Algorithms. Lecture Notes 6

Problem: Sequence Comparison (String Editing)

Given: two strings $A = a_1 \dots a_n$ and $B = b_1 \dots b_m$, where the a_i, b_j are characters from a fixed, finite alphabet.

Goal: Transform A into B by a minimum number of edit steps. An edit step is to insert or delete a character, or to replace a character with another one.

The *edit distance* of A and B is the minimum number of necessary edit steps. The problem can be reformulated as follows. We define a *gap* symbol that does not already appear in the alphabet. An *alignment* of A and B is a pair of strings A' and B' of equal length, obtained from A and B by inserting gaps before, after or between the symbols. A *mismatch* in an alignment is a pair of different symbols (real symbols or gaps) at the same position in A' and B'. Then, our problem is equivalent to computing an alignment of A and B with a minimum number of mismatches.

Generalized versions of the problem assign costs to the different edit steps. The costs may even depend on the characters.

Motivations:

Searching and information retrieval: Finding approximate occurrences of keywords in texts. Keywords are aligned to substrings of the text. Mismatches can stem from misspellings or from grammatical forms of words.

Archiving: If several, slightly different versions of the same document exist, and all of them shall be stored, it would be a waste of space to store the complete documents as they are. It suffices to store one master copy, and the differences of all versions compared to this master copy. The deviations of any document from the master copy are described in a compact way by a minimum sequence of edit steps.

Molecular biology: Comparison of DNA or protein sequences, searching for variants, computing evolutionary distances, etc.

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 recursion formula for $OPT(i, j)$ requires some effort and problem analysis.

An idea for a complete case distinction is to consider the "fate" of the last character of A. While transforming A into B by edit steps, what can happen to a_n? We consider two cases: a_n is deleted or not. (1) We may delete a_n, and transform $a_1 \dots a_{n-1}$ into B. (2) We may keep a_n, and convert it into another character or not. In this case, a_n or its "conversion product" is some character b_j of B. We consider two subcases: $j = m$ or $j < m$. (2.1) If $j = m$, we have to transform a_n into b_m, and $a_1 \dots a_{n-1}$ into $b_1 \dots b_{m-1}$. (2.2) If $j < m$, then the edit sequence must have created b_m. (Since $j < m$, and the order of characters is preserved, no character of A can be turned into b_m.) Since b_m is a new character, we have to transform A into $b_1 \dots b_{m-1}$. Now we have covered all cases.

We define an auxiliary function by $\delta_{ij} = 1$ if $a_i \neq b_j$, and $\delta_{ij} = 0$ if $a_i = b_j$. Now the above case distinction can be readily expressed a recursive formula, if applied to the currently last positions i, j rather than to n, m:

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

Note that, in the middle case, a_i is mapped onto b_j, so we need an edit step if and only if this character changes. 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 construct the alignment of A and B from behind. (It is recommended to do some calculation examples.)

Searching, Sorting, and Divide-and Conquer

The greedy approach and dynamic programming are two main algorithm design principles. They have in common that they extend solutions from smaller sub-instances *incrementally* to larger sub-instances, up to the full instance. 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 in-

stances. These are solved independently. **Conquer:** These partial solutions are combined appropriately, to get a solution of the given 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 basically by counting of operations in loops and summations, this is not so simple for divide-and-conquer algorithms, due to their recursive structure. We will need a special techique for time analysis: solving **recurrences**. Luckily, a special type of recurrence covers almost all 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.

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.

Divide-and-Conquer. Binary Search

As an introductory example for divide-and-conquer we discuss the perhaps 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. Suppose 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?)

A fast strategy is: Compare x to the central element c of S. (If $|S|$ is even, take one of the two central elements.) Assume that x belongs to S at all.

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 suppose that the elements of S are stored in an array. Hence we can always compute the index of the central element of a subarray. If the subarray is bounded by positions i and j, the central element is at position $(i + j)/2$ rounded to the next integer.

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 accurately 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: coordinates 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 the obvious algorithm is that is uses linearly many update operations to insert only one new rectangle. This is quite wasteful. The key observation for a faster algorithm is that merging two arbitrary skylines is not much 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 keeping the higher horizontal line at each position. 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 our previous algorithm, which is $T(n) = T(n-1) + O(n)$, with solution $T(n) = O(n^2)$.

