Kordel France
Lab 3
Class 605.202 Section 81

Lab 3 Analysis Document

FranceLab3 focuses on the construction of a multivariate polynomial expression evaluator through the use of a circular singly linked list abstract data type. Operations on *FranceLab3* is a software module written in Python that performs multiplication, division, and subtraction between polynomial expressions of up to three different terms and evaluates them for integer numbers.

Description of Chosen Data Structures

The data structures used for this program were linked lists, particularly circular singly linked lists (CSL lists). A multivariate polynomial term was encoded into a node object called *Node* such that each node contained five parameters – a coefficient, pointer to the next node, a degree for *x*, a degree for *y*, and a degree for *z*. Multiple polynomial terms were encoded into a single CSL list called *CSLList* that represented an entire polynomial expression. For example, the terms $4x^2y^3z^4$ and $2x^4y^3z^2$ are encoded into two separate nodes while the expressive combination between those two ($4x^2y^3z^4 + 2x^4y^3z^2$) are encoded into a CSL List object to represent the full expression. Towards the end of the project, I found it useful to have the ability to save each polynomial into a separate list ADT for the purposes of iterating over for evaluation. To this point, each CSL List was itself stored into another CSL List such that there exists a circularly linked list of circularly linked lists that represent a series of polynomial expressions. This object is monotonously defined as *CSLListOfCSLLists* within the module and proved very useful for cleanly evaluating the expressions and printing them to the screen. Arrays in other implementations were used sparingly but only for operations having nothing to do with the expression evaluation, such as cleaning the input text file or storing command-line arguments.[1] My extraction algorithm looked for multivariate pattern in the format $sCx^ay^bz^ck$, where *s* was the sign of the term, *C* was the coefficient, *a* was the degree of the x-term, *b* was the degree of the y-term, *c* was the degree of the z term, and *k* was either an operator or a carriage return character. This was all formatted into a node object.

Justification for Design Decisions and chosen ADTs

While the use of a circular singly linked list was advised in the lab assignment, I also found that it seemed a natural choice for the polynomial representation. A CSL list is singly linked, so there

[1] In office hours with Dr. Chlan on Sunday, 11 April 2021, I asked whether data from the input file could be stored into an array buffer just for cleaning the data and checking for invalid characters prior to feeding it into the CSL list. She mentioned that this was acceptable so long as the data is read in character by character and is only used for the purposes of cleaning data. The timestamp of the question is between 8:15 pm and 8:20 pm ET (T+15 min to T+20 min) in the Office Hours Recordings on Blackboard.

is only the housekeeping of keeping track of one node pointer versus two node pointers in a doubly linked list. Making the linked list circular also allowed me to easily traverse from the head of the list to the tail. Although this can be an issue for lists with large sizes since traversing from index *i* to index *i - 1* would take significant computing time, I was confident that my data size would be small and that this would not pose a problem. I initially planned to implement recursion to decompose the data buffer into CSL lists of polynomial terms, then polynomial expressions, and then polynomial series. However, during the development of the algorithm, there became several base cases that made the implementation of recursion difficult. Due to this, I pivoted toward the use of an iterative algorithm. Given more time, I am confident I could have figured out an eloquent way to configure tail recursion. The iterative function allowed me to optimize my space complexity, but the resulting algorithm was a bit messy. Nevertheless, it seems a great fit for the job.

Leveraging the CSL structure I used in building the polynomial expressions into the construction of polynomial series allowed me to leverage a lot of my existing code between both structures. The polynomial representation object, *CSLList*, contains several methods for adding, subtracting, multiplying and dividing polynomials that became simple to utilize with the CSL structure once the Node objects were correctly formatted.

The *Node* object representing a polynomial term was rebuilt several times, but the resulting structure worked really well. Initially, I represented each variable within a polynomial term as its own node, but encountered a lot of friction for multiplication and division between polynomial expressions. The eventual reconfiguring of the object with the coefficient and all three degrees gave me much better time complexities on my multiplication and division functions.

Complexity and Algorithm Analysis

Please refer to *Table 1* for a complete analysis of time and space complexities for every function in the *FranceLab3* module.

