1 Objectives

There are a number of key objectives for this assignment:

- Understand how a real-world problem can be implemented by different data structures and/or algorithms.
- Evaluate and contrast the performance of the data structures and/or algorithms with respect to different usage scenarios and input data.

In this assignment, we focus on implementing a *spreadsheet*.

2 Background

Spreadsheets are an essential application, for anyone needing to record and share tabular data. While not the most exciting of applications, they are the workhorse of data analysis and professional services and provide an ideal application setting for studying data structures and their efficiencies.

In this assignment, we will focus on implementing data structures needed for developing a spreadsheet application. We will develop several data structures and compare their performances. These data structures are arrays (that is, Python list), linked lists and Compressed Sparse Rows (CSR), which will be further described below. Please read them carefully. Latest updates and answers for questions regarding this assignment will be posted on the Discussion Forums (Ed Forums). Please check the post frequently for important updates.

Array-Based Spreadsheet

Python's built-in 'list' is equivalent to 'array' in other languages. In fact, it is a dynamic array in the sense that its resizing operation (when more elements are inserted into the array than the original size) is managed automatically by Python. You can initialize an empty array in Python, add elements at the end of the array, remove the first instant of a given value by typing the following commands (e.g., on Python's IDLE Shell).

```
>>> array = []
>>> array.append(5)
>>> array.append(10)
>>> array.append(5)
>>> array
[5, 10, 5]
>>> array.remove(5)
>>> array
[10, 5]
```

In the array-based spreadsheet implementation, we use the Python list (a data structure) as the basis to implement common operations for a spreadsheet. A spreadsheet is a 2D structure of cells, and each cell can hold different data types, e.g., *general*, *number* etc. For this assignment, we will concentrate on cells only holding floats. Hence to implement a spreadsheet, we will need to implement a 2D array.

The 2D array is indexed by the tuple (row,column) in the spreadsheet. An example is if we use numbers to index rows, and letters for columns, we might have (10,8) to specify the cell of the 11th row, 9th column (we assume indices start at 0).

Doubly Linked-List-Based Spreadsheet

A linked list is precisely what it is called: a list of nodes linked together by references. In a doubly linked list, each *node* consists of a data item, e.g., a string or a number, a reference that holds the memory location of the next node in the list (the reference in the last node is set to None) and a memory reference to the previous node in the list (the reference to the first node is set to None). Each linked list has a `head`, which is the reference holding memory location of the first node in the list and a `tail` reference. Once we know the `head` or `tail` of the list, we can access all nodes sequentially by going from one node to the next using references until reaching the last node.

In the linked-list-based implementation of a spreadsheet, we use an **unsorted** doubly linked list. As we need to implement a 2D structure, this needs to be a linked list of linked list. You can use the implementation of the linked list in the workshop as a reference for your implementation. Each node stores as data the cell contents (a float), a reference to the next node and a reference to the previous node.

CSR (Compressed Sparse Row) based Spreadsheet

CSR (Compressed Sparse Row) is an array based representaion used to store spreadsheets using less space. It consists of *three* arrays, ColA, SumA and ValA.

Consider a spreadsheet consisting of r rows, c columns and NZV number of cells with values in them.

- *ColA* is of length NZV and for each row, denotes which columns of cells with values in them.
- *ValA* is of length NZV and for each row, denotes the values of each cell that has values in them. *ColA* and *ValA* should have the same lengths.
- *SumA* is of length $r + 1$ and stores the cumulative sum up to the ith row. E.g., $sumA[0]$ is the cumulative sum up to 0th row (it should always equal 0), $sumA[1]$ is the cumulative sum up to the 1st row.

As an example, consider we have the following spreadsheet:

$$\begin{bmatrix} - & - & 3 \\ - & 4 & - \\ 6 & - & -2 \end{bmatrix}$$

Then we would have the following arrays:

- $ColA = [2, 1, 0, 2]$, as in row 0, column 2 has a value, for row 1, column 1, and for row 2, columns 0 and 2.
- $ValA = [3, 4, 6, -2]$, these are the values that correspond to each cell that has a value. Note it corresponds with the $ColA$ array.
- $SumA = [0, 3, 7, 11]$, as the cumulative sum up to row 0 (not including row 0) is 0, up to row 1 is 3, up to row 2 is 7 (3+4) and up to row 3, or the whole spreadsheet, is 11 (7 + 6 - 2).

From these three arrays, we are able to reconstruct the 2D spreadsheet. For example, we know from SumA[1] that row 0 has a total of 3. We then know from ValA[0] that row 0 only contains a value of 3, which is located in column 2 (from ColA[0]). The location of the other values in the spreadhseet can be inferred from the three arrays.

3 Tasks

The assignment is broken up into a number of tasks, to help you progressively complete the assignment.

