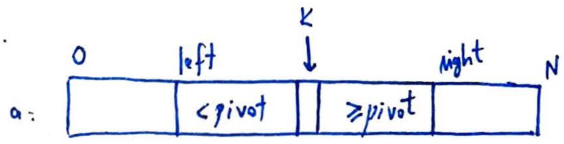
MOISE GABRIEL

IMPERATIVE PROGRAMMING 1

QUESTION 2

(a) Our partition function:

def partition(a: Array[int], left: int, right: int): ~~Unit~~ int = ...



has the following effect:

- $a[0..left) = a\_0[0..left)$ && $a[right..N) = a\_0[right..N)$ &&
  $a[left..right)$ is a permutation of $a\_0[left..right)$ with $pivot = a\_0(left)$,
  returning K s.t. $a[left..K) < pivot$ && $a[K..right) \geq pivot$, $left \leq K \leq right$

Then, the sorting function is:

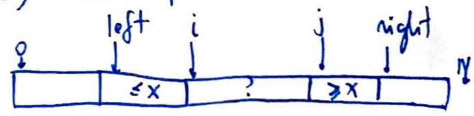
```
def Sort(a: Array[int], n: int) : Unit = QuickSort(a, 0, n)
  ~~Var k = partition~~
def QuickSort(a: Array[int], left: int, right: int) : Unit = {
  Var k = partition(a, left, right)
  if (k-left > 1) QuickSort(a, left, k)
  if (right-k > 1) QuickSort(a, k, right)
}
```

- For an array of length 1, it is already sorted
- For an array of length >1, it is partitioned according to the scheme above, therefore we only need to sort the 2 ~~halves~~ sub-arrays formed now until we get to length 1.

(b) The implementation of Partition where we split into 2 groups, one less than pivot and one $\geq$ pivot, can make QuickSort inefficient $(O(n^2))$, since on ~~the~~ an increasingly sorted array, the pivot is always equal to a(left), which is the smallest element, therefore partition(a, left, right) = left and therefore we reduce the problem by 1 each step: $T(n) \leq O(n) + T(n-1) \Rightarrow T(n) = O(n^2)$.

partition makes $O(n)$ steps as it goes through the array from left to right to separate it.

(c)




```
def Partition2 (a: Array[Int], left: Int, right: Int) : Int = {
    var i = left
    var j = right
    var pivot = a(left)
    // Invariant I: a[left..i) ≤ pivot && a[j..right) ≥ pivot && ~~i ≤ j~~
    //              left ≤ i ≤ j ≤ right && a0[0..left) = a[0..left) && a[right..N) = a_0[right..
    //              && a[left..right) is a permutation of a_0[left..right)
    ~~// Variant: j-i~~
    while (j > i) {
        while ((j>i) && (a(j) >= pivot)) j -= 1
        while ((j>i) && (a(i) <= pivot)) i += 1
        if (j>i) { var t = a(i); a(i) = a(j); a(j) = t ; i += 1; j -= 1 }
    }
    // j ≤ i => a[left..i) ≤ pivot && a[i..right) ≥ pivot => we return i
    i
}
```

Unfortunately, for increasingly-sorted arrays, the problem remains, since partition still gets left as the result and it reduces the problem by 1 at each step, so the algorithm is still quadratic.

QUESTION 1

1/N represented by $0 \le j < k$ and $d[0..k)$, k = minimum s.t.

$1/N = 0.d_0 d_1 d_2 \dots d_{j-1} \; d_j \, d_{j+1} \dots d_{k-1}$

repeating

(a)
```
def print (d: Array [Int], j: Int, k: Int): Unit = {
    ~~println ("0.")~~ print ("  "); print ("  ")   // 2 spaces for "0."
    for (i = 0 to j-1) print (" ")
    for (i = j until k) print ("-")
    println()
    print ("0.")
    for (i = 0 until k) print (d(i))
    println()
}
```

(b) ~~We will use an array of integers up to N, each~~

We will use an array of size N of Booleans, with num(i) ~~mean an~~ = true meaning that we arrived at an earlier stage of the division process at i and now the division produces recurring units. There will be a corresponding array pos, with pos(i) = -1 if num(i) = false and pos(i) = the place where we first encountered i during the division.