The most demanding tasks on time complexity was actually an enhancement and not part of the core project requirements. The processing of the input file technically proved the most demanding on time complexity. However, this is because the program contains three nested loops to detect the input and output files for the program (iterating through each directory, then iterating through each subdirectory, then iterating through each file). While the maximum number of directories should be 1, the maximum number of subdirectories per directory should be 2, and the maximum number of files per subdirectory should be 8, I decided to use a bit of defensive programming in case the user mistakenly creates subdirectories into the module while trying to run it. Due to these reasons, the functions possessing in *error_handling.py* are highly unlikely to exhibit $\boldsymbol{O(n^3)}$ time complexities. The space complexities for these functions are negligible because the functions are only ever holding a Boolean *file_found* value in memory.

FranceLab3 Complexity Analysis				
			Worst Case	
#	File	Function	Time Complexity	Space Complexity
1	__init__.py	*[file]*	N/A	N/A
2	__main__.py	*[file]*	43	161
3	constants.py	*[file]*	7	198
8	Node.py	*init*	1	1
9		*getNodeXDegree*	1	1
10		*getNodeYDegree*	1	1
11		*getNodeZDegree*	1	1
12		*getNodeCoefficient*	1	1
13		*getNodeType*	1	1
14		*getNextNode*	1	1
15		*setNodeXDegree*	1	1
16		*setNodeYDegree*	1	1
17		*setNodeZDegree*	1	1
18		*setNodeType*	1	1
19		*printNode*	1	1
20	CSLList.py	*init*	1	1
21		*getNode*	n	1
22		*appendToList*	n	1
23		*printFullList*	n	n
24		*returnFullList*	n	n
25		*__add__*	n	1
26		*__sub__*	n	1
27		*__truediv__*	n	n/2
28		*__mul__*	n	n/2
29		*getProductOfTerms*	n	n
30		*getQuotientOfTerms*	n	n
31		*evaluateList*	n	1
32	CSLListOfLists.py	*init*	1	1
33		*appendToList*	n	1
34		*printFullList*	n	n
35		*returnFullList*	n	n
36	file_processing.py	*process_file_data*	n	1
37		*buildCircularSinglyLinkedList*	n^2	n
38		*evaluateAndDisplayPolynomials*	2n + n^2	n
39	error_checking.py	*check_project_for_input_file_in_correct_directory*	n	1
40		*check_project_for_input_file*	n^3	1
41		*find_input_file*	n^3	1
42		*check_project_for_output_file_in_correct_directory*	n	1
43		*check_project_for_output_file*	n^3	1
44		*find_output_file*	n^3	1

Table 1 - Worst-Case Time and Space Complexities for functions in FranceLab3

Probabilistically, the highest complexity functions in both space and time will be the *file_processing.py* functions. The function *process_file_data* manages the reading and cleaning of all input data character by character and poses only ***O(n)*** time complexity. Since the input file is only reading a character at a time, the time complexity maintains an appealing ***O(1).*** Upon data cleaning, the data buffer moves to the *buildCircularSinglyLinkedLists* function to build each

node, and two nested sets of circularly linked lists. The construction of these two lists delivers two nested while loops which lends to a time complexity of $\boldsymbol{O(n^2)}$ but only a space complexity of $O(n)$. With more time, I believe I may be able to bring this time complexity down through the implementation of tail recursion. The result will likely be $\boldsymbol{2*O(n)}$.

The formatted polynomials resulting from the construction of the CSLLists in *buildCircularSinglyLinkedLists* moves to an evaluation process in *evaluateAndDisplayPolynomials*. This function provided the worst time complexity for the core project requirements with $\boldsymbol{2*O(n)+O(n^2)}$. This large magnitude is partially attributed to the method in which data is displayed to the user. Though the actual size of n is very small for this program, the algorithm iterates over every math operation $(A + B)$ for every polynomial $(x^2y^3+z^4)$ for every set of evaluation points $(x = 1, y = 2, z = 3)$. The way in which I wanted to display the results of these evaluations to the user is the catalyst in this nested-loop sequence. Through all of this, the algorithm still maintains a space complexity of $\boldsymbol{O(n)}$.

