HOMEWORK 3 - Fall 2020

HOMEWORK 3 - due Tuesday, October 6th no later than 5:00PM

REMINDERS:

- **Be sure your code follows the coding style for CSE214.**
- **Make sure you read the warnings about academic dishonesty. *Remember, all work you submit for homework or exams MUST be your own work.***
- **Login to your grading account and click "Submit Assignment" to upload and submit your assignment.**
- **You may use (and are encouraged to use) any Java API Data Structures you like to implement this assignment.**
- **You may use Scanner, InputStreamReader, or any other class that you wish for keyboard input.**

Most programming languages are organized as structured blocks of statements, with some blocks nested within others. Functions, which are examples of such blocks, execute statements and other blocks contained within them. Similarly, flow control structures, such as `for` and `while` loops, are blocks which can be executed several times subject to some condition. The Python programming language is an example of a language which follows this principle, and is even flexible enough to allow functions to be nested within functions!

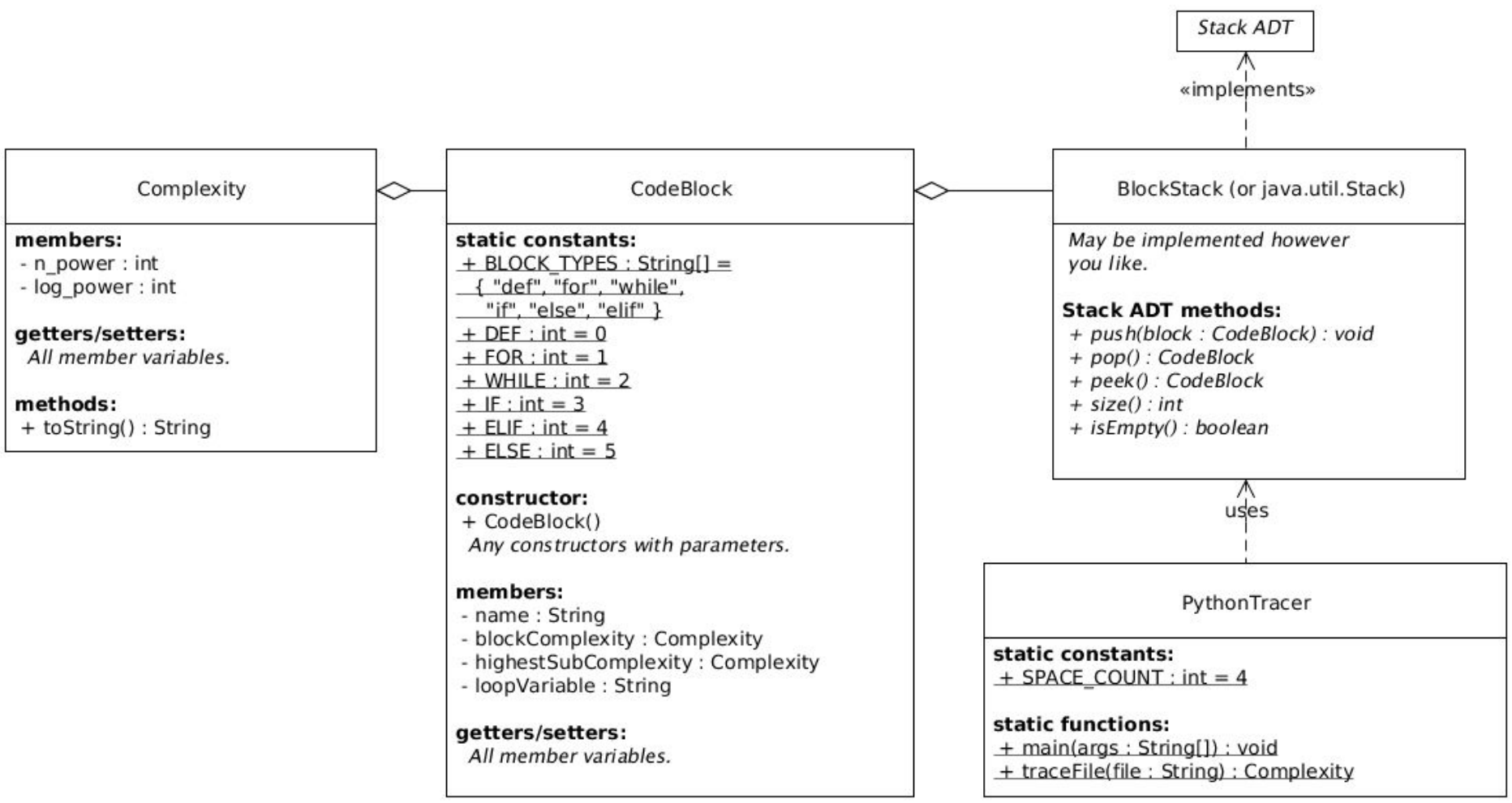
In this homework assignment, you will create a code tracer program which takes the name of a Python file containing a single function and outputs the Big-Oh order of complexity of that function. To make things easier, several restrictions will be made on the format of the input code, and some techniques for text parsing will be described below. You must implement a `BlockStack` class to determine the complexity of blocks with nested blocks, and use the rules of Big-Oh complexity to determine the total complexity for the function. You are encouraged to use the Java API to help you complete this assignment, but you may implement the stack however you like.

