The University of Texas
Computer Science

313E Programming Assignment 04
Big O

Setup

This assignment may be completed individually or with a pair programming partner. Be sure to include your and your partner's name and EID in the Python file header. If you are working alone, you may delete one of them.

Complete these steps to prepare for the Assignment.

Check	Description
▢	Download bigo.py, test_bigo.py, and time_graph.py. You will be working on the bigo.py file.
▢	Place all files in the same folder/directory.
▢	You may not change the file names. Otherwise, the grading script will not work.

To visualize your code's execution time, you must install matplotlib. To do so, run the command in a terminal.
For Mac/Linux:
```
python3 -m pip install matplotlib
```

For Windows, if your python3 is installed as python:
```
python -m pip install matplotlib
```

If you would like to turn on autosave in VSCode, click on File -> Autosave.

Problem Description

To show the impact of Big O, we will be writing three solutions to the same problem. We will show how small changes can make big differences in Big O. One solution will have time complexity O(N^3), another will have time complexity O(N^2), and the last solution will have time complexity O(N). The problem is:

Given a string s, find the length of the longest substring without repeating characters.

A **substring** is defined as a contiguous, non-empty sequence of characters within a string. You can obtain a substring via slicing; such as in the below snippet:

```
my_str = "Thirty Six All Terrain Tundra Vehicle"
my_substr = my_str[7:10]
```

```
print(my_substr)
# The above line prints "Six"
```

For the purposes of this class, assume that creating a substring via slicing (e.g. line 2 of the above snippet) has time complexity O(k), where *k* is the length of the requested substring.

Note that the above snippet is best for making a *copy* of a substring. If you simply need to analyze a substring without requiring a copy, as you'll be doing in this project, then you can logically define a substring as two indexes i and j (where i is the starting index, and j is the ending index of the substring).

Example 1:

Input: s = "abcabcbb"
Output: 3
Explanation: The answer is "abc", "bca", or "cab", each with a maximum length of 3.

Example 2:

Input: s = "bbbbb"
Output: 1
Explanation: The answer is "b", with the length of 1.

Example 3:

Input: s = "pwwkew"
Output: 3
Explanation: The answer is "wke" or "kew", each with the length of 3.
Notice that the answer must be a substring, "pwke" is a subsequence and not a substring, and a substring must have all characters next to each other.

Approaches

Below are suggested approaches for achieving the required big O time complexities. You are welcome to try other approaches or deviate slightly from our recommendations, but to receive credit for your code, **you must achieve the required big O**.

You will modify the following functions to implement the functionality described above.

length_of_longest_substring_n3(s)
This function accepts a string s and should return the length of the longest substring of s which has all non-repeating characters. The Big O for this function must be O(N^3).

This approach is called the brute force approach. We will be using nested loops to generate every possible substring. For each generated substring, we will be using a frequency list to check if it is a substring without repeating characters, and if the substring is valid, then we will update the maximum length.

1. Loop over each possible starting index for a substring.
2. Make a nested loop that begins at the starting index, and ends at the length of s, representing the ending index.
3. In the nested loop
    a. Create a list of size 256 of all zeros, which will represent our frequency list.

b. Create the substring using your starting and ending index (both should be included)
c. Check the entire substring, updating the frequencies of each character of the substring in the frequency list. This can be done by converting the character into the corresponding numeric representation, and using that numeric representation as an index. Increment that index by 1 to represent the character count. If you are unfamiliar with how to convert to the numeric representation, please refer to Chapter 1: Extended Review, Strings.
d. If the frequency list has no frequencies greater than 1, update the maximum length if the current length is longer.

Note: you may be wondering whether creating a frequency list of size 256 will make the Big O worse. However, regardless of the size of this function's input string, the size of our frequency list never changes – it is always of size 256. Creating a list of size 256 is a *constant time* operation, so it will not change the Big O of the function.

length_of_longest_substring_n2(s)
This function accepts a string s and should return the length of the longest substring of s which has all non-repeating characters. The Big O for this function must be O(N^2).

Starting at each index, maintain a frequency list to track occurrences of characters as you expand the substring. We no longer create a literal substring using slicing.

