

Assignment 2

October 12, 2024

The assignment is due on **Friday, October 25, 2024, 9:00PM**. Late submissions will be accepted until Monday, October 28, 2024 9:00PM with a penalty of 10%. After that no submissions will be accepted.

1 Written Questions

Question 1 (15 points) :

Write pseudocode for a recursive function that outputs all **permutations** of $[1, \dots, n]$ (the list of all numbers from 1 to n) and justify why it works.

You should use this function prototype:

```
def permutations(n):
```

Question 2 (15 points)

In this problem, you will implement a stack using a single queue. A stack follows the Last In First Out (LIFO) principle, while a queue follows the First In First Out (FIFO) principle. The goal is to simulate stack behavior using queue operations.

Note: Only use one queue to implement the stack. All operations should run in $O(n)$ time, where n is the number of elements in the stack.

Task: Implement a class StackUsingQueue with the following methods:

```
class StackUsingQueue:
    def __init__(self):
        """
        You can assume this Queue class has the following methods:
        pop()      : return the top of the queue and pop it
        top()      : return the top of the queue
        enqueue()  : add an element to the back of the queue
        """
        self.queue = Queue()

    def push(self, element):
```

```

        Add the element to the queue, and
        rotate the element to the front. This
        simulates pushing the element onto the top of the stack.
        '''
        pass

    def pop(self):
        '''Remove the front element.'''
        pass

    def top(self):
        '''Checking the top of the stack. '''
        pass

    def is_empty(self):
        '''Return 0 if the queue is empty.'''
        pass

    def size(self):
        '''Return the number of elements. '''
        pass

```

Example:

```

stack = StackUsingQueue()
stack.push(1)
stack.push(2)
stack.push(3)
print(stack.top())      # Output: 3
print(stack.pop())      # Output: 3
print(stack.top())      # Output: 2
stack.push(4)
print(stack.pop())      # Output: 4
print(stack.is_empty()) # Output: False

```

2 Coding Questions

Question 1 (35 points):

Imagine a new programming language called FractalScript where arithmetic expressions can include recursively nested sub-expressions denoted by square brackets [...]. The result of a sub-expression is squared first before being used in the outer expression.

Examples:

- Input: `"[(2 + 3) * [4 - 1]]"`
Output: 2025
Explanation: This is a standard arithmetic expression, evaluating to $(2 + 3) * (4 - 1) * (4 - 1) = 45$. This is squared to give 2025
- Input: `"[2 + 3]"`
Output: 25
Explanation: The sub-expression $2 + 3$ evaluates to 5, which is then squared 25.
- Input: `"2 + [3 * 4]"`
Output: 146
Explanation: The sub-expression $3 * 4$ evaluates to 12. This is squared to give 144, and then added to 2 to give 146.
- Input: `"[2 + 3 * [4 - 1]]"`
Output: 841
Explanation:

```
[ 2 + 3 * [ 4 - 1 ] ]
[ 2 + 3 * [ 3 ] ]
[ 2 + 3 * 9 ]
[ 29 ]
841
```

The sub-expression $4 - 1$ evaluates to 3, and is squared to give 9. Then $2 + 3 * 9$, evaluates to 29, which is squared to give 841.

- Input: `"[1 + 2 * [3 - [4 / 2]]]"`
Output: 9

Explanation:

```
[ 1 + 2 * [ 3 - [ 4 / 2 ] ] ]
[ 1 + 2 * [ 3 - [ 2 ] ] ]
[ 1 + 2 * [ 3 - 4 ] ]
[ 1 + 2 * [ -1 ] ]
[ 1 + 2 * 1 ]
[ 3 ]
9
```

- Input: `"[[[1 + 2] + 3] + 4]"`
Output: 21904
Explanation:

```
[ [ [ 1 + 2 ] + 3 ] + 4 ]
[ [ [ 3 ] + 3 ] + 4 ]
[ [ 9 + 3 ] + 4 ]
[ [ 12 ] + 4 ]
[ 144 + 4 ]
[ 148 ]
```

- Input: "2 * 3 + [4 - 1] * 5"

Output: 51

Explanation:

```
2 * 3 + [ 4 - 1 ] * 5
2 * 3 + [ 3 ] * 5
2 * 3 + 9 * 5
6 + 45
51
```

Write a Python function `fractal_eval` that takes a string `expr` representing a valid FractalScript arithmetic expression and evaluates it.

