CS 4349 Lecture—September 18th, 2017

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

Prelude

- Homework 3 is due Wednesday, September 20th.
- Office hours have been rescheduled. Kyle's run from 2:00pm–3:00pm on Tuesday. Jon's are 4:00pm–6:00pm on Mondays. Other 4349 TAs hold their hours in the shared TA labs. See the course webpage for a schedule.

Longest Increasing Subsequence

- As usual, let A[1 .. n] be an array of numbers.
- A *subsequence* of A is a sequence <A[i_1], A[i_2], …, A[i_k]> such that 1 ≤ i_1 < i_2 < … < i_k ≤ n.
- They don't have to be consecutive, but they do have to appear in order.
- A subsequence is *increasing* if A[i_1] < A[i_2] < … < A[i_k].
- Our goal today is to design an algorithm that given A[1 .. n] finds *the length* of the longest increasing subsequence.
- As you might guess from last week's lecture, we're going to use dynamic programming.

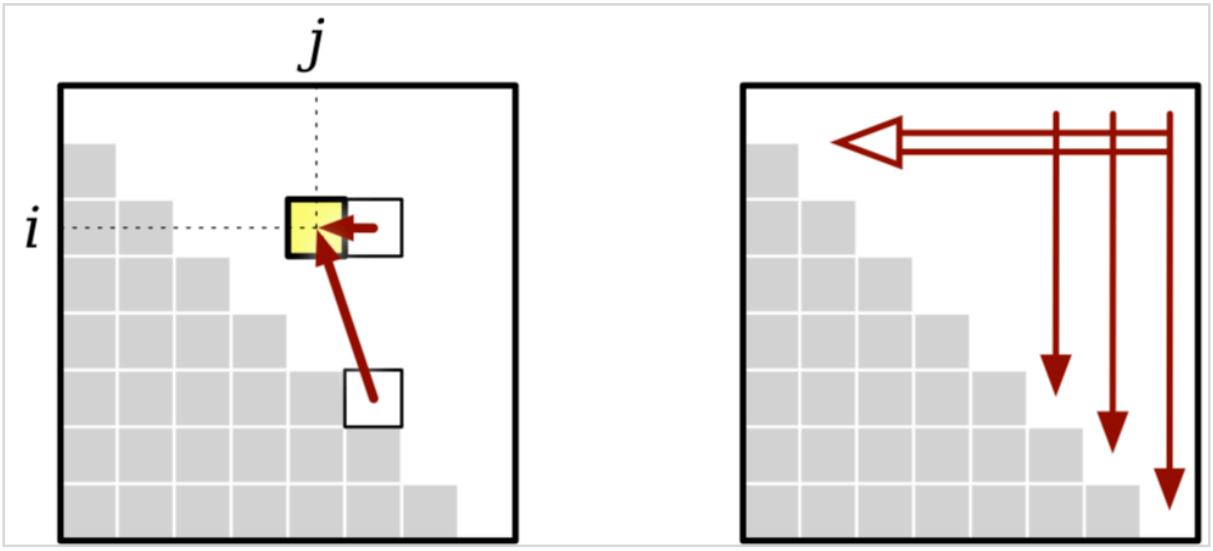
- So, the first step is to find the recursive structure so we can design a recurrence.
- And this usually involves making some first choice and dealing with the consequences.
- A sequence is the first element followed by the rest of the sequence.
- So, a useful first choice is whether or not to include the first member of your array.
- The longest increasing subsequence of A[1 .. n] is either
  - The longest increasing subsequence of A[2 .. n] or
  - A[1] followed by… **what has to come next?**
- Right, we'd not only want a longest increasing subsequence of A[2 .. n], but we'd need one with the extra restriction that each element is larger than A[1].
- Sometimes, your first choice has consequences beyond just solving the rest optimally, so you might need to think of a slightly different problem to solve.
- But at that point, we're fine. Just need to make sure when we do the recursion that we recognize what the previously chosen value was.
- What would this look like as a recursive algorithm?
- LISBigger(prev, A[1 .. n]) returns the length of the longest increasing subsequence of A[1 .. n], where each member of the subsequence is larger than prev.

