SIMPLE SPREADSHEET PROGRAM

COMP3600 Final Project – Final Deliverables

Lucas Hempton

u6942724

1. Overview

This report provides algorithmic analysis for my simple spreadsheet program.

Spreadsheets are useful for organising, analysing and storing tabulated data. The spreadsheet will be a non-infinitely extending two-dimensional array of cells that can only contain integers. The user is able to read, modify, save and load files. They also have the ability to search for and sort elements within the spreadsheet. The user of my program has access (but is not limited) to the use of these capabilities.

1.1 Programming Language & Operating System

My project is written in C++. I have worked on it from my Mac OS computer. I have also ensured it also works on the ANU lab Linux machines.

1.2 Assumptions

Basic assumptions underpin the program. They define limitations such that it can be run on almost all modern home computers.

- Each cell can only contain an integer data type.
- The program will be capable of running on a computer with <= 2GB of RAM.
- Maximum number of columns: 2,500.
- Maximum number of rows: 250,000.

2. List of Functionalities

The following functionalities have been implemented in full for marking.

- B6. Red-black trees
- B7. Hashing
- B8. Dynamic Programming

Please view the following subheadings (3.1, 4.1 and 5.1) for descriptions of their roles in the functionality of my application.

3. Red-Black Trees

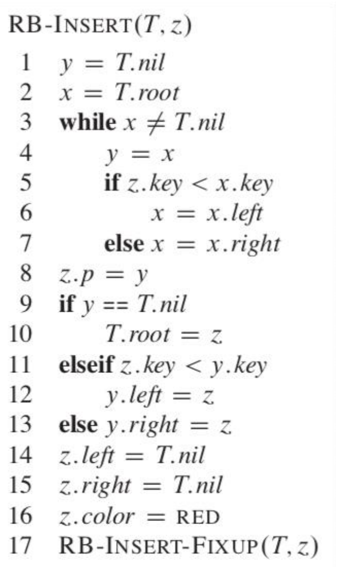
3.1 A short description of their role and why they are suitable for the purpose of my application

As red-black trees are very efficient self-balancing binary trees (with a height of O(log n)), I believe they are efficient for use in my program. Moreover, Red-Black trees have strong support for both insertion and deletion, as opposed to AVL trees for example which have higher computational costs for deletion. This is useful in a spreadsheet as they often require both large-scale deletion and insertion of items. Moreover, an RB Tree can be flattened into an array with a defined rule and doing this is an efficient solution to store data.

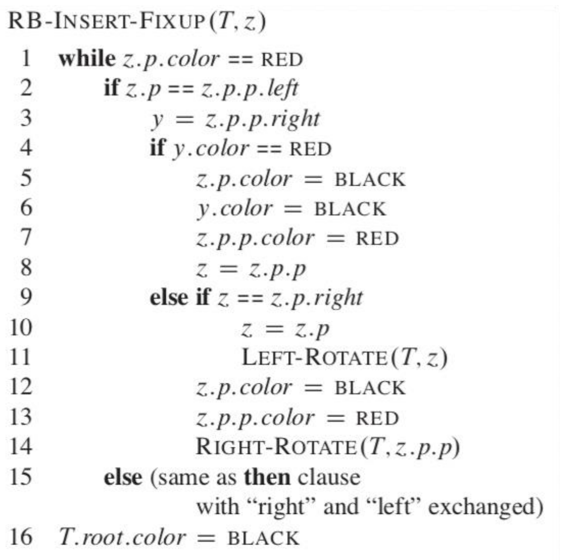
The red-black tree I have implemented in my program works with a node constructor and `RBTree` object. Functions that are implemented are insertion, deletion, search, and rotation (left and right about a node).

