605.202: Data Structures

Peter Rasmussen

Lab 3 Analysis

Due Date: April 13, 2021

Dated Turned In: April 15, 2021

Lab 3 Analysis

This analysis examines the 1) use of the circular lists and node data dictionaries to add, subtract, multiply, display, and evaluate polynomial expressions and 2) code used to do so. I justify the use of a double-linked circular list as an appropriate method for this lab because that data structure 1) allows the program to efficiently cycle through polynomial expressions and 2) frees the user from managing a null pointer. I justify the use of a dictionary keyed on polynomial terms (e.g., $x^2y^1z^0$) because that approach makes the consolidation of terms fast and simple. I consider an alternate structure that might be better: using hashing to quickly retrieve polynomial expressions, operations, and variable-value evaluation sets. Although I did not use recursion in this lab, it may have been useful for parsing polynomial expressions.

I organized my program as a Python package comprised of modules and submodules, and grouped submodules by theme (e.g., node submodules were organized under the nodes module). The program also generated complexity statistics which I used to analyze the program's time and space efficiency.

I analyzed the complexity of the program and hypothesize that the runtime efficiency of the program is O(N) with respect to the number of evaluation sets processed. If we factor in the length of each expression M (e.g., x + y versus x + y + z) as a variable then we would have an O(M*N) problem. The space complexity for my implementation is analogous to runtime complexity because, although I read each character into memory one at a time and processed each in turn, I ended up loading it all into memory. I learned that I should have taken a top-down approach rather than the bottom up one I took, as doing so may have yielded a more direct approach. I will aim to begin Lab 4 sooner than I did Lab 3.

The rest of this analysis covers the following:

- Justification of the design of the program.
- Description and justification of circular list and node data dictionary implementation.
- Consider whether hashing may have been a better implementation.
- Discussion of the time and space efficiency of the program.
- Summarization of lessons learned and what I could have done differently.

<u>Justification of the Design of the Program.</u>

As was the case in previous labs, the Lab 2 program is organized as a Python package[1]. However, this time I grouped like Python files as submodules. This approach yielded a more coherent, organized construction of the program. In addition, the design of the program supports up to 26 variables (a-z) and allows the user to efficiently organize, retrieve, and combine polynomial terms because of its use of node data dictionaries.

The package is organized as follows:

- lab3: Program module
    - __main__: Conductor file that parses command-line arguments and executes the run script.
    - __init__: Limits imports to the run function.
    - run: Executes file IO, polynomial parsing, polynomial arithmetic, and polynomial evaluation functions.

[1] Per Scott Almes, proj0 Python package: "Packages are groups of modules that work together to act as a library or program".

- symbols: Used throughout the program, this module provides the sets of accepted symbols and classifies symbols by kind (e.g., operators, variables, etc.).
- Miscellaneous top-level scripts.
- evaluators: Comprised of submodules that combine (i.e., add, subtract, multiply) and evaluate polynomial expressions.
  - combine: Adds, subtracts, and multiplies polynomial expressions.
  - evaluate: Evaluates combined polynomial expressions for given set of variable-value assignments.
- parsers: This module reads polynomial and evaluation inputs and organizes each into its respective list.
- lists: Organizes all the lists under one module and includes, but is not limited to:
  - polynomial_list: Its class inherits a circular list class and holds polynomial expressions as nodes.
  - Variable_value_list: Modified singly linked list that includes validation methods and a name-based get_node method.
- nodes: Its submodules make up the building blocks of the list data structures.
  - polynomial_node: Each polynomial node holds a polynomial expression.
  - variable_value_node: Each node holds one variable-value pair. The node class inherits from SimpleNode, which is a singly-linked list structure.
- tests: Performs unit tests for list and node classes.

The program generates an output similar to the example below.

