Algorithms. Lecture Notes 7

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 the 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 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 which are 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 considerably 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 (more 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.

Reductions between Problems

In general, any new problem requires a new algorithm. But often we can solve a problem X using a known algorithm for a related problem Y. That is, we can **reduce** X to Y in the following informal sense: A given instance x of X is translated to a suitable instance y of Y, such that we can use the available algorithm for Y. Eventually the result of this computation on y is translated back, such that we get the desired result for x. This sounds very

abstract? Here we illustrate the idea by an example:

Suppose that we want an algorithm for multiplying two integers, and there is already an efficient algorithm available that can compute the square of an integer. It needs $S(n)$ time for an integer of n digits. Can we use it somehow to multiply arbitrary integers a, b efficiently, without developing a multiplication algorithm? (The assumptions are a bit made up, but we will see below that the example is meaningful.)

Certainly, squaring and multiplication are closely related problems. In fact, we can use the identity $ab = ((a+b)^2 - (a-b)^2)/4$. We only have to add and subtract the factors in $O(n)$ time, apply our squaring algorithm in $S(n)$ time, and divide the result by 4, which can be easily done in $O(n)$ time, since the divisor is a constant.

Thus we have reduced some problem X (multiplication) to some problem Y (squaring). Namely, we have taken an instance of X (the factors a and b), transformed it quickly into two instances of Y (with operand $a+b$ and $a-b$, respectively), solved these instances of Y by the given squaring algorithm, and finally applied another fast computation to the results (an addition, and a division by 4) to obtain the solution ab to the instance of problem X.

It is crucial that not only a fast algorithm for Y is available, but the transformations are fast as well. Note that the time for our multiplication algorithm is $O(S(n))$. The $O(n)$-time transformations are already counted in this time bound, as $S(n)$ is certainly not faster than $O(n)$.

Doing multiplication through an algorithm specialized to squaring may appear somewhat strange. But we get an interesting insight from this reduction: One might conjecture that squares can be computed faster than products of arbitrary numbers, since this problem is only a very special case of multiplication. (Moreover, in applications with a lot of squarings it would be nice to have such a faster algorithm.) But due to our reduction, these hopes come to nothing, and we can give firmly a negative answer: Any faster algorithm for squaring would immediately yield a faster algorithm for (general) multiplication, too. We conclude that squaring is not easier than multiplication!

We have identified two different purposes of reductions: (1) solving a problem X with help of an already existing algorithm for a different problem Y, and (2) showing that a problem Y is at least as difficult as another problem X.

Note that (1) is of immediate practical value, and even usual business: Ready-to-use implementations of standard algorithms exist in software packages and algorithm libraries. One can just apply them as black boxes, by

using their interfaces, and without caring about their internal details. This is nothing but a reduction! Point (2) might appear less practical, but it gives us a way to compare the difficulty of problems without determining their "absolute" time complexities. It can be useful to know these comparisons, for example: If Y is at least as difficult as X, then research on improved algorithms should first concentrate on the easier problem X. Some applications (as in cryptography) even rely on the hardness of certain computational problems, rather than efficient solvability.

Reductions – Now More Formally

After this informal introduction we approach the abstract definitions of reductions that are needed to build up a **complexity theory** of computational problems.

Let X, Y be any two problems. By $|x|$ we denote the length of an instance x of problem X. We say that X *is reducible to* Y *in* $t(n)$ *time*, if we can do the following in $t(n)$ time for any given instance x with $|x| = n$: Transform x into an instance $y = f(x)$ of problem Y, and transform the solution of y back into a solution for x.

Symbol f merely denotes the function describing how an instance is transformed. f must be computable in $t(n)$ time. Note that the time needed by the algorithm for problem Y is not counted in $t(n)$. Only the transformations of instances and solutions are charged. These are the extra costs for using the algorithm for Y, so to speak. Assuming that we have an algorithm for problem Y with time bound $u(n)$, we can solve an instance x of problem X in time $t(|x|) + u(|f(x)|)$. Since $|f(x)| \leq t(n)$ (why?), this is bounded by $t(n) + u(t(n))$.

Loosely speaking we can conclude: If Y is an easy problem and the reduction is fast, then X is an easy problem, too. Conversely, if X is a hard problem, and we have a fast reduction to problem Y, then Y is a hard problem, too. In this sense, a reduction allows a comparison of the difficulty of two problems.

These comparisons become much easier to handle formally when we restrict attention to so-called **decision problems**. A decision probem is simply a computational problem that has to output a Yes or No answer (e.g., the instance has a solution or not). This is not a severe restriction. Every optimization problem can be viewed as a series of decision problems. Instead of asking "give me a solution where the objective value is minimized" we can

ask "does there exist a solution with objective value at most t?", for various thresholds t. Informally, if the optimization problem is easy to solve, then the corresponding decision problem is also easy, for every threshold t. (We just compare an optimal solution to the threshold.) By contraposition, if already the decision problem is hard, then the corresponding optimization problem is also hard.

Finally we define reductions between decision problems X and Y: We say that X is reducible to Y in $t(n)$ time, if we can compute in $t(n)$ time, for any given x with $|x| = n$, an instance $y = f(x)$ of Y such that the answer to x is Yes *if and only if* the answer to y is Yes. (In other words, instances x, y of the decision problems X, Y are equivalent.) If the time $t(n)$ needed for the reduction is bounded by a polynomial in n, we say that X is **polynomial-time reducible** to Y.

Problem: Clique

A **clique** in a graph $G = (V, E)$ is a subset $K \subseteq V$ of nodes such that all possible edges in K exist, i.e., there is an edge between any two nodes in K.

Given: an undirected graph G.

Goal: Find a clique of maximum size in G.

Motivations:

This is a fundamental optimization problem in graphs. Many other problems can be rephrased as a Clique problem. A setting where it appears directly is the following: The graph models an interaction network (persons in a social network, proteins in a living cell, etc.), where an edge means some close relation between two "nodes". We may wish to identify big groups of pairwise interacting "nodes", because such groups may have an important role in the network.

Problem: Independent Set

An **independent set** in a graph $G = (V, E)$ is a subset $I \subseteq V$ of nodes such that no edges in I exist.

Given: an undirected graph G.

Goal: Find an independent set of maximum size in G.

Motivations:

The same general remarks as for the Clique problem apply. A setting where it appears directly is the following: The graph models conflicts between items, and we wish to select as many as possible items conflict-free. For example: Goods shall be packed in a box, but for security reasons certain goods must not be packed together. How many items can we put in the same box?

Problem: Vertex Cover

A **vertex cover** in a graph $G = (V, E)$ is a subset $C \subseteq V$ of nodes such that every edge of G has at least one of its two nodes in C.

Given: an undirected graph G.

Goal: Find a vertex cover of minimum size in G.

Motivations:

Vertex covers are of interest in "facility location" problems. A toy example is the question: How can we place a minimum number of guards in a museum building so that they can watch all corridors?

Another application field is combinatorial inference. As a bioinformatics example, consider some genetic disease that appears if some rare bad variant of a certain gene is present. Geneticists want to figure out what the bad gene variants are. Their number is expected to be small, as a result of a few unfortunate mutations. Every person carries two copies of the gene. Given the genetic data of a group of persons having the disease, we know that each person has at least one bad variant in his/her pair of genes. Now we can try and explain the data by a minimum number of different bad gene variants.