3.2 Theoretical Time Complexity Analysis

Unfortunately, due to time constraints, I was unable to fully implement the RB Tree to search and store values in my spreadsheet. However, I did successfully implement an RBTree object that has functions including `insert(int key)` and `insertFixup(NodePtr k)` defined on it. These are what my report will analyse.



```
RB-INSERT($T, z$)
 1  $y = T.nil$
 2  $x = T.root$
 3  while $x \neq T.nil$
 4      $y = x$
 5      if $z.key < x.key$
 6          $x = x.left$
 7      else $x = x.right$
 8  $z.p = y$
 9  if $y == T.nil$
10      $T.root = z$
11  elseif $z.key < y.key$
12      $y.left = z$
13  else $y.right = z$
14  $z.left = T.nil$
15  $z.right = T.nil$
16  $z.color$ = RED
17  RB-INSERT-FIXUP($T, z$)
```



```
RB-INSERT-FIXUP($T, z$)
 1  while $z.p.color$ == RED
 2      if $z.p == z.p.p.left$
 3          $y = z.p.p.right$
 4          if $y.color$ == RED
 5              $z.p.color$ = BLACK
 6              $y.color$ = BLACK
 7              $z.p.p.color$ = RED
 8              $z = z.p.p$
 9          else if $z == z.p.right$
10                  $z = z.p$
11                  LEFT-ROTATE($T, z$)
12              $z.p.color$ = BLACK
13              $z.p.p.color$ = RED
14              RIGHT-ROTATE($T, z.p.p$)
15      else (same as then clause
                with “right” and “left” exchanged)
16  $T.root.color$ = BLACK
```

Figure 1: Red-Black Tree pseudocode (from lecture slides)

Above is the pseudocode as implemented in my `RBTree`. `insert(int key)` inserts a given element into the `RBTree`, while `insertFixup(NodePtr k)` rotates the tree such that it still obeys the rules of a red-black tree.

Lines one through to sixteen of RB-Insert is equal to O(log n).

The rotations in an insertion are equal to O(1). This is due to the fact that, for insertion, there’s at most 2 rotations. Rotation only occurs in case two and case three of RB-Insert-Fixup. Moreover, case two will always be followed by case three, which both cause rotations. Case three concludes the method, resulting in the complexity of O(1).

Note: recolouring, however, will take O(log n).

3.3 Empirical Time Complexity Analysis

The following is an approximation of the real time that occurred when calling the `insert(int key)` method on my RBTree as it grew. The x-axis is n, while the y-axis is time-taken.

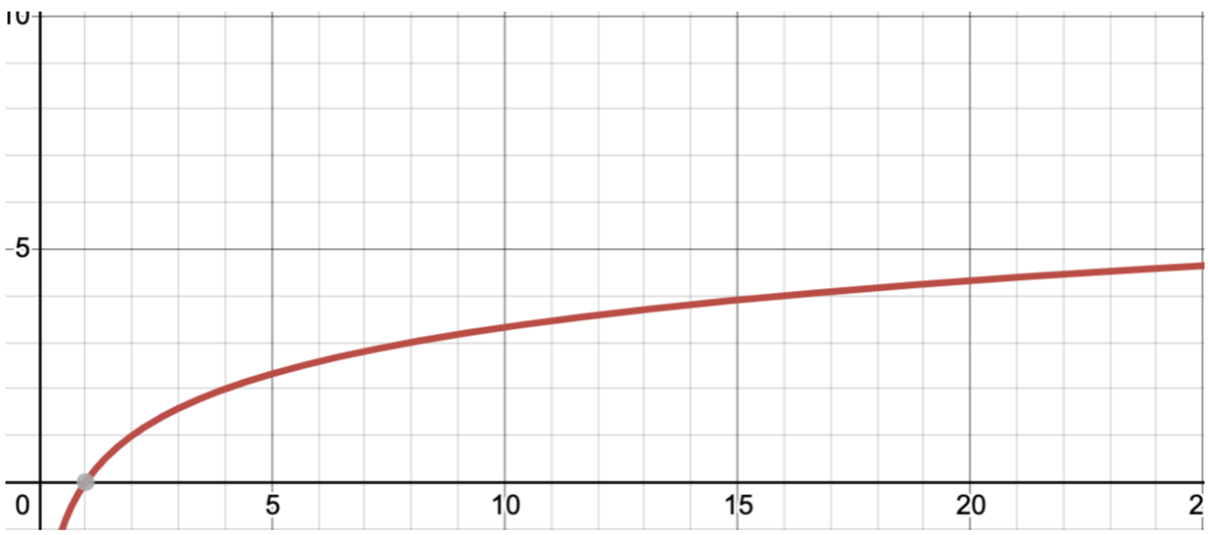


Figure 2: n x time taken for insert in RBTree.

This curve (not logarithmically adjusted) follows the expected complexity values of insertion into a red-black tree, thus the complexity of implanted inset method is O(log n).

4. Hashing

4.1 A short description of their role and why they are suitable for the purpose of my application

Hashing algorithms are essentially used to map data of arbitrary size to fixed-size values. This use of hashing aids in references to cells. This means it can be used to decrease the complexity of the search algorithm. While my program did not implement this aspect, it would also be useful in modifying data – for example calculations on the data, specifically where one cells value is based off a reference to another cell.

