UF UNIVERSITY of FLORIDA
Herbert Wertheim College of Engineering

COP3504C

Lab 04: Time Complexity & Profiling

Overview

This project is designed to familiarize students with "Big-O" notation through the algorithm analysis. Algorithms have an associated "Big-O" notation that describes how they run according to the input size. This models how fast the number of calculations grows as the size of the input grows larger. Programs with low growth rates are preferred because, no matter the size of the input, the program will handle it in a timely manner. In this lab exercise, students will compare two different search algorithms by timing them reflecting on how their time complexities compare to their runtimes.

Specification

You will write a main method and two static methods implementing binary and linear search algorithms. The linear search will step through the elements one at a time until the correct element is found; the binary search will operate on a pre-sorted set and use this knowledge to divide the search space in half at each step. Students are recommended to review the algorithms listed in their textbooks, as they are implemented directly.

Provided Implementation

Students are provided with the **`stringdata`** module. It contains a single data element:

`dataset``: tuple(str)`
This tuple is the complete set of possible lowercase 5 character strings (containing 17,576 entries).

Student Implementation

Students will implement the **`analyzer`** module with a main entry point and two search methods as follows.

Entry Point (main)

This module will have a **`main()`** function which will execute if the module is directly invoked. The main function should complete the following steps, documenting results and answer questions in the write-up:

1) Access the data set from the **`stringdata`** module (via the **`dataset`** value).
2) Search for **`"not_here"`** using both algorithms. Capture the time each method requires to execute.
3) Search for **`"mzzzz"`** using both algorithms. Capture the time each method requires to execute.
4) Search for **`"aaaaa"`** using both algorithms. Capture the time each method requires to execute.

Search Methods

`linear_search``(container: tuple(str), element: str) -> int`
Uses linear search to find specified **`element`** in **`container`**. Returns index of element, or **`-1`** if not found.

`binary_search``(container: tuple(str), element: str) -> int`
Uses binary search to find specified **`element`** in **`container`**. Returns index of element, or **`-1`** if not found.

Timing your Methods

To time your methods, use the **`time()`** function in Python's **`time`** module – i.e., **`time.time()`**. It reports the time in seconds as a floating-point number.

1) Capture the begin time using **`time.time()`**.
2) Run the method.
3) Immediately after running, capture the ending time.
4) Subtract the beginning time from the ending time; display it to the screen.

NOTE: ***You should take a screenshot of the results for each run of each method.***

Report

In your report, include the screenshots (mentioned above) for each run of each method, along with the answers to these questions:

1) Why is a search for **`"not_here"`** the worst-case for linear search and binary search?
2) Why is a search for **`"mzzzz"`** the average-case for linear search?
3) Why is a search for **`"aaaaa"`** the best-case for linear search?
4) How do the results you saw compare to the Big-O complexity for these algorithms?
5) Why do the binary search results appear so similar, while the linear search results are so divergent?

Submissions

Files: analyzer.py, Lab04-Report.pdf
Method: Submit on Canvas