3.1 Task A: Implement the Spreadsheet and Its Operations Using Array, Doubly Linked List, and CSR data structure (9 marks)

In this task, you will implement a spreadsheet that allows adding and inserting rows and columns, updating values, finding which cells containts a certain value and enumerating cells with values, using three different data structures: Array (Python's list), Linked List, and CSR. Each implementation should support the following operations:

- Build a spreadsheet from a list of tuples of (row, column, value).
- Append a new row to the bottom of the spreadsheet.
- Append a new column to the right end of the spreadsheet.
- Insert a new row (can be between any existing rows) into the spreadsheet.
- Insert a new column (can be between any existing rows) into the spreadsheet.
- Update a cell's value (which could include adding a value to a cell that didn't have any values before, or removing a value by assigning to None).
- Return the number of rows in the spreadsheet.
- Return the number of columns in the spreadsheet.
- Find a value in the spreadsheet and returns a list of cells that has that value. The returned list can be empty.
- Enumerate all cells that have values in them.

3.1.1 Implementation Details

Array-Based Spreadsheet. In this subtask, you will implement the spreadsheet using Python's lists. In this implementation, all standard operations on lists are allowed. Other data structures should NOT be used directly in the *main* operations of the array-based spreadsheet. See the Background Section for more details and an example.
Dual Linked-List-Based Spreadsheet. In this subtask, you will implement the spreadsheet by using a doubly linked list. Other data structures should NOT be used directly in the *main* operations of the linkedlist-based spreadsheet (but Python's list can be used to store intermediate data or the input/output). See the Background Section for more details and an example.
CSR-Based Spreadsheet. In this subtask, you will implement the spreadsheet using the CSR data structure. See the Background Section for more details and an example.

3.1.2 Operations Details

Operations to perform on the implementations are specified in the command file. They are in the following format:

```
<operation> [arguments]
```

where `operation` is one of {AR, AC, IR, IC, U, R, C, F, E} and `arguments` is for optional arguments of some of the operations. The operations take the following form:

- `AR` – appends a row to the end of the spreadsheet.
- `AC` – appends a column to the end of the spreadsheet.
- `IR r` – inserts a new row, after/below current row r. If inserting a row to the start of the spreadsheet, i.e., before current row 0, use `IR -1`. If r is less than -1 or greater or equal to the current number of rows in the spreadsheet, return false.
- `IC c` – inserts a new column, after/to the right of current column c. If inserting a column to the start of the spreadsheet, i.e., before current column 0, use `IC -1`. If c is less than -1 or greater than or equal to the current number of columns in the spreadsheet, return false.
- `U r c v` – updates cell r,c (row r, column c) with value v. If cell doesn't exist, should return false.
- `R` – returns the number of rows in the spreadsheet.
- `C` – returns the number of columns in the spreadsheet.
- `F v` – find a value v in the spreadsheet and returns a list of cells that has that value. The returned list can be empty.
- `E` – enumerate all cells that has a value in it. The returned list can be empty if all cells in the spreadsheet do not have a value.

As an example of the operations, consider the input and output from the provided testing files, e.g., `sampleData.txt`, `sampleCommands.in`, and the expected output, `sample.exp` (Table 1).

Note, you do NOT have to do the input and output reading yourself. The provided Python files will handle the necessary input and output formats. Your task is to implement the missing methods in the provided classes.

sampleData.txt	sampleCommands.in (commands)	sample.exp (expected output)
9 9 2.0	AR	Call to appendRow() returned success.
2 5 7	AC	Call to appendCol() returned success.
3 1 6	F 6	Printing output of find(6.0): (3,1)
8 5 -6.7	F -6.0	Printing output of find(-6.0):
	F -6.7	Printing output of find(-6.7): (8,5)
	R	Number of rows = 11
	C	Number of columns = 11
	U 2 5 -1	Call to update(2,5,-1.0) returned success.
	U 10 10 1	Call to update(10,10,1.0) returned success.
	U 11 11 2.5	Call to update(11,11,2.5) returned failure.
	E	Printing output of entries(): (2,5,-1.00) | (3,1,6.00) | (8,5,-6.70) | (9,9,2.00) | (10,10,1.00)
	IR 1	Call to insertRow(1) returned success.
	IC 4	Call to insertCol(4) returned success.
	IR -2	Call to insertRow(-2) returned failure.
	R	Number of rows = 12
	C	Number of columns = 12
	E	Printing output of entries(): (3,6,-1.00) | (4,1,6.00) | (9,6,-6.70) | (10,10,2.00) | (11,11,1.00)
	U 2 5 -2	Call to update(2,5,-2.0) returned success.
	E	Printing output of entries(): (2,5,-2.00) | (3,6,-1.00) | (4,1,6.00) | (9,6,-6.70) | (10,10,2.00) | (11,11,1.00)

Table 1: The file `sampleData.txt` provides the cells with values, while `sampleCommands.in` and `sample.exp` have the list of input commands and expected output. Consider the lines in data.txt is used to initialise the spreadsheet, then each command in the commands.in column are feed into the program one at a time, then the expected output from executing each command is in the correspond line in the output.exp column.