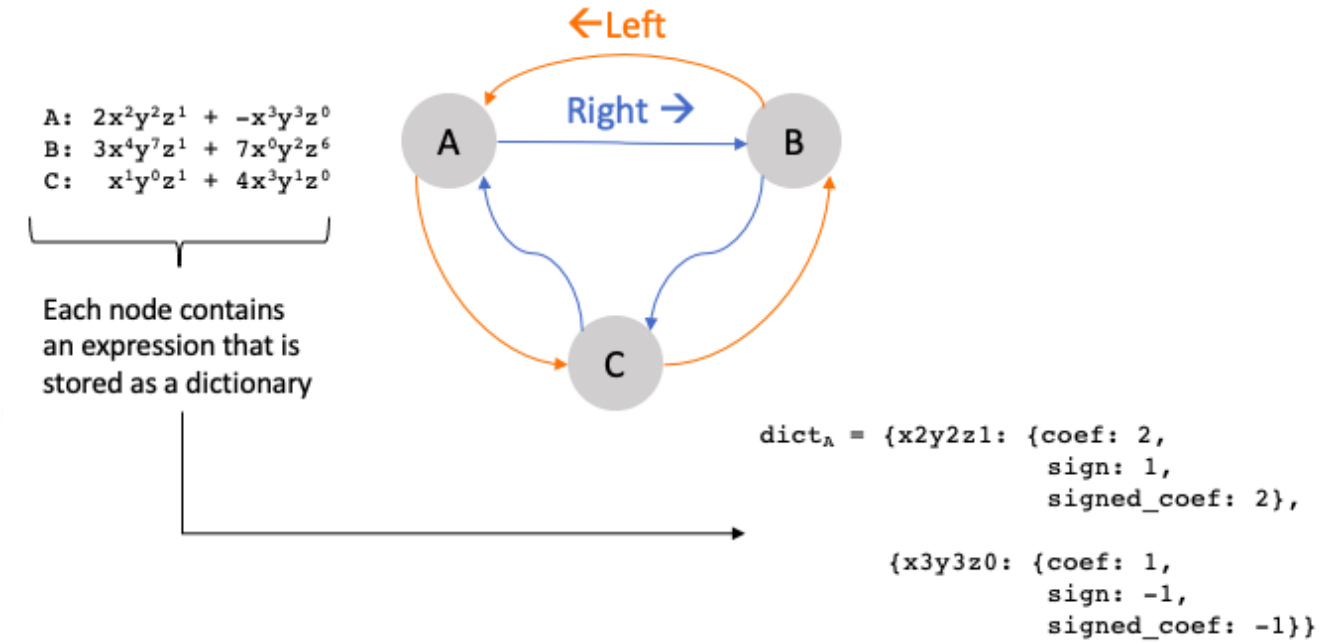
```
  # Peter Rasmussen, Lab 3
    # Polynomial simplification and evaluation
    # Input files:
    # /Users/peter/additional_polynomial_input_04.txt
    # /Users/peter/resources/additional_evaluation_input_02.txt
    # Output file: /Users/peter/resources/additional_output_04_02.txt

    @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
    Polynomial input expression definitions
    A = 31a3b4c3d4e1+11a4b1c0d0e0-1a4b1c3d3e3-13a2b3c2d4e0+24a1b3c4d4e3
    B = 7a1b4c2d3e1-29a1b2c3d0e2-25a3b2c0d4e3+6a4b2c0d1e2-1a1b0c2d0e3
    C = a2b1c3d1e1-10a2b1c0d3e4+23a4b2c2d3e2+16a0b2c2d3e2-5a2b1c1d1e2
    D = 4a3b1c0d1e2+24a4b4c2d0e2+28a2b3c2d1e0+18a1b3c0d2e1-16a1b4c4d0e2
    A+B
    Input:  (31a3b4c3d4e1+11a4b1c0d0e0-1a4b1c3d3e3-
13a2b3c2d4e0+24a1b3c4d4e3)+(7a1b4c2d3e1-29a1b2c3d0e2-25a3b2c0d4e3+6a4b2c0d1e2-
1a1b0c2d0e3)
    Output: 31a3b4c3d4e1+11a4b1c0d0e0-1a4b1c3d3e3-
13a2b3c2d4e0+24a1b3c4d4e3+7a1b4c2d3e1-29a1b2c3d0e2-25a3b2c0d4e3+6a4b2c0d1e2-
1a1b0c2d0e3
    Evaluation Set                  Answer
    a0b4c1d5e4                      0
    a2b3c5d0e1                      -64772
```

Description and justification of circular list and node data dictionary implementation.

The program can efficiently traverse the double-linked circular list to obtain the desired polynomial expression 1) without the risk of a pointer falling off the list and 2) can traverse in either direction to find the closet node. The diagram below illustrates the circular list, its nodes that hold polynomial

expressions, and the dictionaries that organize the data in those expressions. The use of node dictionaries has the benefit of making it simple and fast – because the dictionary is hashed – to combine like terms: they simply need to share the same term.



