Assignment One – 15%

Algorithms and Data Structures – COMP3506/7505 – Semester 2, 2023

Due: 3pm on Friday September 1st (week 6)

Summary

The main objective of this assignment is to get your hands dirty with some simple data structures and algorithms to solve basic computational problems. These data structures will also come in handy for your second assignment, so you should take your time to think about your implementations and try to make them as efficient as possible.

1 Getting Started

Before we get into the nitty gritty, we will discuss the skeleton codebase that will form the basis of your implementations, and provide some rules that must be followed when implementing your solutions.

1.1 Codebase

The codebase contains a number of data structures stubs that you should implement, as well as some scripts that allow your code to be tested. Figure 1 shows a snapshot of the project directory tree with the different files categorized.

```
├── structures
│   ├── m_extensible_list.py
│   ├── m_single_linked_list.py
│   └── m_stack.py
├── data
│   ├── large.refgrid
│   ├── medium.refgrid
│   ├── small.refgrid
│   └── tiny.refgrid
├── execute_refgrid.py
└── test_structures.py
```

Figure 1 The directory tree organized by data structures (inside the `structures` directory), test data (inside the `data` directory), and the two executable programs (in the root directory).

1.2 Implementation Rules

The following list outlines some important information regarding the skeleton code, and your implementation:

- The code is written in Python and, in particular, should be executed with Python 3 or higher. The EAIT student server, `moss`, has Python 3.6.8 installed by default. We recommend using `moss` for the development and testing of your assignment, but you can use your own system if you wish.
- You are not allowed to use built-in methods or data structures – this is an algorithms and data structures course, after all. If you want to use a `dict` (aka `{}`), you will need to implement that yourself. Lists can be used as "dumb arrays" by manually allocating space like

`myArray = [None] * 10` but you may not use built-ins like `append, clear, count, copy, extend, index, insert, pop, remove, reverse, sort, min, max,` and so on. List slicing is also prohibited, as are functions like `sorted, len, reversed, zip`. Be sensible – if you need the functionality provided by these methods, you may implement them yourself. Similarly, don't use any other collections or structures such as `set` or `tuple` (for example; `mytup = ("abc", 123)`).

- ▷ You are not allowed to use libraries such as `numpy`, `pandas`, `scipy`, `collections`, and so on.
- ▷ Exceptions: The only additional libraries you can use are `random` and `math`. You are allowed to use `range` and `enumerate` to handle looping.

If you have any doubts, please direct your questions to the Ed discussion.

2 Task 1: Data Structures (7 marks for 3506, 9 marks for 7505)

We'll start off by implementing some fundamental (but very important) data structures. As you will see, both linked lists (Task 1.1) and extensible lists (Task 1.2) can be used as the basis of a variety of other structures including stacks (Task 1.3).

Task 1.1: Fix the Singly Linked List (1 mark)

Your first task is to examine the `m_single_linked_list.py` file to see how a canonical linked list can be implemented. There are two classes inside this file: the SingleNode is the structure that holds both the satellite data object, as well as a pointer (aka reference) to the location of the next node; the SingleLinkedList is the structure that keeps track of the head of the list, the size of the list (the number of nodes in the chain), and also the implementation of various functionality.

You should read each method and the constructors to understand the code, and find the *two* bugs in the implementation. You should fix those bugs once you find them, but do not edit anything else. There is a sample test suite that you can run via `python3 test_structures.py --linked-list`.[1] You are free to modify this test method as much as you want — this is a good way to locate and fix the bugs! Note that the autograder will also give you some basic test results, so you can check your solution there (by submitting your work) as well.

Task 1.2: Implement an Extensible List (2 marks)

Unlike the linked lists discussed above, which can store nodes at any abitrary location in memory, we often prefer to have data items stored contiguously (consecutively in memory), allowing us to access an element x at some index i in constant time. One such way to achieve this is through the use of an extensible list (aka a dynamic array).

The file `m_extensible_list.py` contains a skeleton object for you to implement. You should store your data in self._data, and you can add any other member variables to your ExtensibleList object. Each function that needs to be supported is provided as a stub. Your implementations should be efficient and correct. You will need to implement your own tests and run them using `python3 test_structures.py --ex-list`.

