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 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.

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 $T(n) = 2T(n/2) + O(n)$, which would have the 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. Recall that sorting takes $O(n \log n)$ time, which does not destroy the desired time bound. Wouldn't it be enough to compute the median in $O(n)$ time, without sorting? Yes, it is enough for the first step, but we will recursively split the point set further, on the lower recursion levels. Sorting the points once in the beginning is simpler and cheaper (in terms of the hidden constant factors) than median computations on every recursion level.

Then, of course, we compute the closest pairs in both subsets recursively. Let d be the minimum of the two minimum distances. The 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 in each subset. The candidates for such pairs of points are in a stripe of breadth d on both sides of the separating line. Moreover, each point has only constantly many partners (at distance smaller than d) on the other side, hence $O(n)$ 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 implementation, all steps in the conquer phase run in $O(n)$ time as desired.

Graphs

Graphs are used to model networks or abstract binary relations of any kind. A **graph** $G = (V, E)$ consists of a set V of **nodes** and a set E of **edges** which are pairs of nodes. We call G an **undirected (directed)** graph if these pairs are unordered (ordered). "Unordered" means that (u, v) and (v, u) are identical. We can also view an undirected graph as a directed graph where for every directed edge $(u, v) \in E$, the reverse edge (v, u) belongs to E as well. Two vertices joined by an edge are called **adjacent**. We will always denote the number of nodes and edges of a graph by n and m, unless stated otherwise. A node and an edge are **incident** if the edge contains this node. Also, two edges that share a node are called incident.

There are two standard ways to represent a graph in computer memory. The **adjacency lists** data structure contains, for each node u, a list of all v with $(u, v) \in E$. An **adjacency matrix** A has a row and a column for each node, and entry $A(u, v)$ equals 1 if $(u, v) \in E$, and 0 otherwise. In most graph algorithms, adjacency lists are preferable, as they do not waste space for non-edges.

The **degree** of a node is the number of incident edges. In directed graphs we distinguish between **in-degree**, the number of incoming edges, and **out-degree**, the number of outgoing edges.

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 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.

Problem: Satisfiability (SAT)

A **Boolean variable** has two possible values: True (1) or False (0). A **literal** is either a boolean variable x or its negation $\neg x$. A **Boolean formula** is composed of literals joined by operations AND (conjunction, $\wedge$), OR (disjunction, $\vee$), and perhaps further negations. An **assignment** gives a truth value to every variable in a Boolean formula. An assignment is said to be **satisfying** if the formula evaluates to 1. A **clause** is a set of literals joined by OR. Note that "if-then" conditions can be rewritten as clauses. A boolean formula is in **conjunctive normal form (CNF)** if it consists of clauses joined by AND. We remark that every Boolean function can be written equivalently as a CNF formula.
Given: a boolean formula, either in general form or in CNF.

Goal: Find a satisfying assignment (if there exists some).

Motivations: This is a fundamental problem in logic and related fields like Artificial Intelligence. The following is only an example of a more concrete application scenario.

Certain objects can be described as vectors of boolean variables, where each variable indicates whether the object has a certain property or not. Suppose that the properties of many objects are stored in a database. Now we want to retrieve an object that satisfies a given set of conditions, expressed as clauses. Before we process an expensive database query, it may be good to check whether the conditions are satisfiable at all. (The given specification may be overconstrained.) Furthermore, if the result is positive, we may search the database for occurrences of the satisfying assignments, which is much faster and simpler than testing the conditions for each database entry. (However, speed of the latter application depends on the number of satisfying assignments we have to try.)