1. Loop over each possible starting index for a substring.
2. Create a list of size 256 of all zeros, which will represent our frequency list.
3. Make a nested loop that begins at the starting index, and ends at the length of s, representing the current index.
4. In the nested loop:
    a. Increment the frequency of the character at the current index in the frequency list by 1. This can be done by converting the character into the corresponding numeric representation, and using that numeric representation as an index to the frequency list. Increment that index by 1 to represent the character count.
    b. If the frequency list has no frequencies greater than 1, update the maximum length of the substring if the current length is longer.

length_of_longest_substring_n(s)
This function accepts a string s and should return the length of the longest substring of s which has all non-repeating characters. The Big O for this function must be O(N). You may choose to challenge yourself by implementing the sliding window approach for this problem, which is O(N).

Starting at each index, maintain a frequency list to track occurrences of characters as you expand the substring. The main difference between the O(N^2) approach and this O(N) approach is that we stop early when a duplicate character is detected.

5. Loop over each possible starting index for a substring.
6. Create a list of size 256 of all zeros, which will represent our frequency list.
7. Make a nested loop that begins at the starting index, and ends at the length of s, representing the current index.
8. In the nested loop:
    a. Increment the frequency of the character at the current index in the frequency list by 1. This can be done by converting the character at the current index into the corresponding numeric representation, and using that numeric representation as an index to the frequency list. Increment that index by 1 to represent the character count.

b. If the frequency was updated to be greater than 1, break out of the loop immediately.
c. Otherwise, update the maximum length of the substring if the current length is longer.

Requirements

Your three solutions will each be in their own functions.

1. The solutions you implement must each have the Big O time complexities corresponding to the requirements above. You may choose to use the visualizer we provide to see if the Big O looks correct.
2. **You may not change the names of the functions listed. They must have the functionality as given in the specifications. You can always add more functions than those listed.**
3. **You may not import any additional external libraries in your solution.**

Testing Framework

We will provide you with test cases in test_bigo.py. This file contains all visible test cases for this assignment, and it uses the Python `unittest` test framework. This framework uses the following format:

```
import unittest
import sys

from bigo_solution import (
    length_of_longest_substring_N3,
    length_of_longest_substring_N2,
    length_of_longest_substring_N,
)


class TestLongestSubstringN3(unittest.TestCase):
    """Test O(N^3) implementation."""

    def test_length_of_longest_substring_n3_1(self):
        """O(N^3) approach with input 'bevo', answer is 4."""
        self.assertEqual(length_of_longest_substring_n3("bevo"), 4)

def main():
    unittest.main()

if __name__ == "__main__":
    main()
```

Each test is contained in its own method, and the **name of each method must begin with "`test`"**. Each function has its own test *class*. You'll learn more about Python classes this semester, but for now just know that the class serves as a way to group together all our tests.

The tests each check if the expected output matches the actual output (i.e. the output of your traversal.py functions when run with the given test input). If these values match for all tests, then you will see something like the output below (note that you'll see a lot more than just one test!).

```
----------------------------------------------------------------------
Ran 1 test in 0.000s
OK
```

If the expected and actual values do not match on one or more tests, then error(s) will be thrown:

```
F
======================================================================
FAIL: test_length_of_longest_substring_n3_1 (__main__.TestLongestSubstringN3)
O(N^3) approach with input 'bevo', answer is 4.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_bigo.py", line 15, in test_length_of_longest_substring_n3_1
    self.assertEqual(length_of_longest_substring_n3("bevo"), 4)
AssertionError: None != 4

----------------------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=1)
```

Throughout your implementation of all your programming assignments, we highly encourage you to take advantage of the existing test framework by writing your own tests. For whichever of the function(s) you'd like to test, simply create a new `test` method within the corresponding class and create your testing parameters (e.g. a grid for the row_traversals function). Within this test method, provide the expected (correct) output of the test, then check to see if your actual output matches the expected output.