[1] Depending on your version of Python or how it was installed, you might need to substitute the `python3` call with something else.

Task 1.3: Implement a Stack or two... (4 marks)

Recall that a stack provides *last-in-first-out* (LIFO)[2] ordering of the data, with constant-time access to the most recently pushed element.

Our last objective in this part is to implement two different stacks (see `m_stack.py`). The first is called an EStack and will inherit our ExtensibleList as the underlying data storage mechanism. The second is called an LStack and will inherit the SinglyLinkedList as the storage mechanism.

Since we have two stack versions, you will need to decide which methods to override (if any) from the base classes, and implement the `push(x)`, `pop()`, `peek()`, and `empty()` functionality in the stack classes themselves. It may also be a good idea to override any functions from the base classes that should *not* be called by your stack (we wouldn't want to call `find_and_remove(x)` on a stack, for example) to avoid any accidents.

Implement test cases and run them using `python3 test_structures.py --ex-stack` and `python3 test_structures.py --linked-stack`.

Task 1.4: Benchmark Stacks (2 marks for COMP7505)
→ Optional and Unmarked for COMP3506

Our final objective in this part is to benchmark your stacks. In particular, you should look at the `benchmark_stacks()` function inside `test_structures.py` to see how a basic microbenchmark can be implemented. In this case, we simply generate a fixed number of random integers, push them all onto the stack, and pop them all out again, measuring the time taken. The benchmark can be executed via `python3 test_structures.py --bench-stacks 1000` where `1000` represents the number of random integers to generate in the benchmark.

Consider the expected rate of growth of runtime when increasing n – the expected complexity of both `push(x)` and `pop()` – what do you expect to see happen as you increase n in the benchmark?

Submission

You should run the benchmark with 10 different values of n (your choice, but try to use values that provide timings greater than 0.1 seconds at a minimum) and record the output in a file called `stackbench.txt`. You should use the specified format as we will be using scripts to extract the information (space or tab delimiting is OK). Please see Figure for more details.

If you are a COMP3506 student and you wish to submit your benchmark, please use the filename `stackbench-3506.txt`.

```
1 [n val] [EStack time] [LStack time]
2 [n val] [EStack time] [LStack time]
3 [n val] [EStack time] [LStack time]
4 [n val] [EStack time] [LStack time]
5 [n val] [EStack time] [LStack time]
...
```

```
1 1000     3.5   4.7
2 10000    10.1  11.6
3 100000   22.8  31.3
4 1000000  32.2  40.6
5 10000000 50.7  101.5
```

Figure 2 The format of the `stackbench.txt` file (left) and an example output (right). Note: The values are examples only, and only the first 5 of 10 lines are displayed.

[2] This is the same as saying *first-in-last-out*, FILO.

3 Task 2: Problem Solving with Data Structures (8 marks)

You are a bioinformatician working at Frankenstein Labs™, a world leader in futuristic genome editing technology that has promising applications for anti-ageing, cloning, and superhuman intelligence. As a bioinformatician, you often work with DNA data. DNA data is represented as a string S of length $|S|$ over an alphabet $\Sigma = \{\texttt{a}, \texttt{c}, \texttt{g}, \texttt{t}\}$. Each character represents a different base (`a` is *adenine*, `c` is *cytosine*, `g` is *guanine*, and `t` is *thymine*). For example, a sequence might look like $S = \texttt{gaatacgg}$ where $|S| = 8$.

One of the latest innovations from Frankenstein Labs is known as the DNA-RefGrid which, among other things, can be used to fold multiple strands of DNA together in interesting (and sometimes frightening!) ways. The DNA-RefGrid is a simple $n \times |S|$ matrix containing n different strands of DNA, each of the same initial length $|S|$, and can be represented as a text file. The following example shows a DNA-RefGrid file containing $n = 4$ strings with $|S| = 12$:

```
gtacggttaacc
gctctggactct
atggtgtcctca
ctgctgccccta
```

Figure 3 The format of the `.refgrid` file type.

