# Algorithms and Data Structures Miniproject:

Dimitrios Papaoikonomou, Oliverio Bombicci Pontelli, Hudhayfa Ahmed

## Member Contributions:

Oliverio began by creating a GitHub repository to which we all could work on. He worked on the ArrayListWithUndo question, implementing all the functions (append, remove, insert and undo). The code was tested and then debugged until everything worked as expected. Oliverio also helped with question 2, fixing a bug which prevented test 13 from printing by adding a bounds-check in the undo function which was missing.

Hudhayfa worked on question 2 (NetworkWithUndo) implementing the add, root, merge and undo. He tested the implementation and identified an issue with test 13, which was later resolved by Oliverio. Additionally, Hudhayfa worked on the report, writing the member contributions and implementation description.

Dimitrios provided assistance with question 1, helping Oliverio fix some bugs.
He also worked on the third question – implementing the ‘Gadget’ class. This meant he worked on the ‘add’, ‘subnets’, ‘connect’, ‘clean’ and ‘undo’ functions. Initial testing showed the code to work as expected after some debugging. However, upon further testing with the testing script, there seems to be a problem with the code which we were unable to figure out.

## Implementation Description:

### Question 1 – ArrayListWithUndo:

ArrayListWithUndo is an extension of the ArrayList class given which allows the programmer to undo the operations that occurred in reverse order (i.e. first-in, last-out). This is achieved via a stack. In order to achieve this, basic ArrayList functionality was rewritten to include 'undo' functionality.

The 'undo' functionality works on the basis that each function, whether it be append, insert or remove, will push an undo instruction onto the aforementioned stack. This instruction is a triple made up of what operation needs to be done, the relevant index and finally, if required, a value. Then, once the undo function is called, the top of the stack is checked, and the relevant operation is carried out.

For example, if we append the number '5', the remove operation would be added to the stack, indicating that the element in index i (for an empty arraylist we would have appended to index 0) should be removed. When the undo function is then called, the 'count' (indication of the array size) would decrease by one, and then all the elements following index i would be moved to the left once – essentially overwriting 'i'. In our case, first element would just become empty as there is nothing to replace it. If need be, the arraylist can be resized upwards.

If the operation was to set, then the element in index i would be replaced by whatever value is in 'v'.
If the operation was to insert, then all elements from index i onwards are shifted one to the right and then the new value is inserted into i. If the array becomes full it is resized.

### Question 2 – NetworkWithUndo:

The NetworkWithUndo class manages a network of nodes, each represented by an integer, and organizes them into clusters. These clusters are represented as trees, with each node pointing to its parent or root. The class allows adding new nodes, finding cluster roots with path flattening, merging clusters, and undoing operations.

The constructor initializes the network with N nodes, each starting as its own cluster. It uses an ArrayListWithUndo to store node pointers and sizes, ensuring efficient undo operations. Each node starts with a size of -1, indicating it is a root of a single-node cluster. A stack is also initialized to track the number of undo operations.

The class allows the addition of new nodes via the 'add' function, which are appended as a new cluster onto the ArrayListWithUndo with the value -1. Then a value of 1 is pushed onto the networks undo stack.

The ‘root’ function locates the root of the cluster of a node. This is simply done by starting at the given node and following pointers from node to node till a pointer goes negative. During this a list of visited nodes is taken and at the end, all of these are made to point to the root node (path flattening) to improve efficiency.

The ‘merge’ function merges clusters when given two root nodes. The larger root node will become the root node of the merged clusters.

The ‘undo’ function works slightly differently to the one with ArrayListWithUndo as it works ontop of it. The undo function here simply indicates how many undo instructions should be completed from the stack. For example, assume ‘2’ is pushed onto the ‘NetworkWithUndo’ undo stack. Once undo is called, the undo will call the undo function of the ArrayListWithUndo twice. This is needed as the merge function requires that two values are set in the arraylist.

## Question 3 – Gadget:

The ‘Gadget’ class extends the functionality of ‘NetworkWithUndo’ in order to manage multiple networks.

