ECS529U Algorithms and Data Structures **Mini-Project**

Deadline: Monday 2 December (week 11) at 10am

*Read the project description carefully, including **Essential Guidelines** on page 8.*

What to submit

You should submit **exactly both** of the following files:

1. A **PDF file** `report.pdf` with a report, in which you should include:
   a) A section describing each group member's contribution. There should be one paragraph for each member and the whole section should cover about 1 page.
   b) A short description of your implementation. This should be no longer than 3 pages and should explain the main points of each function implemented. You should not try to explain your code line-by-line.
   c) Your code implementations, leaving out the code for `ArrayList` and `Stack`.

   In order to be able to check submissions for plagiarism:
   - The report should be written electronically.
   - Do not embed the code in the report as an image; rather, copy-and-paste it in your document as text.
2. A **Jupyter file** called `solutions.ipynb` containing your code:
   - the file should contain the full classes `ArrayListWithUndo`, `NetworkWithUndo` and `Gadget`, as well as `ArrayList` and `Stack` (these two are given).

   We will mark your code automatically, so make sure it is correctly indented and without syntax errors, and that it can run without errors !!!

   Do not include any module imports, etc.; include only your classes.

   Do not include any testing code (this is typically a source of errors).

   Put all your code in one cell (i.e. a single "box" of code) with all classes inside it.

Mark allocation

There are three marked components for this project: the report, the code and the viva, with corresponding marks **R**, **C** and **V** out of 100. All of these contribute to the group mark (**G**):

- if either **R** or **V** is below 20 then **G** = 0;
- otherwise, **G** = 0.1 * **R** + 0.85 * **C** + 0.05 * **V**

Your code mark **C** will be calculated automatically using a set of tests. **We will also look at the code to ensure that essential guidelines are respected (see page 8)**, and deduct marks if they are not. The tests are going to be revealed only after all projects have been marked, although we will reveal a part of them before submission to assist you.

Each group member's individual mark (**I**) will be calculated from **G** using information from the report and viva:

- if the member did not contribute at all then **I** = 0;
- if the member clearly had little contribution then **I** = **p** * **G**, for some **p** < 1;
- otherwise, **I** = **G**.

The Planet is Under Attack!

An alien civilisation has unleashed a cyber-physical attack and splintered all human computer networks. Can your team implement an efficient data structure to restore connectivity and protect networks from unsafe connections before it is too late?

Below is a plan the ECS529U module leaders have entrusted you..

You are asked to use a data structure for grouping network nodes together into **clusters** in such a way that connecting two clusters together and figuring out whether two nodes are in the same cluster is very efficient (almost constant amortised complexity). At the same time, the data structure supports an undo operation to allow for reverting network connections that are deemed unsafe – intelligence on unsafe connections can come in belatedly, at which point all connections from the unsafe one on need to be undone. We shall call this data structure `NetworkWithUndo`.

The `NetworkWithUndo` data structure shall in turn be based on `ArrayListWithUndo`, a more efficient implementation of the `ArrayListWithUndo` that you were asked to work on in Lab 5. It supports the standard array list operations along with an undo operation that restores the array list to how it was before the last operation on it was performed.

Question 1 – ArrayListWithUndo [30 marks]

You are asked to implement an `ArrayListWithUndo` data structure which extends array lists by addition of an undo functionality, similarly to Question 7 of Lab 6. The implementation we followed in Lab 6 is space- and time-inefficient as for each operation performed on the array list it needed to create whole copy of the whole array list. Instead, here we shall follow a different idea: for each operation performed we shall store in our data structure an **undo instruction** on how the operation can be undone.

An undo instruction will simply be a triple `(op,i,v)` with `op` being a string, `i` an index and `v` a value (or `None`). The string `op` specifies how we need to change the array list:

- if `op` is `"set"` then we need to set element `i` of the array list to `v`
- if `op` is `"rem"` then we need to remove element `i` from the array list (`v` is not used)
- if `op` is `"ins"` then we need to insert the value `v` in position `i` of the array list

Each `ArrayListWithUndo` object will contain a stack of undo instructions, stored in a variable called `undos`.

Each of the operations that modify the stack, namely `set`, `append`, `remove` and `insert`, will need to push an undo instruction onto the `undos` stack. For example, here is the implementation of the `set` function (note that we do not need to worry about out of bounds errors – humanity is in danger!):

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

Thus, before setting element `i` to `v`, we push onto `undos` an instruction to set element `i` back to its current value.

You are asked to implement the following functions for the `ArrayListWithUndo` class. The first three functions should also push on `undos` an undo instruction that reverts the operation they perform.