Getting Started

We have provided you with an executable program called `execute_refgrid.py` that contains a RefGrid class. You need to implement the functionality inside that class to solve the following tasks. To get you started, we provide code that can read a `.refgrid` file into a singly linked list (`self.linkedlist`) or an extensible list (`self.extlist`) once correctly implemented. You are free to (and should) utilize your data structures from Task 1 within the implementation of your RefGrid class where appropriate.

Task 2.1: Sequence Reversal (2 marks)

Given the RefGrid loaded into the `self.linkedlist` structure, your first task is to implement a method that reverses the kth sequence of the RefGrid in *linear time*. This functionality can be tested using the `--reverse-k` argument to the `execute_refgrid.py` program. For example, assuming the example file in Figure 3 is stored in `example.refgrid`, then running `python3 execute_refgrid.py --refgrid my.refgrid --reverse-k 2` should output:

```
gtacggttaacc
gctctggactct
actcctgtggta
ctgctgccccta
```

Note the value of k is zero-indexed, and that you should validate the value of k and handle out-of-bounds cases elegantly (just ignore the reversal and store the file as-is). This method should be *destructive* (it should modify the stored data in place, not generate a new copy of the data). Hint: One of your stacks might come in handy here.

Task 2.2: Grid-Based Cutting and Splicing (3 marks)

Your next task is to implement *cut-and-splice* routine. The idea is as follows. Given a *pattern* P and a target sequence T, replace *all occurrences* of P with T. To simplify this task, you may assume that both P and T make use of each base *just once* (that is, you can not supply a pattern or target like `gg` or `ttccc`) and that both P and T are not empty.

For example, given an initial string $S =$ `gtcaggtcccaccgcc`, $P =$ `ca` and $T =$ `cgt`, the cut-and-splice would look like:

```
gtcaggtcccaccgcc      (before)
gtcgtggtcccgtccgcc    (after)
```

Although we are not operating on a single string S, but rather a RefGrid with n strings, the cut-and-splice operation works the same on a per-sequence basis. Thus, you need to take care not to allow pattern matches across sequences, only within each sequence. Again, this method should be *destructive* and modify the stored data in place, instead of generating a new copy of the data. You should use the linked list to store/modify the original RefGrid, and use the extensible list to store the length of each row; implement the `stringify_spliced_linkedlist` function to handle printing the spliced data. Test via: `python3 execute_refgrid.py --refgrid my.refgrid --cut-and-splice pattern:target`

Task 2.3: Cloning Viability (3 marks)

The holy grail of Frankenstein Labs is their cloning technology. However, in order to determine whether a series of DNA sequences (as provided in a RefGrid) are viable for cloning, an algorithm needs to determine if there is an L-path through the sequences. An L-path is simply described as a path through the RefGrid from the top-left to the bottom-right that matches the following conditions: (1) The path consists of *entirely the same base*; and (2) The path can only ever go downwards or to the right.

```
gtaca        atgca
gggaa        attgt
ccggg        tcgca
ctgag        ggtca
```

Figure 4 The sequences in the left RefGrid are viable for cloning, but those in the right are not.

You need to implement an algorithm that returns `True` if an L-path exists, or `False` otherwise. Luckily, your colleague and renowned algorithms legend Barry Malloc has been thinking about how to solve this problem and has given you his notes. Use Barry's notes to help you implement the `left` and `below` helper functions, and to then implement his solution in Algorithm 1. Test via: `python3 execute_refgrid.py --refgrid my.refgrid --check-clone` To make things a little bit easier, you may assume that the input RefGrid has not been subject to any cutting and splicing (so all rows are of the same length).

Barry Malloc to the Rescue

- ▷ We will read the input data, in order, into one single extensible list (A in Algorithm 1).
- ▷ We will initialize a stack to keep track of our traversal, and a linked list to keep track of indexes we have visited.

- We need to write two helper functions that allow us to map from a given array index i to a new index j, see Figure 5:
  - The first will map index i to the index of the element to the right; call it `right(i)`. It will return 0 if there is nothing to our right.
  - The second will map index i to the index of the element below; call it `below(i)`. It will return 0 if there is nothing below.
- We can now solve our problem using a *stack of array indexes* (Algorithm 1).