1. Running All Tests:

If you run the script without any arguments, all the test cases defined in the file will be executed:

```
python3 test_bigo.py
```

2. Running All Tests for a Function:

If you run the script with the test name, it will run only the test cases for that function:

```
python3 test_bigo.py [test_name]
```

Example:

```
python3 test_bigo.py n3
```

3. Running A Specific Test:

If you'd like to run a specific traversal test, you can provide the name of the test followed by the grid dimensions for that test. The format for this command is:

```
python3 test_bigo.py [test_name] [test_number]
```

Example:

```
python3 test_bigo.py n3 1
```

This command will run the length_of_longest_substring_N3() function for input from test 1.

4. Command-Line Arguments

- **`test_name`**: The name of the specific traversal test you want to run. The valid options are:
  - n3: length_of_longest_substring_N3() tests
  - n2: length_of_longest_substring_N2() tests
  - n: length_of_longest_substring_N() tests
- **`test_number`**: There are 10 tests numbered 1-10 for N^3, 20 tests numbered 1-20 for N^2, and 30 tests numbered 1-30 for N.

Execution Time Visualizer

This assignment comes with a visualizer to show off your hard work! It uses **`matplotlib`** to plot the total time it takes your code to run for increasing input sizes onto a graph.

To visualize your code's execution time, you must install matplotlib. To do so, run the command in a terminal:

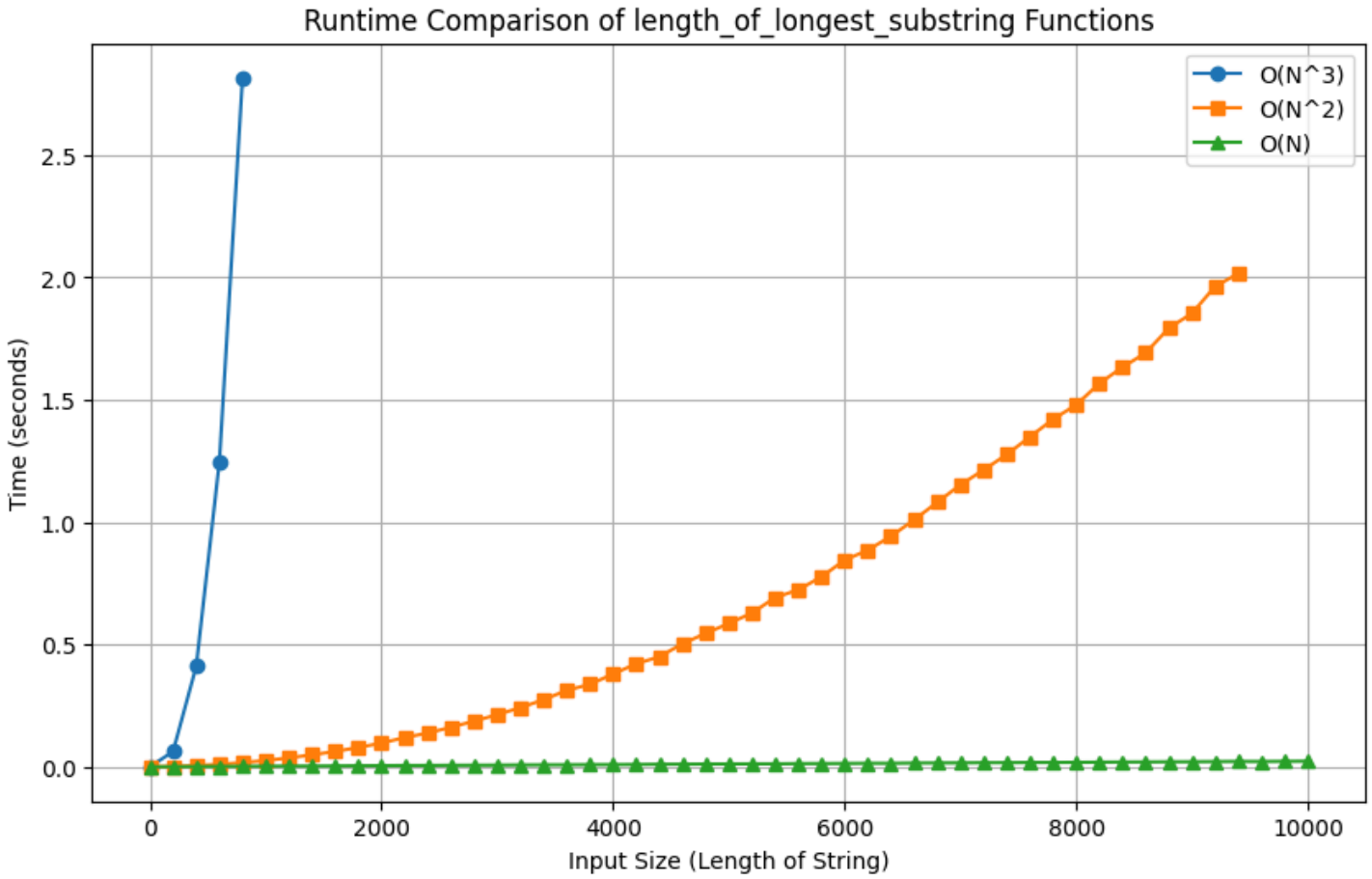
For Mac/Linux:
`python3 -m pip install matplotlib`