- `append(self, v)`: appends the value `v` at the end of the array list.

- `remove(self, i)`: removes from the array list the element in position `i`.
- `insert(self, i, v)`: inserts in position `i` of the array list the value `v`.
- `undo(self)`: pops the last undo instruction from `undos` and performs the corresponding operation on the array list (i.e. on `self`). If `undos` is empty, it returns without altering the array list.

We have provided you with the code for the constructor, `set` and `str`, as well as array list functions inherited from `ArrayList` (e.g. `get`). You should not change these.

Note that operations `append` and `insert` may lead to resizing up the internal array of the array list. These resizing-up sub-operations should not be taken into account by `undo`, i.e. you should not try to revert them by resizing down.

Question 2 – NetworkWithUndo [40 marks]

We call endpoints in a network **nodes** – you can imagine these being computers, phones, routers, etc. Our job is to build a `NetworkWithUndo` class to keep track of which nodes are connected with which nodes, either directly or via other nodes. We will achieve this by grouping nodes into **clusters**. Two nodes shall belong to the same cluster if, and only if, they are connected. A **network** is a grouping of nodes into clusters such that each node belongs to exactly one cluster. For economy, we denote nodes by integers 0,1, 2, 3, … .

In the language of sets: we have a set of integers {0, 1, …, *N*-1}, representing the nodes, a cluster is a subset *A* of *N*, and a network is a partition of {0, 1, …, *N*-1}.

Clusters are represented using lists (in fact, trees): a pointer from node *x* to node *y* means that *x* and *y* belong to the same cluster. Each node has exactly one outgoing pointer (which can be `None`) and no pointer cycles are allowed. For example, suppose our network has nodes 0, 1, 2, 3, 4, 5, 6, 7, 8 and its clusters are:

{0, 3, 4}, {1, 2}, {5}, {6, 7, 8}.

This network can be represented by the set of lists in Figure 1, where pointers to `None` are written as ending in a bullet (•). Nodes that point to `None` are called **roots**.

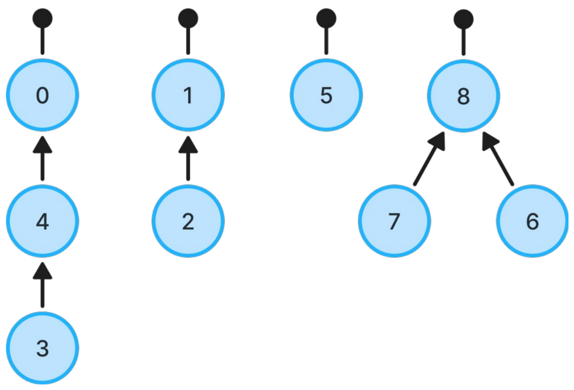
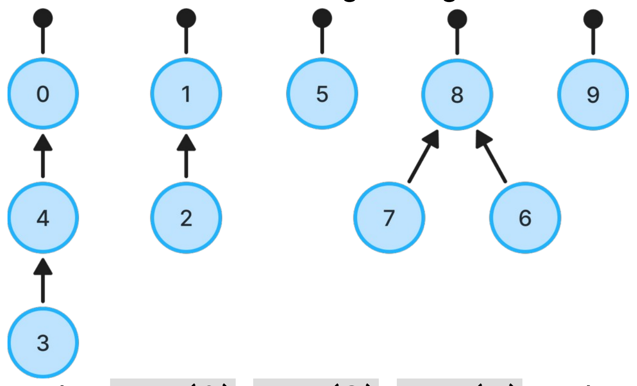


Figure 1: A network with nodes {0, 1, …, 8}.

Remember that clusters are simply sets of nodes. Therefore, the above is just one of the ways to represent the given network. There are several other ways, e.g. instead of the list `3 → 4 → 0 → None` we could have used `0 → 3 → 4 → None`, etc.

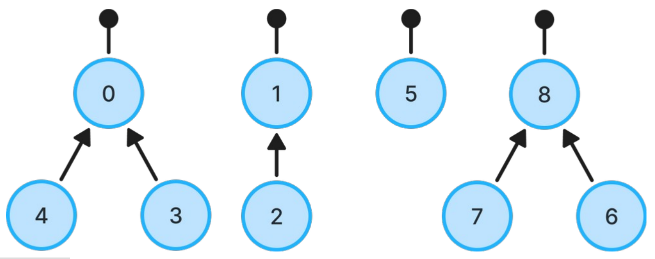
The functions that we need to perform on networks are: addition of a new node (`add`), finding the root node of cluster (`root`), and merging of two clusters (`merge`). Additionally, we need an **undo** function that reverts the last performed operation. We first explain these operations informally and then go over their intended implementation.

