Algorithms. Lecture Notes 2

The Need for Speed

Consider a problem like Interval Scheduling. We may immediately solve it in $O(n2^n)$ time by **exhaustive search**: Generate all subsets of the given set of intervals, check whether they are free of intersections, and take the largest feasible solution.

But such obvious and trivial algorithms are hopelessly slow. Typically, naive algorithms compute the same things over and over again, thereby wasting time. The problems have structure that can be used. For example, if a set X of intervals is already discarded because some intervals intersect, it is not necessary to consider any further sets that contain X.

Myth: Fast algorithms are not needed, because the hardware is fast.

We can quickly see that this is wrong. Just compare several functions, such as $n!$, 2^n, n^2, $n \log n$, ... directly, by computing their values for $n = 1, 2, 3, \ldots$ Moreover, multiply the latter functions with some large constant factors. Even then, the former functions will get much higher values very soon. Next, suppose that we can double the speed of the processor. See how this affects the size n of instances that can be solved within a given time budget.

These considerations should convince everyone of the necessity and power of problem analysis and algorithm design. Because an algorithm must be created only once for a problem but will be applied many, many times, good algorithm design is definitely worth the effort.

Algorithm for Interval Scheduling

As a first example let us try and develop a fast algorithm for Interval Scheduling, replacing the naive one.

Here is a good starting point for algorithm development: *Suppose that we are already able to solve the problem for smaller instances, how can we use the partial solutions to solve the overall instance?* Since instances of most computational problems can be split into smaller instances of the same problem in natural ways, this is a very fruitful approach.

In the case of Interval Scheduling we may ask more specifically: Suppose we know already how to find the best solution for less than n intervals. Can we perhaps make a decision for one interval (to put it in the solution X or not), and then solve the remaining instance? However, we have many options to choose this one distinguished interval.

A tempting idea is to serve the first request, i.e., the one with smallest s_i. That is, we put this interval $[s_i, f_i]$ in X and, of course, remove all intervals that intersect $[s_i, f_i]$, and continue with the same rule. But the drawback is obvious: The first interval could be very long. It could even intersect all others. Apparently we should take the lengths $f_i - s_i$ into account. Let us make another attempt: First put the shortest interval $[s_i, f_i]$ in X. Unfortunately, also the shortest interval does not necessarily belong to an optimal solution. The smallest counterexample has only three intervals and is easy to see! Hence this rule does not work either. We should analyze the reasons for these failures and learn from them. Our selection rules were bad because the selected intervals may overlap too many other intervals. This suggests yet another idea: Put an interval in X that intersects the smallest number of other intervals. Sadly, this rule fails, too. (But this time it is a little harder to find a counterexample.)

It is doubtful whether we will find, in this way, any algorithm that works. It is time to apply creativity. What else could be a good candidate for an interval to put in X? We may try further choices until we are lucky, or we decide to give up at some point. What we can try depends on the problem structure. At least, the principle of reducing a given problem to smaller instances guides our search for an idea.

> *"Ever tried. Ever failed. No matter.*
> *Try again. Fail again. Fail better."*
> (Samuel Beckett)

For the Interval Scheduling problem, persistent search is finally crowned by success: Reviewing the counterexamples again, we may notice that in the last two attempts the selected intervals were somewhere in the middle of the schedule. What if we come back to the original idea and look at the beginning of the schedule? But taking the interval with earliest starting point was bad. Instead, let us take the interval with the earliest endpoint! The rationale is that this interval is in conflict with the smallest number of other intervals in the remaining instance to the right of its endpoint. Before we investigate whether this rule constitutes a correct algorithm, we write the proposed algorithm more explicitly:

Earliest End First: Sort the intervals according to their right endpoints. That is, re-index them so that $f_1 < f_2 < \ldots < f_n$. Put the interval with the smallest f_i in X, and discard all intervals that intersect this first interval. Repeat this step until every interval is either in X or discarded.

This time we will not detect counterexamples. But after the bad experiences where we saw plausible rules breaking down, it should be clear that we need an optimality proof. It is not enough to say that no counterexamples are known. There might exist some, but they might be relatively large and hard to see (as it happened with some of the wrong algorithms above).

For the proposed algorithm Earliest End First we claim: There exists some optimal solution Y containing the interval $x := [s_1, f_1]$.

If this claim is true, we can already conclude correctness of Earliest End First. Why? The claim says that it is safe to put x in X. After this first step, the algorithm removes all intervals intersecting x. This is also safe, because a solution with x cannot contain any of these discarded intervals. It remains to compute an optimal solution for the remaining intervals, and we can repeat the argument. You may have recognized that this reasoning is **induction** on n, where the claim provides the induction step.

Finally we prove the claim. Consider any optimal solution Y with $x \notin Y$. We will transform Y into an optimal solution that contains x. If x has no conflicts with any intervals in Y, we can add x to Y, contradicting the assumption that Y was optimal. Thus, x intersects some interval $y \in Y$. At this point we use that x has the smallest f_i. (This fact must play a role somewhere!) Note that x intersects only one $y \in Y$, since the intervals in Y are pairwise disjoint. Thus we can replace y with x and obtain another optimal solution with x, as desired. The claim is proved, and the algorithm is correct.