In hindsight, the implementation of tail recursion would have decreased my time complexities and optimized my code, especially for the *buildCirularlSinglyLinkedLists* and *evaluateAndDisplayPolynomials* function. Although my space complexity metrics would likely increase, for this particular project, the data population remains relatively small which discounts any worry of the recursive stack exploding, especially because of the *tail* flavor of recursion.

Based off of the above analysis and, in consideration of the data presented in *Table 1*, the *FranceLab3* program has a worst-case time complexity of $\boldsymbol{O(n^3)}$ and a worst-case space complexity of $\boldsymbol{O(n^2)}$. However, the *FranceLab2* module probabilistically has worst-case time and space complexities of $\boldsymbol{O(n^2)}$.

Lessons Learned

With the luxury of additional time, I would have focused more on implementing tail recursion into the construction of the CSL List objects and the evaluation algorithm. Using tail recursion would allow me to keep my memory low and clean up my code. The biggest issue, still, would be trying to find a low number of base cases to make recursion attractive. My extraction algorithm looked for multivariate pattern in the format $sCx^ay^bz^ck$, where s was the sign of the term, C was the coefficient, a was the degree of the x-term, b was the degree of the y-term, c was the degree of the z term, and k was either an operator or a carriage return character. This was all formatted into a node object. Given the nature of how data could be input into the input file by the user, there is a lot of cleaning that is necessary just to extract a simple polynomial term into a node. For example:

1) there is no coefficient present in the term if the coefficient is implicitly 1, so it must be added.
2) there are no exponents for variables that are implicitly of power 1, so this must be detected and added.

3) there are no variables or exponents for variables of power 0, so this must be recognized and added.
4) negative signs must be distributed to the polynomial whereas positive signs do not.

The "base" cases all give rise to several exceptions that must be handled to determine what happens with operators, operands, and terms that lead to a very specific algorithm.

Distributing multiplication and division among a series of terms proved much more difficult than multiplication and division between two terms. This process alone took a majority of my effort. I eventually ended up building two functions where one function performs multiplication/division among a series of terms while the other distributes the multiplier and divisor to each term individually.

All in all, *Lab3* provided me with another angle of appreciation for ADTs – the right ADT can reduce the complexity in interpreting user input. I gained a lot of insight on how ambiguity in user input can actually direct the decision on which ADT to select for an algorithm. The amount of edge cases the evaluation of polynomials contains pushed me away from recursion for this project, but additional insight gained while building an iterative version.

Optimizations for Later Versions

As specified previously, I would choose to implement at least two of my algorithms utilizing tail recursion. Additionally, my code contains a mix of camel case (camelCase) and "python_style" (python_style) syntax which I would like to clean up. Working on the project across multiple periods while working across multiple languages with work is the driver behind the inconsistency. Finally, I would like to alter *Lab3* to accommodate floating point values for both coefficients and exponents.

Addressing Requirements in *Lab 3*

FranceLab3 addresses the requirements as specified in the *Lab 3* handout. Specifically, the program reads polynomial expression data from an input text file character by character. The program continues by appropriately parsing the data and detecting polynomial terms as nodes of a circular linked list in the form of $sCx^ay^bz^ck$. The program then allocates these terms into a circular linked list to represent a full polynomial expression. Then, each expression is input into a separate, higher level of circular singly linked list for storage. The program evaluates each of the required 8 operations from the *Lab3* handout by multiplying, adding, and subtracting the four expressions below using the corresponding mathematical operations:

Expressions

A) 5x2y2z3 + 2xz4 + y2z + z

(B) 3x4y4z4 + 12x3 + y2z2 + 2yz3

(C) 2xyz

(D) 25xyz - 3xz4 - 12yz3

Operations

A + B, A + C, A + D, B + C , B + D, C + D, B – A, B – D,

A * B, A * C, A * D, B * A, B * C, B * D, C * D

Following the completion of these operations, the resultant expressions are presented to the user on the screen and also written to the output file. Then, each of the four expressions is evaluated using the following values for x, y, and, z:

x=0, y=1, z=2,

x=1, y=2, z=3

x=2, y=1, z=0

x=4, y=-1, z=-4

Under the *Course Modules* for the class on *Blackboard,* there was an additional file stating it as required evaluation input for the assignment. It is shown below:

1) x0y1z2
2) x1y2z3
3) x2y1z0
4) x4y-1z-4

As the *Lab3* handout did not make it clear which sequence of values to evaluate the resultant polynomials for, I perform the evaluation on both sets of functions within *FranceLab3.*

Finally, these resulting values from these evaluations are presented to the user on the screen and also written to the output file. In total, 8 expressions are evaluated for 4 different sets of data points.

Enhancements

The program provides 4 major enhancements (along with many other minor enhancements) on top of the original requirements.

1) The first major enhancement is centered around the command line arguments. In order to execute the program, the user is required to specify the name of their input and output text files such as `python -m Lab2 input.txt output.txt.` The user is directed to put these files into the *io_files* folder of the module. In the event that the user puts these files into the wrong folder, the program searches the entire program for both files and, if found, relocates them to the designated *io_files* folder. In the event that the user specifies these files and the program cannot find either one of them, the program automatically generates 1) an input file called *input.txt*, and 2) an output file called *output.txt.* Both files are located in the *io_files* folder and the *input.txt* file is auto-populated with demonstration data containing 7 polynomial expressions. So, in short, if the user never creates input or output files or moves them to the incorrect file path, the program will still execute successfully.

2) The second major enhancement incorporated time delays. When the program is conventionally executed, the command prompt flashes a plethora of data and finishes executing in under a second. The user is then forced to go back and read through the steps executed. *FranceLab3* incorporates time delays to allow the user to read information in the command prompt as processing steps are completed. For example, when the program starts up and accepts the command line arguments, a time delay of 3 seconds allows the user to know whether or not their *args* were accepted. Additionally, when the program successfully imports the I/O files, this information is logged to the command prompt and a time delay of 3 seconds halts processing so that the user can take in this information accordingly. After each mathematical operation is performed on a polynomial, processing is briefly paused to allow the user to visualize the operators, operands, and result. Finally, after each polynomial expression finishes its evaluation, a time delay of 1 second briefly holds processing for each evaluated set of points as well to allow the user to see whether or not that step failed.

3) The next major enhancement revolves around error handling. If a polynomial expression contains illegal characters, the program notifies the user through both the console and the output file. However, instead of simply letting the user know that the expression contained invalid characters, they are notified of exactly which characters that were illegal in their expression. Additionally, all white space is concatenated so that any gaps between characters in an expression (i.e. **3 4*) are stripped and any new lines within the input file are not counted as polynomial expressions.

4) Finally, the fourth major enhancement builds a summary metric for the user. At the end of both the console output and output file, a summary of exactly how many prefix expressions were processed is displayed along with how many successively converted to postfix expressions and how many failed.

5) The conversion and evaluation algorithm performs division between polynomial terms. While only addition, subtraction, and multiplication were labeled as requirements for *Lab3,* I built *FranceLab3* incorporate division. A small series of tests were performed to validate.

6) *FranceLab3* accepts negative exponents and performs the evaluation accordingly. This took quite a bit of logic surrounding sign handling in the extraction algorithm, but I was able to obtain success on tested conditions. A small series of tests were performed to validate.

More On Recursive vs. Iterative Evaluation

The constraints posed in *Lab 1* and *Lab 2* illuminate the pros and cons of using both recursive and iterative procedures in evaluating mathematical expressions. While I know that iterative methods can occasionally out-perform recursive methods in time and space complexities, both *Lab 1* and *Lab 2* showed consistent algorithmic behavior across my code in this respect. Under both scenarios, the project produced worst-case time complexities of $\boldsymbol{O(n^3)}$ and worst-case space complexities of $\boldsymbol{O(n^2)}$. However, the iterative solution wins my preference due to the facts that it is easily represented the mathematical evaluation rules, resulted in less lines of code, and worked seamlessly with the circular singly linked list ADT.