Given a `NetworkWithUndo` object with *N* nodes (which must be 0, 1, …, *N*-1), adding a new node amounts to simply adding node *N* and include it in a cluster of its own. For example, adding a node to the network of Figure 1 gives us:

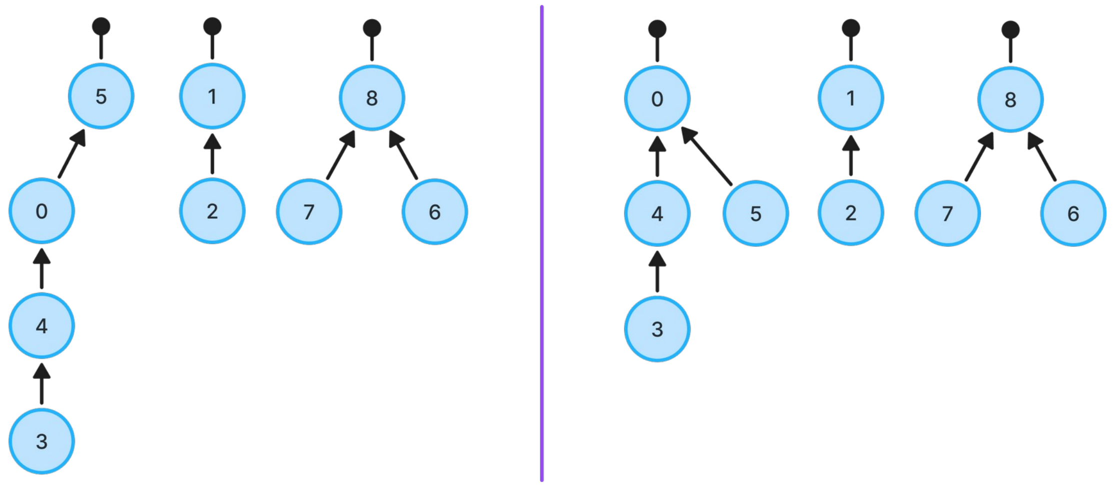


On the other hand, running `root(0)`, `root(3)`, `root(4)` on the network of Figure 1 should all return `0`. A first solution for `root(i)` is to simply follow pointers up from `i` to the corresponding root node and return the latter.

An important observation to make about `root` is that its worst-case time complexity is linear to the length of the longest list (i.e. path) in the network. Thus, `root` is most efficient if our network is shallow, i.e. our clusters have low heights. This leads to the following optimisation for `root`. When following up pointers from a node, we remember the nodes we visit on the way and, once the root node is found, we make all remembered nodes point directly to the root. We call this process *path flattening*. For example, calling `root(3)` on the network of Figure 1 will return `0` but also change the data structure so that node 3 points directly to its root, i.e. to 0. Thus, the resulting network will be:



We next look into `merge`. Merging of two clusters is done by making the root node of one cluster point to the root node of the other one. Which of the two root nodes is made to point to the other can be important for efficiency of future `root` operations. For example, suppose that in the network of Figure 1 we want to merge the clusters with roots 0 and 5. The two options for merging are depicted below:



On the left-hand side, node 0 is linked to 5 and this leads to an increase of the longest path in the network (the longest path was 2, now it is 3). On the right-hand-side, node 5 is linked to 0 we do not get such an increase. Thus, it is more efficient to connect shallow clusters to deeper ones. In practice, for optimal efficiency, it suffices to connect smaller clusters to larger ones (wrt their number of nodes), and that what we are going to follow.

We next look at the implementation of `NetworkWithUndo` in more detail.

Since our nodes are the form 0, 1, …, *N*-1, for some *N* ≥ 0, we can represent lists such as the above implicitly, using an array list of length *N*: each node corresponds to a position of the array list, and to represent a pointer from node *i* to node *j* we simply store *j* in position *i*. For example, the network of Figure 1 will be represented by the array list:

None	None	1	4	0	None	8	8	None
0	1	2	3	4	5	6	7	8

An additional piece of data that we need to store, for efficiency reasons explained above, is the **size** of each root node. The size of each such node is simply the number of nodes that eventually point to it, including itself. This number is actually the size of the cluster that the node belongs to. For simplicity, if node *i* is a root and has size *k* then we shall store *-k* in position *i*. This is a bit of a hack, but it works because in all non-root nodes we store non-negative integers. Thus, the above network will be represented by the array list:

