Analysis of Algorithms
CS 375, Fall 2022
Project 4
Due **BY 11:59pm** on **Monday, Dec. 12**

Project 4: Edit Distance and Dynamic Programming!

In this assignment, you'll work in teams of up to four people to implement a dynamic programming algorithm from a recursive definition, with applications to natural language processing (*NLP*). The goals of this project are:

- to give you practice converting a recursive definition into an iterative dynamic programming algorithm;
- to give you practice implementing both recursive and iterative dynamic programming methods;
- to introduce you to *edit distance* as a concept for NLP applications;
- to give you practice presenting your algorithms and ideas in a written document and short video-recorded demo of your work;
- to give you practice working with other students as a team.

A few **Important Notes** about this project assignment:

- All of your implementation in this assignment must be in Python or Java. Please pick one of those languages and use it for all of your team's coding.
- **Restriction on resources to use: Do not use any outside resources about *edit distance*** as part of this assignment, because most of what's available contains information about implementation that you are expected to do yourselves as part of learning about dynamic programming.

  If you have questions about edit distance, please let me know! If you're looking for outside resources regarding a particular part of edit distance, please ask me, and I will try to find something to help with your questions that does not affect your ability to have this project be completely your own work.

  You are welcome to use outside resources about Python or Java programming in general. It is only edit distance-specific resources that are restricted for this assignment. As always, please let me know if you have any questions!

The Project Assignment[1]

Here are the parts of this assignment:

1. **Edit Distance** Your team will be given a recursive definition of edit distance, and you will create and implement both *top-down recursive* and *bottom-up iterative dynamic programming* algorithms to compute edit distance.

2. **Spelling Correction** Your team will use your edit distance calculations to create and implement a general-purpose *spelling checker*, and you will demonstrate its effectiveness.

3. **Improvements** Your team will implement a specialized spell checker with improved performance on CS375-specific documents, and your team will come up with at least one other way to improve upon your original, general-purpose spell checker.

4. **Presentation of Your Work** Along with a write-up document, your team will record a short video demo that emphasizes features of your dynamic programming implementation and your improvements to the original algorithm.

1 Edit Distance

Several natural language processing (*NLP*) and computational biology applications rely on metrics of similarity between words (or documents, genome sequences, etc.). One common way to measure word similarity is by *edit distance*—a measure of how much work it takes to transform a string S into a string T.

1.1 Definition and Examples

For this project, we'll consider only three possible operations to transform one string into another:

Replace Replace one character by another. For example, "cat" is transformed to "car" by a single *replace* operation—replacing 'r' with 't'.

Insert Insert one character into a string. For example, "cat" is transformed to "cart" by a single *insert* operation—inserting 'r' into the word "cat."

Delete Delete one character from a string. For example, "char" is transformed to "car" by a single *delete* operation—deleting 'h' from the word "char."

The *edit distance* from string S to string T is defined to be the *minimum* number of operations required to transform S to T. For example, the edit distance from *analysis* to *algorithms* is 8. Here's one way of transforming *analysis* to *algorithms* in 8 operations, shown below in left-to-right order of acting on string *analysis*:

[1]Acknowledgment: Thanks to Prof. Amanda Stent (Davis AI Institute, Colby College) for helpful conversations as part of preparing this Project

Operation	String
[Starting string]	analysis
replace 'n' with 'l'	a*l*alysis
replace 'a' with 'g'	al*g*lysis
replace 'l' with 'o'	alg*o*ysis
replace 'y' with 'r'	algo*r*sis
delete 's'	algoris
insert 't'	algori*t*s
insert 'h'	algorit*h*s
insert 'm'	algorith*m*s

There are ways to transform *analysis* to *algorithms* in more than 8 operations, but because 8 is the minimum possible, the edit distance from *analysis* to *algorithms* is 8.

This idea of edit distance leads to a recursive definition (which is somewhat similar to what we used for the *Longest Common Subsequence* problem in CS375). Here's a recursive definition of function $editDistance(S, T)$, giving the edit distance from S to T:

Base case When S or T are empty, the edit distance is the length of the non-empty string: transforming the empty string to T would take $|T|$ insert operations; transforming S to the empty string would take S delete operations. Thus, if we let "" stand for the empty string, $editDistance(\text{“”}, T) = len(T)$, and $editDistance(S, \text{“”}) = len(S)$.

