Dynamic Programming

1 Deciding whether or not a string is an interleaving

For bit strings \( X = x_1 \ldots x_m \), \( Y = y_1 \ldots y_n \) and \( Z = z_1 \ldots z_{m+n} \), say that \( Z \) is an interleaving of \( X \) and \( Y \) if there is a way to interleave the bits of \( X \) and \( Y \), maintaining the left-to-right order of each, that yields \( Z \). For example, if \( X = 101 \) and \( Y = 01 \), then \( x_1 x_2 y_1 x_3 y_2 = 10011 \) is an interleaving of \( X \) and \( Y \), while \( 11010 \) is not.

**Problem.** Given strings \( X \), \( Y \), and \( Z \), determine whether \( Z \) is an interleaving of \( X \) and \( Y \).

**Find an efficient algorithm for this problem.** You should give the complexity of your algorithm as a function of \( m = |X| \) and \( n = |Y| \).

**Question 1.** What algorithmic strategy would you use to solve this problem?

*Answer:* We can rule out divide-and-conquer as a likely candidate, as there does not seem to be a reasonably efficient na"ïve algorithm for this problem. We could try to develop a greedy algorithm, but the natural ideas – greedily matching a symbol first against \( X \) then \( Y \), say – clearly fail. So we are left with dynamic programming as the only technique we have seen so far that has any reasonable chance of working.

**Question 2.** What kind of subproblems would you consider? Can you bound their number?

*Answer:* Natural choices are to try to match either the first or last symbol of \( Z \) with the corresponding, first or last (respectively) symbol of \( X \) or \( Y \), and recursively match the suffix/prefixes of \( Z \) against the rest of \( X \) and \( Y \).

As for bounding the number of solutions, here is a first, loose bound. Each subproblem in which \( Z \) has length \( m + n - k \) matches that prefix/suffix of \( Z \) against prefix/suffixes of \( X \) and \( Y \) of total length \( m + n - k \), where these \( k \) symbols could either have been taken from \( X \) or \( Y \)—note that the order does not matter, only how many have been taken from \( X \) or \( Y \), respectively. Between 0 and \( k \) of these symbols were taken from the end of \( X \) and the rest must have been taken from the end of \( Y \). It’s easy to see that there are at most \( \sum_{k=0}^{m+n}(k+1) = (m+n)(m+n+1)/2 = O((m+n)^2) \) subproblems.

But actually, we can do better: at most \( m \) symbols can be taken from \( X \) and at most \( n \) can be taken from \( Y \). So actually, the subproblem is described by the number of symbols remaining in
each string (which determines the number of symbols remaining in Z). This shows that actually, there are only \((m + 1)(n + 1) = O(mn)\) subproblems.

**Question 3.** So, what is your proposed algorithm?
There are two possible kinds of answers. One is given by a recursive algorithm that “memoizes” (stores) the solutions to subproblems, and the other forms a \((m + 1) \times (n + 1)\) table indicating how much of X and Y has been matched in the subproblem, with the original problem corresponding to \((0, 0)\). For the latter kind of algorithm, \((0, 0)\) is feasible if either the end symbols of Z and X match and \((1, 0)\) is feasible or if the ends of Z and Y match and \((0, 1)\) is feasible. In general, \((i, j)\) is feasible if either the \(i\)th character from the end of X matches the \(i + j\)th character from the end of Z and \((i + 1, j)\) is feasible, or if the \(j\)th character from the end of Y matches the \(i + j\)th character from the end of Z and \((i, j + 1)\) is feasible. In \((m, n)\), our base case, no characters remain in X, Y, or Z and so this is taken to be trivially feasible (if we can reach it).

**Question 4.** What is your proof of correctness?
Answer: we prove correctness by induction on the length of Z that remains to be matched. Our base case will be when Z is the empty string, and X and Y must also be empty. This is trivially a match, which is what our algorithm reports.

For the induction step, we suppose our algorithm can detect whether or not a string Z of length at most \(m + n - 1\) matches strings X and Y of total length at most \(m + n - 1\) (i.e., when we are filling in cell \((i, j)\), all cells \((i', j')\) with \(i' + j' > i + j\) correctly indicate whether or not such an interleaving is possible). Now, if Z is an interleaving of X and Y, some character at the end must come from X or from Y. Then if we delete the last character of that string, and the last character of Z, we obtain new strings \(Z', X', \) and \(Y'\) where \(Z'\) has length \(m + n - 1\), and \(X'\) and \(Y'\) have total length \(m + n - 1\), and for which our original interleaving with the last character deleted demonstrates that \(Z'\) is indeed an interleaving of \(X'\) and \(Y'\). Thus, in this case by our induction hypothesis, our recursive call or corresponding table entry correctly indicates that this is feasible and the end character of Z indeed matches the corresponding end character of X or Y. So in this case, our algorithm correctly reports that these strings are an interleaving.

Conversely, let’s suppose that our algorithm reports that Z is an interleaving of X and Y. Then by construction, the end character of Z must match one of the end characters of X or Y, for which the corresponding subproblem in which \(X'\) and \(Y'\) have total length \(m + n - 1\), obtained by deleting that string’s end character, is reported as “feasible” by the algorithm. By the induction hypothesis, again, in this case it must be that \(Z'\) really is an interleaving of \(X'\) and \(Y'\). In this case, we can extend the interleaving of \(X'\) and \(Y'\) to an interleaving of X and Y by adding back this last character, which gives us that Z must also be an interleaving. So, we see that the algorithm reports that Z is an interleaving if and only if it actually is.
Question 5. What is the running time of your algorithm?

Answer: The complexity of the table-filling algorithm is clearly $O(mn)$, since it only does a constant number of comparisons to fill each of the $(m + 1)(n + 1)$ cells, and filling the final cell is trivial. As long as we are careful with how we handle the memoization, the recursive algorithm also only does $O(1)$ work per call and is only invoked $O(mn)$ times, giving an $O(mn)$ algorithm as well.