-3	-2	1	4	0	-1	8	8	-3
0	1	2	3	4	5	6	7	8

Figure 2: A network of nodes {0, 1, …, 8} *using an array list.*

The `add`, `root` and `merge` operations are all going to be simply operating on the underlying array list. For example, adding a new node changes the array list above to:

-3	-2	1	4	0	-1	8	8	-3	-1
0	1	2	3	4	5	6	7	8	9

On the other hand, finding the root of a node amounts to pointer chasing around the array list. In this process, we keep the nodes we visit (e.g. in a stack) and, once the root is found, make each of them point to the root, in order to apply path flattening. For example, calling `root(3)` on the network of Figure 2 should return 0 and change the array list to:

-3	-2	1	0	0	-1	8	8	-3
0	1	2	3	4	5	6	7	8

Merging is performed by connecting the roots of two clusters, with the smaller cluster made to point to the larger one. For simplicity `merge(i,j)` should only applied if *i* and *j* are root nodes – otherwise the function throws an error. Thus, applying `merge(0,5)` on the network of Figure 2 should first compare the sizes of the corresponding clusters: the size of the cluster rooted at 0 is 3 (we have stored -3 in position 0), while the cluster rooted at 5 has size 1 (we have stored -1). We therefore make 5 point to 0, and update the size of the (extended) cluster rooted at 0. The resulting array list is:

-4	-2	1	4	0	0	8	8	-3
0	1	2	3	4	5	6	7	8

On the other hand, if we call `merge(0,8)` on the network of Figure 2, we note that the sizes of these clusters are both 3. In this case, we simply make the first cluster (i.e. the one corresponding to the first argument of `merge`) point to the other one: that is, 0 is made to point to 8. The resulting array list then is:

8	-2	1	4	0	-1	8	8	-6
0	1	2	3	4	5	6	7	8

Finally, the `undo` function should simply rely on the `undo` function of the underlying `ArrayListWithUndo`. As each of the network operations (i.e. `add`, `root` and `merge`) involve a number of array list operations, undoing a network operation amounts to undoing a corresponding number of array list operations. Thus, we can simply use a stack and remember the number of array list operations for each network operation performed. In order to undo the last such operation we pop from the stack its number of array list operations, say *n*, and perform n `undo` operations on the underlying array list.

You are tasked with implementing a `NetworkWithUndo` class based on the representation of clusters described above. For underlying array list, you will use an `ArrayListWithUndo` in order to support reverting of operations. Here is the code for the constructor:

```
def __init__(self, N):
     self.inArray = ArrayListWithUndo()
     for _ in range(N): self.inArray.append(-1)
     self.undos = Stack()
     self.undos.push(N)
```

We have also provided you implementations for `getSize` (returns the number of nodes in the network), `toArray` and `str`. You should <u>not change these</u>.

Implement the following functions. The first three of them should also count the number of array list operations they perform and push that number onto `undos` at the end.

- `add(self)`: adds a new node to the network. If the cluster contains *N* elements then the new element added should be the integer *N*. Moreover, *N* should be in a cluster containing only *N*.
- `root(self, i)`: finds the root node of the cluster where *i* belongs to and returns it. The function should apply path flattening as described above.
- `merge(self, i, j)`: merges the clusters rooted at nodes *i* and *j*. The merging should depend on the size of the clusters, as explained above. If *i* or *j* is not a root node then the function should `assert(0)`. If *i* and *j* are the same root node then it should return without altering the network.
- `undo(self)`: reverts the last operation performed on the network. If there are no previous operations then the function should return without altering the network.

Question 3 – Gadget [30 marks]

This question asks you to build a data structure to perform and monitor the safe rebuilding of human networks. We call this data structure `Gadget`. A gadget object is initialised with an empty network and provides functions for adding new nodes to the network, connecting two nodes if they are not connected, and reverting the network to a previous state by going back a number of steps. It will use `NetworkWithUndo` and save the world!

You are asked to implement a class `Gadget` as follows. The nodes in a gadget are strings – you can think of these as IP addresses or unique node identifiers. We use a hashmap (i.e. a dictionary) to map these strings to unique integers, so that if there are *N* nodes in a gadget then our map assigns them the integers 0, 1, …, *N*-1. This is in order to be able to use a `NetworkWithUndo` data structure: the nodes of the gadget are going to correspond to the nodes of the network object, while connectivity between gadget nodes will coincide with connectivity of their corresponding network nodes.

Thus, two gadget nodes will be connected if, and only if, their corresponding network nodes are in the same cluster. We let a **subnet** be a set of connected gadget nodes. The **subsize** of a gadget is the number of subnets it contains.