3.1.3 Testing Framework

We provide Python skeleton codes (see Table 2) to help you get started and automate the correctness testing. You may add your own Python modules to your final submission, but please ensure that they work with the supplied modules and the Python test script.

Debugging. To run the code, from the directory where `spreadsheetFilebased.py` is, execute (use 'python3' on Linux, 'python' on Pycharm):

```
> python3 spreadsheetFilebased.py <approach> <data filename> <command filename>
                                                              <output filename>
```

where

- `approach` is one of the following: `array`, `linkedlist`, `csr`,
- `data filename` is the name of the file containing the initial set of row, column and values of cells in the spreadsheet,
- `command filename` is the name of the file with the commands/operations,
- `output filename` is where to store the output of program.

file	description
`spreadsheetFilebased.py`	Code that reads in operation commands from file then executes those on the specified data structure. For **debugging** your code. DO NOT MODIFY.
`spreadsheet/cell.py`	Class representing a cell in the spreadsheet. DO NOT MODIFY.
`spreadsheet/baseSpreadsheet.py`	The base class for the spreadsheet implementation. DO NOT MODIFY.
`spreadsheet/arraySpreadsheet.py`	Skeleton code that implements an array-based spreadsheet. COMPLETE all the methods in the class.
`spreadsheet/linkedlistSpreadsheet.py`	Skeleton code that implements a doubly linked-list-based spreadsheet. COMPLETE all the methods in the class.
`spreadsheet/csrSpreadsheet.py`	Skeleton code that implements a trie-based spreadsheet. COMPLETE all the methods in the class.

Table 2: Table of Python files. The file `spreadsheetFilebased.py` is the main module and should NOT be changed.

For example, to run the test with csr, type (in Linux, use 'python3', on Pycharm's terminal, use 'python'):

```
> python3 spreadsheetFilebased.py csr sampleData.txt sampleCommands.in sample.out
```

Then compare `sample.out` with the provided `sample.exp`. In Linux, we can use the `diff` command:

```
> diff sample.out sample.exp
```

If nothing is returned then the test is successful. If something is returned after running `diff` then the two files are not the same and you will need to fix your implementation.

Automark script. We will use another Python script to automatically run your code through a number of test cases. These tests are fed into the script which then calls your implementations. The outputs resulting from the operations are stored, as well as error messages. The outputs are then compared with the expected output. To mark your implementation, we will use the same Python script and a set of **different** input/expected files that are in the same format as the provided example. **To avoid unexpected failures, please do not change the** `spreadsheedFilebased.py`. If you wish to use the script for your timing evaluation, make a copy and use the unaltered script to test the correctness of your implementations, and modify the copy for your timing evaluation.

3.1.4 Notes

- If you correctly implement the "To be implemented" parts, you in fact do not need to do anything else to get the correct output formatting because `spreadsheetFilebased.py` will handle this.
- We will run the automated test script on your implementation on the university's core teaching servers, e.g., `titan.csit.rmit.edu.au`, `jupiter.csit.rmit.edu.au`, `saturn.csit.rmit.edu.au`. If you write codes on your own computer, **make sure they run without errors/warnings on these servers before submission**. If your codes do not run on the core teaching servers, we unfortunately won't have the resources to debug each one and cannot award marks for testing.

- Please **avoid including non-standard Python modules not available on the servers**, as that will cause the test script to fail on your submission.
- All submissions should run with no warnings on **Python 3.6.8**, which is the default version on the core teaching servers.

3.2 Task B: Evaluate your Data Structures for Different Operations (11 marks)

In this second task, you will evaluate your implementations in terms of their time complexities for different operations. You will perform the empirical analysis and report the process and the outcome, and provide comparisons, comments, interpretations and explanations of the outcome, and your overall recommendations. The report should be no more than **5 pages**, in font size 12 and A4 pages (210×297 mm or 8.3×11.7 inches). See the assessment rubric (Appendix A) for the criteria we are seeking in the report.

Data Generation and Experiment Setup

You'll need to generate data to experiment. You could take real spreadsheets and convert them to csv, then into the necessary input format. Another option is to write a separate program to generate datasets. Either way, in the report you should explain in detail how the datasets are generated and why they support a robust empirical analysis.

We suggest you to use datasets of various sizes (at least six sizes) ranging from small (e.g., 100, 500), medium (e.g., 1000, 5000), to large (10000, 50000). You may find these numbers too be too small (if your computer hardware is fast) or too large (if your computer hardware is older and slower). Use appropriate ranges to be able to see differences across the data structures. You should explain how the spreadsheets to be tested are generated in detail as well.

To summarise, data generation and experiments have to be done in a way that guarantees reliable analysis and avoids bias caused by special datasets or special input parameters chosen for evaluated operations, and must be reported in detail.