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 in algorithm design (see below).

Due to the vague similarity to Sorting, it should be possible to apply divide-and-conquer. We could split the sequence in two halves, say, $A = (a_1, \ldots, a_m)$ 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. 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 $T(n) = 2T(n/2) + O(n)$ will apply. 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 in two halves, and then to multiply them 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 $T(n) = O(n^{\log_2 4}) = O(n^2)$. ... Too bad! Unfortunately, this is not an improvement.

Was this a futile approach? No, we have just failed to fully exploit the power of the idea. The key observation suggesting that the usual algorithm might be unnecessary slow was that it executes the same multiplications of digits many times. Simple geometry gives an idea how to save one of the four recursive multiplications: 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 the area of a rectangle, 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 only $T(n) = 3T(n/2) + O(n)$ time, which yields $T(n) = O(n^{\log_2 3}) = O(n^{1.59})$. This is indeed considerably better than $O(n^2)$.

A minor remark is that this analysis was not completely accurate: The 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. Hence this minor technicality can be easily fixed.

But now, why don't we use this clever 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 administrative 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 their prime factors. These methods use multiplications of large numbers. They 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 beyound the scope of this course. 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. Very large matrices can appear in calculations and simulations in mechanics or economy.

Problem: Closest Points

Given: a set of n points in the plane, specified by their Cartesian coordinates (x_i, y_i).

Goal: Find a pair of points with minimum Euclidean distance (i.e., the usual distance in the plane, which is the length of the straight line segment connecting the points).

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.

Divide-and-Conquer in Geometry: Closest Points

Fast geometric calculations are needed in computer graphics, computer-aided design, robotics, planning (transport optimization, facility location), chemistry (modelling molecules and their dynamics), for extracting information from geographic databases, etc. The amount of data can be huge (e.g., elements of a picture), such that efficient algorithms make a difference.

Divide-and-conquer is suitable for various geometric problems, because instances can be divided in a natural way. However, the conquer phase is usually less trivial. To give at least an impression, we discuss another geometric problem example: finding a pair of closest points among n given points in the plane.

An obvious algorithm would compute all pairwise distances and determine the minimum in $O(n^2)$ time. Instead, we aim at a divide-and-conquer algorithm satisfying the recurrence $T(n) = 2T(n/2)+O(n)$, hence with time complexity $T(n) = O(n \log n)$.

It is natural to divide the set by a straight line. To make the calculation details simple, we first sort the points by their x-coordinates, and then halve the set by a vertical separator line. More formally, we take the median z of all x-values and put all points with coordinate $x < z$ and $x > z$, respectively, in the two sets. Sorting takes $O(n \log n)$ time and needs to be done only once in the beginning, which does not destroy the desired time bound.

Then, of course, we compute the closest pairs in both subsets recursively. Let d be the minimum of the two minimum distances. The more tricky part is to combine the partial solutions. The global solution could be the best of the two closest pairs from the two subsets, but there could also exist a pair of points with distance smaller than d, having one point on each side of the separator line. But now some geometry helps:

The candidates for such pairs of points are in a stripe of breadth d around the separator line. Moreover, each point has only constantly many partners (at distance smaller than d) on the other side, hence only $O(n)$ such pairs of close points must be considered. These pairs can be identified in $O(n)$ time, if all points are already sorted by their y-coordinates as well. With careful

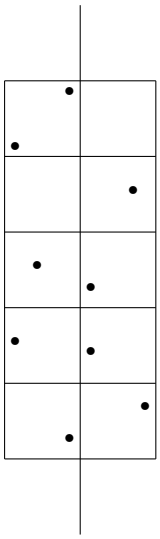

Figure 1: This is why the conquer phase needs only $O(n)$ time. Pairs with points on both sides of the separating line can only be taken from a stripe of breadth $2d$. For clarity, let us partition this stripe into squares with side length d. Since the points on each side must keep a distance at least d (yes, the points must keep some distance, too, not only people ...), there can be only one or two points in every square. Moreover, we need to measure the distances of points only in incident squares, since other points are clearly too far away. These are $O(1)$ candidate partners for each point. Once we have sorted the y-coordinates in the beginning (not during the recursion!), we only have to traverse some sorted list of points.

implementation details (that we omit here), all steps in the conquer phase run in $O(n)$ time as desired.