I have implemented a `HashTable` object program. This includes an array of buckets that can be hashed, which can go on to sort rows or columns of the spreadsheet. The hash table has insertion, deletion and search functions defined on it.

4.2 Theoretical Time Complexity Analysis

The hashing in my program works to first put the element into a linked list, and secondly to find the frequency of each element.

The time complexity for the first step can be derived by

$$T(n_1) = n\ x\ c_1$$

C_1 here can be defined by the computational cost required to add an element to the linked list for a bucket.

The time complexity for the second step can be derived by

$$T(n_2) = n\ x\ c_2$$

Where c_2 is the cost to get size of linked list plus the cost to add to that list.

It can be derived that Hash Tables are O(1) average complexity, and O(n) worst case complexity.

5.3 Empirical Time Complexity Analysis

Unfortunately, my Hash Table was not implemented fully in the program, however, there is a skeleton still there.

The following is a graph for the expected insertion time for my implemented hash table.

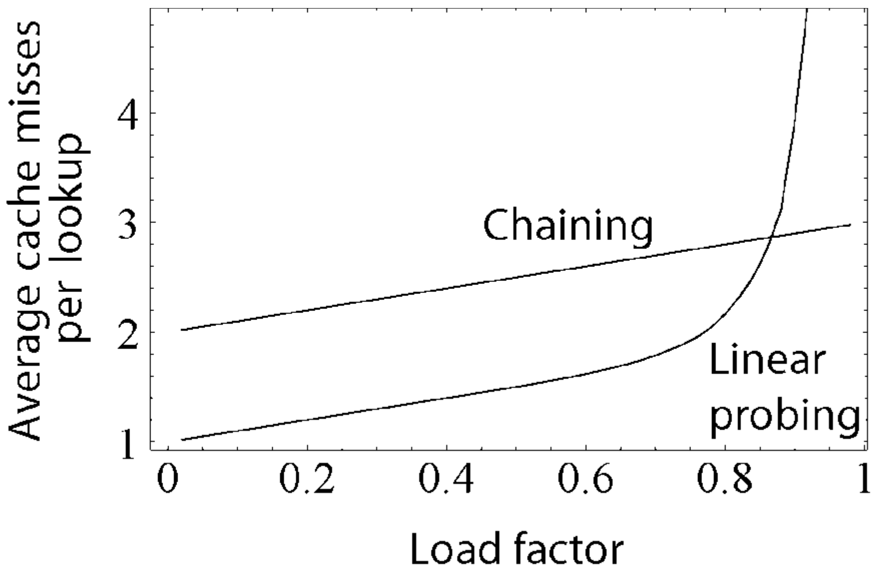


Figure 3: Average Hash Table Insertion Time

5. Dynamic Programming

5.1 A short description of their role and why it is suitable for the purpose of my application

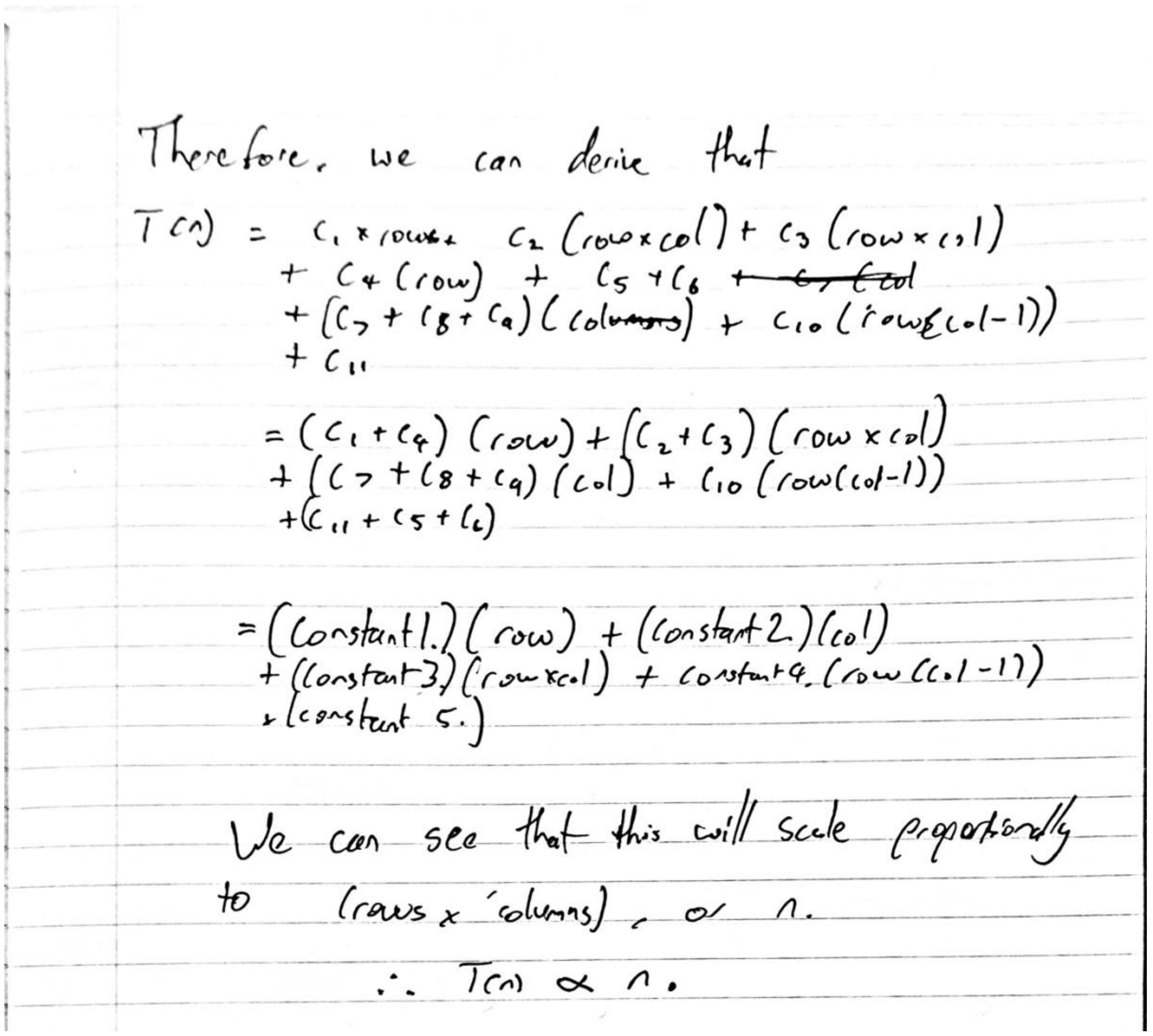
Dynamic programming involves breaking down an algorithm into smaller pieces such that the system optimises the use of memory. An example is the divide-and-conquer approach. A good use of dynamic programming would be to overcome the problem that the entire array of possible cells is too large be efficiently stored as a two-dimensional array with int data type as it approaches the RAM limit. This is despite the fact that this is how it is visualised to the user. Dynamic programming would allow more efficient use of memory.

