Algorithms. Lecture Notes 6

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 non-negative constants. We obtain the following type of recurrence:

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

We remark that $a \geq 1$, $b \geq 2$, $c > 0$, 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.

Since $T(n)$ recurs to $T(n/b)$, it is simpler to restrict attention first to arguments n which are powers of b, say $n = b^m$ for integer m. Starting with $T(b^0) = T(1) = c$ we can explicitly write down the $T(b^m)$ step by step, simply by applying the recurrence $T(b^m) = aT(b^{m-1}) + cb^{mk}$:

$$T(b^0) = c$$

$$T(b^1) = c(a + b^k)$$

$$T(b^2) = c(a^2 + ab^k + b^{2k})$$

$$T(b^3) = c(a^3 + a^2b^k + ab^{2k} + b^{3k})$$

Maybe these steps are enough to see the general pattern. The expression for general m is:

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

Formally this may be proved by induction on m. In this explicit form, $T(n)$ is merely a geometric sum. We will simplify it, using the O-notation. The ratio b^k/a is decisive for the final 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, then 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 sophisticated.

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: 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.

Mergesort

One obvious idea to solve the Sorting problem is called Bubblesort: The elements are stored in an array, and whenever two neighbored elements are in the wrong order, we swap them. It can be easily shown that Bubblesort needs $O(n^2)$ time. Bubblesort is worst if many elements are far from their proper places, because the algorithm moves them only step by step. The Insertion Sort algorithm overcomes this shortage: It uses k rounds to sort the first k elements ($k = 1, \ldots, n$). To insert the $(k+1)$st element we search for the correct position, using binary search. Hence we need $O(n \log n)$ comparisons in all n rounds. This seems to be fine. Unfortunately, this result does not imply $O(n \log n)$ time: If we use an array, we may be forced to move $O(k)$ elements in the kth round, giving again an overall time complexity of $O(n^2)$. Using a doubly linked list instead, we can insert an element in $O(1)$ time at a desired position, but now we cannot apply binary search for finding the correct position, because no indices are available. Without further tricks we have to apply linear search, resulting once more in $O(n^2)$ time for all n rounds. One could implement Insertion Sort with a dictionary data structure. This achieves $O(n \log n)$ time, however with an unnecessarily large hidden constant. In contrast, the fastest sorting algorithms are genuine divide-and-conquer algorithms.

Mergesort divides the given set arbitrarily in two halves, sorts them separately, and merges the two ordered sequences while preserving the order. This conquer phase runs in $O(n)$ time: We scan both ordered sequences simultaneously and always move the currently smallest element to the next position in the result sequence. The time complexity satisfies the recurrence $T(n) = 2T(n/2) + O(n)$, with solution $T(n) = O(n \log n)$.

Mergesort has a particularly simple structure, but it is not the fastest sorting algorithm in practice. It executes too many copy operations besides the comparisons. Moreover, in every merging phase on every recursion level it moves all elements of the merged subsets into a new array. Thus, we also need extra memory, which can be another practical obstacle in the case of large n.

There are several alternative algorithms for sorting which also need $O(n \log n)$ time, but with different hidden constant factors. These factors are hard to analyze theoretically, but some informal reasoning as above gives at least some hints, and runtime experiments can figure out what is really faster.

Quicksort and Random Splitters

One of the favorable sorting algorithms is **Quicksort**. It needs only one array of size n for everything, apart from a few auxiliary memory cells for rearrangements. An algorithm with these properties is said to work *in place*, or (Latin) *in situ*. Bubblesort is an *in place* algorithm as well, however a bad one.

Quicksort divides the given set according to the following idea. Let p be some fixed element from the set, called a **splitter** or **pivot**. Once we have put all elements $< p$ and $> p$, respectively, in two different subsets, it suffices to sort these two subsets independently. Then, concatenating the sorted sequences, with the splitter in between, gives the final result. Thus the conquer phase is trivial here.

What makes Qicksort quick is the implementation details of the divide phase: Two pointers starting at the first and last element scan the array inwards. As long as the left pointer meets only elements $< p$, it keeps on moving to the right. Similarly, as long as the right pointer meets only elements $> p$, it keeps on moving to the left. When both pointers have stopped, we swap the two current elements. This requires one additional memory cell where one element is temporarily stored. As soon as both pointers meet, the set is divided as desired.