It should be noted that you do not need to actually learn Python to complete this assignment - simply following the instructions as detailed in this specification should suffice. However, if you would like some background on language, a good starting point would be the Python website, which should provide you with enough information to get a feel for how the language works.

Required Classes

The following sections describe classes which are required for this assignment. Each section provides a description and the specifications necessary to complete each class. If you feel that additional methods would be useful, you may feel free to add them during your implementation as you see fit. However, all the variables and methods in the following specifications must be included in your project.

UML diagram of the Java class structure.



1. Complexity

Write a fully-documented class named `Complexity` which represents the Big-Oh complexity of some block of code. Since Big-Oh notation can get quite messy (*e.g. log(n^(1/2) / n!)*), we will restrict the possible orders to powers of two base types: `n`, and `log_n`. Following this practice, include two member variables in the `Complexity` class: `nPower` (int) and `logPower` (int). These two variables will keep track of what power each of the base types is present in the complexity object.

For example, a Complexity object with `nPower= 4` and `logPower = 2` would represent the complexity `O(n^4 * log(n)^2)`. Similarly, a Complexity object with `nPower = 1` and `logPower = 1` would represent the complexity `O(n * log(n))` (note that the exponents only appear when the power value is greater than 1). A case to consider is the situation in which both variables are 0, which indicates that the complexity is `O(1)`.

- **`public Complexity` - constructor (you may include a constructor with parameters)**
- **Two int member variables: `nPower` and `logPower`.**
- **Getter and setter methods for both member variables.**
- **A toString() method which prints human-readable Big-Oh notation. (e.g `O(n^4 * log(n)^2)`.**

2. CodeBlock

Write a fully-documented class named `CodeBlock` which describes a nested block of code. There are different types of code blocks you must consider (*e.g. a `for` block, a `while` block, etc.*), so create a static final array of six String variables named `BLOCK_TYPES` to enumerate the types of blocks available for nesting. `"def", "for", "while", "if", "elif", "else"`. You should also declare six static final int variables corresponding to the indices of the `BLOCK_TYPE` array: `DEF = 0, FOR = 1, WHILE = 2, IF = 3, ELIF = 4, ELSE = 5`.

NOTE 1: When parsing Python code for the keywords, be careful not to accidentally parse the "for" in "fortune" (which might be a variable name) as the start of a `for` block. To avoid this, make sure there is a single space (`" "`) before and after the keyword before starting a new block.

NOTE 2: An alternative to the above combination of a single static array `BLOCK_TYPES` and six static final ints representing the indices is to use a Java `Enum` type. It is not required for this assignment, but you may find it to provide a simpler and more elegant solution.

Include a member variable `blockComplexity` (Complexity) to keep track of the Big-Oh complexity of this block, and a member variable `highestSubComplexity` (Complexity) to keep track of the Big-Oh complexity of the highest-order block nested within this block. The difference between these Complexity objects is that the `blockComplexity` represents the order of the block ignoring the statements inside (*e.g. `O(n)` for a `while` block looping from n to 1*), whereas the `highestSubComplexity` represents the highest complexity of all the blocks nested inside this block.