I have used dynamic programming to implement the actual `Spreadsheet` object. A spreadsheet can be loaded (from file) into the program as an object. At this stage, the spreadsheet object has functions `save`, `load`, `print`, `search`, `sort`, etc. defined on it, which make use of dynamic programming for optimisation

5.2 Theoretical Time Complexity Analysis

This section will analyse the theoretical time complexity of the `print` method for the `Spreadsheet` object, which employs principals of dynamic programming. It can assumed that the input for this function is the two-dimensional array `currentSpreadsheet[][]`, where the input size n can be derived from `rows` multiplied by `columns`. i.e. the number of elements in the current spreadsheet. Note: `rows` and `columns` are indexed from zero in the program, however, in this theoretical analysis, they will be indexed from one.

print()	Cost	Times
`for (int i = 0; i < rows; i++)`	C_1	rows
`  for (int j = 0; j < columns; j++)`	C_2	rows x columns
`    if (j == (columns - 1))`	C_3	rows x columns
`      if (i == (rows - 1))`	C_4	rows
`        output += to_string( currentSpreadsheet[i][j] );`	C_5	1
`        break;`	C_6	1
`      output += to_string( currentSpreadsheet[i][j] );`	C_7	columns
`      output += "\n";`	C_8	columns
`      break;`	C_9	columns
`    output += to_string( currentSpreadsheet[i][j] ) + ",";`	C_{10}	rows x (columns – 1)
`Return output;`	C_{11}	1



