Algorithms. Lecture Notes 5

Dynamic Programming for Segmentation Problems

Segmentation problems deal with sequences, too. We consider the penalty minimization version first.

Let e_{ij} denote the penalty for segment $(x_i \ldots, x_j)$ in the given sequence. Being already trained in dynamic programming, we define $OPT(j)$ to be the smallest possible sum of penalties in a segmentation of $(x_1, \ldots, x_j)$. The last segment may start at any position $i \leq j$, therefore we have:

$$OPT(j) = \min_i OPT(i-1) + e_{ij},$$

where the minimum is taken over all i with $1 \leq i \leq j$. A new phenomenon is that it takes $O(j)$ time to evaluate every $OPT(j)$. Thus, the time complexity is $O(n^2)$, plus the time for computing all e_{ij}. It depends on the penalty function f how difficult these computations are. In the quality maximization version we proceed similarly, replacing min with max.

There is yet another way to solve a segmentation problem with quality maximization: Instead of developing a new dynamic programming algorithm, we may *reduce* the problem to Weighted Interval Scheduling, realizing the similarity of these problems. Namely, we consider every segment $(x_i, \ldots, x_j)$ as an interval from i to j, and its quality as the weight. Then, the best segmentation is an optimal solution to this instance of Weighted Interval Scheduling.

For correctness of this reduction it is important to notice the following: Since *all* segments became intervals, an optimal solution cannot contain gaps betweeen the selected intervals.

As for the time analysis, we have to be careful: The time complexity of Weighted Interval Scheduling was $O(n)$ (when intervals are already sorted), however, this n denoted the number of intervals. Since we have $O(n^2)$ intervals here, we are back to the time bound $O(n^2)$. Therefore we have not gained anything in terms of time complexity. Perhaps an advantage of this approach is that we reused an already existing algorithm.

In enhanced versions of segmentation problems, only some maximum number of segments may be allowed in a segmentation. Then, our dynamic programming formula needs a second parameter counting the segments we have already used up.

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 instances. 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.

Divide-and-Conquer. First Example: 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.

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)

"Mathematicians are machines turning coffee into theorems."
(P. Erdős)

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:

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

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

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

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.

Problem: Sorting

Given: a set of n elements, where an order relation is defined, e.g., a set of numbers, equipped with the natural $\leq$ relation. Comparison is assumed to be an elementary operation, that is, any two elements can be compared in $O(1)$ time.

Goal: Output the n given elements in ascending order.

Motivations: obvious

Problem: Center of a Point Set on the Line

Given: n points $x_1, \ldots, x_n$ on the real line.

Goal: Compute a point x so that the sum of distances to all given points $\sum_{i=1}^{n} |x - x_i|$ is minimized.

Motivations:

Imagine a village consisting of only one long street with n houses, with irregular spaces in between. A building for a new shop shall be erected at a "central" position in this street, that is, the average distance to the houses shall be minimized.

The scenario becomes more interesting in a usual "2-dimensional" village. However, for simplicity let us assume that streets go only in north-south and west-east direction, this road network is complete, and houses stand somewhere in these streets. What would now be the ideal position for the shop, if minimizing the average distance to all houses is the only criterion? (Here, distances are understood as walking or driving distances along the streets, not as Euclidean distances.)

More complicated (and more realistic) facility location problems with various objectives appear in infrastructure planning.

Problem: Selection and Median Finding

Given: a set of n elements, where an order relation is defined, and an integer k.

Goal: Output the element of rank k, that is, the kth smallest element.

If $k = n/2$ (rounded), we call the kth smallest element the *median*. The term “Selection problem” is a bit unspecific but established. The problem is also called Order Statistics.

Motivations:
In statistical investigations, the median is often better suited as a “typical” value than the average, because it is robust against outliers. For example, the average wealth in a population can raise when only a few people become extremely rich. Then the average gives a biased picture of the wealth of the majority. This does not happen if we look at the median. Changing the median requires substantial changes of the wealth of many people.

More generally speaking, median values, or values with another fixed rank, are often used as thresholds for the discretization of numerical data, because the sets of values above/below these thresholds have known sizes.

We remark that, once we know the *kth smallest element*, we can also find the *k smallest elements* in another $O(n)$ steps, just by $n - 1$ comparisons to the rank-k element.

Problem: Counting Inversions

Given: a sequence $(a_1, \ldots, a_n)$ of elements where an order relation $<$ is defined.

Goal: Count the inversions in this sequence. An inversion is a pair of elements where $i < j$ but $a_i > a_j$.

Motivations:
This problem is obviously related to sorting, but here the goal is only to measure either the “degree of unsortedness” of a given sequence, or the dissimilarity of two sequences containing the same elements but in different order. One of them can be assumed to be $(1, 2, 3, \ldots, n)$, and the other sequence is a permutation of it. One natural measure of unsortedness or dissimilarity, among several others, is the number of inversions.

The problem appears, e.g., in the comparison of rankings (e.g., of web pages returned by search engines), and in bioinformatics (measuring the dissimilarity of two rearranged genome sequences).