Algorithms. Lecture Notes 5

Dynamic Programming for Sequence Comparison

The “linear” structure of the Sequence Comparison problem immediately suggests a dynamic programming approach. Naturally, our sub-instances are the pairs of prefixes of A and B, and we try to align them with a minimum number of mismatches. Accordingly, we define $OPT(i, j)$ to be the minimum number of edit steps that transform $a_1 \dots a_i$ into $b_1 \dots b_j$. What we want is $OPT(n, m)$. But here, already the construction of a recursive formula for $OPT(i, j)$ requires more cereful thinling and problem analysis. If we are smart, we eventually observe that one of these three cases must appear in any alignment:

(1) a_n and b_m are aligned. It follows that $a_1 \dots a_{n-1}$ and $b_1 \dots b_{m-1}$ are aligned.

(2) a_n is not aligned to any symbol of B but comes later. It follows that $a_1 \dots a_{n-1}$ and $b_1 \dots b_m$ are aligned.

(3) Symmetrically, b_m is not aligned to any symbol of A but comes later. It follows that $a_1 \dots a_n$ and $b_1 \dots b_{m-1}$ are aligned.

Here it is “easy to be too complicated”: In case (2), some more symbols of A may come later than b_m in the alignment, say a_j is aligned to b_m, and $a_{1+1}, \dots, a_n$ have gap symbols as alignment partners, after B. We may want to specify this position j as well. But this would lead to multi-way choices (one subcase for each j) and to an unncessarily complicated dynamic programming formula.

The above distinction of only three cases will make the formula rather simple. But first we need an auxiliary function, because the edit distance depends on whether the aligned symbols are equal or different. We define $\delta_{ij} = 1$ if $a_i \neq b_j$, and $\delta_{ij} = 0$ if $a_i = b_j$. Now our case distinction, applied to the prefixes until positions i, j (rather than to n, m), immediately yields:

$$OPT(i,j) = \min\{OPT(i-1,j-1)+\delta_{ij}, OPT(i-1,j)+1, OPT(i,j-1)+1\}.$$

In case (1) we have mapped a_i onto b_j, hence we need an edit step (replacement) if and only if this character changes. The $+1$ term in cases (2) and (3) comes from deletion of a_i and insertion of b_j, respectively. Initialization is done by $OPT(i,0) = i$ and $OPT(0,j) = j$. As usual, the time complexity is the array size, here $O(nm)$, and an optimal edit sequence can be recoverded from the stored edit distances $OPT(i,j)$ by backtracing: Starting from $OPT(n,m)$ we review which case gave the minimum, and we construct the alignment of A and B from behind. (It is recommended to do some calculation examples, to see how simple it is.)

Problem: Searching

Given: a set S of n elements, and another element x.

Goal: Find x in S, or report that x is not in S.

Motivations:
Searching is, of course, a fundamental problem, appearing in database operations or inside other algorithms. Often, S is a set of numbers sorted in increasing order, or a set of strings sorted lexicographically, or any set of elements with an order relation defined on it.

Searching, Sorting, and Divide-and Conquer – An Introduction

Both the greedy approach and dynamic programming extend solutions from smaller sub-instances *incrementally* to larger sub-instances. The third main design principle still follows the pattern of reducing a given problem to smaller instances of itself, but it makes jumps rather than incremental steps.

Divide: A problem instance is split into a few significantly smaller instances. These are solved independently.

Conquer: These partial solutions are combined appropriately to obtain a solution of the entire instance.

Sub-instances are solved in the same way, thus we end up in a **recursive** algorithm.

A certain difficulty is the time analysis. While we can determine the time complexity of greedy and dynamic programming algorithms essentially by counting the operations in loops and adding these numbers, the matter is more tricky for divide-and-conquer algorithms, due to their recursive structure. We will need a special technique for the time analysis: solving **recurrences**. Luckily, a special type of recurrences covers most of thel usual divide-and-conquer algorithms. We will solve these recurrences once and for all, and then we can just apply the results. This way, no difficult mathematical analysis will be needed for every new algorithm.

Among the elementary algorithm design techniques, dynamic programming is perhaps the most useful and versatile one. Divide-and-conquer has, in general, much fewer applications, but it is of central importance for searching and sorting problems.