Therefore, we can derive that

$$
\begin{aligned}
T(n) = {} & c_1 \times \text{rows} + c_2(\text{row} \times \text{col}) + c_3(\text{row} \times \text{col}) \\
& + c_4(\text{row}) + c_5 + c_6 \cancel{+ c_7\,\text{col}} \\
& + [c_7 + c_8 + c_9](\text{columns}) + c_{10}(\text{row}(\text{col}-1)) \\
& + c_{11}
\end{aligned}
$$

$$
\begin{aligned}
= {} & (c_1 + c_4)(\text{row}) + [c_2 + c_3](\text{row} \times \text{col}) \\
& + [c_7 + c_8 + c_9)(\text{col}] + c_{10}(\text{row}(\text{col}-1)) \\
& + (c_{11} + c_5 + c_6)
\end{aligned}
$$

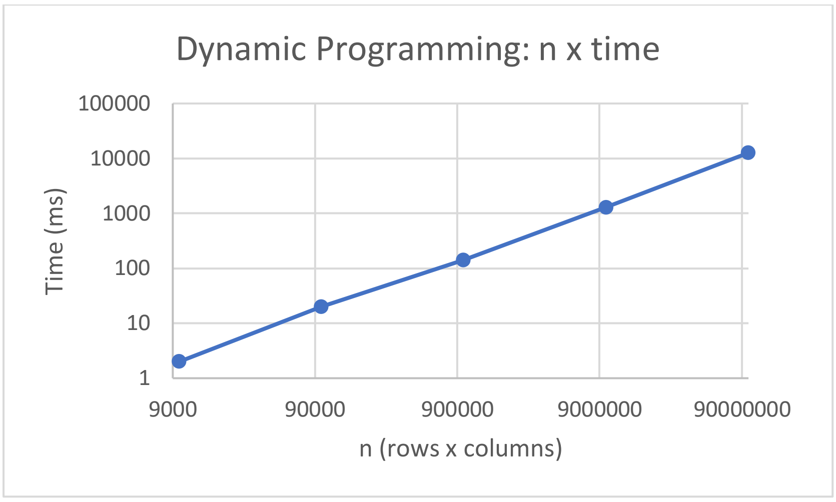
$$
\begin{aligned}
= {} & (\text{constant 1.})(\text{row}) + (\text{constant 2.})(\text{col}) \\
& + (\text{constant 3.})(\text{row} \times \text{col}) + \text{constant 4.}(\text{row}(\text{col}-1)) \\
& + (\text{constant 5.})
\end{aligned}
$$

We can see that this will scale proportionally to (rows x columns), or n.

$\therefore T(n) \propto n$.

5.3 Empirical Time Complexity Analysis

Measuring the empirical time taken to fully execute this method for increasing value of n (`rows` x `columns`), we obtain the following graph.



From this, we can derive that the time complexity T(n) is approx. equal to n, as expected.

6. Testing

To test my program, as well as provide the ability for users of the program to test it, it made far more sense to me to provide the user the option of running the program. In doing this, it gives the user the ability to test each function independently.

This is due to the easily navigable user interface. Specifically, the “help” command which reminds the user how to navigate the interface, and leaves control and testing up to them.

An example with the dynamic programming functionality is when the user types “print” and there is a spreadsheet that exists, the terminal will tell the user the number of milliseconds taken to perform that task. The user can then compare the input value (rows multiplied by columns in this case), with the time taken in order to derive empirical time complexity for the functionality.

7. Reflection

Ultimately, despite the challenges, I am happy with how my program and analysis turned out. I am excited for future studies into the intricacies of algorithms

7.1 What I have learnt

I have learnt how to apply some of the algorithms I have been learning about in this course, which I find to be exciting. I have also learnt about the implementation of relevant data structures and can see how they can be applied to real-world projects. I believe doing this through a complexity perspective is a useful educational skill.

7.2 What I would do differently

One thing I intend to work on is red-black trees. Specifically, how they can be implemented in my program. Many of their functionalities can be more simply utilised with simple hash tables, which I have already implemented. I can however see potential to simplify searching and sorting with the lower complexity of red-black trees. This needs further investigation.

8. References

- Some of RBTree psuedo code taken from here: https://www.geeksforgeeks.org/c-program-red-black-tree-insertion/.

- Lecture slides, regarding RBTree psuedocode and explanation for **Figure 1**.

- Hash Map analysis taken from lecture slides.

- **Figure 3** taken from https://commons.wikimedia.org/wiki/File:Hash_table_average_insertion_time.png