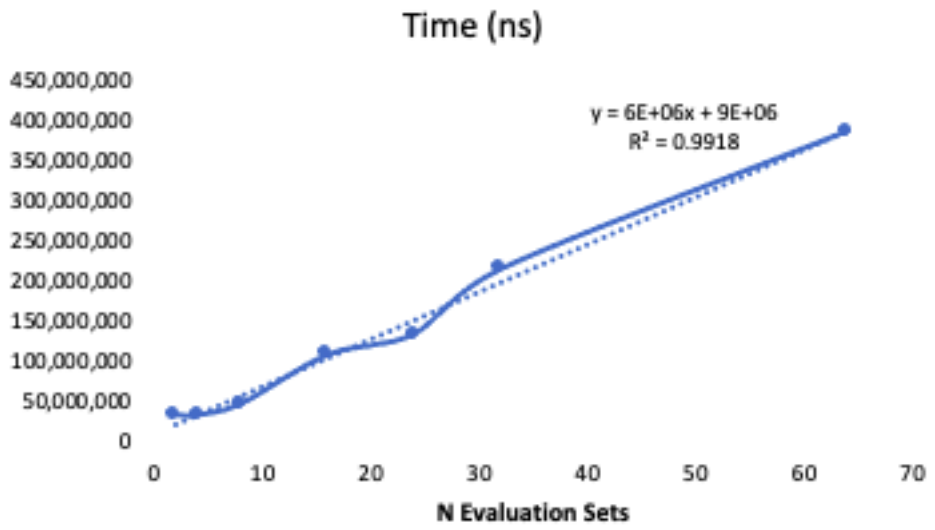
Consider whether hashing may have been a better implementation.

The hashed approach worked well for organizing, retrieving, and consolidating polynomial terms, and it would be considerably more straightforward to use this approach than the circular linked list one. We could hierarchically organize our dictionary into key: value pairs of dictionaries. For instance, the label A would be the key for its expression, and that expression could be the key for the components (i.e., coefficient and sign) that comprise it. Retrieval is simple and fast, and if we encountered any collisions we could refactor our hashing function accordingly.

Discussion of the time and space efficiency of the program.

Runtime

As the chart below indicates, the runtime efficiency of the program is O(N) with respect to the number of evaluation sets processed. If we factor in the length of each expression as a variable, then the complexity would be O(M*N). Although the number of operations (e.g., A+B, A*B, etc.) was constant in this lab, it is reasonable to imagine it would not be in another case. In that case, the complexity would be O(M*N*O), where O would be the number of operations.



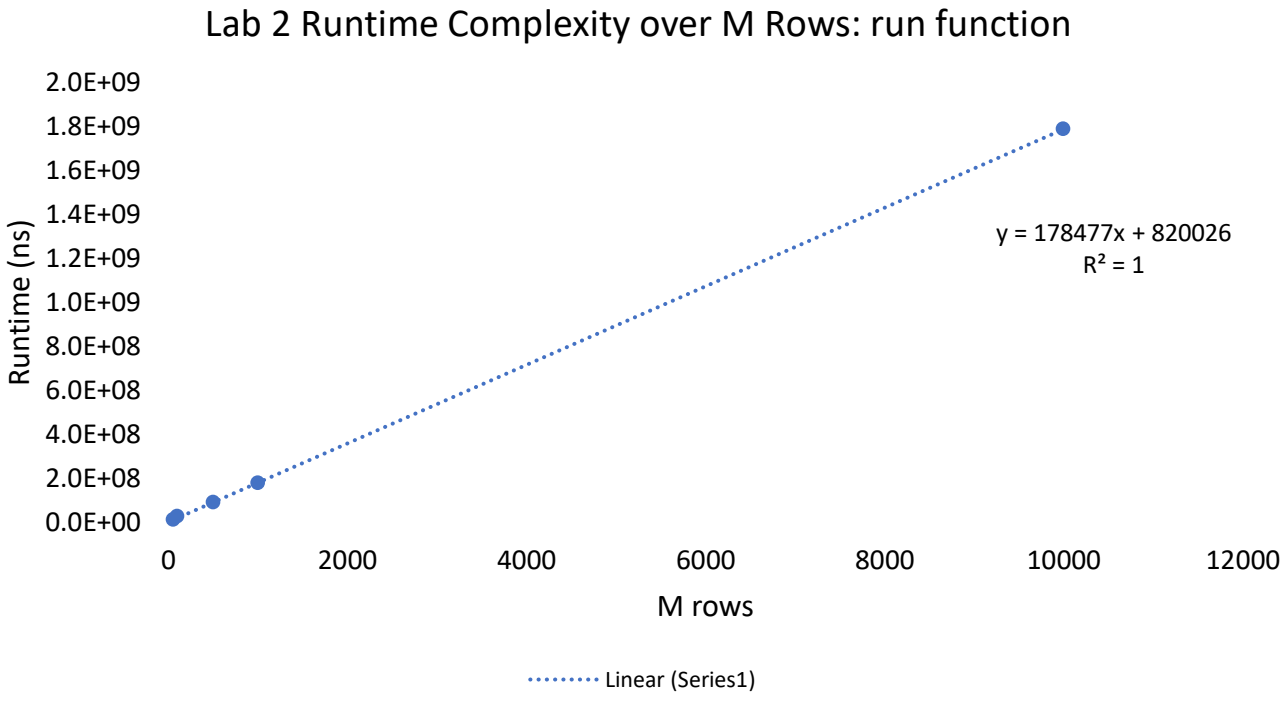
Space efficiency

In the implementation I used, the space complexity would be analogous to the runtime complexity. Although I read characters one-by-one, they were eventually all loaded into memory for processing. Therefore, the total space in memory would increase by O(M*N). However, if I had only loaded one evaluation set and one polynomial expression into memory at a time, writing to output with each line of input, we could achieve O(1) space complexity.