For Windows, if your python3 is installed as python:
`python -m pip install matplotlib`

You can run your code by doing:

```
python3 time_graph.py
```

On our laptops, this takes about 41 seconds to run. When the command is completed, it will pull up a chart showing the total time each one of your functions runs. Your code should look something similar to this:



While it may not be exactly the same, the general trends should be the same. If yours does not match, it may be because:

- Your Big O is incorrect for some functions.
- Your computer is being overworked, and it could be impacting the timing.
- Your code has a bug.

Consider rerunning it again and seeing if the trend changes, which may just mean your computer is being overworked.

Grading

Visible Test Cases - 60/100 points

The program output exactly matches the sample solution's output. To receive full credit, the program must produce the correct output based on all requirements in this document and pass all the test cases.

Hidden Test Cases - 30/100 points

The program output exactly matches the sample solution's output. To receive full credit, the program must produce the correct output based on all requirements in this document and pass all the test cases.

Style Grading - 10/100 points

The bigo.py file must comply with the PEP 8 Style Guide.

If you are confused about how to comply with the style guide, paste the error or the error ID (e.g. C0116 for missing function or method docstring) in the search bar here in the Pylint Documentation.

The TAs will complete a manual code review for each assignment to confirm that you have followed the requirements and directions on this document. Deductions will occur on each test case that fails to follow requirements.

Submission

Follow these steps for submission.

Check	Description
▢	Verify that you have no debugging statements left in your code. These will cause your test cases to fail.
▢	Run the visible test cases on your implementation. This is recommended prior to submission, as you have a maximum of 15 tries to test your code on the hidden test cases via Gradescope.
▢	Ensure that you are following PEP 8 style guidelines via Pylint. ***NOTE***: To receive full credit for style points on your programming assignment, you must have addressed ALL problems detected by Pylint.
▢	Submit bigo.py to the assignment in Gradescope. You **must** submit via Github, regardless of if you are in a group. To submit as a group, follow the instructions here. This will run the grading scripts (hidden + visible tests).
▢	When the grading scripts are complete, check the results. If there are errors, evaluate the script feedback, fix your code, test your code, and then re-submit the file to Gradescope. You can submit on Gradescope up to 15 times until the due date. **You will not be able to submit after your 15th submission**. So, if you're failing hidden test cases, try to narrow down and fix your bug prior to submitting! More techniques on this are described in the Gradescope guide here and in the VS Code Debugger guide here. ***NOTE***: By default, only the most recent submission will be considered for grading. If you want to use a previous submission for your final grade, you must activate it from your submission history on Gradescope before the due date.

Academic Integrity

Please review the Academic Integrity section of the syllabus. We will be using plagiarism checkers as well as checking for AI-generated code. Remember, the goal in this class is not to write the perfect solution; we already have many of those! The goal is to learn how to problem-solve, so:

- don’t hesitate to ask for guidance from the instructional staff, and
- be sure you stay within the Academic Integrity discussion guidelines outlined in the syllabus.

Attribution

Thanks to Dr. Carol Ramsey for the instruction template.