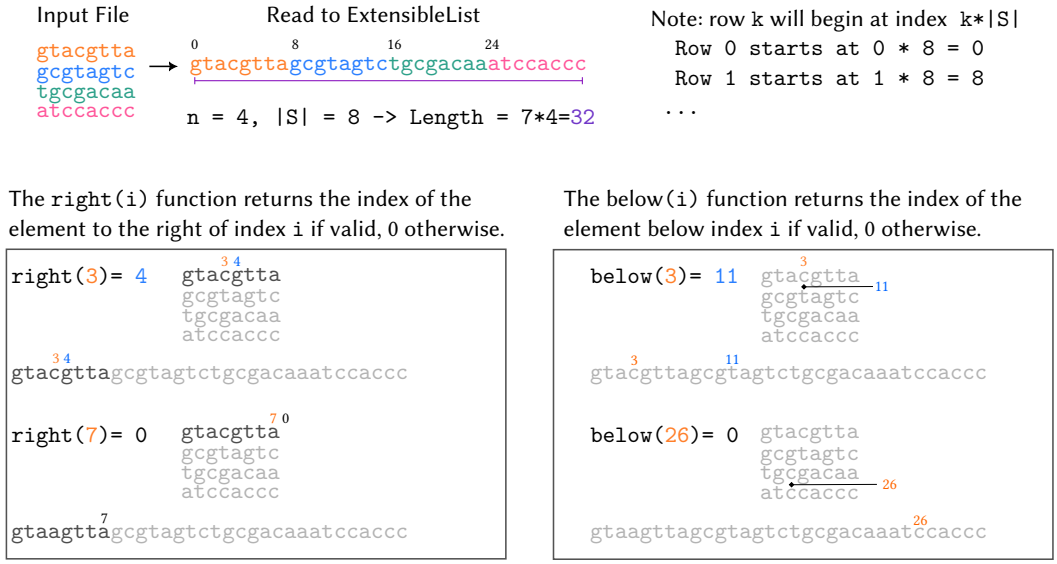

Figure 5 Barry's diagrams showing how to map the 2-dimensional RefGrid into a single list, and then how the helper functions work on that single list.

Algorithm 1 A stack-based algorithm to solve the cloning viability problem. Assumes you have read the input into an ExtensibleList called A that can be accessed throughout the computation.

```
1:  procedure IsViable(A)
2:      myStack ← ∅                                          ▷ Make an empty stack
3:      visited ← ∅          ▷ Make an empty linked list to keep track of visited indexes
4:      cur ← 0                                              ▷ Start index is the top-left
5:      end ← A.length − 1                                   ▷ End index is the bottom-right
6:      base ← A[cur]                                        ▷ This is the target base character
7:      Push(myStack, cur)
8:      Insert(visited, cur)
9:      while NotEmpty(myStack) and cur != end do
10:         cur = Pop(mystack)                               ▷ Get the next candidate cell (index)
11:         r_idx = Right(A, cur)                            ▷ Compute the right cell index
12:         b_idx = Below(A, cur)                            ▷ Compute the below cell index
13:         ▷ If the element to our right matches the target base and has not been visited
14:         if r_idx != 0 and A[r_idx] == base and r_idx is not in visited then
15:             Push(myStack, r_idx)                         ▷ This is a candidate path
16:             Insert(visited, r_idx)                       ▷ Mark as visited
17:         end if
18:         ▷ If the element below matches the target base and has not been visited
19:         if b_idx != 0 and A[b_idx] == base and b_idx is not in visited then
20:             Push(myStack, b_idx)                         ▷ This is a candidate path
21:             Insert(visited, b_idx)                       ▷ Mark as visited
22:         end if
23:     end while
24:     if cur == end then                                   ▷ We found an L-Path!
25:         return True
26:     end if
27:     return False          ▷ We have exhausted our options and there is no L-Path
28: end procedure
```

4 Assessment

This section briefly describes how your assignment will be assessed.

4.1 Mark Allocation

Marks will be provided based on an extensive (hidden) set of unit tests. These tests will do their best to break your data structure in terms of time and/or correctness, so you need to pay careful attention to the efficiency and the validity of your code. Each test passed will carry some weight, and your autograder score will be computed based on the outcome of the test suite. If you did not rigorously test your programs/code, you should go back and do so!