Recursive cases If the last symbol in S equals the last symbol in T, no work has to be done transforming that symbol from S into that symbol from T. If we let m stand for $len(S)$ and n stand for $len(T)$, this means when $S[m-1] == T[n-1]$, the edit distance from S to T is just the edit distance from $S[0..m-2]$ to $T[0..n-2]$—i.e., all but the last characters of both strings.

If $S[m-1] \neq T[n-1]$, however, then some work has to be done to address that difference. Because we're using three possible operations for transforming strings, there are three options for that:

- *replace* $S[m-1]$ with $T[n-1]$ and then transform $S[0..m-2]$ to $T[0..n-2]$;
- *delete* $S[m-1]$ and then transform $S[0..m-2]$ to $T[0..n-1]$; or
- *insert* the last symbol of T onto the end of S and then transform $S[0..m-1]$ into $T[0..n-2]$.

Each of these requires 1 operation plus the remaining transformation, and the edit distance will be whichever option requires the minimum number of operations. So, for the recursive case, $editDistance(S, T)$ is defined as follows (recall that $S = S[0..m-1]$ and $T = T[0..n-1]$):

$$\begin{aligned} editDistance(S,T) \;=\; 1 + \min(\;& editDistance(S[0..m-2], T[0..n-2]), \\ & editDistance(S[0..m-2], T[0..n-1]), \\ & editDistance(S[0..m-1], T[0..n-2])\;). \end{aligned}$$

So, to put it all together in one definition:

- When S or T are empty,
  $editDistance(\text{“”}, T) = len(T)$, and
  $editDistance(S, \text{“”}) = len(S)$

- If $S[m-1] == T[n-1]$,
  $editDistance(S, T) = editDistance(S[0..m-2], T[0..n-2])$

- If $S[m-1] \neq T[n-1]$,
  $editDistance(S, T) = 1 + \min($ $editDistance(S[0..m-2], T[0..n-2])$,
  $editDistance(S[0..m-2], T[0..n-1])$,
  $editDistance(S[0..m-1], T[0..n-2])$ $)$

That is our full definition of *editDistance*(), with the first two equations being base cases and the last two being recursive cases for when S and T are both non-empty.

1.2 Implement Edit Distance

Working from that recursive definition of edit distance, create algorithms for **two versions** of a function to compute the edit distance of two input words:

- a straightforward top-down recursive implementation; and

- a bottom-up, iterative dynamic programming implementation.

In addition, implement those two algorithms in your team's programming language of choice (Python or Java). Analyze the time complexity of the two versions, using empirical complexity measures from actual runs of your code as well as asymtotic complexity analyses to describe differences between the versions' runtime efficiencies.

For asymptotic complexity analysis of your recursive version, be sure to *present and explain* a recurrence that describes the runtime of the function. Although *you are not required to analyze it* to arrive at a Θ bound to get a good grade on this project, *you might choose to attempt to do so*, to show greater depth and command of relevant concepts. (As always on CS375 project assignments, showing depth and command of related concepts will be considered as part of your grade.)

(**Hint:** If you are going to try analyzing your recurrence to get a Θ bound, try setting up a recursion tree. It may be challenging to solve it using only the techniques on which we primarily focused in CS375, but you may be able to analyze the tree to get a lower bound (Ω) and an upper bound (Big-O) on the overall complexity, which may help. Consider the number of levels in the tree (for the worst case): What's the smallest number of levels it could be? What's the largest? How can you use that information—along with some summation formulas we've seen this semester—to come up with asymptotic complexity bounds?)

To analyze the asymptotic complexity of your iterative version, standard methods will probably be helpful. For empirical data about runtime, feel free to use whatever methods you think would be most helpful, perhaps including language-specific tools to help analyze actual time spent running by the implementation; be sure to run your implementations on large

enough examples to demonstrate that difference in time efficiency. (The *SCOWL* dictionary for use in Section 2 below may be more than sufficient!)

Present the algorithms for these two implementations of edit distance in your write-up along with your analyses of efficiency and your explanations of how the algorithms correctly implement the definition of edit distance.

You will also submit the code of your implementations, as described in Section 5 below, which must be fully commented and easily readable, with overall good style. Your code should be straightforward implementations of your algorithms—a reader should immediately see how the code follows directly from the pseudocode. If there are any essential implementation details to include, you may describe those in your write-up as well as in comments in your code, but ideally, the code should follow from pseudocode straightforwardly and not require such extra documentation. In general, for all code in this project, if code isn't easily readable, or if it is not easily understandable as a straightforward implementation of your algorithms, it will not earn full credit.

