CPSC 335
Fall 2014
Project #1 — empirical analysis

Introduction

In this project you will design, implement, and analyze straightforward algorithms for two problems. For each problem, you will design an algorithm, describe your algorithm using clear pseudocode, analyze it mathematically, implement your algorithm in Python, measure its performance, compare your experimental results with the efficiency class of your algorithm, and draw conclusions.

The hypothesis

This experiment will test the hypothesis that *for large values of n, the mathematically-derived efficiency class of an algorithm accurately predicts the observed running time of an implementation of that algorithm.*

The problems

Both problems involve string processing.

1. The *largest digit in string* problem is:

   input: a string s of length n
   output: the greatest digit character (0–9) that appears in s, or `None` if s contains no digits
   size: n

   There is a straighforward decrease-by-one algorithm that solves this problem in $O(n)$ time.

2. The *longest repeated substring* problem is:

input: a string s of length $n > 0$
output: the longest nonempty substring u of s such that u appears more than once in s, or `None` if no such u exists
size: n

Long repeated substrings often show up as a result of mistakenly copy-pasting something twice, so it can be helpful to detect them.

There is a straighforward decrease-by-one algorithm, possibly involving four nested **`for`** loops, that solves this problem in $O(n^4)$ time.

Algorithm design, pseudocode, and mathematical analysis

First, design an algorithm for each of the problems. This is not intended to be particularly difficult; this project focuses on empirical analysis, not algorithm design. So do not be surprised if your algorithms are simple. As stated above, there are concise algorithms based on **`for`** loops that solve the problems in $O(n)$ and $O(n^4)$ time respectively.

Once you have worked out your algorithms, write clear pseudocode for each. As discussed in lecture, we consider pseudocode to be clear when a typical student in this class could implement it without any further explanation.

Then, analyze each algorithm mathematically. The goal is to prove that each algorithm's worst case running time should be $O(f(n))$, for some specific $f(n)$. I expect that your algorithms' efficiency classes are most likely to be $O(n)$ and $O(n^4)$, but if not, they will almost certainly be one of the "top 8" efficiency classes.

Implementation

Implement each of your algorithms in Python 3. You must write your own code for the algorithm implementations, which should correspond directly to your pseudocode. The Python library may have built in functions to solve parts of these problems, but you can only use them if they really do correspond to the algorithm described by your pseudocode.

Each algorithm should be encapsulated as a single clearly-named function (which may call helper functions if you wish).

I have provided a template Python source file to help get you started. You may freely use any part of the template. It shows how to encapsulate an algorithm in a function, access command line arguments, load a text file into a Python string, and how to use the `time.perf_counter()` function to measure the run time of Python code.

Empirical analysis

To analyze the algorithms empirically you will need to run them against representative inputs of various sizes n, measure the elapsed running time (in seconds) of each run, graph these results on a scatter plot, and try to infer which complexity class each plot corresponds to.

Section 2.6 of the Levitin textbook, and section 2.4 of the lecture notes, describe how to conduct an empirical analysis in general terms. As discussed there, you should create a *test harness* program that runs your code and measures the elapsed time of the code corresponding to the algorithm in question. Your test program should perform the following steps:

1. Load an instance text file from disk, establish an n value, and form a string s from the first n characters of the file. The provided template does this by reading command line arguments for the file name and value of n; you may use this code, but do not need to.
2. Print out the instance file name and value of n to standard output.
3. Use your algorithm to find the greatest digit in the string, while measuring how long the process takes.
4. Print out the output of your algorithm.
5. Print out the elapsed time.
6. Repeat steps 3–5 for your other two algorithms.

You will need to find your own text files to use as problem instances. In order to use truly large values of n, these files may need to be quite large (e.g. at least a hundred kilobytes). One source of free, large text files is Project Gutenberg (`http://gutenberg.org`) which provides works of literature as plain text files.

Sample output

The following shows the output of a solution program when given the first $n = 100$ characters of the Project Gutenberg edition of *The Adventures of Huckleberry Finn*.

```
Loaded "pg76.txt" of length 593144
n = 100
largest digit = None
elapsed time = 3.717001527547836e-05
longest repeated substring = [ of ]
elapsed time = 0.004114371025934815
```

The output for $n = 5000$ with the same file:

```
Loaded "pg76.txt" of length 593144
n = 5000
largest digit = 7
elapsed time = 0.002042291220277548
longest repeated substring = [

The Widows

Moses and the "Bulrushers"

Miss Watson

Huck Stealing ]
elapsed time = 73.71632703999057
```

What to measure

The goal is to draw a scatter plot graph for each algorithm's running times (two plots). The values of n should be on the horizontal axis (x-axis) and the time

values should be on the vertical axis (y-axis). Each plot should have a title and axis labels and be legible.

Each plot also needs to have enough data points to interpolate a fitting curve. 5 is the smallest number that might be reasonable. So run each algorithm for at least 5 different values of n. If possible, include at least one problem instance that's large enough to make each of the three algorithms run for *at least one minute.*

Since your algorithms will probably have differing time complexities, some of your implementations may run significantly faster than others. You may need to interrupt your program with CTRL-C, or temporarily disable some of your algorithms with `if` statements, in order to gather all the time data.

You can generate your best fit lines in software such as Excel or Mathematica, or draw them by hand.

Deliverables

Produce a written project report. Your report should include the following:

1. Your name(s), CSUF-supplied email address(es), and an indication that the submission is for project 1.
2. Two scatter plots meeting the requirements described above.
3. Your pseudocode for both algorithms.
4. Output from your program, for one instance of size $n = 200$ and another of size $n = 2000$.
5. Your complete Python source code.
6. Answers to the following questions, using complete sentences.
   (a) What did you use as problem instances? How representative are your problem instances of real-world printable strings, and why?
   (b) What is the efficiency class of each of your algorithms, according to your own mathematical analysis?

(c) Are the best fit lines on your scatter plots consistent with these efficiency classes? Justify your answer.

(d) Is this evidence *consistent* or *inconsistent* with the hypothesis stated on the first page? Justify your answer.

Your document *must* be uploaded to Titanium as a single PDF file.

Due Date

The project deadline is Thursday, 9/18, 11:55 pm. Late submissions will not be accepted.