CS 6363.005.19S Lecture 8—February 7, 2019

Main topics are #dynamic_programming with #example/edit_distance.

Prelude

- Homework 2 is due Tuesday, February 19th. Get started early. Dynamic programming is not easy!

Edit Distance

- We're going to continue with dynamic programming today by looking at a way to compare pairs of strings.
- The *edit distance* between two strings is the minimum number of character insertions, deletions, and substitutions required to transform one string into the other.
- The edit distance between FOOD and MONEY is at most four:

```
FOOD → MOOD → MON D → MONED → MONEY
                ^
```

- This distance function was independently proposed by Vladimir Levenshtein (working in coding theory), Taras Vintsyuk (working on speech recognition), and Stanislaw Ulam (working on biological sequences), so it's often referred to as Levenshtein or Ulam distance (but never Vintsyuk distance for some reason).
- We can visualize the editing process by aligning the two strings on top of one another.

```
F O O   D
M O N E Y
```

- There's a gap in the first word for every insertion and a gap in the second word for every deletion. If a column contains two *different* characters then it represents a substitution.
- Another example, ALGORITHM vs ALTRUISTIC. The edit distance is at most 6.

```
A L G O R   I   T H M
A L   T R U I S T I C
```

- But how do we compute it exactly? Let's design a dynamic programming algorithm that, given two strings A[1 .. m] and B[1 .. n], returns the edit distance between A and B.

Recursive Structure

- The first step in any dynamic programming algorithm is developing a recursive algorithm or recurrence.
- This alignment representation suggests a recursive structure.
- **If we remove the last column from the optimal alignment structure, then the remaining columns must represent the shortest edit sequence for the remaining prefixes.**

- There's an easy proof by contradiction here. If the prefixes had a shorter edit sequence, then gluing the last column onto the shorter sequence would give us a better sequence for the original strings than we started with.
- So, we should guess what to do with the last column and have the Recursion Fairy take care of the rest.
- Let's describe this another way.
- The alignment we're looking for represents a sequence of editing decisions ordered (for no particular reason) from right to left.
- We need to make a sequence of decisions, one for each column of the alignment.
- In the middle of these decisions, we have already aligned a suffix of one string with a suffix of the other.

- The total cost of the alignment is just the number of mismatched columns, so our remaining decisions don't depend on the editing operations we've already chosen. They only depend on the prefixes we haven't aligned yet. The Recursion Fairy only needs to care about the two prefixes.

- So let's describe the problem we're giving to the Recursion Fairy using a recurrence. Let Edit(i, j) denote the edit distance between A[1 .. i] and B[1 .. j]. We need to compute Edit(m, n).

Recurrence

- When i and j are both positive, there are three possibilities for the last column.
  - Deletion: The last entry of the bottom row is empty. Here, the edit distance is equal to 1 + Edit(i - 1, j). The 1 is for the deletion and the recursive call is to handle alignment of A[1 .. i-1] and B[1 .. j].

  - Insertion: The last entry of the top row is empty. Here, the edit distance is equal to 1 + Edit(i, j - 1).

  - Substitution(?): Both rows have characters in the last column. If the characters are different, the edit distance is 1 + Edit(i - 1, j - 1). Otherwise, the “substitution” is free so the edit distance is Edit(i - 1, j - 1).
- This case analysis doesn't work when one of i or j is equal to 0, but we can handle the

boundary cases easily.

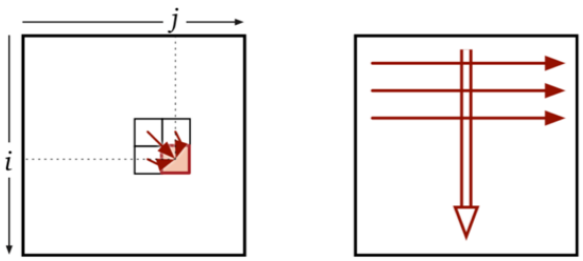
- Converting an empty string into a string of length j requires j insertions, so Edit(0, j) = j.
- Converting a string of length i into the empty string requires i deletions, so Edit(i, 0) = i.

- And that implies, the edit distance of two empty strings Edit(0, 0) = 0. Checks out.
- So the Edit function follows this recurrence.

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \left\{ \begin{array}{l} Edit(i-1,j)+1, \\ Edit(i,j-1)+1, \\ Edit(i-1,j-1)+\left[A[i] \neq B[j]\right] \end{array} \right\} & \text{otherwise} \end{cases}$$

Dynamic Programming

- And now we can find our efficient algorithm using the mechanical memoization process.
  - Subproblems: Each recursive subproblem takes a pair of indices 0 ≤ i ≤ m and 0 ≤ j ≤ n.
  - Memoization: So we can memoize all possible values of Edit(i, j) in a two-dimensional array Edit[0 .. m, 0 .. n].
  - Dependencies: Each entry Edit[i, j] depends only on its three neighboring entires Edit[i - 1, j], Edit[i, j - 1], and Edit[i - 1, j - 1]. Here's a picture of what it looks like.



