LAB 2 ANALYSIS REPORT

by Chih Yuan Chan ID: 28A8CA

Date: 03/28/2023

1. PROBLEM DEFINITION

1.1 General approach

The approach of Lab 2 is to convert a Prefix expression into a Postfix expression using recursion.

The structure of the prefix to postfix conversion naturally suits a recursion approach since the prefix notation may be defined recursively (refer below algorithm).

In this case the Prefix expression will be traversed / read from left to right and the following sequence of operations may be repeated recursively (ref 1):

- **STEP 1** - *The first character in a Prefix expression is always operator. The recursive method stores the first character and saves it to a variable, say 'op'.*
- **STEP 2** – *To obtain the left operand (say 'op1') the method checks if the next character in the string is an operator. If it is an operator, a recursive call is triggered until an operand is detected. At which point, this is output as op1.*
- **STEP 3 –** *Step 2 is repeated for the right operand after op1 is output to obtain op2.*
- **STEP 3** - *A string or list is declared to progressively store the resultant postfix sub-expression, which is formed by combining:*
  op1 + op2 + op
- **STEP 4** – *Per steps 2 and 3, the base case of the recursion is reached when a sub-expression is reduced to a single operand, which is then added to the postfix expression.*
- **STEP 5** - *When the last recursion returns from the call stack, the string or list contains the final postfix expression.*

1.2 Data structures implemented in recursive algorithm

A recursive approach does not require a specific ADT to maintain state, such as a Stack data structure for an iterative approach.

The data structures that have been implemented in the recursive code are simple Python List (dynamic array) or String data structures to store the results of reading and writing the pre and post-fix expressions respectively.

When calling by value is not an issue a simple String has been used. However, when recursion is implemented, the intermediate results of variables can be lost if these are stored by value. When call by reference is required so that updated variables are not lost when the stack frame calls are un-wound, a list has been adopted.

The worst-case performance for reading and writing to a string / dynamic array Python list is O(n), which is justifiable for an algorithm that needs to individually scan all characters of an expression.

2. JUSTIFICATION OF APPROACHS

2.1 Justification for converting Prefix to Postfix using recursion

Recursion is an appropriate implementation for converting Prefix to Postfix for the following reasons:

- No separate data structures i.e. stack are required to maintain the state of the operations. From this standpoint, memory overheads are minimised.
- The input prefix string is read left to right and this sequence is implemented using recursion without needing separate operations to reverse the order of the string.
- The recursive algorithm mirrors the actual sequence of operations when traversing an expression tree, hence is intuitive and directly interpretable.
- The recursive code can also be very concise versus the iterative approach.
- The time complexity performance is O(n) (see separate discussion below) and this is comparable to alternative iterative (stack based) approaches.

2.1.1 Recursion Vs Stack based approaches

In general, the 2 approaches to converting Prefix to Postfix are a Recursive approach and Iterative approach (represented by e.g. using a stack ADT).

A comparison of the 2 separate approaches is as follows:

Recursive approach

Advantages: Per previous section

Disadvantages:

- A recursive algorithm requires the execution of stack frame calls. Python does not support tail call recursion (ref 1), hence, making the stack frame calls will generate memory overheads and slow down the algorithm vs iterative approaches where a stack frame is not required. When the prefix expressions are very large, this can result in stack overflow issues. If other languages such as C++ are used to support tail recursion, then the differences in memory overhead due to the single stack frame calls vs iterative approaches is minimized.
- Recursion can cause debugging to become more complex, particularly when multiple stack frame calls will need to be tracked.

Iterative stack based approach

Advantages:

- Pushing the characters from the Prefix String (left to right) onto the stack naturally arranges the characters in the correct reverse (right to left) order as they are popped.

- A stack naturally maintains the precedence of the operators by storing them in the order they are encountered as the stack is traversed.
- Once the entire Prefix expression has been traversed, the stack naturally stores the characters in the correct Postfix order, which can be efficiently retrieved by sequentially popping the characters from the stack.
- The algorithm does not require intermediate elements within the stack to be accessed, only the top (head) element in the stack. Pushing and popping from the top of the stack can be implemented with O(1) time and space complexity and this is highly efficient.

