Algorithms. Lecture Notes 7

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 some integer $k \leq n$.

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 for 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: After k rounds of Insertion Sort, the first k elements ($k = 1, \ldots, n$) are sorted. 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.

However, 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 $T(n) = O(n \log n)$. (We remark that our Skyline algorithm implicitly uses Mergesort to sort all endpoints of the rectangles.)

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. In every merging phase on every recursion level we have to move *all* elements of the merged subsets into a new array. Thus, we also need additional 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 *in situ* (for Latin lovers). Bubblesort is an *in place* algorithm as well, but too slow.

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, where 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 partitions in two sets. In fact, a strict analysis confirms that $O(n \log n)$ time is needed on expectation. We do not give the 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 of them as the splitter. This needs a few extra operations, but much more operations are saved due to the better partitions.

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 s and compare all elements to s in $O(n)$ time. Now we know the rank r of s. If $r > k$ then throw out s and all elements larger than s. If $r < k$ then throw out s and all elements smaller than s, and set $k := k - r$. If $r = k$ then return s. 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 primary 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 operation? 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).

An Algorithm for Counting Inversions

Next we want to count the number of inversions in a sequence, faster than by the obvious $O(n^2)$ time algorithm. This problem example is instructive as it combines divide-and-conquer with some general issue that sometimes plays a role in algorithm design.

Due to a vague similarity to Sorting, it should be possible to apply divide-and-conquer. We could split the sequence in two halves, $A = (a_1, \ldots, a_m)$ and $B = (a_{m+1}, \ldots, a_n)$, where $m \approx n/2$, and count the inversions in A and B separately and recursively. In the conquer phase we would count the inversions between A and B, that means, those involving one element in each of A and B, and sum up. But it is not easy to see how to execute this conquer phase better than in $O(n^2)$ time. At this point we need a creative idea.

Intuitively, it would be much easier to do the conquer phase when the the two halves were sorted. (We will look at this in detail.) What if we also *sort* the sequence while counting the inversions? This idea may appear counterintuitive: Sorting is not what we originally wanted, and one might think that a problem becomes only harder by extra demands. But in fact, sorting serves here as a tool to make the conquer phase of another algorithm efficient! Figuratively speaking, our inversion counting algorithm will be piggybacked by a recursive sorting algorithm. That is, we extend our problem to Sorting AND Counting Inversions, and solve it recursively.

As the underlying sorting algorithm we take the conceptually simple Mergesort. If we manage to merge two sorted sequences A and B, and simultaneously count the inversions between A and B, still everything in $O(n)$ time, then the recurrence $T(n) = 2T(n/2) + O(n)$ applies; remember that its solution is $T(n) = O(n \log n)$. In fact, this $O(n)$ time merging-and-counting is easily done, using some pointers and counters. We proceed as in Mergesort, and whenever the next element copied into the merged sequence is from B, this element has inversions with exactly those elements of A not visited yet. Hence we only need $O(n)$ additions of integers, on top of the copy operations.

Faster Multiplication

This is one of the most amazing classic results in the field of efficient algorithms. Recall that the "school algorithm" for multiplication of two integers, each with n digits, needs $O(n^2)$ time. For simplicity let n be a power of 2, otherwise we may fill up the decimal representations of the factors with dummy 0s. This "padding" can at most double the input size, hence the (polynomial) time complexity is increased by some constant factor only.

An attempt to multiply through divide-and-conquer is to split the decimal representations of both factors into two halves, and then to multiply with help of the distributive law:

$$(10^{n/2}w + x)(10^{n/2}y + z) = 10^n wy + 10^{n/2}(wz + xy) + xz$$

.

That is, we reduce the multiplication of n-digit numbers to several multiplications of $n/2$-digit numbers and some additions. Then we apply the same equation recursively to all the $n/2$-digit numbers. This algorithm satisfies the recurrence $T(n) = 4T(n/2) + O(n)$, since additions and other auxiliary operations cost only $O(n)$ time. Factor 4 comes from the four recursive calls. Note that only w, x, y, z are multiplied recursively, whereas multiplications with powers of 10 are trivial: Append the required number of 0s. Since $2^1 < 4$, the master theorem yields $T(n) = O(n^{\log_2 4}) = O(n^2)$. Unfortunately, this is not an improvement.

However, we have not fully exploited the power of the idea. The key observation suggesting that the usual algorithm might be unnecessary slow was that it does the same multiplications many times. Simple geometry gives an idea how to save one multiplication: Consider a rectangle with side lengths $w + x$ and $y + z$. We need the area sizes of three parts of this rectangle: $xz, wy, wz + xy$. The last term is not a rectangle area, but looking at the the whole rectangle we see that

$$(w + x)(y + z) = wy + (wz + xy) + xz$$

.

Hence we obtain the desired numbers by only three multiplications:
$(w + x)(y + z)$, wy, and xz. The term $wz + xy$ is obtained by subtractions,

which are cheaper than another multiplication. Altogether we need $T(n) = 3T(n/2) + O(n)$ time, which yields $T(n) = O(n^{\log_2 3}) = O(n^{1.59})$. This is impressively better than $O(n^2)$.

This analysis was not completely accurate: Factors $w+x$ and $y+z$ can have $n/2+1$ digits. But then we can split off the first digit, which gives us recursive calls to instances with (now accurately) $n/2$ digits, plus some more $O(n)$ terms in the recurrence which do not affect the time bound in O-notation.

Why don't we use this algorithm in everyday applications? It must be confessed that the acceleration takes effect only for rather large n (moer than some 100 digits). The main reason is the adminisrative overhead for the recursive calls. The simple traditional algorithm does not suffer from such overhead. Multiplication by divide-and-conquer is not suitable for numerical calculations, since the factors have barely more than a handful digits. Still the algorithm is not useless. Some cryptographic methods rely on the fact that integers are easy to multiply but hard to split into integer factors. These methods use the multiplication of large numbers which have no numerical meaning but encode messages and secret keys instead. In such applications, n is large enough to make the asymptotically fast algorithm really fast also in practice.

The above algorithm is not yet the fastest known multiplication algorithm. An $O(n \log n \log\log n)$ time algorithm is based on convolution via Fast Fourier Transformation, but this is a more advanced topic. It is not known whether one can multiply even faster.

Finally we mention that similar divide-and-conquer algorithms exist also for matrix multiplication, with similar provisoes. However, very large matrices can appear in calculations and simulations in mechanics or economy.