Project 4: dynamic programming

CPSC 335 - Algorithm Engineering
Summer 2018
Instructor: Kevin Wortman (kwortman@fullerton.edu)

Abstract

In this project you will implement and analyze dynamic programming algorithms for the same substring and subsequence problems that we attacked in project 3.

Mathematical analysis suggests that the new subsequence algorithm is substantially faster than the old one. The dynamic programming subsequence algorithm takes $O(n^3)$ time, while the exhaustive search algorithm takes $O(2^n n)$ time. The dynamic programming substring algorithm has the same $O(n^3)$ time complexity as the exhaustive search algorithm, so implementing it is more of a "warm-up" exercise than an optimization achievement.

The Dynamic Programming Substring Algorithm

We will solve the following variant of the substring problem.

longest common substring
input: a string a of length m and a string b of length n *output:* the longest string s such that s is a substring of both a and b; in the case of ties, use the substring that comes first alphabetically

(The only difference from project 3 is that we are handling ties differently, to make it easier for us to tell when your implementation is correct.)

The problem can be broken into smaller sub-problems by searching for the longest common *suffix* between a and b. String s is a suffix of a if s and a end with identical characters, but don't necessarily start with the same characters. For example, all of the following are suffixes of "banana": "" (empty string), "a", "na", "nana", "anana", and "banana".

We can solve this problem with a dynamic programming algorithm that uses a 2D array A with the invariant

$$A[i][j] = \text{the longest common suffix between } a[:i] \text{ and } b[:j]$$

where $a[:i]$ means "the first i characters of a" and $b[:j]$ means "the first j characters of b."

This invariant induces the base cases $A[i][0] =$ "" (empty string) for all i, since when $j = 0$ we are not allowed to use any characters from b, so the only substring we can form is "". Symmetrically $A[0][j] =$ "" (empty string) for all j.

In the general case we need to initialize $A[i][j]$ for $i,j > 0$. When $a[i-1] == a[j-1]$, the strings $a[:i]$ and $a[:j]$ end with the same character, so the longest common subsequence between them includes that character, at the end of whatever was in $A[i-1][j-1]$. But when $a[i-1] \neq a[j-1]$, the only common suffix between $a[:i]$ and $a[:j]$ is the empty string. So we have the following general case:

```
if a[i-1] != b[j-1]:
        A[i][j] = ""
else:
        // + means appending one character to the end of string A[i-1][j-1]
        A[i][j] = A[i-1][j-1] + a[i-1]
```

We can find the solution, which is a longest common substring, by searching for the longest string in any $A[i][j]$.

The following pseudocode incorporates all these ideas.

```
dynamic_substring(a, b):
        A = new (m + 1) × (n + 1) 2D array of strings
        Initialize each A[i][0] = "" for all i ∈ [0, m]
        Initialize each A[0][j] = "" for all j ∈ [0, n]
        For i from 1 to m inclusive:
                For j from 1 to n inclusive:
                        if a[i-1] != b[j-1]:
                                A[i][j] = ""
                        else:
                                A[i][j] = A[i-1][j-1] + a[i-1]
        Let s be the longest string in A; in the case of ties, prefer the string that comes first alphabetically
        Return s
```

The two nested loops mean that the main loop iterates $O(n^2)$ time, and in the "Else" case a length-n string is created which takes $O(n)$ time, for a grand total of $O(n^3)$ time.

The Dynamic Programming Subsequence Algorithm

Again we will use a variant of the subsequence problem that defines how to handle ties in a convenient way.

longest common subsequence
input: a string a of length m and a string b of length n *output:* the longest string s such that s is a subsequence of both a and b; in the case of ties, use the substring that comes first alphabetically

The dynamic programming algorithm for subsequences is similar to the one for substrings. Both involve a 2D array of strings, base cases, and a general case. However they are different problems so a few things are different. In the subsequence problem we use the invariant

$$A[i][j] = \text{the longest common subsequence in } a[:i] \text{ and } b[:j].$$

This time, the subsequence in $A[i][j]$ is not obligated to include the last character of $a[:i]$ or $b[:j]$. The base cases are the same; when $i = 0$ or $j = 0$, the only possible subsequence is the empty string.

In the general case, when $a[i-1] == a[j-1]$, we can include the last character $a[i-1]$ in whatever subsequence was already computed for $A[i-1][j-1]$. But when $a[i-1] \neq a[j-1]$, we don't have to settle for the empty string. Instead, we can form a subsequence that ignores the last characters of $a[:i]$ and $b[:j]$. We have two to choose from, one at $A[i-1][j]$ and another at $A[i][j-1]$, and we prefer whichever is longer.

