Algorithms. Lecture Notes 6

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 ($k = 1, \dots, n$) 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, 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, 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 **rank** of an element in a set is defined as the position of this element if the set were already sorted. That is, the kth smallest element has rank k. (Ties are broken arbitrarily if some elements are identical.) The **median** is the element of rank $n/2$ (rounded to an integer if n is odd). 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.

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: Closest Points

Given: a set of n points in the plane (given as Cartesian coordinates (x_i, y_i)).

Goal: Find a pair of points with minimum distance.

Motivations:
Some approaches to hierarchical clustering of data take the two closest data points and combine them to a cluster by replacing these two points by their midpoint, and this step is repeated until one cluster remains.