CS 4349 Lecture—September 20th, 2017

Main topics for #lecture include #dynamic_programming and #edit_distance.

Prelude

- Homework 3 is due.
- Homework 4 due Wednesday, September 27th.

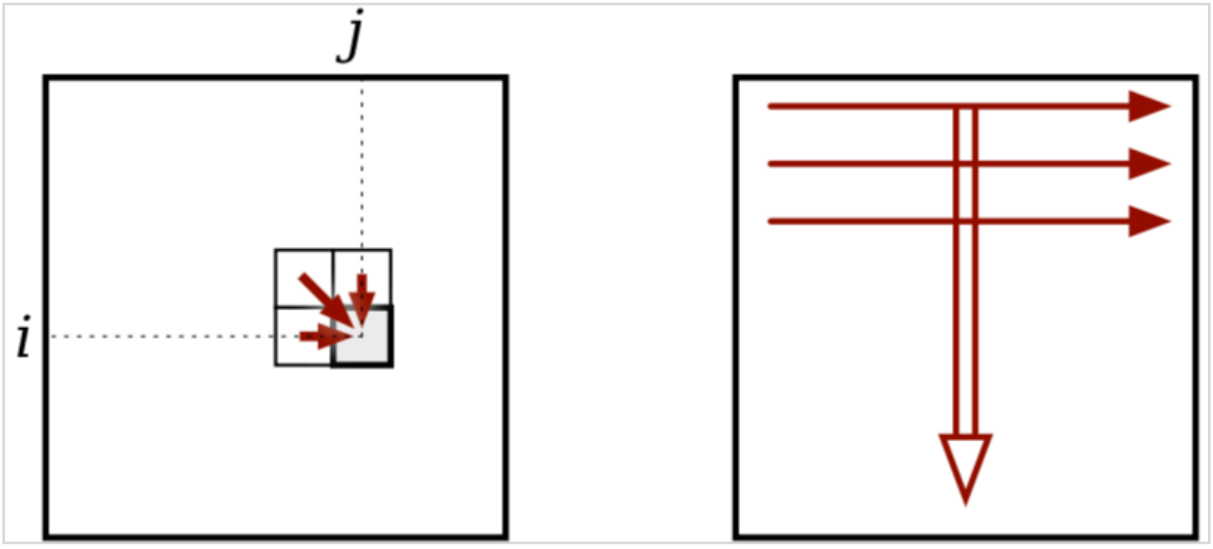
Edit Distance

- So far we've been playing with single arrays of numbers. Now, let's try something different.
- 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 ➔ MOND ➔ MONED ➔ MONEY
- You could also show this by placing the two strings on top of each other with a gap in the first string for every insertion and a gap in the second string for every deletion. Columns with two different characters represent substitutions, so the number of editing steps is the number of columns that don't contain the same character twice.
  - F O O _ D
  - M O N E Y
- 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
- Let's design an algorithm that, given two *strings* A[1 .. m] and B[1 .. n], returns Edit(A[1 .. m], B[1 .. n]), the edit distance between A and B.
- Once again, we need to find some recursive structure, and it all comes down to making some first choice. Look at the gap representations I drew. We need to decide what pair goes in the last column.
- Earlier, we had a choice of _ C, M _, or M C.
- If the optimal gap representation for the whole strings ends with M C, then it starts with the optimal gap representation for the substrings to the left.
- So let's write Edit(A[1 .. m], B[1 .. n]) using a recursive definition that handles all the choices we could make for that last column.
- For a proposition P, let [P] = 1 if P is true and [P] = 0 otherwise.
- Ignoring base cases for now, Edit(A[1 .. m], B[1 .. n]) = min{
  - Edit(A[1 .. m], B[1 .. n - 1]) + 1 We do an insertion.
  - Edit(A[1 .. m - 1], B[1 .. n]) + 1 We do a deletion.

    - Edit(A[1 .. m - 1], B[1 .. n - 1]) + [A[m] ≠ B[n]]}
- The base cases occur whenever a string is empty. To convert an empty string to one of length n, we use n insertions. We use m deletions to convert a string of length m to an empty string. So
    - Edit(eps, B[1 .. n]) = n, and Edit(A[1 .. m], eps) = m. Both imply Edit(eps, eps) = 0.