Note: You do not *need* to use specifications, formal loop invariants, or inductive arguments to explain the structure of your algorithms or code—the intention is for these explanations to less formal rather than a structured correcntess argument—although if you find that doing so makes your explanation clearer, you are welcome to do so. As always, clarity is essential for good explanations!

2 Spelling Correction

One NLP application built upon edit distance is *spelling correction*—a program reads through a body of text and, word by word, suggests possible alternates for words that may be misspelled. You've probably used spell checkers before! They can start out with a dictionary (i.e., a word list, which can be in a separate file) of known words that they consider correctly spelled, and then as they read through some text, for each word they encounter that isn't in the dictionary, they suggest words that *are* in the dictionary and have similar spellings.

That's where edit distance comes in: Upon encountering some word w that's not in the dictionary, edit distance is used to determine words in the dictionary that are similar to w, to suggest as alternates. You'll come up with an algorithm to do that spell checking, and then you'll implement your algorithm in a working spell checker, test it, and describe your results in a way that illustrate both the strengths and limitations of your spell checker!

2.1 Your Spelling Correction Algorithm

In your write-up for this project, give an algorithm that meets the following specifications:

> **Input**: A string of text T, consisting of the words to be spell checked; and a dictionary (word list) D of correctly spelled words.
> **Output**: For each word w in T that does not occur in D, 5 words in D with minimal edit distance from w—i.e., 5 suggestions for a corrected spelling of w, which must be the 5 words with minimal edit distance from S; in case of ties for least edit distance from S, the algorithm can pick any of them with that minimal edit distance. (To be clear, your alorithm should provide output for every word in T that's not in D.)

Once you have your algorithm, implement and test it, reporting results of the tests to illustrate the effectiveness and limitations of your algorithm. For this part of the project, your algorithm and code should use your iterative dynamic programming edit distance implementation (written for Section 1.2), without re-describing or re-implementing it, and go linearly / exhaustively through dictionary D in a brute-force manner. (You do not *need* to do anything other than such a linear / exhaustive search approach for any part of this project, although if you do something else as part of making improvments to your algorithm or your implementation, that should be presented as part of your work for Section 3 below.

For the dictionary D for your general-purpose spell checker, use the *Spell Checking Oriented Word Lists (SCOWL)*, which can be downloaded from the CS375 Project Assignments website, or from

https://sourceforge.net/projects/wordlist/

For this project, please download and use the US wordlist, just to make sure work is standardised across all teams submitting this assignment. (It might be fun to explore the other English-language wordlists available there, but that won't be part of this project!) When you download and unzip the zipped folder, one of the files will be `en_US-large.txt`—please look at it, and you'll find it's a long list of words, which you should take as correctly spelled for purposes of this assignment.

Your write-up for the project should include your algorithm in psuedocode, an English description of how it works, an asymptotic complexity analysis, and a correctness argument (as usual). As part of the project, you will also submit your code—fully commented, easily readable, and in good style—but do not include it in the write-up. The write-up should be self-contained and fully explain the algorithm, also including illustrative examples of how it works (i.e., tests of your implementation), showing its effectiveness and its limitations; if the write-up is not self-contained, and I need to look at your code to understand your algorithm or its correctness / complexity, you will not receive full credit on this part of this project. CS375 is, after all, an *algorithms* class!

3 Improve Spell-Checking Performance

There may be several different ways to improve the performance of your spelling correction algorithm or implementation, and for this part of the project, you'll create and present at least two of them: one will be using **a *domain-specific* dictionary**; the other will be **your team's choice**!

As always, improvements that demonstrate greater depth and command of relevant concepts will earn a higher grade. If you'd like to discuss your team's proposed improvements with me, I'll be happy to meet with you—please contact me as early as possible in your process!

3.1 Use A Domain-Specific Dictionary

The SCOWL dictionaries are general-purpose word lists, but for particular application domains, spell-checking performance can be improved by incorporating domain-specific knowledge and tools. One straightforward way to do this is to create a domian-specific dictionary to use alongside a general-purpose one. For this part of the project, improve performance on CS375-specific documents by doing the following:

- Create a CS375-specific word list / dictionary file, derived from course lecture notes, course assignments, and the course textbook, using whatever methods you'd like. Be sure that your CS375-specific dictionary will be sufficient to let you demonstrate credit-worthy improvements in performance!

- Once you've created your CS375-specific dictionary, use it with your spell checker code to find and correct errors in spelling or diction in this project assignment document! (There are some. And they're all put there on purpose ... no, really, they are!) In addition, do spell checking on this project assignment document using your algorithm from Section 2.1 with only the SCOWL word list, and compare that to the performance of your algorithm with your domain-specific dictionary.