Test Results

Running the command `python -m Lab3 input.txt output.txt.` produced appropriate results for an input file called *input.txt* containing the above mentioned polynomial expressions. A log of this output may be found in the Appendix. The content for this command-prompt output are identical to that which is written to the output file.

Conclusion

The *FranceLab3* Python module performs mathematical operations on multivariate polynomial expressions specified by the user and returns (if available) its corresponding polynomial expression. The program then evaluates the expression for four specific sets of values. The module does this explicitly through the implementation of custom circular singly linked lists and the custom nodes. In hindsight, there are some optimizations that could be made to enhance the

time and space complexities and fine-tune the conversion algorithm through tail recursion. The module contains several enhancements that aid in usability, user interface, error handling, and functionality. All tested polynomial expressions correctly output the corresponding postfix expression and appropriately communicates any errors to the user.

References

1) Almes, Scott; et al. (2021). Stacks Office Hours, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 4 February and 8 February 2021.

2) Almes, Scott; et al. (2021). Stacks Office Hours, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 12 February 2021.

3) Almes, Scott; et al. (2021). Queues and Lists Office Hours, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 19 February 2021.

4) Almes, Scott; et al. (2021). Lab 2 Office Hours, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 25 February 2021.

5) Almes, Scott; et al. (2021). Lab 0 Sample Project, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 26 February 2021. Web. URL: https://blackboard.jhu.edu/bbcswebdav/pid-9469422-dt-content-rid-100642966_2/xid-100642966_2

6) Almes, Scott; et al. (2021). Lab 2 Office Hours, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 1 March 2021.

7) Chlan, Eleanor; et al. (2021). ADT and Complexity Notes, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 28 January – 2 February 2021.

8) Chlan, Eleanor; et al. (2021). ADT and Complexity Notes, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 2 February– 9 February 2021.

9) Chlan, Eleanor; et al. (2021). Stacks Office Hours, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 14 February 2021.

10) Chlan, Eleanor; et al. (2021). ADT and Complexity Notes, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 10 February– 16 February 2021.

11) Chlan, Eleanor; et al. (2021). Queues Notes, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 17 February – 18 February 2021.

12) Chlan, Eleanor; et al. (2021). Lists Notes, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 17 February – 18 February 2021.

13) Chlan, Eleanor; et al. (2021). Queues and Lists Office Hours, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 21 February 2021.

14) Chlan, Eleanor; et al. (2021). Queues and Lists Office Hours, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 28 February 2021.

15) Miller, B. N., & Ranum, D. L. (2014). Problem solving with algorithms and data structures using Python (2nd ed.). Decorah, IA: Brad Miller, David Ranum.

16) Chlan, Eleanor; et al. (2021). Graphs Notes, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 10 March– 15 March 2021.

17) Chlan, Eleanor; et al. (2021). List Notes, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 10 March– 15 March 2021.

18) Almes, Scott. (2021). Homework 7 Office Hours, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 11 March 2021.

19) Chlan, Eleanor. (2021). Homework 7 Office Hours, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 14 March 2021.

20) Almes, Scott. (2021). Lab 2 Office Hours, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 19 March 2021.

21) Chlan, Eleanor. (2021). Lab 2 Office Hours, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 21 March 2021.

22) Almes, Scott. (2021). Homework 10 Office Hours, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 26 March 2021.

23) Chlan, Eleanor. (2021). Homework 9 Office Hours, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 28 March 2021.

24) Almes, Scott. (2021). Homework 9 Office Hours, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 29 April 2021.

25) Almes, Scott. (2021). Homework 10 Office Hours, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 1 April 2021.

26) Chlan, Eleanor. (2021). Homework 10 Office Hours, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 4 April 2021.

27) Almes, Scott. (2021). Homework 10 Office Hours, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 5 April 2021.

28) Almes, Scott. (2021). Lab 3 Office Hours, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 8 April 2021.

29) Chlan, Eleanor. (2021). Lab 3 Office Hours, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 11 April 2021.