Lastly, include two `String` member variables `name` and `loopVariable`. The `name` member will be used to keep track of the nested structure of the blocks. The first block in the stack will always be named "1". All blocks included directly under a block will be numbered increasingly using a dot "." separator after the block's own name (e.g. blocks nested under block 1 will start with "1.1" and proceed to "1.2", "1.3", etc). Similarly, all blocks included directly under the block named "1.2" will be numbered "1.2.1", "1.2.2", "1.2.3", etc. For more detail, see the sample I/O below. The `loopVariable` member will only be used for `while` blocks in this assignment, as `for` blocks will not alter their variable during execution in the input code. In the constructor, this variable can be initialized to `null` and should only be updated when a `while` block is traced and the name of it's loop variable has been determined. See the sample algorithm below for further information.

Include getter and setter methods for all member variables.

- **`public CodeBlock` - constructor (you may include a constructor with parameters)**
- **A static final array of six String constants `BLOCK_TYPES` representing the types of code blocks:**
  - `public static final String BLOCK_TYPES = {"def", "for", "while", "if", "elif", "else"};`
- **Six static final int variables corresponding to the indices of the `BLOCK_TYPES` array:**
  - `public static final int DEF = 0, FOR = 1, WHILE = 2, IF = 3, ELIF = 4, ELSE = 5;`
- **Two Complexity member variables:**
  - `private Complexity blockComplexity;`
  - `private Complexity highestSubComplexity;`
- **Two String member variables:**
  - `private String name;`
  - `private String loopVariable;`
- `Getter and setter methods for each member variable.`

3. BlockStack (or java.util.Stack)

Since the complexity of each `CodeBlock` depends on the complexity of any `CodeBlocks` nested inside of it (following standard order of complexity rules), your tracer will need to use a Stack to determine the total complexity of a function. In this assignment, you can write your own Stack class or you may use the standard Stack class that is in the java.util package. Go to the HELP section and view the online documentation for Stack from the Java API.

CAUTION: Although the Stack class has push, pop and peek methods, the Stack class is a subclass of Vector. Therefore, all of the methods of Vector are also accessible in the Stack class. However, if you use any of the inherited Vector methods in your solution, you will be penalized in this assignment, since some of the Vector methods are not supposed to apply to a Stack ADT in general. (That is, the designers of Java basically define that a Stack is a special type of Vector, but it really isn't.)

4. PythonTracer

Write a fully-documented class named `PythonTracer` which contains a `main` method. The class will also contain a static final int variable `SPACE_COUNT = 4` which will be used to determine the indentation of each statement (see below). In addition, include a static `traceFile` method, which will take as a parameter the name of a file containing a Python function. The `traceFile` method will open the indicated file, trace through the code of the Python function contained within the file, and output the details of the trace and the overall complexity to the console.

During the operation of the `traceFile` method, the user should be updated to the changes being made to the stack via a *stack trace*. Every time a new block is pushed on to the stack, the block's name (see the sample I/O for naming convention), complexity, and highest sub-complexity should be printed to the console. In addition, every time the Complexity at the top of the stack is updated, the new values should be printed to the console. Lastly, after a block is popped from the stack, the new top of the stack should be updated (it's `highestSubComplexity` variable should be changed if necessary), and it should be printed to the console, regardless of whether it was changed or not. These operations correspond to 'entering' a new block, determining the complexity of the current block, and 'leaving' a block, respectively. For more detail, see the sample I/O below.