After the general cases have completed, $A[m][n]$ contains a longest subsequence, but if it is tied with another subsequence, the other one might come first alphabetically. So there is a post-processing step of searching for the longest string, which can only be in the last row or last column.

The complete pseudocode is below.

```
dynamic_subsequence(a, b):
    A = new (m + 1) × (n + 1) 2D array of strings
    Initialize each A[i][0] = "" for all i ∈ [0, m]
    Initialize each A[0][j] = "" for all j ∈ [0, n]
    For i from 1 to m inclusive:
        For j from 1 to n inclusive:
            if a[i-1] == b[j-1]:
                A[i][j] = A[i-1][j-1] + a[i-1]
            Else if len(A[i-1][j]) > len(A[i][j-1]):
                A[i][j] = A[i-1][j]
            Else:
                A[i][j] = A[i][j-1]
    Let s be the longest string in row A[m][?] and column A[?][n]; in the case of ties, prefer the string
    that comes first alphabetically
    Return s
```

Just like the substring algorithm, the bottleneck is the $O(n)$ string creation operation, inside two nested loops, for a total of $O(n^3)$ time.

Implementation

You are provided with the following files.

1. `dynamic_subsequence.hpp` is a C++ header that defines skeleton functions for the two algorithms described above. You are responsible for implementing these two functions, using the pseudocode given above.
2. `exhaustive_subsequence_test.cpp`, `exhaustive_subsequence_timing.cpp`, `README.md`, `rubrictest.hpp`, `timer.hpp`, and `Makefile` function essentially the same as in project 3.

Obtaining and Submitting Code

This document explains how to obtain and submit your work:

GitHub Education / Tuffix Instructions

Here is the invitation link for this project:

https://classroom.github.com/g/1MsYio0L

What to Do

First, add your group member names to `README.md`. Then implement the two skeleton functions. Use the test program to check whether your code works.

Once you are confident that your algorithm implementations are correct, do the following:

1. Gather empirical timing data for each of the two algorithms by running them for various values of n.
2. Draw two scatter plots. Each plot should have the instance size n on the horizontal axis, elapsed time on the vertical axis, a fit line, title, and labels on both axes.
3. Conclude whether or not your empirically-observed time efficiency data is consistent, or inconsistent, with the mathematically-derived big-O efficiency classes.

Finally, produce a brief written project report ***in PDF format***. Your report should be submitted as a checked-in file in GitHub. Your report should include the following:

1. Your names, CSUF-supplied email address(es), and an indication that the submission is for project 3.
2. Two scatter plots meeting the requirements stated above.
3. Answers to the following questions, using complete sentences.
    a. Consider the two algorithms you've implemented for the substring problem (the exhaustive search algorithm in project 3, and the dynamic programming algorithm in this project). How do their performance compare; is one significantly faster than the other, and which one; or are they roughly equivalent? If there is a speed difference, is it enough to make the difference between the algorithm being practical to use, and impractical?
    b. Answer the same questions, but for the subsequence problem.
    c. Did you find implementing this algorithm to be easy, difficult, or in between? What was the hardest part, and why? How does the difficulty level compare to the exhaustive search algorithms from project 3?

Grading Rubric

Your grade will be comprised of three parts: *Form, Function,* and *Analysis*.

Function refers to whether your code works properly as defined by the test program. We will use the score reported by the test program, when run inside the Tuffix environment, as your Function grade.

Form refers to the design, organization, and presentation of your code. A grader will read your code and evaluate these aspects of your submission.

Analysis refers to the correctness of your mathematical and empirical analyses, scatter plots, question answers, and the presentation of your report document.

The grading rubric is below.

1. Function = 10 points, scored by the unit test program
2. Form = 9 points, divided as follows:
    a. README.md completed clearly = 3 points
    b. Style (whitespace, variable names, comments, helper functions, etc.) = 3 points
    c. C++ Craftsmanship (appropriate handling of encapsulation, memory management, avoids gross inefficiency and taboo coding practices, etc.) = 3 points
3. Analysis = 9 points, divided as follows
    a. Report document presentation = 3 points
    b. Scatter plots = 3 points
    c. Question answers = 3 points

Legibility standard: As stated on the syllabus, submissions that cannot compile in the Tuffix environment are considered unacceptable and will be assigned an "F" (50%) grade.

Deadline

The project deadline is Friday, June 29, 11 am.

You will be graded based on what you have pushed to GitHub as of the deadline. Commits made after the deadline will not be considered. Late submissions will not be accepted.