  - Evaluation order: We only need things from earlier in the same row or from the previous row, so let's fill the array in row-major order: row by row from top down, each row from left to right.
  - Space and time: At this point, we can already figure out the running time without writing any code! The memoization structure uses **O(mn) space**. We can compute each entry Edit(i, j) in O(1) time once we know its predecessors, so computing all the entries will take **O(mn) time**.
- And finally, here's the algorithm! **[swap uses of ins and del]**

```
EditDistance(A[1..m], B[1..n]):
    for j ← 0 to n
        Edit[0, j] ← j

    for i ← 1 to m
        Edit[i, 0] ← i
        for j ← 1 to n
            ins ← Edit[i − 1, j] + 1
            del ← Edit[i, j − 1] + 1
            if A[i] = B[j]
                rep ← Edit[i − 1, j − 1]
            else
                rep ← Edit[i − 1, j − 1] + 1
            Edit[i, j] ← min {ins, del, rep}

    return Edit[m, n]
```

- We started by finding the recursive structure and eventually got to a relatively simple iterative process.
- Now, similar to how we have recursion trees to explain what's going on with our recursive algorithms, we can also look at how the array is filled. I want to emphasize again, though, that **designing the recursive algorithm was the key step**. We're only looking at the array to see what happens in hindsight.

		A	L	G	O	R	I	T	H	M
	0	1	2	3	4	5	6	7	8	9
A	1	0	1	2	3	4	5	6	7	8
L	2	1	0	1	2	3	4	5	6	7
T	3	2	1	1	2	3	4	4	5	6
R	4	3	2	2	2	2	3	4	5	6
U	5	4	3	3	3	3	3	4	5	6
I	6	5	4	4	4	4	3	4	5	6
S	7	6	5	5	5	5	4	4	5	6
T	8	7	6	6	6	6	5	4	5	6
I	9	8	7	7	7	7	6	5	5	6
C	10	9	8	8	8	8	7	6	6	6

- A number at position (i, j) is the value of Edit(i, j). The arrows indicate which predecessors could have defined each entry. A horizontal arrow means we did a deletion. A vertical arrow represents insertion, and a diagonal arrow represents substitution. Red arrows are the “free" substitutions that don't increase the cost.
- The algorithm we wrote only computes the total edit distance, not the optimal sequence of operations. However, any path of arrows from (0, 0) to (m, n) represents an optimal

sequence of operations.

- If we want to recover an optimal sequence, we can figure out which arrows lead into any entry (i, j) in O(1) time by checking its three dependences. In O(m + n) additional time, we can reconstruct an optimal sequence by tracing *backwards* from (m, n). So we might find any of these three optimal transformations.

```
A L G O R I   T H M
A L T R U I S T I C

A L G O R   I   T H M
A L   T R U I S T I C

A L G O R   I   T H M
A L T   R U I S T I C
```

- As I said before, if you want to find an optimal solution for some problem using dynamic programming, its easiest to write an algorithm to find the *value* of the solution and then make any changes necessary to get the solution using the values.

Common Patterns

- The two examples we've seen so far of rod cutting and edit distance are examples of *sequence dynamic programming*. We're given one or more sequences (arrays) as input, and our goal is to compute an optimal sequence as output: a sequence of word boundaries or a sequence of editing operations, for example.
- Usually, the input to recursive subproblems in these cases consists of *prefixes* or *suffixes* of the input arrays, possibly with additional information.
- For example, if I wanted to use at most k substitutions, then I'd use a third parameter telling me how many substitutions I have remaining in my recursive subproblem.

Saving Space vs. Finding the Solution

- Recall how for Fibonacci numbers we could replace our O(n) size array with a pair of integers to take the space usage down to O(1).
- We can do a similar trick for edit distance. That inner loop only relies on the previous row of the array. So we could just keep one previous row around for a space usage of O(n).
- But now we have an issue if we want to recover the sequence of edits. We've discarded almost all of the array, so there's no tracing backwards from (m, n).
- There are ways to get the best of both worlds, though. In 1975, Dan Hirschberg described a way to combine divide-and-conquer with dynamic programming so you could recover the optimal sequence of edits in O(mn) time using only O(m + n) space.
- The details are beyond the scope of this course, but the high level idea is to use dynamic programming to figure out a value h where the sequence transforming A[1 .. m] into B[1 .. n] first transforms A[1 .. m/2] into B[1 .. h] and then transforms A[m / 2 + 1 .. n] into B[h +

1 .. n].

- Then we recursively compute those optimal sequences.
- Somehow it all works out to run in O(mn) time with O(m + n) space. See Erickson D if you're interested in these details or want to learn more advanced dynamic programming techniques.
- Next Tuesday we'll go back to only computing optimal solution values, but the problems will have more interesting recursive structures.

Greed is Bad, Actually

- Let's finish with a rant.
- If you're very lucky, you might be able to run a greedy algorithm for whatever problem you're trying to solve.
- This means committing to a good looking first decision directly, without looking at any recursive subproblems, and then letting the Recursion Fairy do everything else. So backtracking without ever looking tracking back.
- It seems natural, but very few problems can be solved correctly in this way.
- For example, for rod cutting, you might just take the most expensive piece possible and then commit to recursively segment the rest of the string.
- But even a simple example like <1 1 1 2> shows that doesn't work.
- So remember, GREEDY ALGORITHMS NEVER WORK (except very rarely).
- If you ever get an inkling that a greedy approach might work, you really want to use dynamic programming instead. Really.
- Later in the semester, we'll go over what's involved in designing a greedy algorithm and proving it correct. Until then, don't even think about using one in my class.