- LISBigger(prev, A[1 .. n]):
  - if n = 0
    - return 0
  - else
    - max ← LISBigger(prev, A[2 .. n])
    - if A[1] > prev
      - L ← 1 + LISBigger(A[1], A[2 .. n])
      - if L > max
        - max ← L
    - return max
- To get the length of the longest increasing subsequence of A[1 .. n], call LISBigger(-∞, A[1 .. n]) since every element of A is fine for the start of the sequence.

- But how are we going to use dynamic programming here? prev is an integer, and A[..] is a subarray. That seems like a lot of subproblems. And we need some way to address them in a data structure.
- There are two things here that help.
- First, prev is always either -∞ or an element of the input array. So we can use an index i and say prev = A[i]. For simplicity, we can say A[0] = -∞.
- Second, we alway recurse on a *suffix* of the input array, so we can address the suffix by its first index j.
- From there, we can write a recurrence that directly describes how to fill a data structure. Let LIS(i, j) denote the length of the longest increasing subsequence of A[j .. n] with all elements larger than A[i].
- Our goal is to compute LIS(0, 1) **you need to specify where the final answer lies sometimes since it's not always obvious like in a standard recursive algorithm**
- For all i < j, LIS(i, j) =
  - 0 if j > n
  - LIS(i, j + 1) if A[i] ≥ A[j]
  - max{LIS(i, j + 1), 1 + LIS(j, j + 1)} otherwise

- The recurrence tells us exactly what subproblems we need to solve and how to index them in an array. **it's often much easier or sometimes only possible to do these next steps with a recurrence compared to a traditional recursive algorithm**
- **what data structure should we use?**
- We'll use a two-dimensional array LIS[0 .. n][1 .. n] where each LIS[i][j] will equal LIS(i, j) when we're done.
- We can already tell the algorithm will use O(n^2) space to store the array and O(n^2) time to fill its entries.

- But what order to we fill in the array? Let's use a picture.



- LIS[i][j] depends only on entries one column to the right. So let's fill in the array in decreasing column major order, column by column, each column from, say, top down.
- Here's our final algorithm:
- LIS(A[1 .. n]):
  - A[0] ← -∞ To easily handle i = 0
  - for i ← 0 to n Do the base cases
    - LIS[i][n + 1] ← 0
  - for j ← n down to 1
    - for i ← 0 to j - 1
      - if A[i] ≥ A[j]
        - LIS[i][j] ← LIS[i][j + 1]
      - else
        - LIS[i][j] ← max{LIS[i][j + 1], 1 + LIS[j][j + 1]}
  - return LIS[0][1]
- Again, you don't need to write the iterative algorithm if you have already given a recurrence, given a data structure, and described way to fill the data structure.

- We didn't get to it in lecture, but if you're interested, here's another recurrence we could have used to solve this problem.
- In LIS(i, j), we use the first parameter to remind us of a choice we already made. Instead, let's use the fact that we're even evaluating a particular subproblem represent our choice.
- Let LIS'(i) denote the length of the longest increasing subsequence of A[i .. n] *that starts with* A[i]. LIS'(0) will start with A[0] which we are pretending is -∞. A[0] isn't really a member of A[1 .. n], so the longest increasing subsequence of A[1 .. n] has length LIS'(0) - 1.
- For $0 \leq i \leq n$, LIS'(i) =
  - 0 if i > n
  - 1 + max{LIS'(j) | j > i and A[j] > A[i]} otherwise

        - (If no elements qualify for the max's condition, we'll say the max is 0.)
- Now we have one parameter, so we'll use O(n) space. Each evaluation takes O(n) time to check all j > i, so the total time is O(n^2).
- We can use an array LIS'[0 .. n] to store recursive answers.
- [[] ← ← ←] We depend on larger indices, so fill the array in reverse order.
- LIS2(A[1 .. n]):
    - A[0] ← -∞
    - LIS'[n+1] = 0
    - for i ← n down to 0
        - LIS'[i] ← 1
        - for j ← i + 1 to n
            - if A[j] > A[i] and 1 + LIS'[j] > LIS'[i]
                - LIS'[i] ← 1 + LIS'[j]
    - return LIS'[0] - 1

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. **What is a single choice we can make?**
- We can decide what goes in one of the extreme columns. To make the final algorithm easier to describe, we'll say our choice is what goes in the last column. Earlier, we had a choice of _ C, M _, or M C.

- We picked MC. **What is true about the gap representation regarding the remaining substrings to the left of those characters?**
- 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.
- But we'll have to do that Wednesday.