- Come up with one or more other examples of spell checking CS375-related material to demonstrate your improvements. For CS375-specific material, you can use your own submitted HW files, lecture notes, passages from the textbook, course assignments, or other sources you choose—or you can take such sources and modify them specifically to demonstrate your improvements—but please make sure they're *CS375-specific*, and be sure to say where they're from in your write-up.

In your write-up, present examples and data that will convince a reader that your work has indeed improved spelling correction for CS375-specific documents—please use whatever metrics you think will be most effective to demonstrate the improvement. For the particular task of finding misspelled words in this document, please present your work so that readers can fully understand your process and replicate your results—this will include fully (but concisely!) describing the process by which you took text from this document for spell checking, and presenting all of the output from your spelling checker, i.e., all the misspellings your algorithm found, along with all the alternate words suggested for spelling correction.

3.2 Your Team's Choice!

Create and present at least one other improvement to your original, general-purpose spell checker. One way to do this is to pick another domain of your choice, and improve its performance on text that's specific to that domain—you are welcome to do this by creating another domain-specific dictionary and following the same general procedure as in Section 3.1 above, although doing something *too* similar to that in Section 3.1 will not demonstrate as much depth or command of concepts as if something more novel were at least *considered* as part of your work.

You are welcome to come up with other ways to improve your original spell checker, too! You might consider modifying your algorithm, using other auxiliary word lists / dictionaries or data structures, or other approaches. The choice is yours! I encourage your team to consult with me early in your process about options you're considering, however, in case I can offer perspective that will be helpful for you.

For whatever improvement(s) your team elects to implement, your write-up should describe them clearly and present examples and data that demonstrate the extent of your improvements. Any new or modified algorithms should be presented with pseudocode, an English description, and brief correctness and complexity arguments if the changes from previous algorithms are deep enough to warrant new correctness / complexity analyses. Any auxiliary files used (including new dictionaries / word lists) should be included as separate files when submitting your work, as noted in Section 5 below.

4 Present Your Work: A Demo

For Project 2, you were part of a team that gave a presentation 15–20 minutes in length. For Project 3, you were part of a team that gave a presentation 10–12 minutes in length. It's important to be able to give presentations of varying lengths, to present your ideas to different audiences in different contexts!

For this Project, your team will give a demo of your work in a video recording that is **5–7** minutes in length. For full credit, your team must record an effective demo lasting between 5–7 minutes; anything outside that range may not receive full credit. Meeting these time constraints may require planning; please feel free to ask me questions early in your process about what to include in your demo!

For this demo, you should assume that your audience is *me*. (Or, equivalently, someone with extensive background knowledge who wants to confirm the high quality of your work.) I know what edit distance is and what the recursive definition is that you were given, but I want to verify that your team understands what you've done with it. I want to confirm that your implementations work correctly and efficiently (but for full credit, I only want to be shown algorithm-level details, not implementation-level details). I want to make sure your team understands the strengths and limitations of your implementations, and I want to make sure I don't miss any of the coolness demonstrated in your improvements!

Your demo should include **at least** the following:

1. A description and demonstration of your iterative dynamic programming spell-check algorithm, along with its time complexity and a comparison of its performance to the recursive spell-check algorithm. You may not have time to include a full complexity argument in the demo, but at least say what its complexity is, give a *one-sentence-long* description of the key to its complexity analysis (e.g., "It has a single loop that goes through the input doing constant work each iteration, so it's a linear algorithm."), and present some data showing that it's better (I hope!) than the recursive version.

2. A description and demonstration of your CS375-specific dictionary and your application(s) of it.

3. A description and demonstration of at least one other improvement.

Your team should choose what to include to make the demo as effective as possible within the available time. This is not a standard technical presentation, and you do not *need* to have slides, but it might be helpful to do so, to present your ideas and the strengths of your work in a form that's easy to understand in 5–7 minutes. You could instead choose to show your code running in real time, explaining it as it runs; or, you could combine those two approaches, interleaving some of both. Your team should decide the most effective way for you to present your work in a short demo.

As with previous Projects, ideally, the entire team would learn about all parts of the topic being presented, and workload would be balanced. For this demo, your presentation should consist of each team member presenting for a single shift of 1–2 minutes in duration, followed by a different teammate. Presentations not following this structure may not receive full credit for this assignment. If there are questions about what's expected in terms of the division among teammates of time spent presenting, please let me know!