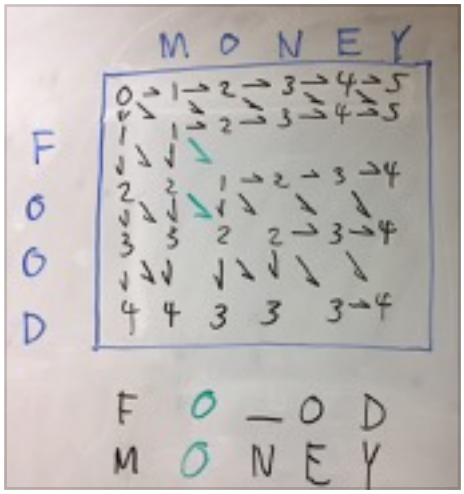
- To use dynamic programming, we need some simple way to index these subproblems. For longest increasing subsequence, we use the first index of a suffix of the input array. **what can we use here for a simple index or indices?**
- We're always working with prefixes, so let Edit(i, j) denote the edit distance between the prefixes A[1 .. i] and B[1 .. j]. We need to return Edit(m , n).
- Edit(i, j) =
    - i if j = 0
    - j if i = 0
    - min{
        - Edit(i, j - 1) + 1,
        - Edit(i - 1, j) + 1,
        - Edit(i - 1, j - 1) + [A[i] ≠ B[j]]} otherwise

- And now we can find our efficient algorithm.
- We need to store all the subproblems. **what data structure should we use?**
- We'll use a two-dimensional array Edit[0 .. m][0 .. n].
- We can already figure out the space and running time. **what are they?** O(m n) space. O(m n) time.