The next step is to think about implementation details that make the algorithm efficient. We may create two copies of each interval, sort them by ascending f_i and ascending s_i, respectively, and put them in two linked lists. The two copies of each interval are connected by pointers, too. In the f_i list we can always find the smallest f_i in $O(1)$ time: It is simply the first element. The intervals that intersect this $[s_i, f_i]$ are exactly those with $s_j < f_i$. That is, we can take the first elements from the other list, until value f_i is reached. These intervals are deleted in both lists, using the pointers. We spend only $O(1)$ time on each copy of an interval. Thus we need $O(n)$ time in total, plus the time for sorting.

To What End Do We Study Algorithm Theory?

Various computational problems are important, and as we have seen, fast algorithms are important. Does that mean that we have to learn a suite of fast algorithms for the most frequent problems? Yes, but this is not enough. Practical problems rarely arise in nice textbook form, and usually we cannot take an algorithm from the shelves. Often we must adjust or combine algorithms that are known for similar problems or for parts of a given problem. To be able to do this, we need a profound understanding of **how and why** these algorithms work. We have to understand the underlying ideas of algorithms, not only the

particular steps. Moreover, new computational problems will in general require new algorithms. There is no universal recipe for developing good algorithms, except some general guidelines and techniques. The main part of this course provides some of these general design techniques, a basic toolkit so to speak. We illustrate and practice them on various problem examples. But still the actual algorithm design for a given problem remains a trial-and-error process. (Compare it to craft: Even if one knows one's trade, every application is a bit different.) The selected problems are, hopefully, also of some relevance by their own. But the emphasis is on the design process, rather than on the ready-to-use algorithms for specifc problems. In the Interval Scheduling example we were quite detailed about solution attempts, including failing attempts, in order to stress this creative process and not only the final algorithm.

We also have to learn how to **prove correctness** of a new algorithm.

Myth: Correctness proofs are not needed, we can simply test our algorithms on some instances.

Already the Interval Scheduling example has shown that this is not true! Various algorithms appeared to be plausibe, but they failed. By not caring about correctness proofs we may happily accept erroneous algorithms. Proofs are not a luxury, just made to intellectually please a few researchers. Rather, it should be clear that testing alone cannot guarantee correctness. We may pick, by good luck, some test instances where our algorithm yields the correct result, but it may have a hidden error that shows up in other instances. This can have fatal consequences, especially in sensible technical systems controlled by algorithms, and so this matter touches even questions of ethics in engineering.

In a more general perspective, proving correctness means to *prove that there exists no counterexample*, and proving non-existence of an object can be tricky: In order to show that white rabbits exist, it suffices to present one. But how do we prove that no blue rabbits exist?! It is not enough to say "I have never seen one", but we need general arguments to prove that we will never see one in the future either. Theferfore, admittedly, proving properties such as correctness can be a difficult business, but who else should do it, if not the persons who develop the algorithms?

Now we can summarize our main goal: *Develop correct algorithms with low worst-case time bounds, which are, preferably, moderate polynomials.*

About Greedy Algorithms

Earliest End First is an example of a **greedy algorithm**. These are algorithms which, at every moment, make the currently best choice, according to some simple optimality criterion. (This is not a formal definition, just a circumscription.) In this sense, greedy algorithms are "myopic".

Myth: Take the best, ignore the rest. If we take an optimal decision in every step, then the overall result will be optimal, too.

In fact, *most greedy algorithms are wrong*, and counterexamples are often amazingly small. As we have seen, we do need correctness proofs. The scheme we used to prove optimality of Earliest End First is quite common for greedy algorithms: Consider any optimal solution. Then transform it, to get another optimal solution which is "one step closer" to the solution our algorithm would produce. By induction, we reach the algorithm's solution in this way, hence it is optimal. A transformation step typically consists of exchanging some items of a solution. Therefore we speak of an *exchange argument*.

A Greedy Algorithm for Interval Partitioning

This time we give a correct algorithm straight away, without discussing the development process. Subsets $X_1, X_2, X_3, \ldots$ are initially empty. Sort the intervals such that $s_1 < \ldots < s_n$, and consider them in this order. Always put the current interval x into X_i with the smallest index i, where x does not intersect any other interval in X_i.

Optimality may be proved again by an exchange argument, but here we illustrate another nice proof technique: We give a simple bound for the value d to be optimized, and then we show that our solution achieves this bound, hence it is optimal. Specifically, let d be the maximum number of intervals that contain the same point. Since d such intervals must be put into d distinct subsets, any solution needs at least d subsets. Our greedy algorithm uses only d subsets: Whenever a new interval x is considered, at most $d-1$ earlier intervals can intersect x, because these intervals must contain the start point of x. Hence we can always put x is some of the first d subsets.

Problem: Weighted Interval Scheduling

Given: a set of n intervals $[s_i, f_i]$, $i = 1, \ldots, n$, on the real axis. Every interval has also a positive weight v_i.

Goal: Select a subset X of these intervals which are pairwise disjoint and have maximum total weight.

Motivations:
Similar to Interval Scheduling, but here the requests have different importance. The weights might be profits, e.g., fees obtained from the customers.