Project 1: Empirical Analysis

Hypothesis

This project will test the following hypothesis:

> *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.*

To test this hypothesis, you will implement three algorithms; analyze each empirically; analyze each mathematically; and determine whether the empirical and mathematical analyses are in agreement.

You will produce two deliverables: 1) C++ code implementing the algorithms, and 2) a project report. There are two separate Gradescope "assignments" for submitting your deliverables.

The Problems and Algorithms

The first two problems involve searching a vector of integers for a part that is “balanced” in a certain way.

Dip Search

In the *dip search problem,* we search for two equal elements with a smaller element in between.

> *dip search problem*
> **input:** a vector V of n integers
> **output:** the position of the last *dip* in V, or None if V does not contain a dip

Here a dip is a sequence of three contiguous elements v_i, v_{i+1}, v_{i+2} where $v_i = v_{i+2}$ and $v_{i+1} < v_i$, v_{i+2}. For example, the sequence 5, 2, 5 is a dip. The integers are not required to be positive, so -4, -6, -4 is also a dip. Note that the problem definition asks for the last-occurring dip. So if V contains multiple dips, a correct algorithm must return the later one (i.e. the one at the higher index). The following algorithm solves the dip search problem.

```
dip_search(V):
    last_dip = None
    for i from 0 through n-3:
        if V[i] == V[i+2] and V[i+1] < V[i]:
            last_dip = i
    return last_dip
```

Note that this algorithm does not return as soon as it finds a dip. This is intentional, and will guarantee that you will observe worst-case performance every time you run this algorithm.

Longest Balanced Span

In the *longest balanced span problem,* we search for a large sub-vector whose elements sum to zero.

> *longest balanced span problem*
> **input:** a vector V of n integers
> **output:** the span of vertices s, e $\in [0, n]$ with $s < e$, where $(\sum_{i=s}^{e-1} V[i]) = 0,$ and the number of elements in the span is maximized. In the event of a tie, return the later span.

A balanced span is a sequence of integers that adds up to zero. So $4, 2, -3, -3$ is a balanced span because

$4 + 2 - 3 - 3 = 0$. Also any sequence of zeroes counts as a balanced span, such as $0, 0, 0$. In this problem we search for the longest balanced span, or in other words the span containing the most elements. So if

$$V = [3, -1, 1, 6, 2, -3, 1, 2]$$

then a correct algorithm would return the indices $s = 5, e = 8$ because the span at indices $5, 6, 7$ contains the values $-3, 1, 2$ which is balanced. V also contains a shorter balanced span $-1, 1$ but a correct algorithm prefers the longer 3-element span over the shorter 2-element span.

The following algorithm solves the longest balanced span problem.

```
longest_balanced_span(V):
    best = None
    for s from 0 through n:
        for e from s+1 through n:
            if the sum of elements in V[s] up to but not
including V[e] is zero:
                if best is None or [s,e) contains more elements
than best:
                    best = [s, e)
    return best
```

Telegraph Style

The third problem involves manipulating a string to put it into *telegraph style.*

> *telegraph style string problem*
> **input:** a string s
> **output:** a string s' that is a copy of s reformatted into telegraph style

A telegraph was an early form of text message, common in the 19th century, that predated digital networks. Due to technical limitations, telegraphs could only transmit a limited range of characters, and often ended in the string "STOP." so that people could tell where one message ended and the next began.

The input string s should be transformed into the telegraph-style s' as follows:

1. All lower-case letters are converted to upper-case.
2. Punctuation characters !?; are converted to periods.
3. Only some characters are allowed. After the conversions stated above, any character that does not match one of the following allowed categories, is removed:
    1. uppercase letters
    2. digits
    3. space
    4. period
4. There can only ever be one space in a row. Multiple contiguous spaces are replaced with a single space.
5. The string must end in "STOP.". If s does not already end in "STOP." then append "STOP." to the end.

For example, if s = "ab#c" then s' = "ABCSTOP."

There is no provided algorithm for this problem. You will design, describe, and analyze your own algorithm.

Team Formation

As stated in the syllabus, you may work in a team of 1-3 students. Your first task is to decide on who will be on your team before you do any of the work described below.

In particular, you need to communicate with your team members, and reach consensus on the members of the team, before accepting the assignment invitation in Github. Teams cannot be changed in Github, so once you form a team there, you are stuck with it for the remainder of this project. (You can form different teams on other assignments.)

Algorithm Design and Mathematical Analysis

After forming your team, your next task is to design your own algorithm for the telegraph style string problem. Your algorithm does not need to be complicated; the naïve pattern should suffice. As usual, write pseudocode for your algorithm, and revise the pseudocode until it is clear, correct, and terminates.

Then, do a mathematical analysis of each of the three algorithms: the given dip search algorithm; the given longest balanced span algorithm; and your own telegraph style string algorithm. For each, perform a chronological step count, and prove the efficiency class of the algorithm.

You will present this information in your report document, described below.

Implementation

Next, implement all three algorithms in C++.

Obtaining and Pushing Code

We are using GitHub Education to distribute starter code; Tuffix to run and grade code; and GradeScope to collect submissions.

