CSE541 Class 3

Jeremy Buhler

January 23, 2019

1 Problem: Longest Common Subsequence

Let's do some stringology!

- Consider the problem of diffing two text files X and Y, each consisting of zero or more lines.
- To diff X and Y, we first find an ordered list of common lines between them.
- Then, we insert and delete lines between the common lines to turn X into Y.
- **Example**: compare X

```
foo
bar
baz
quux
```

to Y

```
bar
xyzzy
plugh
baz
foo
quux
```

- The 3 lines "bar, baz, quux" occur in the same order in each file.
- Resulting diff from X to Y is:

```
-foo
 bar
+xyzzy
+plugh
 baz
+foo
 quux
```

- To minimize the number of insertions and deletions needed to transform one file to the other, we want to find the longest possible ordered list of common lines between files.

More formally...

- Let $X[1..n]$ and $Y[1..m]$ be two strings of symbols (think of each line of text as a "symbol" over a ginormous alphabet)
- Find a sequence of index pairs $(x_1, y_1), (x_2, y_2) \ldots, (x_q, y_q)$ such that
  1. $x_1 < x_2 \ldots < x_q$, and $y_1 < y_2 \ldots < y_q$
  2. $X[x_j] = Y[y_j]$ for $1 \leq j \leq q$
  3. the length q of the sequence is maximal.
- This is the *longest common subsequence (LCS) problem.*

2 Recursive Solution Strategy

How to proceed?

- We want to look at ways to match up symbols between X and Y.
- A greedy approach might pick the first available pair, last available pair, or ???
- Once again, none of these simple choices are optimal for every problem instance!

Let's consider a recursive decomposition that could lead to DP.

- A common subsequence can be divided into its last common pair (x_q, y_q) and the "rest," which is confined to $X[1..x_q - 1]$ and $Y[1..y_q - 1]$.
- Intuitively, we want to identify that last common pair, then recursively find a long common subsequence on those prefixes of X and Y.

On with the algorithm!

- Suppose we want an LCS for $X[1..n]$ and $Y[1..m]$.
- Here are two possible first choices that we can always make: leave $X[n]$ unmatched, or leave $Y[m]$ unmatched (not mutually exclusive).
- If $X[n] = Y[m]$, we also have a third choice: match $X[n]$ to $Y[m]$.
- **Complete Choice**: for every input, is one of these three choices consistent with a longest common subsequence?
- If $X[n] = Y[m]$, then the LCS *might* match them up, or not.
- It is not possible to match *both* $X[n]$ and $Y[m]$ to distinct characters of the other sequence, as they are the last in their sequences, and matches must respect order of both seqs.

- Hence, an LCS that does not match $X[n]$ to $Y[m]$ must match at most one of $X[n]$ and $Y[m]$.
- Conclude that an LCS must make one of the three choices above. QED
- **Inductive Structure**: making any one of the three choices leaves us with a smaller subproblem of the original without external constraints.
- **Pf**: the 3 choices leave LCS subproblems for:
  - $X[1..n-1]$ and Y
  - X and $Y[1..m-1]$
  - $X[1..n-1]$ and $Y[1..m-1]$.
- Any solution to each of these subproblems can feasibly be combined with its corresponding first choice. QED
- **Optimal Substructure**: for each first choice, if we can solve the corresponding subproblem optimally and combine it with that choice, we get a common substring of the full strings that is as long as possible *given that choice*.
- **Pf**: Let $\text{LCS}(A, B)$ be a longest common subseq of A and B.
- Then for the 3 cases, we have that $|\text{LCS}(X, Y)|$ is respectively given by
  - $|\text{LCS}(X[1..n-1], Y)| + 0$,
  - $|\text{LCS}(X, Y[1..m-1])| + 0$, and
  - $|\text{LCS}(X[1..n-1], Y[1..m-1])| + 1$.
- In each case, note that we have shown the costs to be separable and so can apply the standard contradiction argument. QED

3 Formulating the DP Recurrence

We have shown correctness of a recursive algorithm for LCS. Now to formulate a DP solution!

- As before, think about the general shape of the subproblems that the recursive algorithm must solve.
- A general subproblem computes LCS for some prefixes $X[1..i]$ and $Y[1..j]$ of X and Y.
- Call this subproblem $[i, j]$, since these are the two free variables needed to uniquely identify it.
- The result of solving this subproblem is $\text{LCS}(X[1..i], Y[1..j])$. Let $L(i, j)$ be the size of this solution.

We can now write our recurrence in terms of our subproblem notation.

- Recursive procedure solves up to three smaller subproblems of problem $[i, j]$ and takes the best result.
- More formally, it computes $L(i, j)$ as

$$L(i,j) = \max \begin{cases} L(i-1,j) \\ L(i,j-1) \\ L(i-1,j-1)+1 \quad (\text{if } X[i] = Y[j]) \end{cases}$$

- *Base cases*: $L(i, 0) = L(0, j) = 0$ (no symbols for LCS).
- *Goal*: compute $L(n, m)$.

To finish up and prove efficiency, what is our computation order for subproblems?

- To compute $L(i, j)$, we need to know (possibly) $L(i-1, j-1)$, $L(i-1, j)$, and $L(i, j-1)$.
- Hence, before we compute $L(i, j)$, we need to know the values for subproblems that are smaller in one or both dimensions.
- Easiest to think about this as filling in a 2D "table" of LCS sizes:

- One of many feasible orderings of subproblems: compute $L(i, *)$ for each i from 1 to n, filling in each row in order of increasing j.
- Each subproblem's optimal cost can be computed in $O(1)$ time from previously solved subproblems, since combining operation is just a max over 3 things.
- We solve $\Theta(n \times m)$ subproblems, in constant time per subproblem, for a total algorithmic cost of $\Theta(nm)$.

What does the traceback for LCS look like?

- Solution is implicit in the sequence of choices we made to compute $L(n, m)$.
- If, when we fill in the table of values for L, we remember which choice we made at each step (i.e. which term gave the max), then we can trace back through these choices to reconstruct the solution.