When a new node is added to the gadget, a check is done against the hashmap to make sure there are no duplicates. If not, the node is added, and the undo stack position is stored.

The connect method connects two named nodes. It checks if both nodes exist in the network. If they do, it checks whether they are already part of the same cluster. If they aren't connected, the method merges their clusters. If they are already connected, no changes are made.

The subnets method returns a list of clusters (subnets) in the network. It identifies all root nodes in the network, and stores them in an array. It then groups nodes that are connected to each root and inserts them into the array. It returns this array, which represents the nodes in each cluster.

The undo method undoes the last n operations made on the network. It loops n times, performing the last instruction on the undos stack, and performing specific instructions depending on the operation stored in the popped triple. If the instruction is rem, then it will remove the name from the nameMap and helper, and reduce the size of the gadget and internal network accordingly, then calls the given number of undo operations. If the instruction is brk, it increases the subsize by 1, then calls the given number of undo operations. Otherwise, if the instruction is oth, it just calls the undos.

The clean method removes all changes from when a specific name was added to the network, completely reverting the network back to the state it was in at that moment. It does this by calculating the number of steps that have happened between the current moment and the moment the name was added and calling undo with that number as an argument to perform that many undos.

## Code Implementation

```
class ArrayListWithUndo(ArrayList):
  def __init__(self):
    # already implemented
    super().__init__()
    self.undos = Stack()

  def set(self, i, v):
    # already implemented
    self.undos.push(("set",i,self.inArray[i]))
    self.inArray[i] = v

  def append(self, v):
    # TODO
    self.undos.push(("rem", self.count, None))
    self.inArray[self.count] = v
    self.count += 1
    if len(self.inArray) == self.count:
      self._resizeUp()

  def insert(self, i, v):
    # TODO
    self.undos.push(("rem", i, None))
    for j in range(self.count,i,-1):
      self.inArray[j] = self.inArray[j-1]
    self.inArray[i] = v
    self.count += 1
    if len(self.inArray) == self.count:
      self._resizeUp()

  def remove(self, i):
    # TODO
    self.undos.push(("ins", i, self.inArray[i]))
    self.count -= 1
    val = self.inArray[i]
    for j in range(i,self.count):
      self.inArray[j] = self.inArray[j+1]
    return val

  def undo(self):
    if self.undos.size == 0:
      return
    toUndo = self.undos.pop()
    if toUndo[0] == "set":
      self.inArray[toUndo[1]] = toUndo[2]
    elif toUndo[0] == "ins":
      for j in range(self.count,toUndo[1],-1):
            self.inArray[j] = self.inArray[j-1]
          self.inArray[toUndo[1]] = toUndo[2]
          self.count += 1
        else:
          val = self.inArray[toUndo[1]]
          for j in range(toUndo[1],self.count):
            self.inArray[j] = self.inArray[j+1]
          self.count -= 1
          return val

    def __str__(self):
      # already implemented
      return str(self.toArray())+"\n-> "+str(self.undos)


## NetworkWithUndo
## Each element in arraylist holds the value of the node it is linked to in the network
## -1 means the node is a root
class NetworkWithUndo:
  def __init__(self, N):
    # already implemented
    self.inArray = ArrayListWithUndo()
    for _ in range(N): self.inArray.append(-1)
    self.undos = Stack()
    self.undos.push(N)

  def getSize(self):
    # already implemented
    return self.inArray.length()

  def add(self):
    # DONE - To be tested
    self.inArray.append(-1)
    self.undos.push(1)
    return

  def root(self, i):
    # DONE - Check complexity - To be tested
    rememberedNodes = ArrayList()
    while self.inArray.get(i) > -1:
      rememberedNodes.append(i)
      i = self.inArray.get(i)