We mention an alternative algorithm. Sort all $2n$ left and right ends according to their order on the horizontal line, then scan this sorted sequence of these points, and for every such point determine the largest height immediately to the right of the point. (Again, details are simple.) This construction needs $O(n)$ time, when implemented with some care. However, note that the rectangles were given as an (unordered) set, hence the preceding sorting phase is necessary. Fast sorting needs $O(n \log n)$ time (as we will see later, or as you already know from a data structure course) and works by divide-and-conquer as well.

Solving a Special Type of Recurrences

It is time to provide some general tool for time complexity analysis of divide-and-conquer algorithms. Most of these algorithms divide the given instance of size n in some number a of instances of roughly equal size, say n/b, where a, b are constant integers. (The case $a \neq b$ is quite possible. For example, in binary search we had $b = 2$ but only $a = 1$.) These smaller instances are solved independently and recursively. The conquer phase needs some time, too. It should be bounded by a polynomial, otherwise the whole algorithm cannot be polynomial. Accordingly we assume that the conquer phase needs at most cn^k steps, where c, k are other constant integers. We obtain the following type of recurrence:

$$T(n) = aT(n/b) + cn^k.$$

We remark that $a \geq 1$, $b \geq 2$, $c \geq 1$, and $k \geq 0$. Also assume that $T(1) = c$. This is probably not true for the particular algorithm to be analyzed, but we can raise either $T(1)$ or c to make them equal, which will not affect the O-result but will simplify the calculations.

> *"Explain things as simply as possible. But not simpler."*
> *(A. Einstein)*

To solve our recurrence we can start expanding the terms. Since T satisfies the recurrence for every argument, in particular for n/b, we have:

$$T(n/b) = aT(n/b^2) + c(n/b)^k.$$

We plug in this term in the recurrence:

$$T(n) = a^2T(n/b^2) + ca(n/b)^k + cn^k.$$

Next we do the same with $T(n/b^2)$ and obtain:

$$T(n) = a^3T(n/b^3) + ca^2(n/b^2)^k + ca(n/b)^k + cn^k.$$

And so on. For simplicity we first restrict our attention to arguments n which are powers of b, that is: $n = b^m$ for some m. Now it is not hard to derive the final result of our repeated substitution:

$$T(n) = ca^m(b^0)^k + ca^{m-1}(b^1)^k + ca^{m-2}(b^2)^k + \ldots + ca^2(b^{m-2})^k + ca^1(b^{m-1})^k + ca^0(b^m)^k.$$

Or shorter:

$$T(n) = ca^m \sum_{i=0}^{m} (b^k/a)^i$$

. In this form $T(n)$ becomes a geometric sum which is easy to evaluate. The ratio b^k/a is decisive for the result. Three different cases can appear. Remember that $n = b^m$, hence $m = \log_b n$, and recall some laws of logarithms.

If $a > b^k$ then the sum is bounded by a constant, and we simply get

$$T(n) = O(a^m) = O(a^{\log_b n}) = O(n^{\log_b a}).$$

If $a = b^k$ then the sum is $m + 1$, and a few steps of calculation yield

$$T(n) = O(a^m m) = O(n^k \log n).$$

If $a < b^k$ then only the mth term in the sum determines the result in O-notation:

$$T(n) = O(a^m (b^k/a)^m) = O((b^m)^k) = O(n^k).$$

These three formulae are often called the **master theorem** for recurrences.

So far we have only considered very special arguments $n = b^m$. However, the O-results remain valid for general n, for the following reason: The results are polynomially bounded functions. If we multiply the argument by a constant, the function value is changed by at most a constant factor as well. Every argument n is at most a factor b away from the next larger power b^m. Hence, if we generously bound $T(n)$ by $T(b^m)$, we incur only another constant factor.

Most divide-and-conquer algorithms lead to a recurrence settled by the master theorem. In other cases we have to solve other types of recurrences. Approaches are similar, but sometimes the calculations and bounding arguments may be more tricky.

> *"Mathematicians are machines turning coffee into theorems."*
> (Pál Erdős)