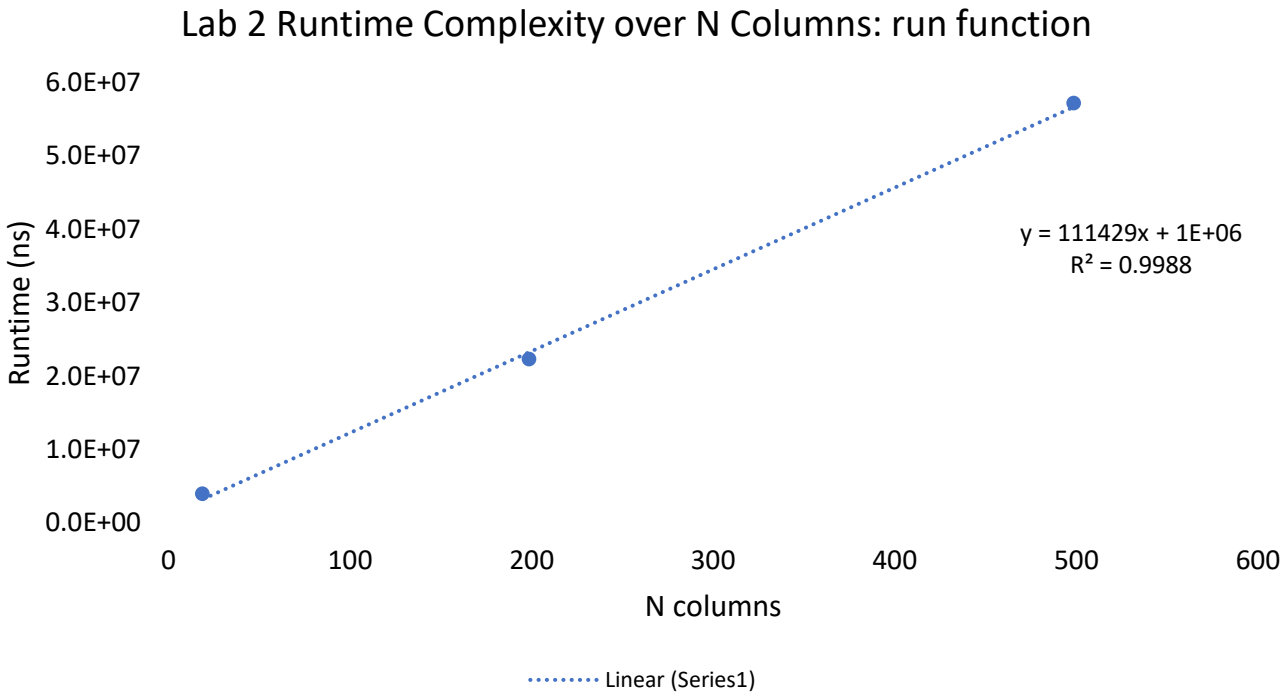
<u>Discussion of the Time and Space Efficiency of the Program</u>

Runtime efficiency

The chart below depicts runtime complexity for the recursive solution as M rows increases. We can clearly see that the runtime complexity is linear in M, or O(M).



In the chart below, we see that the runtime complexity as the number of columns N increases for a fixed number of rows M is O(N), although admittedly there are fewer data points to work with due to recursion depth limitations for N greater than 500 and less than 1000.



Space Efficiency

The space efficiency of the stack is O(M*N): As the number of characters per line M and the number of lines N increase, the space complexity increases accordingly. However, in practice we could assume that N columns tends to be constant, and this reduces our space efficiency to O(M).

Summarization of Lessons Learned and What I Could Have Done Differently.

Although life is hectic, I should have begun this lab sooner. I also should have taken a more top-down approach. In the bottom-up heavy approach I took, I developed the underlying classes and base structures, slowly building up to the highest-level data structures and adding features along the way that I thought would enhance the value of the program. I think I would have been more efficient had I instead focused on finding a simple and direct solution to compute the required outputs and from there drilled down to write the underlying data structures.