This document explains how to obtain starter code and push it to GitHub: GitHub Education / Tuffix Instructions

This Youtube video by David McLaren explains how to get started with GitHub: https://youtu.be/1a5L_xsGIm8

Here is the invitation link:
https://classroom.github.com/g/mwO5m1Za

Code Layout

You will need to work with the following files:

- algorithms.hpp is a C++ header that defines data types and functions for the algorithms described above. The function definitions are effectively empty; you will need to rewrite them to actually work properly.
- algorithms_test.cpp is a Google Test-based unit test program that checks whether the code in algorithms.hpp works or not. You can run the corresponding algorithms_test program to see whether your algorithm implementations are working correctly. The unit tests depend on the Google Test library, specifically /usr/lib/libgtest.a, which is included in Tuffix. Do not modify this source file. The autograder uses the original version of this file to prevent tampering.
- algorithms_timing.cpp is a C++ program with a main() function that measures one experimental data point for each of the algorithms. You can expand upon this code to obtain several data points for each of your algorithm implementations.
- README.md is a markdown file with a place to write the names and CSUF-supplied email addresses of your team members. You need to modify this file and add the contact information for all team members.

- Makefile is a GNU make configuration file that automates compiling, running, and grading the project.

The repository also contains the following files, which you must leave unchanged, and may ignore. (You're welcome to look inside if you wish.)

- .gitignore configures git to ignore temporary files.
- LICENSE assigns the MIT License to the starter code.
- timer.hpp defines the Timer class that is used in algorithms_timing.cpp.
- grade.py is the autograder (script) that computes a score for your submission (see below).

Make Targets

You can use the "make" command inside a Tuffix shell to compile, run, test, and grade your submission.

To **compile** the test and timing programs:

```
$ make
```

To **test** your algorithm implementations for correctness:

```
$ make test
```

To **grade** your submission:

```
$ make grade
```

Rejected code: the autograder rejects a submission if it does not compile; crashes the unit tests (i.e. segfault); tdoes not have names in README.md; is simply a copy of the starter code with no changes; or the unit test source code has been modified. Rejected code corresponds to an "F" grade (approximately 50%).

To **clean** (delete) all compiled binaries and temporary files:

```
$ make clean
```

We recommend that you use these make commands inside Tuffix to confirm that your code works, and preview how your submission will be graded, before submitting your code to Gradescope.

Empirical Analysis

After you have finalized your code, conduct an empirical analysis of each of the three algorithms. For each algorithm:

1. Gather empirical timing data by running your implementations for various values of n. As discussed in class, you need enough data points to establish the shape of the best-fit curve (at least 5 data points, maybe more), and you should use n sizes that are large enough to produce large time values (multiple seconds or even minutes) to minimize instrumental error.
2. Draw a scatter plot and fit line for your timing data. The instance size n should be on the horizontal axis and elapsed time should be on the vertical axis. Your plot should have a title; and each axis should have a label and units of measure.
3. Conclude whether or not your empirically-observed time efficiency data is consistent, or inconsistent, with your mathematically-derived big-O efficiency class. (*Hint:* Our hypothesis is quite well established, so if your results are inconsistent, that most likely means that there's a problem with your math analysis, implementation, or empirical analysis.)

Gradescope Code Submission

Before the deadline, submit your work to the Gradescope code assignment. Follow the Code submissions instructions at gradescope.com. We recommend using the "Submit a GitHub repository" method, which is less error-prone than the "Drag and drop your code file(s) into Gradescope" method.

Shortly after you make your submission, Gradescope will run the autograder to compute a numerical score for your code. This will be your score on the code part of the project. You can see the score and how it was computed at Gradescope.

We set up the autograder so that you can get frequent, early, feedback about the correctness of your code and how it will be graded. We recommend that you start early. With effective time management, every team should be able to get a perfect or near-perfect score.

Report Document

Finally, produce a brief written project report in PDF format. Submit your PDF to the Gradescope project report assignment. Your report should include the following:

1. A title page that indicates that the report is about project 1; and has the name and CSUF-issued email address of every team member.
2. Pseudocode describing your telegraph style string algorithm.
3. Mathematical analyses for each of the three algorithms.
4. Scatter plots for each of the three algorithms.
5. Answers to the following questions. (Each answer should be at least one complete sentence.)
    1. What is the efficiency class of each of the algorithms, according to your own mathematical analysis? (You are not required to include all your

math work, just state the classes you derived and proved.)
2. Between the dip search and longest balanced span algorithms, is there a noticeable difference in the running speed? Which is faster, and by how much? Does this surprise you?
3. Are the fit lines on your scatter plots consistent with the efficiency classes predicted by your math analyses? Justify your answer.
4. Is all this evidence consistent or inconsistent with the hypothesis stated on the first page? Justify your answer.

Grading and Deadline

The project deadline is Tuesday April 6 11:59pm. Late submissions will not be accepted.

As stated above, the code assignment will be graded automatically by the "make grade" autograder.

The report will be graded according to a rubric involving the completeness, content, and presentation of the pseudocode, scatter plots, and question answers.

As usual, detailed rubric scores and feedback will be visible in Gradescope. Canvas will only contain total numerical scores. Grades will not be synced to Canvas until grading is complete, so your submission may show as "MISSING" in Canvas until grading is complete.

Be advised that we may use automated tools to detect plagiarism. Do not plagiarize. As stated in the syllabus, a submission that involves academic dishonesty will receive a 0% score (on both the code and report) and the incident will be reported to the Office of Student Conduct. A repeat

offense will result in an "F" in the class and will be reported to the Student Conduct office.