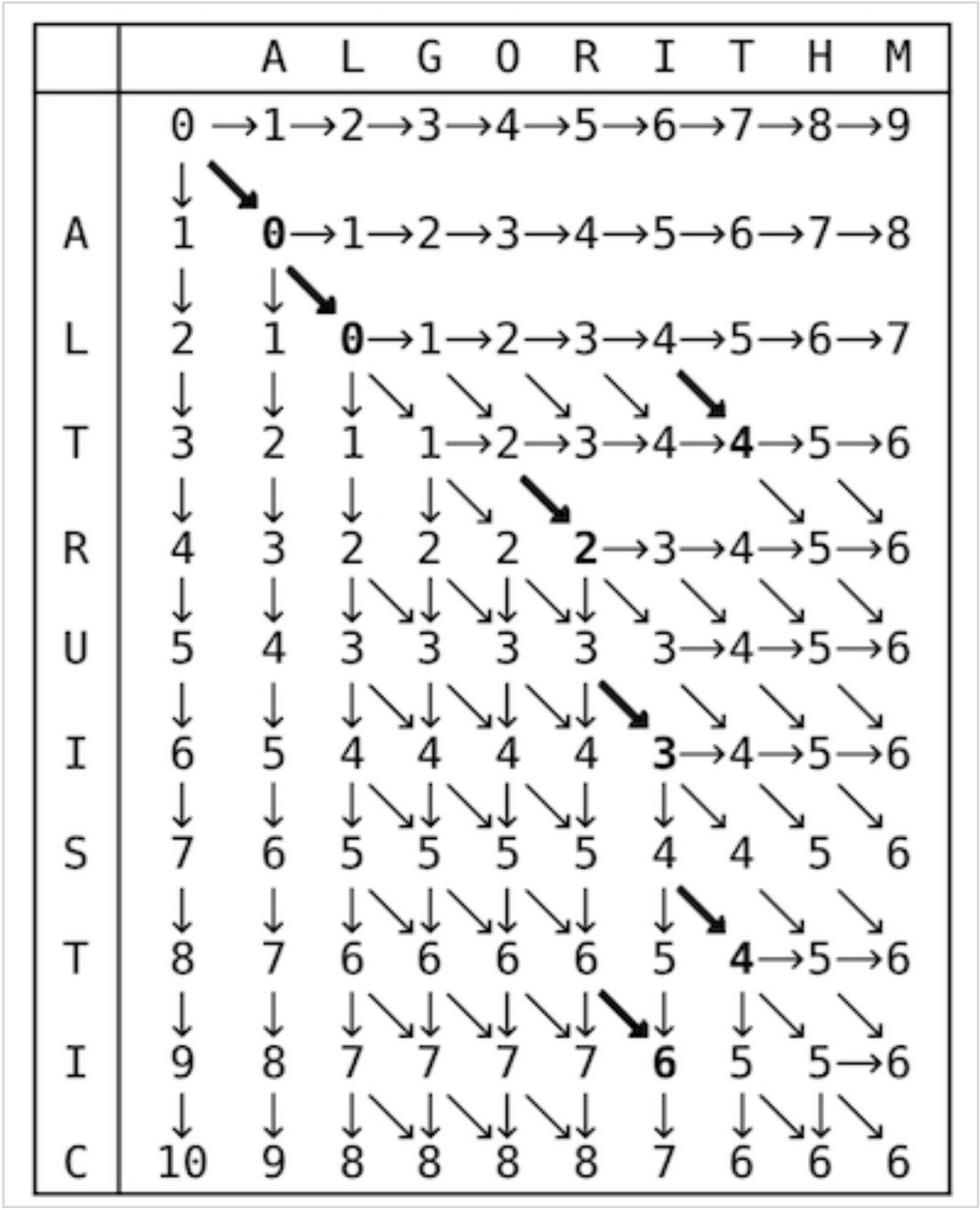


- We depend only on these three items, and we can fill the array in row-major order—row by row from top down, each row from left to right
- And putting it all together, we get the following algorithm. Again, you if did everything else up to this point, you wouldn't need to write out the pseudocode.

- EditDistance(A[1 .. m], B[1 .. n]):
  - for j ← 0 to n Fill row 0
    - Edit[0][j] ← j
  - for i ← 1 to m
    - Edit[i][0] ← i
    - for j ← 1 to n
      - if A[i] = B[j]
        - Edit[i][j] ← min {Edit[i - 1][j] + 1, Edit[i][j - 1] + 1, Edit[i - 1][j - 1]}
      - else
        - Edit[i][j] ← min{Edit[i - 1][j] + 1, Edit[i][j - 1] + 1, Edit[i - 1][j - 1] + 1}
  - return Edit[m][n]
- OK, so let's look at an example of how to fill in this table. I'll draw arrows for each subproblem's optimal choices.



- So the answer is 4.
- For ALGORITHM vs. ALTRUISTIC:



- Before we move on, let's consider two ways we can extend or improve upon the edit distance algorithm.
- First, we have been focusing so far on just computing costs. We wanted the maximum selling price for cutting rods, but not the actual lengths to cut. We wanted the length of the longest increasing subsequence, but not the actual subsequence. We wanted the edit distance, but not the sequence of edits.
- But, it turns out computing the cost is usually just as difficult as computing the solution leading to that cost. So we focus on cost to simplify things a bit.
- What if you want the actual sequence of edits? Well, we already computed it!
- You can do one of two things:
    1. extend the array to store both the value Edit(i, j) and a pointer to the cell you used to

get that value. After filling the array, just walk backwards along these pointers from Edit[m][n] to Edit[0][0].

2. Do a search where you start at Edit[m][n]. For each element Edit[i][j] you look at, you check its dependencies again, and walk to one that would give you the the value at Edit[i][j]. Stop when you reach Edit[0][0].

- Either way, you discover how you computed each entry. Stepping horizontally means an insertion. Stepping vertically means a deletion. Stepping diagonally means either a substitution or the characters already matched.
- **do FOOD to MONEY example**
- Generally, you can focus on finding the value of a best solution, say, the maximum selling price for cutting a rod, and then do a backwards search in the dynamic programming table to learn what choices made that best value possible.

- Now, let's go back to focusing on just the value. This algorithm for edit distance used O(m n) space. That's O(n^2) if m = n. Could we have done better?
- To warm up, let's consider our first example of dynamic programming. We wrote this iterative algorithm for Fibonacci numbers.
- IterFibo(n):
  - F[0] ← 0
  - F[1] ← 1
  - for i ← 2 to n
    - F[i] ← F[i-1] + F[i-2]
  - return F[n]
- This algorithm takes O(n) space. **Can anybody recommend a way to reduce the space used?**
- We don't need *all* the past entries. Just the last two. So why are we keeping the whole array around?
- IterFibo2(n):
  - prev ← 0 F(0)
  - curr ← 1 F(1)
  - for i ← 2 to n
    - next ← curr + prev
    - prev ← curr
    - curr ← next
  - return curr
- Now we're using O(1) space.
- Back to edit distance. We have this O(m n) sized array that we're filling in row-by-row. **Can anybody recommend a way to reduce the space used?**
- We only need the previous row!

- EditDistance2(A[1 .. m], B[1 .. n]):
  - for j ← 0 to n
    - curr[j] ← j
  - for i ← 1 to m
    - next[0] ← i
    - for j ← 1 to n
      - if A[i] = B[j]
        - next[j] ← min {curr[j] + 1, next[j - 1] + 1, curr[j - 1]}
      - else
        - next[j] ← min{curr[j] + 1, next[j - 1] + 1, curr[j - 1] + 1}
    - for j ← 0 to n
      - curr[j] ← next[j]
  - return curr[n]
- Great, we're only using O(m + n) space, which is the minimum needed just to hold the input strings!
- So can we do both things? Yes, but it involves some more advanced techniques and solving a different recurrence. See Erickson 6 if you're interested in how.