Edit: to reduce memory, we can get rid of num, since num(i) = (~~place~~ pos(i) != -1) so we'll only use pos.

```
def decimal (N : int) : ~~(int,~~ (Array[int], int, Int) = {
  var pos = new Array[int] (N)
  for (i <- 0 until N)  pos(i) = -1

  var current = 10
  var recur = false
  var j=0 ; var k = 0                                            Before recur = true
  var d = new Array [int] (N)  // can't have more since there are at most N possibilities

  ~~while (! recur)  if (current < N) { current = current * 10 ; d (k) = 0~~

      ~~if (current < N)  {~~
                         ~~pos (current) = k~~
                         ~~current = current * 10~~
                         ~~d (k) = 0~~
                         ~~k += 1~~
                     ~~}~~

      ~~else {~~
  while (! recur)
    if (current < N)  {   if (pos (current) != -1)
                           {  j = pos ( current)
                              recur = true
                           }
                        else  {  pos (current) = k
                                 current = current * 10
                                 d (k) = 0
                                 k += 1
                              }
                      }

  else {  ~~if (pos (curr~~  if (pos (current) != -1)  { j = pos(current); recur = true }
                        else  {  pos (current) = k
                                 ~~current = current~~
                                 d (k) = current / N
                                 current = current % N
                                 k += 1
  }                           }
```

return (d, j, k)
}

The program runs in time proportional to N since at each step we either terminate or we update a position from pos (there are up to N different cases $\Rightarrow$ after at most N+1 steps we are done).

(c) At each step, we have $\frac{1}{0.d_0d_1\dots d_k} < N$ and when we find that we get to a previous state, we can be sure that the pattern repeats itself since the steps will go in a cyclic direction.

- pos(i) = the step at ~~wit~~ which we first encountered i in the division (K)
- K = the number of steps so far
- when we repeat the pattern, j is the place where the infinite loop begins

We mentioned at (b) the reason why the program always terminates whatever the given value of N.

IMPERATIVE PROGRAMMING 2

QUESTION 3

(a) ~~// Abs:~~

trait intSet {

// State: $set \in P(int)$

// init: $set = \{\}$

// Post: $set = set_0 \cup \cancel{\{x\}}\ \{elem\}$

def add(elem: int): Unit

// Post: $set = set_0$ && return $elem \in set$

def isIn(elem: int): Boolean

// Post: $set = set_0 \setminus \{elem\}$

def remove(elem: Int): Unit

// Post: $set = set_0$ && return $|set|$

def size: int

}

(b) trait intSet {

state: $set \in P([0..N))$

~~//Pre~~

~~val N = ...~~

// Pre: $0 \le elem < N$

// Post: $set = set_0 \cup \{elem\}$ && return $(elem \notin set_0)$

def add(elem: Int): Boolean

// Pre: $0 \le elem < N$

// Post: $set = set_0 \setminus \{elem\}$ && return $(elem \in set_0)$

def remove(elem: Int): Boolean

...

}

// Pre: $elem \in [0..N)$

// Post: $set = set_0$ && return $elem \in set$

def isIn(elem: Int): Boolean

(c) // Abs.: set = { i ∈ [0..N) | a(i) == true }

// DTI : ~~(∀) i ∈ [0..N), a(i) == true <=> i ∈ set~~

size_ = # true values in a[0..N)

```
class BitMapSet ~~f~~ extends intSet {
  var a = new Array[Boolean](N)
  var size_ = 0
  // O(1)
  def add(elem : int) : Boolean = {
    require(elem < N) ; require(0 <= elem)
    var result : Boolean = a(elem)
    if (! result) size_ += 1
    a(elem) = 1
    return(! result)

  // O(1)
  def isIn(elem: int): Boolean = {
    require(elem < N) ; require(0 <= elem)
    return(a(elem))
  }
  // O(1)
  def remove(elem: int) : Boolean = {
    require(elem < N) ; require(0 <= elem)
    var result = a(elem)
    a(elem) = 0
    if (result) size_ -= 1
    result
  }
  // O(1)
  def size : int = size_
```

1(a)
```
def sort (xs: Array[Int]): Array[Int] = {
  // Since all the elements of xs are in [0..N) we can put them in a BitMapSet
  // and since they are all distinct, a(elem) = false means elem doesn't appear in xs
  // at all, otherwise it appears exactly once
  var n = xs.len var set = new BitMapSet
  for (i <- 0 until xs.size) set.add(xs(i))  // O(xs.size) since add is O(1)
  for (i <- 0 until N)
    if (isIn(i) set.isIn(i)) print(i + " ")  // O(N)
}
```
So, the sorting is linear.