Divide-and-Conquer. First Example: Binary Search

As an introductory example for divide-and-conquer we discuss the simplest algorithm of this type. Consider the Searching problem. Finding a desired element x in a set S of n elements requires $O(n)$ time if S is unstructured. This is optimal, because in general nothing hints to the location of x, thus we have to read the whole of S. But order helps searching. Assume the following: (i) An order relation is defined in the "universe" the elements of S are taken from, (ii) for any two elements we can decide by a comparison, in $O(1)$ time, which element is smaller, and (iii) S is already sorted in increasing order. In this case we can solve the Searching problem quickly. (How do you look up a word in an old-fashioned dictionary, that is, in a book where words are sorted lexicographically?)

A fast strategy is: Compare x to the central element c of S. (If $|S|$ is even, take one of the two central elements.) If $x < c$ then x must be in the left half of S. If $x > c$ then x must be in the right half of S. Then continue recursively until a few elements remain, where we can search for x directly. We skip some tedious implementation details, but one point must be mentioned: We assume that the elements of S are stored in an array. This is crucial. In an array we can compute the index of the central element of the current su-barray: If its left and right end is position i and j, respectively, then the central element is at position $(i+j)/2$, rounded to the next integer. This would not be possible in a linked list.

Every comparison reduces our "search space" by a factor 2, hence we are done after $O(\log n)$ time. Remarkably, the time complexity is far below

$O(n)$. We do not have to read the whole input to solve this problem, however, this works only if we can trust the promise that S is sorted. The above algorithm is called the **halving strategy** or **binary search** or **bisection search**. It can be shown that it is the fastest algorithm for searching an ordered set.

Binary search is a particularly simple example of a divide-and-conquer algorithm. We have to solve only one of the two sub-instances, and the conquer step just returns the solution from this half, i.e., the position of x or the information that x is not present.

Although it was very easy to see the time bound $O(\log n)$ directly, we also show how this algorithm would be analyzed in the general context of divide-and-conquer algorithms. (Recall that binary search serves here only as an introductory example.) Let us pretend that, in the beginning, we have no clue what the time complexity could be. Then we may define a function $T(n)$ as the time complexity of our algorithm, and try to figure out this function. What do we know about T from the algorithm? We started from an instance of size n. Then we identified one instance of half size, after $O(1)$ operations (computing the index of the central element, and one comparison). Hence our T fulfills this recurrence: $T(n) = T(n/2) + O(1)$. Verify that $T(n) = O(\log n)$ is in fact a solution. We will show later in more generality how to solve such recurrences.

Problem: Skyline

Given: n rectangles, having their bottom lines on a fixed horizontal line.

Goal: Output the area covered by all these rectangles (in other words: their union), or just its upper contour.

Motivations:

This is a very basic example of problems appearing in computer graphics. The rectangles are front views of skyscrapers, seen from a distance. They may partially hide each other, because they stand in different streets. We want to describe the skyline.

Such basic graphics computations should be made as fast as possible, as they may be called many times as part of a larger graphics programme, of an animation, etc.

Solving The Skyline Problem

A more substantial example is the Skyline Problem. Since this problem is formulated in a geometric language, we first have to think about the representation of geometric data in the computer, before we can discuss any algorithmic issues. In which form should the input data be given, and how shall we describe the output?

Our rectangles can be specified by three real numbers: coordinate of left and right end, and height. It is natural to represent the skyline as a list of heights, ordered from left to right, also mentioning the coordinates where the heights change.

A straightforward algorithm would start with a single rectangle, insert the other rectangles one by one into the picture and update the skyline. Since the jth rectangle may obscure up to $j-1$ lines in the skyline formed by the first $j-1$ rectangles, updating the list needs $O(j)$ time in the worst case. This results in $O(n^2)$ time in total.

The weakness of this obvious algorithm is that is uses linearly many update operations to insert only one new rectangle. This is quite wasteful. The key observation towards a faster algorithm is that merging two arbitrary skylines is not more expensive than inserting a single new rectangle (in the worst case). This suggests a divide-and-conquer approach: Divide the instance arbitrarily in two sets of roughly $n/2$ rectangles. Compute the skylines for both subsets independently. Finally combine the two skylines, by scanning them from left to right and always keeping the higher horizontal line. The details of this conquer phase are not difficult, we skip them here. The conquer phase runs in $O(n)$ time. Hence the time complexity of the entire algorithm satisfies the recurrence $T(n) = 2T(n/2) + O(n)$. For the moment believe that this recurrence has the solution $T(n) = O(n \log n)$. (We may prove this by an ad-hoc argument, but soon we will do it more systematically.) This is significantly faster than $O(n^2)$.

Again, the intuitive reason for the much better time complexity is that we made a better use of the same number $O(n)$ of update operations. We can also see this from the recurrence for the bad algorithm, which is $T(n) = T(n-1) + O(n)$, with solution $T(n) = O(n^2)$.