

Acknowledgments

Shout-out to Eric for inspiration of this project. Although it's changed quite a bit, the underlying ground work of this document is still largely his work.

Introduction

In this project, you'll again work in teams 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 writing a report with a level of detail that enables readers to independently analyze your work and replicate / confirm your analysis;
- 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 algorithm design and implementation that you are expected to do yourselves as part of learning about dynamic programming. (Note that many resources about dynamic programming generally also contain information about edit distance that is restricted for this assignment.)

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, explicitly cite all outside sources you use in any way for this assignment—for full credit, be sure to have an Acknowledgments section in your write-up that lists all resources used—and as always, please let me know if you have any questions about this!

The Project Assignment

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. (1.2)
2. **File Comparison** Your team will use your edit distance calculations to create and implement a general-purpose *file comparer*, and you will demonstrate its effectiveness. (2.1)
3. **Improvements** Your team will implement a specialized file comparer with improved performance on CS-specific text files (presumably programs), and your team will come up with at least one other way to improve upon your original, general-purpose file comparer. (3)
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’. The ‘cost’ of this operation will be 3 by default.

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.” The ‘cost’ of this operation will be 2 by default.

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 ‘cost’ of this operation will be 2 by default.

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 18. Here's one way of transforming *analysis* to *algorithms* with a ‘cost’ of 20, shown below in left-to-right order of acting on string *analysis*:

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

It turns out there is a way to transform *analysis* to *algorithms* with a cost of 18 and this way is the minimal way (take a moment to see if you can spot it); because 18 is the minimum possible, the edit distance from *analysis* to *algorithms* is 18.

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 twice 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, $editDistance(“”, T) = 2 \times len(T)$, and $editDistance(S, “”) = 2 \times len(S)$.

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 $\text{len}(S)$ and n stand for $\text{len}(T)$, 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 $T[0..n-2]$ (at a cost of 3);
- *delete* $S[m-1]$ and then transform $S[0..m-2]$ to $T[0..n-1]$ (at a cost of 2); or
- *insert* the last symbol of T onto the end of S and then transform $S[0..m-1]$ into $T[0..n-2]$ (at a cost of 2).

Each of these requires paying some cost (either 2 or 3) plus the remaining transformation, and the edit distance will be whichever option requires the minimum summed cost. So, for the recursive case, $\text{editDistance}(S, T)$ is defined as follows (recall that $S = S[0..m-1]$ and $T = T[0..n-1]$):

$$\begin{aligned} \text{editDistance}(S, T) = \min(& 3 + \text{editDistance}(S[0..m-2], T[0..n-2]), \\ & 2 + \text{editDistance}(S[0..m-2], T[0..n-1]), \\ & 2 + \text{editDistance}(S[0..m-1], T[0..n-2])). \end{aligned}$$

So, to put it all together in one definition:

- When S or T are empty,
 $\text{editDistance}("", T) = 2 \times \text{len}(T)$, and
 $\text{editDistance}(S, "") = 2 \times \text{len}(S)$
- If $S[m-1] == T[n-1]$,
 $\text{editDistance}(S, T) = \text{editDistance}(S[0..m-2], T[0..n-2])$
- If $S[m-1] \neq T[n-1]$,
 $\text{editDistance}(S, T) = \min(3 + \text{editDistance}(S[0..m-2], T[0..n-2]), \\ 2 + \text{editDistance}(S[0..m-2], T[0..n-1]), \\ 2 + \text{editDistance}(S[0..m-1], T[0..n-2]))$

That is our full definition of $\text{editDistance}()$, 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 (i.e., not *memoized*) 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 efficiency / complexity measures from actual runs of your code as well as asymptotic 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 and determine a Θ bound on its runtime.

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.

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 ?? 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 be less formal rather than a structured correctness 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 File Comparison

One application built upon edit distance is *file comparison*: a program which compares two files and determines the changes between them. In particular, a good file comparison program will determine the ‘best’ way to get from one file to the other. Some reasonable metrics could be the transformations of minimal number to get from one file to the other, or the ones ‘most likely’ to have been used to get from one to the other, etc.

This is where edit distance comes in: as a starting reference point, we’ll use our edit-distance calculator to determine the edit-distance between the two files and use that to determine which changes were made according to the edit-distance formula to get from one to the other.

A General Note: The effectiveness of your file comparator, not just its efficiency, is essential for this assignment. It’s okay if it takes a little while to compare files — that does not mean an approach isn’t working or isn’t valid! You are encouraged to think about how to improve time efficiency as part of demonstrating depth and command of relevant concepts for this assignment, but just because an implementation isn’t very fast does not mean it’s incorrect. If your team does not allow your file comparator to run long enough to do its job, you will not earn full credit; on the other hand, you may be appropriately concerned about very slow performance — excessively slow code will not earn full credit either — so if your team has questions about whether your implementation is not sufficiently time efficient, please ask me questions about it early in your process!

2.1 Your File Comparator Algorithm

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

Input: Two files F_1 and F_2 . **Output:** A sequence of changes that transforms F_1 to F_2 .

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).

For **full credit** (meaning like A-tier work), I’m including the following example input/output:

- F_1 : [fib1.py](#)
- F_2 : [fib2.py](#)
- Output: [sampleOutput.txt](#)

Your write-up for the project should include your algorithm in pseudocode, 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 File Comparator Performance

There may be several different ways to improve the performance of your file comparator algorithm or implementation, and for this part of the project, you'll create and present at least one.

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!

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 should be included as separate files when submitting your work, as noted in Section ?? below.

4 Present Your Work: A Demo

For this Project, your team will give a demo of your work in a video recording that is **7 – 10** minutes in length. For full credit, your team must record an effective demo lasting between 7–10 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 algorithm, along with its time complexity and a comparison of its performance to the recursive algorithm. Present some data showing that it's better (I hope!) than the recursive version.
2. A description and demonstration of your 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 7 – 10 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 at least 1 minute in duration. 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 a reader to fully understand the work done and related algorithmic analyses for this project even if they 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.
- Concise and convincing high-level correctness arguments and complexity analyses of your algorithms.
- For your 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!)
- An *Acknowledgments* section listing all outside resources used for this assignment.

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 file comparison, and anything relating to your chosen improvements. As with other scientific reports, your write-up document should contain a level of detail needed for your target audience to fully understand your methods and results, and to independently analyze your work to confirm or replicate your conclusions; write-ups that do not present such details may well not earn full credit or demonstrate full command of relevant concepts. 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!