The default expectation is that you will record your demo as a screen recording in Zoom. If you believe another option would be better for your team, **please see me about it as soon as possible!**

Your Write-Up Document In addition to the demo presentation itself, your team will create an accompanying write-up document, which should enable me to fully understand the work done for this project even if I do not see your demo. This document must be typed (submitted in PDF) and contain all important *algorithmic* details. It should contain at least the minimum needed for a presentation of your algorithms for this project:

- Pseudocode and English descriptions of every algorithm, including separate pseudocode / English descriptions for each improvement that involves new or modified algorithms. As part of this, give brief descriptions of domain-specific dictionaries or other auxiliary files created for your improvements, enough for readers to understand their impact on your tests / demo results.
- Concise and convincing high-level correctness arguments and complexity analyses of your algorithms.
- For the "Your Team's Choice" improvement, include 1–2 sentences about how your team came up with the ideas behind that improvement.
- For each implementation, the results of testing showing that your algorithms and implementations work as intended. (If an implementation does not follow straightforwardly from the pseudocode description of an algorithm, please explain that in the relevant part of your write-up. Hopefully, however, this situation will not occur in your work!)

Overall, please include everything needed to concisely demonstrate your command of relevant techniques and concepts, including recursive design, dynamic programming and iterative bottom-up algorithms, effective domain-specific spell checking, and anything relating to your chosen improvments. Including relevant examples can be especially helpful in communicating your work. Ideally, your write-up should demonstrate that you understand the strengths and limitations of your algorithms and implementations; if you choose to include results of testing that illustrate this, that could improve the effectiveness of your write-up. In addition, consider that whatever results of testing you include in your video demo are probably important for understanding your work, so you may want to include them in your write-up as well; your write-up may also allow you to include additional examples, however, that time constraints prohibit you from including in your demo.

Overall, as with previous projects, more credit will be given to submissions that demonstrate greater scope of work completed within the constraints of the project assignment, greater depth of insight in the work completed, and more correct and effective presentations of the work. As would be expected, this will partly depend on the effectiveness of your algorithms, your analyses, your demo, and your write-up. Please keep your write-up as concise as it can be while still containing all relevant information; write-ups that are much too long or too short are not maximally effective and may not receive full credit. As always, please feel free to ask me questions about what to include in your write-up!

5 Submission Instructions

Deadline: 11:59pm, Dec. 12 A "designated submitter" from each team should submit the following items by emailing them to me (eaaron@colby.edu):

- A file containing all of the code of your implementations for this project, along with thorough instructions on how to run your code, in case readers / graders want to independently valdiate your results. (For full credit, readers / graders must be able to easily run your code after reading your instructions. Please feel free to ask me any questions about this!) This is not intended to be a code-heavy project—it is focused on algorithms and explanations—so to save readers / graders the trouble of needing to switch among multiple small files to read and understand your code, please put all of it in a single file called `CS375_Proj4_Code_Team_<INITIALS>.java` (or `.py`). (This may be somewhat non-standard for file-naming, but it will be helpful in this context.)

- Individual files created for your improvements, including your CS375-specific dictionary. Please give each file a helpful, descriptive filename, with your team's `<INITIALS>` in each filename.

- A PDF file with all of the slides used for the demo, if any. Please put two slides per page (as is done for CS375 course lecture notes) and name the file `CS375_Proj4_Slides_Team_<INITIALS>.pdf`.

- Your write-up document containing your algorithms, analyses, and other content as specified above. This should be a PDF file called `CS375_Proj4_WriteUp_Team_<INITIALS>.pdf`.

- A video file (or link to it) of your demo. *Please put it in your Google Drive space* if it's too large to simply include in an email. Please name the file `CS375_Proj4_Demo_Team_<INITIALS>.mp4`

As usual, `<INITIALS>` is to e replaced by the initials of the team members in the group in the team assignments. E.g., if Eric Aaron and Stephanie Taylor were the teammates, a file from that team would be called `CS375_Proj4_Slides_Team_EA_ST.pdf`.
Note the preferred `mp4` format. If for any reason you cannot submit an `mp4` video, please let me know as soon as possible!

Lateness policy: Because we're at the end of the semester, it is essential that this project be submitted on time. For this project, there will be a deduction of 1% for submission up to 24 hours late—i.e., at any time on Dec. 13—which will take us to the end of the last day before Exams. Work submitted on or after Dec. 14 will receive a 50% deduction—please submit your work before Dec. 14!

As always, extenuating circumstances will be considered—please contact me as soon as possible if any extenuating circumstances are impeding your work on this project!