The marks (percentages) provided in each task above are indicative of the total score available for each part, but marks may be taken off for poor coding style including lack of commenting, inefficient solutions, and incorrect solutions. While the overall grade/score will be calculated mathematically, an indicative rubric is provided as follows:

- **Excellent:** Passes at least 90% of test cases, failing only sophisticated or tricky tests; well structured and commented code; appropriate design choices; appropriate application of data structures for solving Task 2.
- **Good:** Passes at least 80% of test cases, failing one or two simple tests; well structured and commented code; good design choices with some minor improvements possible; good application of data structures for solving Task 2 with some minor improvements possible.
- **Satisfactory:** Passes at least 70% of test cases; code is reasonably well structured with some comments; most design choices are reasonable but significant room for improvement; reasonable application of data structures for solving Task 2, but significant improvements possible.
- **Poor:** Passes less than 70% of test cases; code is difficult to read, not well structured, or lacks comments; design choices do not demonstrate a sound understanding of the desired functionality; little or no suitable application of data structures towards solving Task 2.

4.2 Plagiarism and Generative AI

It's 2023; we know all about generative AI technology like ChatGPT or Github Copilot. The course ECP actually has a statement about the use of such technology, repeated here:

> *It is the position of UQ that the use of AI outputs without attribution, and contrary to any direction by teaching staff, is a form of plagiarism and constitutes academic misconduct. The assessment tasks evaluate students' abilities, skills and knowledge without the aid of Generative Artificial Intelligence (AI). Students are advised that the use of AI technologies to develop responses without attribution is strictly prohibited and may constitute student misconduct under the Student Code of Conduct.*

If you want to actually learn something in this course, our recommendation is that you avoid using such tools: You need to think about what you are doing, and why, in order to put the theory (what we talk about in the lectures and tutorials) into practical knowledge that you can use, and this is often what makes things "click" when learning. Mindlessly lifting code from an AI engine won't teach you how to solve algorithms problems.

If you are still tempted, note that we will be running your assignments through sophisticated software similarity checking systems against a number of samples including including your classmates and our own solutions (including 30 AI generated). Note also that the exam

may contain questions or scenarios derived from those presented in the assignment work, so cheating could weaken your chances of successfully passing the exam.

As part of your submission, you should create a file called `statement.txt` containing the following text:

> *I, [firstname lastname (studentid)], hereby declare that this is my own original work, and that no part of this assignment has been copied from any other source or person except where due acknowledgement is made; no part of the work has been previously submitted for assessment in this or any other institution except where explicitly acknowledged; I/We have read PPL 3.60.04, UQ's Student Integrity and Misconduct Policy and understand its implications; and I have not used Generative AI to assist me with my assignment.*

In the same file, you should then provide attribution to any sources or tools used to help you with your assignment, including any prompts provided to AI tooling.

5 Submission

You need to submit your solution to *Gradescope* under the *Assignment 1: Autograder* link in your dashboard. Once you submit your solution, a series of tests will be conducted and the results will be provided to you. However, the assessment will also include a number of additional *hidden* tests, so you should make sure you test your solutions extensively. You may resubmit as often as you like before the deadline.

Structure

The easiest way to submit your solution is to submit a `.zip` file. The autograder expects a specific directory structure for your solution, and the tests will fail if you do not use this structure. In particular, you should use the same structure as the skeleton codebase that was provided. Your root directory should contain the `execute_refgrid.py` file, and a subdirectory `structures/` with the other data structure files (such as `m_singly_linked_list.py`). Note that you do not need to submit the `data/` directory but you can if you wish. You should also have the `statement.txt` and `stackbench.txt` (or `stackbench-3506.txt` if you are a COMP3506 student participating in Task 1.4). Submissions without the `statement.txt` will be given zero marks.

6 Resources

We provide a number of useful git and/or unix resources to help you get started. Please go onto the Blackboard LMS and see the Learning Resources > Resources directory for more information.

7 Changelog

- V1.1: Updated an erroneous example call to `refgrid.py` throughout the document (should have been `execute_refgrid.py`). Also added **Resources** section.