**CS358: Applied Algorithms**

# Assignment 2: Space-Efficient Edit Distance (due 3/4/2020 )

*Instructor: Sam McCauley*

## Instructions

All submissions are to be done through github. This process is detailed in the handout "Handing In Assignments" on the course website. Answers to the questions below should be submitted by editing this document. All places where you are expected to fill in solution are marked in comments with "FILL IN."

Please contact me at sam@cs.williams.edu if you have any questions or find any problems with the assignment materials.

Unlike Assignment 1, **this assignment should be done alone.** You are encouraged to collaborate to the extent described in the class materials, but each student should ultimately have their own submission.

## Problem Description

INPUT: The input consists of a sequence of tests. Each test begins with a line that has four numbers on it. These numbers are the length of `string1`, the length of `string2`, the length of the intended solution string, and finally the size of the alphabet. Following this line, `string1` is listed in 80-character lines, followed by `string2` and finally the intended solution. These strings each have brief comments to help with navigation that are ignored by the input reader provided to you.

OUTPUT: The output is a string describing a sequence of edits. Each character in the string should be `'i'`, `'r'`, `'d'`, or `'m'`. The string should be null-terminated.

GOAL: The output is a string describing how `string2` can be modified to obtain `string1`. The output is a string of characters, where each character is `'i'`, `'r'`, `'d'`, or `'m'`, representing an insert, replacement, delete, or match, respectively.

Thus, if `string2` is `ac`, and `string1` is `a`, the optimal output string is `md`.

**Please pay attention to ties.** Your algorithm should favor going as far right in the dynamic programming table as possible, if `string1` is represented vertically and `string2` is represented horizontally.

- This means that when implementing the recursive algorithm, if there are multiple splits that have the same cost, you should take the rightmost option.
- Equivalently, if you are implementing a non-recursive algorithm, if there is a tie while backtracking you should use an insert whenever possible, then a match or replace, and finally a delete.

- If your algorithm breaks ties correctly, if `string1` is `aaa` and `string2` is `aa`, it should output `mmi` (not `mim` or `imm`).

**You must have an implementation of Hirshberg's algorithm to receive full credit.** That is to say, you should have an implementation that takes $O(n + m)$ space. Implementations of the space-inefficient edit distance start with 80% credit. You are not required to use Hirshberg's to receive credit on the leaderboard; if you do not use Hirshberg's as your main algorithm, please clearly label your Hirshberg's implementation for grading.

## Testing Parameters

The `main()` method of the testing program (in `test.c`) takes two arguments, each of which is a file containing edit distance instances. You can test your program by first running `make`, and then running `./test.out testData.txt timeData.txt`.

- All instances on this assignment will have an alphabet of size 4 or 256; in either case the input will be taken as a normal array of `char`s. You may optimize your solution under this assumption (it's OK if your algorithm fails with an alphabet of size, say, 26).
- There are several testing and timing instances provided. Your performance will be judged based on the total time across three instances. The three testing instances will look very similar to those provided in `timeData.txt`.
- All 4-character instances have `string1` as an actual subsequence of human DNA, and `string2` as a perturbed version of this DNA. All 256-character instances have `string1` consist of English text (including punctuation), and `string2` as a perturbed version of this text.
- Tiebreaking is unfortunately necessary and unavoidable even in the large instances.
- Two solutions with running times within .1 seconds of each other will be considered tied for the purposes of this assignment.

## Questions

**Problem 1.** Please describe your implementation at a high level. What techniques and data structures did you use? What optimizations did you use? **Please give an analysis of the space usage of your implementation.**

*Solution.*

### External Memory

**Problem 2.** How many I/Os does it take to find the edit distance (not necessarily the sequence of edits) using the standard $O(nm)$ time algorithm? Please explain your answer.

*Solution.*

**Problem 3. (Extra credit: 15pts)** Give an algorithm that can find the edit distance (not necessarily the sequence of edits) in $O(nm/MB + (n+m)/B)$ I/Os. Prove that your algorithm meets this bound. Can this be used to improve the I/O complexity of Hirshberg's algorithm? If so, prove it; if not, explain why not.

*Solution.*

### Shelving Books with Labels

Let's say you work in a library. You have to put $m$ books (let's call them $b_1, \ldots, b_m$) on $n \leq m$ shelves (which we'll call $s_1, \ldots s_n$). The books are numbered using the Dewey Decimal system, and must be placed in order starting on shelf $s_1$. Furthermore, **there may not be an empty shelf between two shelves that contain books**.[1] For example, if book 10 goes on shelf 2, book 11 must go on shelf 2 or shelf 3. Each shelf can hold any number of books; even all $m$ books may be placed on a single shelf.

This would normally be fairly easy—for example, you could just put all books on the first shelf. Unfortunately, this library also keeps track of $k$ *topics* to help people browse for books they may be interested in. Each shelf $s_j$ has a label $\ell_j$ representing the topics of books on that shelf. Similarly, each book $b_i$ has a list of topics $t_i$ representing what topics are covered in that book.

You were instructed to reprint all the labels on the shelves so that the shelves indicate the topics of their books: if book $b_i$ is on shelf $s_j$, then each topic in $t_i$ can be found in $\ell_j$. However, in an effort to stay green you want to keep the labels as-is, and place the books so that they match the current labels as closely as possible (while still retaining Dewey Decimal order).

Let's say that the *cost* of placing book $b_i$ on shelf $s_j$ is the number of topics $t_i$ that do not appear in the list $\ell_j$. This leads to an algorithmic problem: how can the books be assigned to shelves to minimize the total cost; i.e. the number of missing topics over all books on all shelves?

**Problem 4.** Give an algorithm to find the assignment of books to shelves that minimizes the number of mismatches. Your algorithm should run in $O(nmk)$ time and $O(nm)$ space.

*Solution.*

**Problem 5.** Give an algorithm to find the minimum number of mismatches in $O(nmk)$ time and $O(m)$ space (you do not need to find the optimal assignment of books to shelves).

[1] Your boss at the library is a stickler for aesthetics.

*Solution.*

**Problem 6.** Finally, give an algorithm to find the assignment of books to shelves that minimizes the number of mismatches in $O(nmk)$ time and $O(m)$ space.

*Solution.*

## Tips and tricks

- Remember to create a correct program before worrying about creating a fast one! Even more importantly: create a correct program before creating one that cleverly reuses unnecessary space. Even your first Hirshberg's implementation should probably be fairly wasteful!
- I would fairly strongly suggest that you create a working version of the simple (not space-efficient) edit distance algorithm to help you with debugging.
- Keep an eye on memory management! If you are not careful you may wind up with $\Theta(nm)$ memory usage even with a recursive algorithm.
- As I mentioned in class, a correct implementation of Hirshberg's algorithm almost certainly has disjoint (nonoverlapping) subproblems.
- In class we discussed that maintaining the solution using Hirshberg's algorithm can be a bit tricky. I believe that the easiest way is to do it "bottom-up:" construct a solution in the base case, and pass it to the calling function. The calling function can then allocate space for a solution that combines its two recursive calls, and (again) pass it up to its calling function; and so on.
- As a reminder, valgrind is an excellent tool if you are having trouble keeping track of memory. It is very easy to use, and it is available on the lab computers.
- While it is likely possible to implement an $O(\min\{m, n\})$ space algorithm, it is probably better to implement an $O(n + m)$ space algorithm and then work on other avenues of improving efficiency.