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 A New 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 ???

• Unfortunately, none of these simple choices are optimal for every problem instance!

• Rather than keep looking for a single best choice, we’ll extend our approach.

New idea: consider more than one first choice!

• 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 $Y[m]$.

• We will devise an algorithm that finds the longest common subsequence given each of the three choices, and then keeps the best one.

• **Question**: is one of these three choices consistent with a longest common subsequence?

• If $X[n] = Y[m]$, then the LCS *might* match them up.

• If $X[n] \neq Y[m]$, it is not possible to match both $X[n]$ and $Y[m]$ to distinct characters in the other sequence, as they are the last in their sequences, and matches must respect order of both seqs.

• Hence, an LCS must match at most one of $X[n]$ and $Y[m]$.

• Conclude that an LCS must make one of the three choices above. QED.
• This argument is called the *Complete Choice Property*, analogous to the Greedy Choice Property: there exists an optimal solution that makes at least *one* of the first choices considered by the algorithm.

We can now formulate a recursive algorithm analogous to the greedy approach. For each first choice, we recursively solve the remaining subproblem. As before, we’d better check that these subproblems are well-defined, and that we can put them back together with our first choices.

• **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:
  
  - \(X[1..n-1] \text{ and } Y\)
  - \(X \text{ and } Y[1..m-1]\)
  - \(X[1..n-1] \text{ and } Y[1..m-1]\).

• **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 LCS(\(A, B\)) be longest common subseq of \(A\) and \(B\).

  Then for the 3 cases, we have that \(|\text{LCS}(X, Y)|\) is respectively given by
  
  - \(|\text{LCS}(X[1..n-1], Y)| + 0,\)
  - \(|\text{LCS}(X, Y[1..m-1])| + 0, \text{ and}\)
  - \(|\text{LCS}(X[1..n-1], Y[1..m-1])| + 1.\)

• In each case, note that we have shown the costs to be separable and so can apply the standard contradiction argument. QED

As before, we can use these properties to prove inductively that the recursive algorithm yields a global optimum. Detailed proof is similar to that for greedy algorithms and is left as an exercise.

3 Formalizing the Recursive Algorithm

We have shown correctness of a recursive algorithm for LCS. How should we (carefully) write this algorithm down?

• As the recursive algorithm runs, it solves a bunch of *subproblems*, all of which are the same “shape” as the full problem (by IS) but smaller.

• To make it easier to think about this process, we will start by formally defining the set of *all possible subproblems that the algorithm might have to 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 (objective value) of this solution.

Now, how does the algorithm actually derive \(L(i, j)\)?

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

\[
L(i, j) = \max \left\{ \begin{array}{c}
L(i - 1, j) \\
L(i, j - 1) \\
L(i - 1, j - 1) + 1 \quad \text{(if } X[i] = Y[j])
\end{array} \right\}
\]

- This formal description of the recursive algorithm is called a \textit{recurrence}.
- As with other recurrences you’ve seen, to completely specify it, we need to give base(s) for the recursion.
- \textit{Base cases}: \(L(i, 0) = L(0, j) = 0\) (no symbols for LCS).
- Must also specify \textit{our goal}: compute \(L(n, m)\).

4 Alternatives to Recursion

We now have a nifty, formal description of a recursive algorithm for LCS. But do we really want to implement it recursively?

- To solve each subproblem \([i, j]\), we must solve at least two smaller subproblems.
- Subproblems are at most one smaller in each dimension than the original.
- It follows that the total number of subproblems computed by the recursive algorithm is \(\Omega(2^{\min(n,m)})\).
- \textbf{Wait...} I thought we didn’t want exponential-time algorithms!

Why is this approach so slow?

- There aren’t that many \textit{distinct} subproblems. In fact there are only \(\Theta(nm)\) of them (range of \(i\) times range of \(j\)).
- Clearly, then, if we are solving exponentially many subproblems, we’re solving some of them more than once!

Let’s consider an analogy to a well-known mathematical calculation.
• Consider the Fibonacci sequence

\[ F(i) = F(i - 1) + F(i - 2) \]

(with \( F(0) = F(1) = 1 \)).

• If we compute, say, \( F(5) \) by naively following the recurrence, we compute the following tree of subproblems:

• As for our recursive LCS algorithm, it takes \( \Omega(2^n) \) subproblem computations to get our answer.

• But there is trivially a lot of repetition here!

• A much better algorithm would be to solve every subproblem exactly once! Is this possible?

• For Fibonacci, a general subproblem is indexed by the single index \( i \).

• **Problem:** find an computation ordering on the set of all subproblems so that, when we compute a given \( F(i) \), we have already computed the values needed to compute \( F(i) \) according to its recurrence.

• For Fibonacci, we can solve subproblems in increasing order of \( i \).

• This approach lets us compute \( F(n) \) in time \( \Theta(n) \).

Analyzing a recurrence to find an ordering of all subproblems that allows us to compute each subproblem only once is called *dynamic programming*. It can turn naively exponential-time recursive algorithms into polynomial-time algorithms.

• Let’s apply this insight to LCS.

• 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.

• In general, the cost of a dynamic programming algorithm is

\[
(\# \text{ of subproblems}) \times (\text{cost per subproblem}).
\]

• We solve $\Theta(n \times m)$ subproblems, in constant time per subproblem, for a total algorithmic cost of $\Theta(nm)$.

One more thing... we have a fast algorithm to compute the size of the LCS, but how do we get the LCS itself?

• 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 “replay” these choices to reconstruct the solution.

• This is called “computing the traceback”, and it takes time proportional to the solution size.

• (There are standard tricks you can do to compute the traceback efficiently even if you don’t want to store the entire table of subproblem solutions in memory.)