- **`public static final int SPACE_COUNT = 4;`**

- **`public static Complexity traceFile(String filename)`**
  - ***Brief:***
    - **Opens the indicated file and traces through the code of the Python function contained within the file, returning the Big-Oh order of complexity of the function. During operation, the stack trace should be printed to the console as code blocks are pushed to/popped from the stack.**
  - ***Preconditions:***
    - **`filename` is not `null` and the file it names contains a single Python function with valid syntax (Reminder: you do NOT have to check for invalid syntax).**
  - ***Returns:***
    - **A `Complexity` object representing the total order of complexity of the Python code contained within the file.**
- **`public static void main(String[] args)`**
  - ***Brief:***
    - **Prompts the user for the name of a file containing a single Python function, determines its order of complexity, and prints the result to the console.**

Trace Algorithm

A high-level algorithm for the `traceFile` method is given below:

```
Initialize stack to an empty stack of CodeBlocks.
 Open file using filename.
 while file has lines
  line = next line in file.
 if line is not empty and line does not start with '#'
    indents = number of spaces in line / SPACES_COUNT.
   while indents is less than size of stack
      if indents is 0
         Close file and return the total complexity of stack.top.
      else
         oldTop = stack.pop()
         oldTopComplexity = total complexity of oldTop
        if oldTopComplexity is higher order than stack.top's highest sub-complexity
           stack.top's highest sub-complexity = oldTopComplexity
   if line contains a keyword
      keyword = keyword in line.
      if keyword is "for"
         Determine the complexity at end of line ("N:" or "log_N:")
         Create new O(n) or O(log(n)) CodeBlock and push onto stack.
      else if keyword is "while"
         loopVariable = variable being updated (first token after "while").
         Create new O(1) CodeBlock with loopVariable and push onto stack.
      else
         Create new O(1) CodeBlock and push onto the stack.
   else if stack.top is a "while" block and line updates stack.top's loopVariable
      Update the blockComplexity of stack.top.
```

```
    else
        Ignore line.
while size of stack > 1
    oldTop = stack.pop()
    oldTopComplexity = total complexity of oldTop
    if oldTopComplexity is higher order than stack.top's highest sub-complexity
        stack.top's highest sub-complexity = oldTopComplexity
Return stack.pop().
```

Input Code Restrictions

Instead of enclosing blocks of code within curly braces (as is the case in Java and other C-style languages), Python delineates blocks of code using whitespace. In Python code, statements in the same block must have the exact same number of spaces or tabs preceding them. Once a new block is declared (via a new `def`, `for`, `while`, `if`, `else`, or `elif`), all statements contained within the new block must be indented by AT LEAST one additional whitespace character, whether it be a space (`" "`) or a tab (`"\t"`). Once this new block ends, the following statements must have the exact same indentation as the previous statements in the block (see the code in python_example.py for an extensive example of the syntax).

In this assignment, every instance of `def`, `for`, `while`, `if`, `elif` and `else` contains all following statements that are indented exactly 4 spaces further than the block declaration. Although valid Python code may have any system of consistent indentation, it is conventional to use 4 spaces or a single tab. All Python code in this assignment will have 4 spaces as an indentation marker for blocks, which will be stored in the `SPACES_COUNT` variable.

To determine which block a statement is in, count the number of spaces (character ' ') occurring before the statement, and divide this number by `SPACES_COUNT`, which is defined in the `PythonTracer` class. If it is equal to the size of the current `BlockStack`, it is part of the `CodeBlock` at the top of the stack. If this number is less than the size of the stack, pop `CodeBlocks` off of the stack until its size is equal to this number. If this number is greater than the current size of the stack, then it is a syntax error (you do not have to check for erroneous code in this assignment, all test files will contain valid Python code).

Comments in Python come in two varieties. There are line comments, which start at the first instance of "#" and continue to the end of the line (similar to "//" comments in Java). There are also block comments, which start with three consecutive double-quote characters (""") and continue until the next three consecutive double-quote characters (similar to "/* */" comments in Java). This assignment will only contain line comments, and all line comments will exist on a single line (there will be no comment on the same line as any statement). While parsing, you should ignore ANY line that contains at least one "#" character.

When a new block is traced, a new `CodeBlock` object should be created, and it's `blockComplexity` variable should be initialized to some value depending on the block type. All `def`, `if`, `elif`, and `else` blocks should have a `blockComplexity` of `O(1)`. However, `for` blocks and `while` blocks will have varying complexity, and it is up for you to determine the complexity of the block.

In this assignment, all `for` blocks will be of the following variety, where `*` represents some variable name:

```
for * in N                  for * in log_N
            or
  . . .                       . . .
```

The complexity of `for` blocks iterating `*` over `N` is `O(N)`. Similarly, the complexity of `for` blocks iterating `*` over `log_N` is `O(log(N))`.

