var last = newList
  while (curr1.next != null && curr2.next != null) {
    if (curr1.count < curr2.count) {
      last.next = new Node (curr1.word, curr1.count, null)
      last = last.next
      curr1 = curr1.next
    }
    else {
      last.next = new Node (curr2.word, curr2.count, null)
      last = last.next
```

Date:

```
        curr2 = curr2.next
    }
}
if (curr1.next == null) {
    while (curr2.next != null) { last = new Node (curr2.word,
                                 curr2.count, null)
        last = last.next
        curr2 = curr2.next
    }
}
else {
    while (curr1.next != null) {
        last = new Node (curr1.word, curr1.count, null)
        last = last.next
        curr1 = curr1.next
    }
}
newList
}
```

e)

```
def arrange (): Node = {
    var newList = new Node ("", 0, null); var last = newList
    var i = 0
    //Inv: table[0..i) have been added to newList ∧
    // newList is sorted ∧ 0 ≤ i ≤ N
    while (i < N) {
        var tempList = output (table(i))
        merge (newList, tempList)
        i += 1
    }
    newList
}
```

NOTEBOOK Date:

f) On average, each bucket has $\frac{S}{N}$ nodes. Insertion set would take $O\left(\left(\frac{S}{N}\right)^2\right)$ and merging all is linear to the number of words $\Rightarrow O(S)$. ~~Suppose $\frac{S}{N} = d$~~ $\Rightarrow$ overall the average time taken is $O\left(\frac{S^3}{N^2}\right)$.
The worst case is when each bucket has $\frac{S}{N}$ nodes and they are in descending order.

1.

a)

```
def printFrag(d: Array[Int], i: Int, j: Int, k: Int) = {
  print("  " + "  ")  // Two space
  for(i <- 0 until j) print("  ")
  for(i <- j until k) print(" " + "  ")
  println()
  print("0.")
  for(i <- 0 until k) print(d(i))
}
```

b)

```
def frag(N: Int): (Array[Int], Int, Int) = {
  var a = new Array[Int](100) // Assume 100 is enough
  var j = 0; var k = 0; var numerator = 1; var i = 0
  var bitMap = new Array[Boolean](N)
  // Inv: a(i) < 10 ∧ a(i) ≥ 0 ∧ bitMap contains all the
  // remainders we have encountered
  bitMap(1) = true
  while (numerator < N) {numerator *= 10; a(i) = 0; i += 1}
  while (numerator % N != 0 || !bitMap(numerator/N)) {
    a(i) = numerator / N; i += 1; k += 1
    ~~bitMap(a(i)) = true~~
```

Date:

```
    numerator = numerator % N
    bitMap(numerator) = true
}
while(numerator < N) numerator *= 10
while(a[j] != (numerator / N)) j += 1
return (a, j, k)
}
```

c) Every time the numerator is less than N, multiply by 10 and put a zero into the array and increment the length of the "real" array. The code correctly identifies the recurring segment, because it creates a bit-map set, which holds all the remainders we have seen. Using bit-map set allows us to look-up whether & we have seen a remainder in O(1) => the program has complexity O(N). The program correctly terminates because it looks at the two different cases: 1) numerator % N == 0, 2) we have already ~~encountered~~ seen the remainder value.