Disadvantages:

- The stack-based algorithms require explicit manipulation of the stack, this introduces additional complexity into the algorithm.
- Since a stack is required to maintain state, stack-based algorithms may require more memory than their equivalent recursive algorithms.
- The stack-based algorithms can be less intuitive and concise than recursive algorithms, as they do not reflect the sequence of traversing the expression tree as directly as the recursive approach.

2.1.2 Overall justification of algorithm approach

As a general rule, for converting a Prefix to Postfix expression, I would adopt an iterative stack-based approach.

This position has not changed from Lab 1. However, with a new understanding of how the recursive algorithm works, a recursive approach could possibly be preferred to a stack-based approach for "small sized expressions".

The exact crossover between what is a “small / reasonably sized” vs a “large” problem cannot be stated precisely and will depend on the specifics of the problem at hand. Any final choice of algorithm / determination of the crossover should only be made after timing algorithmic performance on a set of representative test cases.

For “smaller sized problems”, recursion may perform more efficiently for the following reasons:

- Using recursion, additional memory overheads and associated complexity for implementing an iterative approach using a stack ADT (which is in turn implemented using a singly linked list) is eliminated.
- The recursive approach follows the procedure for traversing the binary expression tree using the depth-first post-order traversal, hence this is intuitive and interpretable.
- Although recursive approaches require stack frame calls, redundant stack frame calls (e.g. Fibonacci sequence stack frame calls – refer fig. 1) can be eliminated by storing intermediate subtrees and subtree operations in variables (which is done in the adopted code using variables ‘op’, ‘op1’ and ‘op2’). It is anticipated that the differences in overhead between recursive vs stack-based approaches can be minimised for reasonably sized problems.

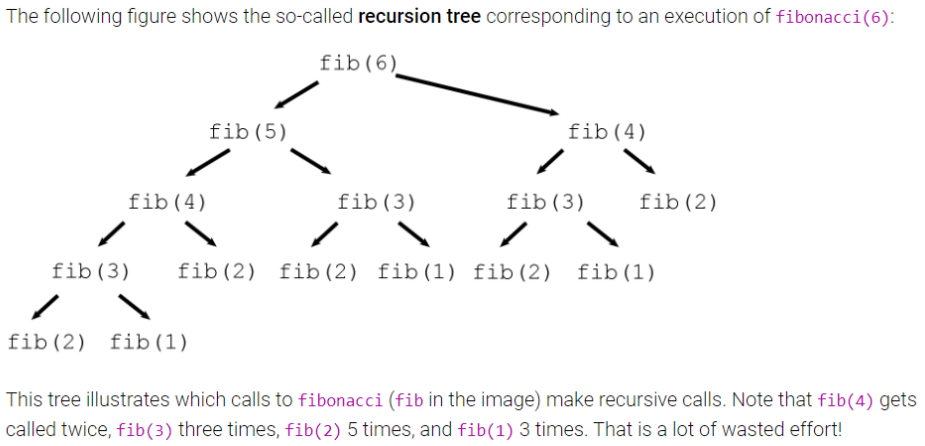

Fig 1. Redundant stack frame calls for Fibonacci(6) (Source: Ref 2)

- The time performance of both recursion and stack based iterative approach is O(n) where n is the size of the prefix expression. However, it is anticipated that the stack based approach may be slightly faster due to the slowing down of the recursive approach when multiple stack frame calls are made. The size complexity of the recursive approach is O(n+m) where m is the number of operators (refer below section on algorithmic complexity derivation). For an equivalent stack based approach, space complexity is O(n).
- When the prefix expressions are very large, the recursive calls made can be very deep and potentially cause stack overflow or result in excessive slow performance. In this case, it is anticipated that the memory overhead from making the stack frame calls (O(n+m) size complexity) starts to negate the advantage of not requiring maintaining state using a stack. In this situation, I would implement an iterative stack-based approach instead of recursion.
- In summary, the differences between a recursive approach and iterative stack based approach are minimal for most practically sized problems since ultimately both approaches are performing the same operations, albeit in different steps. Performance is predicted to diverge once stack frame calls pass a certain threshold, upon which a stack-based approach will be preferred for larger sized problems. Neither approach can be considered 'better' in all categories and an efficient implementation is possible in both cases.