In addition, in this assignment, all `while` blocks will loop from `n` to `1` over some variable. Since the complexity of `while` blocks can only be determined by their update statement, the `blockComplexity` of `while` blocks should be initialized to `O(1)` and changed after the update statement has been located. The update statement will be located within the block and can be either of the following:

```
* = n                       * = n

while (* > 1)                 while (* > 1)
            or
  . . .                       . . .

  * -= 1                      * /= 2
```

`while` blocks which decrement their counter will have an order of complexity of `O(n)`, whereas `while` blocks dividing their counter by 2 will have an order of complexity of `O(log(n))`.

Reading From a File

The file will contain a Python function in the following format (note the number of spaces preceeding each statement):

```
def function(n):                    // Start of the function. (Block 1)
    N = xrange(n);
    log_N = xrange(int(math.log(n, 2)))
    for i in N:                     // First block inside Block 1 (Block 1.1).
        print(i)
```

```
    for j in log_N:                  // Second block inside Block 1 (Block 1.2).
        for k in N:                  // First block inside Block 1.2 (Block 1.2.1)
            print(i, k)
    . . .
    . . .
```

You may use the following code to open an input stream for reading from a text file:

```
FileInputStream fis = new FileInputStream(fileName);

InputStreamReader inStream = new InputStreamReader(fis);

BufferedReader reader = new BufferedReader(inStream);
```

Once the stream is open, you can read one line at a time as follows:

```
String data = reader.readLine();
```

You can also create the reader using a scanner:

```
    Scanner reader = new Scanner(new File(fileName));
```

Check the documentation for BufferedReader or Scanner.

String Handling Methods/Examples

This homework requires string handling and manipulation. Here are some of the String class methods that you may need. For a more thorough explanation please refer to the Java API documentation on Strings here.

- `char charAt(int index)`
  Returns the char value at the specified index.
- `String substring(int beginIndex)`
  Returns a new string that is a substring of this string.
- `String substring(int beginIndex, int endIndex)`
  Returns a new string that is a substring of this string.
- `String trim()`
  Returns a copy of the string, with leading and trailing whitespace omitted.
- `boolean equals(Object anObject)`
  Compares this string to the specified object.
- `boolean equalsIgnoreCase(String anotherString)`
  Compares this string to the specified string, ignoring case.
- `int indexOf(String str)`
  Returns the index within this string of the first occurrence of the specified character.
- `int length()`
  Returns the length of this string.
- `String[] split(String regex)`
  Splits this string around matches of the given regular expression.

Here are some examples of how you might use some of these methods:

```
String testString1 = " This is an example String. "
 String testString2 = "cat"

 testString1.charAt(9) //Will return 'a'
 testString1.trim() //Will return “This is an example String.”
 testString1.substring(5) //returns " is an example String."
 testString1.substring(5, 9) //returns " is ", note that the endIndex is not included in the returned string
 testString1.substring(5, 9).trim() //returns "is"

 testString2.equals("cat") //returns true
 testString2.equals("bat") //returns false

 testString1.indexOf("amp") //returns 14
 testString1.length() //returns 28
 testString2.length() //returns 3

 testString1.split(" ") //returns an array of size 5 consisting of "This", "is", "an", "example", and "String."
```

String manipulation is a very important programming skill. Try to come up with some clever and elegant solutions to String parsing using these methods. As an example, a clever way to count the number of leading spaces in a String is to use a combination of the `indexOf()` method and the `trim()` method.

```
// 's' contains the line of text to be parsed.
 int space_count = s.indexOf(s.trim());
```

Sample Input/Output:

// Comment in green, input in red, output in black

[test_function.py](#)

```python
# This function prints some statements in O(n * log(n)) time.
def test_function(n):

    # Get the range [0, n-1].
    N = xrange(n)
    # Get the range [0, floor(log(n))-1]
    log_N = xrange(int(math.log(n, 2)))

    # Stack record that loops from 0 to n-1.
    for i in N:

        # Nested stack record that loops from 0 to floor(log(n))-1.
        for j in log_N:

            print("This statement prints n * log(n) times.")

    # All 'while' statements will have a variable go from 'n' to 1.
    k = n
    while k > 1:

        print("But this statement only prints log(n) times.")

        # But you will have to determine the order from the
        # update statement (log(n), in this case).
        k /= 2

    print("Since n * log(n) is bigger complexity, the whole "
          "funtion is O(n * log(n).")

