**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."
```

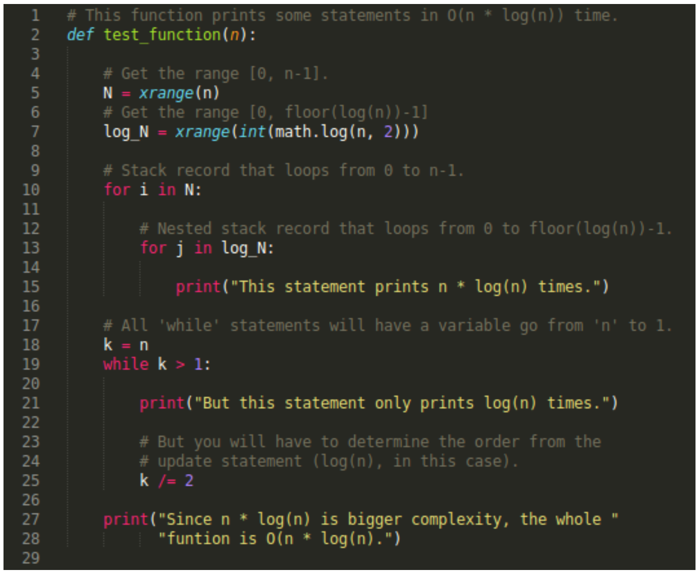
**String manipulation is a very important programming skill. Try to come up with some clever and elegent solutions to String parsing using these methods. As an example, a clever way to count the number of leading spaces in a String is to use a combination of the `indexOf()` method and the `trim()` method.**

```
// '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**

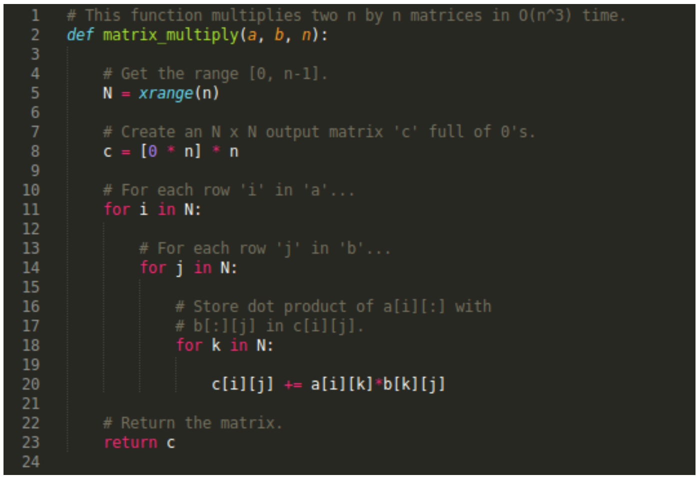
**test_function.py**



```
# 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...
```

Course Info | Schedule | Sections | Announcements | Homework | Exams | Help/FAQ | Grades | HOME