2.2 Time and Space complexity assessment for Recursive approach

Time complexity performance

The recursive algorithm consists of the following operations:

- Recursive traversal of the expression tree. Since intermediate subtrees are stored using "helper" variables, there is minimal redundant function calls and the time performance is proportional to the number of prefix characters in the string, n. Hence time performance is O(n).
- Reading and writing operations. Reading from a string, writing to a string / list and reading from the string / list are all O(n) operations.

- Total time performance is O((X+1).n) where X is the total number of operations per above with a single recursive traversal operation. Therefore, the algorithm has overall simplified O(n) runtime.

Space complexity performance

The space used by the input is O(n) since this is the size of the input expression.

The auxiliary space required in O(m) since for each operator m, a recursive stack frame call is made.

Therefore, total space complexity is O(n+m).

Comparison with Stack-based approach

For the stack-based approach, time performance is O(n). Reading and writing operations are common with the recursive approach. For the actual conversion operation, time complexity is proportional to the number of push / pop operations (each of which have O(1) complexity), hence O(n) complexity in total in proportion to the size of the prefix expression.

For the space complexity, the stack-based approach has O(n) complexity as the expression string determines the size of the stack.

As can be seen, there is minimal difference in the time performances with an advantage to the stack-based approach for space complexity. However, as noted previously, this advantage is offset by the need to use a separate Stack ADT to maintain state.

3. ENHANCEMENTS, LESSONS LEARNT AND FUTURE CONSIDERATIONS

3.1 Incorporated Code Enhancements

The implemented Python code incorporates the following enhancements:

- Alternative approach to implement the building of a binary expression tree and traversing the tree using depth-first post-order traversal to obtain the Postfix conversion. This is to demonstrate that the direct recursive approach is equivalent to traversing the binary expression tree since the output of both approaches is identical. Note: Building a binary expression tree is for learning purposes only - the direct recursive approach eliminates redundancies in time and space complexity from having to build the tree and then separately traversing the tree.
- Evaluation of Prefix integer mathematical expressions to return result. Multiple digit integers may be inputted using specified parentheses bracketing.
- Comprehensive file IO error checking:
  - Incorrect file path. Reproduces input file path for troubleshooting.
  - Incorrect file type for both input and output files.
  - Blank file error due to empty file.
- Comprehensive error checking and reporting for the Prefix expression:
  - Checking of parentheses configuration for implementation of multiple digit integers

- Flexible accommodation of whitespace (whitespace is ignored in the output and the Prefix expression is reported without whitespace)
- Catching of illegal symbols and reporting in the output file
- Checking to ensure that the first and last characters in the Prefix expression are an operator and operand respectively
- Checking for the appropriate positioning of intermediate operators to ensure the recursive operations are fully completed.
- Checking for correct number of operators (one less than total number of operands)

3.2 Lessons Learnt

The following new knowledge was obtained from this programming exercise:

- Understanding how to build a binary expression tree using a node-based class.
- Understanding that the recursive algorithm follows the binary tree post-order traversal.
- Understanding the trade-off impact of auxiliary space usage from triggering stack frame calls in a recursive approach vs penalty of using a stack ADT to maintain the state of an iterative stack-based algorithm.
- Understanding how return calls in a recursive algorithm are tracked and how call by reference vs call by value affects the recursively updated values of variables.
- Understanding how implementing "helper" variables to store intermediate results of recursive function calls improves time performance by eliminating redundant stack frame calls.
- Understanding how different compilers can / cannot implement tail recursion.

3.3 Future Considerations

There is some time and space redundancy in first processing the given prefix expression into a string followed by performing separate error checks on the string.

This time and space redundancy can be eliminated by performing error checks as each character is read directly from the input file i.e. eliminating the intermediate storage list or string, although this may result in more less modularized error checks, more complicated error checking algorithms.

Lastly, I would have liked to run some comparison performance metrics on an implementation of the recursive approach vs stack approach using a varying size of dataset to obtain a feel for the performance.

REFERENCES:

1. Penner, C. (2016, July 26). ***Tail recursion in Python***. Chris Penner Webpage. Retrieved March 28, 2023, from https://chrispenner.ca/posts/python-tail-recursion
2. Rosenbaum, W. (n.d.). ***How Slow is Recursive Fibonacci?*** Will Rosenbaum Webpage. Retrieved March 28, 2023, from https://willrosenbaum.com/teaching/2021s-cosc-112/notes/recursive-fibonacci/