```

```
Please enter a file name (or 'quit' to quit): test_function.py

    // New 'def' block parsed, O(1) by default.
    Entering block 1 'def':
     BLOCK 1:       block complexity = O(1)       highest sub-complexity = O(1)

    // New 'for' block parsed, determined to go from 1 to n.
    Entering block 1.1 'for':
        BLOCK 1.1:     block complexity = O(n)       highest sub-complexity = O(1)

    // New 'for' block parsed, determined to go from 1 to log(n).
    Entering block 1.1.1 'for':
        BLOCK 1.1.1:   block complexity = O(log(n))  highest sub-complexity = O(1)

    // Updating highest sub-complexity of Block 1.1 from O(1) to O(log(n)).
    Leaving block 1.1.1, updating block 1.1:
     BLOCK 1.1:     block complexity = O(n)       highest sub-complexity = O(log(n))

    // Updating highest sub-complexity of Block 1 from O(1) to O(n * log(n)).
    Leaving block 1.1, updating block 1:
        BLOCK 1:       block complexity = O(1)       highest sub-complexity = O(n* log(n))

    // New 'while' block parsed, is O(1) until update statement is found.
    Entering block 1.2 'while':
        BLOCK 1.2:     block complexity = O(1)       highest sub-complexity = O(1)

    // Update statement parsed, block complexity changed from O(1) to O(log(n)).
    Found update statement, updating block 1.2:
        BLOCK 1.2:     block complexity = O(log(n))  highest sub-complexity = O(1)

    // Complexity of Block 1.2 O(log(n) is less than current highest sub-complexity of Block 1 (O(n
* log(n)))
    Leaving block 1.2, nothing to update.
        BLOCK 1:       block complexity = O(1)       highest sub-complexity = O(n * log(n))

    Leaving block 1.

Overall complexity of test_function: O(n * log(n))
```

[matrix_multiply.py](#)

```python
# This function multiplies two n by n matrices in O(n^3) time.
def matrix_multiply(a, b, n):

    # Get the range [0, n-1].
    N = xrange(n)

    # Create an N x N output matrix 'c' full of 0's.
    c = [0 * n] * n

    # For each row 'i' in 'a'...
    for i in N:

        # For each row 'j' in 'b'...
        for j in N:

            # Store dot product of a[i][:] with
            # b[:][j] in c[i][j].
            for k in N:

                c[i][j] += a[i][k]*b[k][j]

    # Return the matrix.
    return c

```

```
// No comments here, try to follow the stack trace using the code above.
Please enter a file name (or 'quit' to quit): matrix_multiply.py

    Entering block 1 'def':
     BLOCK 1:       block complexity = O(1)       highest sub-complexity = O(1)

    Entering block 1.1 'for':
        BLOCK 1.1:     block complexity = O(n)       highest sub-complexity = O(1)

    Entering block 1.1.1 'for':
        BLOCK 1.1.1:   block complexity = O(n)       highest sub-complexity = O(1)

    Entering block 1.1.1.1 'for':
        BLOCK 1.1.1.1: block complexity = O(n)       highest sub-complexity = O(1)

    Leaving block 1.1.1.1 updating block 1.1.1:
        BLOCK 1.1.1:   block complexity = O(n)       highest sub-complexity = O(n)

    Leaving block 1.1.1 updating block 1.1:
        BLOCK 1.1:     block complexity = O(n)       highest sub-complexity = O(n^2)

    Leaving block 1.1 updating block 1:
        BLOCK 1:       block complexity = O(1)       highest sub-complexity = O(n^3)

    Leaving block 1.

Overall complexity of matrix_multiply: O(n^3)

Please enter a file name (or 'quit' to quit): quit
Program terminating successfully...
```