    for j in range(0,rememberedNodes.length()-1):
      self.inArray.set(rememberedNodes.get(j), i)
    self.undos.push(rememberedNodes.length()-1)
    return i
  def merge(self, i, j):
    # TODO
    if (self.inArray.get(i) > -1) or (self.inArray.get(j) > -1):
      # print(self.inArray.get(i), self.inArray.get(j))
      assert(0)
    if i == j:
      # self.undos.push(0)
      return
    if self.inArray.get(i) < self.inArray.get(j):
      # i becomes the root
      self.inArray.set(i, self.inArray.get(i) + self.inArray.get(j))
      self.inArray.set(j, i)
    else:
      # j becomes the root
      self.inArray.set(j, self.inArray.get(j) + self.inArray.get(i))
      self.inArray.set(i, j)
    self.undos.push(2)


  def undo(self):
    if self.undos.size == 0:
      return
    numInstructions = self.undos.pop()
    if numInstructions == 0:
      return
    for _ in range(numInstructions):
      self.inArray.undo()

  def toArray(self):
    # already implemented
    return self.inArray.toArray()

  def __str__(self):
    # already implemented
    return str(self.toArray())+"\n-> "+str(self.undos)


class Gadget:
  def __init__(self):
    # already implemented
    self.inNetwork = NetworkWithUndo(0)
    self.subsize = 0
    self.nameMap = {}
    self.undos = Stack()
    self.helper = {}

  def getSize(self):
        # already implemented
        return self.inNetwork.getSize()

    def isIn(self, name):
        # already implemented
        return name in self.nameMap

    def add(self, name):
        if self.isIn(name):
            self.undos.push(("oth", 0, None))
            return
        else:
            self.nameMap[name] = self.subsize
            self.subsize += 1
            self.inNetwork.add()
            self.helper[name] = self.undos.size
            self.undos.push(("rem",1, name))

    def connect(self, name1, name2):
        if name1 not in self.nameMap or name2 not in self.nameMap:
            return False
        i=self.nameMap[name1]
        j=self.nameMap[name2]
        if self.inNetwork.root(i) == self.inNetwork.root(j):
            return True
        self.inNetwork.merge(self.inNetwork.root(i),self.inNetwork.root(j))
        self.subsize -= 1
        self.undos.push(("brk", 3, None))
        return False

    def clean(self, name):
        if name not in self.helper:
            return
        steps_to_undo = self.undos.size - self.helper[name]
        self.undo(steps_to_undo)

    def subnets(self):
        A= ArrayList()
        counter = 0
        for i in range(self.inNetwork.getSize()):
            if self.inNetwork.root(i) == i:
                A.append(i)
            counter += 1
        for i in range(A.length()):
            temp= ["" for i in range(-self.inNetwork.inArray.get(A.get(i)))]
            ArrCounter=0
            for j in range(self.inNetwork.getSize()):
```

```
            if self.inNetwork.root(j) == A.get(i):
                for name, index in self.nameMap.items():
                    if index == j:
                        temp[ArrCounter] = name
                        break
                ArrCounter += 1
        A.set(i, temp)
    self.undos.push(("oth", counter, None))
    return A.toArray()

def undo(self, n):
    while self.undos.size > 0 and n > 0:
        toUndo = self.undos.pop()
        if toUndo[0] == "rem":
            self.nameMap.pop(toUndo[2], None)
            self.helper.pop(toUndo[2], None)
            self.subsize -= 1
            self.inNetwork.inArray.count -= 1
            for i in range(toUndo[1]):
                self.inNetwork.undo()
        elif toUndo[0] == "brk":
            self.subsize += 1
            for i in range(toUndo[1]):
                self.inNetwork.undo()
        elif toUndo[0] == "oth":
            for i in range(toUndo[1]):
                self.inNetwork.undo()
        n -= 1

def toArray(self):
    # already implemented
    A = self.inNetwork.toArray()
    for s in self.nameMap:
        i = self.nameMap[s]
        A[i] = (s,A[i])
    return A

def __str__(self):
    # already implemented
    return str(self.toArray())+"\n-> "+str(self.nameMap)+"\n-> "+str(self.undos)
```