QUESTION 5

(a)

```
class Node (val word : String, var count: int, var next : Node)
```

(b)

```
class HashBag {
    private val N = 100   // # buckets in the hash table
    private val table = new Array[Node](N)  // the hash table
    // table(i) ~~is~~ represents a linked list of words w s.t. hash(w) = i

    def add (w: String) : Unit = {
        var h = hash(w)
        var current = table(h)
        while ((current != null) && (current.word != w)) current = current.next
        if (current = null) { var n1 = new Node(w, 1, table(h)) ; table(h) = n1 }
        else { current.count += 1 }
    }
```

(c)

```
    def remove (i: int): Node = {
        var n = new Node("?", 0, null) // dummy header
        var current = table(i)
        // We removed the nodes from table(i) up to, but not including current
        ~~while~~
        while (current != null)
        {
            var prev = n
            var curr = prev.next
            // ~~prev.nex~~ Invariant: prev.next = curr && prev.word < current.word }
            while ((curr != null) && (curr.word < current.word)) { curr = curr.next;
                                                                   prev = prev.next }
            prev.next = current
            current.next = curr
        }
        n
    }
```

(d)
```
def sort (list1: Node, list2: Node): Node = {
    var result = new Node ("?", 0, null) ; var end = result
    var curr1 = list1.next
    var curr2 = list2.next
    while ((curr1 != null) || (curr2 != null))
    { else if (curr1.word == curr2.word)
        { var n1 = new Node (curr1.word, curr1.count + curr2.count, null)
          end.next = n1
          end = n1 ; curr1
        }
      else if (curr1.word < curr2.word)
        { ~~var n1 = new Node (curr2.word, curr~~
          curr2.next = null
          end.next = curr2
          end = curr2
          curr2 = curr2.next
        }
      else if (curr1.word > curr2.word)
        { curr1.next = null
          ~~curr~~ end.next = curr1
          end = curr1
          curr1 = curr1.next
        }
      if (curr1 == null)
        {
          end.next = curr2
          curr2 = null
        }
      else if (curr2 == null)
        { end.next = curr1
          curr1 = null
        }
    } result
```

(e) def sortTable : Node = {

~~var~~ var auxtable = table //to not destroy the original table

~~f~~ var size = N

// invariant : we need to merge the table [0.. size)

~~while (size >= 1~~

while (size > 1)

{ var half = (size +1) / 2 // [0..2k) → [0..k) ; [0..2k+1) → [0..k+1)

for (i <- 0 until half) ~~table (i) =~~

{ var list1 = remove (i)

var list 2 = ~~siz~~ remove (size - i - 1)

~~aux~~table(i) = sort (list1, list2)

}

size = half

}

// size == 1 => return table(0)

table = auxtable

table(0)

}

(f) S = number of distinct words input to the program

$\frac{N}{S}$ = constant

~~The worst case is when we have all the words hash to the same bucket, therefore sort runs in~~

Considering that the hash function makes the table have on every bucket a linked list of size close to $\frac{S}{N}$, the sorting should ~~take time~~ need $O(S \log_2 S)$ since we need $\log_2 S$ iterations of the loop, each of them taking time linear to S. The worst-case arises when the hash function is not efficient and puts all the words in a bucket => $O(S^2)$ time complexity.

IMPERATIVE PROGRAMMING 3

QUESTION 7

(a) Polymorphism is a technique by which different constructs in a programming language can process/use ~~different data~~ objects of different data types in appropriate ways.

Polymorphism arises in object-oriented programming in:

- overloading: when the same method is used for several distinct argument datatypes
- ~~or~~ inheritance: code can be written to use an interface/implementation from other code interchangeably
- some code can be written generically so that it can handle argument values ~~of~~ identically without depending on their type

(b)

```
class PartialFn [T,U] (val data: List [(T,U)]) {
    def get (x: T) : Option [U] =
    { var list = data. filter ~~((a,b))~~ ((a,b) => (a == x))
      if (list. length == 0) return None
          else { return Some (list. head._2)
    }

    def remove (x: T) : PartialFn [T,U] =
    {
      var list = data. filter ((a,b) => (a != x))
      ~~ret~~ new PartialFn [T,U] ( list)
    }
```

~~def (x:~~

```
def add (x: T, y : U) : PartialFn [T,U] =
{   val list = (x,y) :: data.filter ((a,b) => (a != x))
    new PartialFn [T,U] (list)
}
```

(c)
```
def compose (that: PartialFn [U,V]) : PartialFn [T,V] =
{   ~~def var list =~~ var list = new List [(T,V)] ()
    for ( pair <- data)
    {  var x = pair._1 ; var y = pair._2
       var aux = that.data.filter ((a,b) => (a == y))
       var z = aux.head._2
       list = (x,z) :: list
    }
    new PartialFn [(T,V)] (list)
}
```

(d) If two type parameters A and B of a generic class have

$A <: B$ implies that $List[A] <: List[B]$

then A and B are covariant.

(e) To ensure that U was covariant