Clearly, the divide phase needs $O(n)$ time. The hidden constant is really small due to the simple procedure. Moreovoer, we only have to move elements being on the wrong side, and these can be much less than n. What about the overall time complexity? If the splitters would exactly halve the sets on every recursion level, we had our standard recurrence $T(n) = 2T(n/2) + O(n)$, with solution $T(n) = O(n \log n)$. Unfortunately, this is not the case if we choose our splitters without care. In the worst case, the splitter is always the minimum or maximum element of the set, and then $O(n^2)$ time is needed. What can we do about it?

The ideal splitter for Quicksort would be the median. But how difficult is it to compute the median? We could sort the set and then read off the element with rank $n/2$. But this would be silly, because sorting was the problem we came from.

Actually, Quicksort goes a different way. A splitter is selected at random! (Indeed, an algorithm can make random decisions. This is not against the general demand that an algorithm must be unambiguous and must not allow for intervention. Random decisions are made by a random number generator rather than by the user.) Now the worst case (rank nearly 1 or n) is very

unlikely. The splitters will mostly have ranks in the middle, and then we get reasonably balanced partitionings in two sets. In fact, a strict analysis confirms that $O(n \log n)$ time is needed on expectation. We do not give this analysis here, as randomized algorithms are beyond the scope of this course.

The speed in practice is further improved by chosing three random elements and taking their median as the splitter. This needs a few extra operations, but much more operations are saved due to the better partitionings.

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.

Algorithms for Selection and Median Finding

First we remark that the solution to the "Center of a Point Set on the Line" problem is the median of the given coordinates, and not the average. (Why? Think about it.)

Surprisingly, the Selection problem can be solved without sorting, in $O(n)$ time. The intuitive reason is that Selection needs much less information than Sorting. Here is a fast algorithm. Again we choose a random

splitter p and compare all elements to p in $O(n)$ time. Now we know the rank r of p. If $r > k$ then throw out p and all elements larger than p. If $r < k$ then throw out p and all elements smaller than p, and set $k := k - r$. If $r = k$ then return p. Repeat this procedure recursively.

If the splitters were always in the middle, the time would follow the recursion $T(n) = T(n/2) + O(n)$, with solution $T(n) = O(n)$. Again, a probabilistic analysis confirms an expected time $O(n)$, whereas the worst case is $O(n^2)$. There also exists a deterministic divide-and-conquer algorithm for Selection, but it is non-standard and a bit complicated. More importantly, its hidden constant in $O(n)$ is rather large, such that the algorithm is only of academic interest, while the random-splitter algorithm is practical.

Information Flow and Optimal Time Bounds

We conclude the discussion of Sorting and Selection with some remarks on optimal time bounds. One of our general goals is to make algorithms as fast as possible. How good are our time bounds?

In order to find a specific element in an ordered set we needed $\log_2 n$ comparisons of elements, by doing binary search. No other algorithm with comparisons as elementary operations can have a better worst-case bound. This holds due to an **information-theoretic** argument explained as follows. How much information do we gain from our elementary operations? Every comparison gives a binary answer ("smaller" or "larger"), thus it splits the set of possible results in two subsets for which either of the answers is true. In the worst case we always get the answer which is true for the larger subset, and this reduces the number of candidate solutions by a factor at most 2. Since there were n possible solutions in the beginning, *any* algorithm needs at least $\log_2 n$ comparisons in the worst case.

The same type of argument shows that no sorting algorithm can succeed with less than $O(n \log n)$ comparisons in the worst case: Since a set with n elements can be ordered in $n!$ possible ways, but only one of them is the correct order, any sorting algorithm can be forced to use $\log_2 n!$ comparisons, and some calculation shows that this is $n \log n$, subject to a constant factor. As we ignore such constants anyway, we can make the calculation very simple:

$$\log_2 n! = \sum_{k=1}^{n} \log_2 k \geq (n/2) \log_2 (n/2).$$

This argument does not apply to the Selection problem: There we have only n possible results, and $O \log n$ is a very poor lower bound. In fact, $O(n)$ is the optimal bound, but for a totally different reason: We have to read all elements, since every change in the instance can also change the result.

It should also be noticed that the information-theoretic lower bounds for Sorting and Searching hold only under the assumptions that (1) nothing is known in advance about the elements, and (2) doing pairwise comparisons is the only way to gather information. Faster algorithms can exist for special cases where we know more about the instance. For example, Bucketsort works in $O(m + n)$ time, if the n elements come from a fixed range of m different numbers. Similarly, a set of words over a fixed alphabet can be sorted in lexicographic order in $O(n)$ time, where n is the total length of the given words. These results do not contradict each other.

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).