The expression may contain:

- Positive integer operands
- The binary operators `+`, `-`, `*`, and `/` (integer division)
- Square brackets `[...]` for recursive sub-expressions
- Parentheses `(...)` for standard grouping and precedence

Standard operator precedence rules apply, with `[...]` having the highest precedence, followed by round brackets, `/` and `*`, then `+` and `-`.

Hint: Operator precedence in an expression with no FractalScript subexpressions can be handled easily with Python's `eval` function. You can use either recursion or a stack based method to evaluate `fractal_eval`. We encourage you to try both ways.

This question will be marked out of 35 for correctness:

- 35/35: Pass all 7 of the provided test inputs.
- 24/35: Pass any 5 of the provided test inputs.
- 14/35: Pass any 3 (or more) of the provided test inputs.
- 7/35: Pass any of the provided test inputs.

Question 2 (35 points)

Overview: Unix paths are used to specify the location of files and directories within a computer's file system. These paths consist of a series of directories separated by forward slashes (/). The paths can also include special components:

- "." refers to the current directory
- ".." signifies the parent directory of the current directory (e.g., the directory above the current one)
- The first / of the path denotes the root directory
- Consecutive forward slashes // are treated as a single slash /

Task: In this question, you will implement a function `simplify_path(path)` that simplifies a given Unix-style file path using a list to represent your stack.

The function must process the path and return the shortest simplified path as a string that:

- Always starts with a / character
- Applies the special characters . or ..
- Contains no consecutive slashes //
- Removes any trailing / characters

If the path does not begin with a /, then return `Invalid Path`.

Examples:

Input: `/home//foo/../bar`

Return: `/home/bar`

Input: `/a/./b/../../../../c/`

Return: `/c`

Input: `/../`

Return: `/`

Input: `/a/b///c/d/././..`

Return: `/a/b/c`

Input: `home/foo/./bar/..`

Return: `Invalid Path`

Specifications:

- Your stack implementation must utilize a list data structure
- Built-in methods for path simplification (such as `os.path`) are **strictly prohibited**
- Use of the `.split()` method is **strictly prohibited**
- The input path will always be a valid string containing only lowercase letters, /, ., and ..
- The input string `path` will have a length of: $1 \leq \text{len}(\text{path}) \leq 100$

Marking Rubric: The code must solve the problem algorithmically. If there is any hardcoding (e.g., `if path == "/foo/bar/../": return "/foo"`) for the provided test cases, zero correctness marks will be given.

This question will be marked out of 35 for correctness:

- 35/35: Pass all 20 of the provided test inputs.
- 24/35: Pass any 19 of the provided test inputs.
- 14/35: Pass any 8 (or more) of the provided test inputs.
- 7/35: Pass any of the provided test inputs.

3 Writing and testing your solution

For the programming questions, you should edit the provided files and add your solution. You can check your solution by running `driver.py`. You can see the output of your program in the **Output/** folder, and any errors in the **Error/** folder.

Please do not change the driver file, only edit the solution file.

You can run the driver in a terminal as follows (assuming your working directory is the assignment folder):

```
$ cd assignment1
$ cd Question1
$ python3 driver.py
All tests passed!
$
```

Please add your name, student number, ccid, operating system and python version at the top of each solution file by replacing the provided comments.

4 Submission Instructions

Please submit a single file to eclass, called `ccid.zip`. Please create a folder with the name as your ccid, put your solutions in that folder, zip it and upload it. It is important for the files to have the same name as described below.

- All written questions should be in one pdf file, ending with `.pdf`.
- Each coding question should be in their own Python file, ending in `.py`.

```
ccid/
  solutionQuestion1.py
  solutionQuestion2.py
  writtenQuestions.pdf
```