The constructor for `Gadget` is as follows:

```
def __init__(self):
     self.inNetwork = NetworkWithUndo(0)
     self.subsize = 0
     self.nameMap = {}
     self.undos = Stack()
     self.helper = None  # you can edit this line
```

Note that an `undos` stack is used. Its entries will be triples of the form `(op, n, s)` with `op` being a string, `n` an integer and `s` a string (or `None`). The string `op` specifies how we need to change the gadget:

- if `op` is `"rem"` then we need to remove the string `s` from the `nameMap`, reduce the subsize by 1, and perform *n* `undo` steps on the internal network
- if `op` is `"brk"` then we need to break a connection, i.e. increase the subsize by 1 and perform *n* `undo` steps on the internal network
- if `op` is `"oth"` then we simply need to perform *n* `undo` steps on the internal network.

We have also provided you implementations for `getSize`, `isIn`, `toArray` and `str`. You should not change these.

Implement the following `Gadget` functions. Note that, in `add`, `subnets` and `connect`, the `undos` stack should also be updated by pushing a new triple.

- `add(self, name)`: adds `name` (which is a string) as a node in the gadget. If `name` is already in the gadget then the function returns without altering it. The function should ensure that `nameMap`, `size` and `inNetwork` are updated accordingly.
- `subnets(self)`: returns an array containing all subnets in the network. Each subnet is returned as an array of nodes. So the function returns an array of arrays of strings. Note this function may change the underlying network. You can use Python dictionaries to construct the array if needed.
- `connect(self, name1, name2)`: checks if nodes `name1` and `name2` are connected. If they are, it simply returns `True`. If not, it merges the corresponding clusters in the underlying network and returns `False`.
- `clean(self, name)`: reverts the gadget to how it was just before node `name` was added. Here you need to implement a mechanism that will allow you to find out how many steps before was `name` added to the gadget, and call `undo` accordingly. You can use the variable `helper` to help you implement this mechanism. The details are up to you, but finding the number of steps should be of constant time complexity, and the mechanism should not overhaul the other gadget functions nor affect their complexity. You can use Python dictionaries in `helper` if needed.
- `undo(self, n)`: reverts the last *n* operations performed on the gadget. If in doing so we are led to an empty gadget, we stop and return, leaving the gadget empty.

Essential Guidelines (see also page 1):

- **Code Specifications**

  Unless stated otherwise, **you are not allowed to use** built-in Python functions. Moreover, and unless specified otherwise, no built-in data structures can be used apart from arrays, tuples and strings. In particular, you cannot use built-in list operations for appending an element to a list. **You can use** substring/subarray-creating constructs like `A[lo:hi]`. You can use helper functions of your own.

  You can use data structures that we saw in the module, **but only for auxiliary purposes** and not for replacing the functions required by the asked data structures. Please ask if not sure.

  Your implementations **should be efficient and make essential use of the defined data structure(s).** For instance:

  - Solutions that find the root of a node by looking at Θ(*N*)-many nodes, where *N* is the number of nodes in a network, or do linear search in a cluster, will get few or no marks. Similarly for linear-time solutions to `merge` or `connect`.
  - As explained, your implementation of `root` should be using path flattening, while `merge` should use the sizes of the corresponding clusters.
  - Implementations of `undo` that store full copies of earlier versions of the data structure will get no marks.

  Your code is going to be automatically tested and marked. **In order to get any marks, you must submit a `solutions.ipynb` file and ensure that:**

  - your code can be imported without errors,
  - you do not change any of the functions already implemented,
  - you do not include any code you used for testing, nor any debugging messages.

- In general, `undo` operations restore a data structure to a previous state. Operations that are undone are removed from the timeline of operations and do not count for future undos. For example, suppose we have an empty array list and we append consecutively the numbers 1, 2, 3, …, 42. The resulting array list is `[1, 2, …, 42]`.

  - If we run `undo` once, this will revert the last `append` and give us `[1, 2, …, 41]`.
  - If we then perform another `undo`, that will revert the `append` of 41 and give us `[1, 2, …, 40]`, and so on.

  Note also that only operations that change the data structure should count towards future undos. E.g. reading from an array list (i.e. calling `get`) does not count.

- **Avoid Plagiarism**

  - This is a group assignment and you should **only collaborate with students in your group**. No help from AI is allowed (e.g. do not use chatGPT).
  - Showing your solutions to other groups is an examination offence.
  - You can use material from the web, so long as you **clearly reference your sources** in your report. However, what will get marked is your own contribution, so if you simply copy code from the web you will get few or no marks.