Complexity Theory

Complexity Theory aims to classify problems (from mathematics and computer science) into different categories, depending on how easy they are to solve. As such, we mostly study algorithms.

Algorithms

An algorithm (derived from the name of mathematician Mohammed ibn-Musa al-Khwarizmi) is a sequence of instructions to follow, transforming an input into an output. A few examples of problems studied :

- Adding or multiplying two numbers a and b.
- Computing a^b where the input is a and b.
- Finding the biggest element in a sequence of numbers.
- Sorting a sequence of numbers from smallest to greatest.

Those are quite mathematical problems, but some are quite applied too :

- Given a map, finding the shortest path between two points (here the map is generally given as a set of points (cities) and roads with distances between each pair of point linked by a road).
- Given an image, deciding whether it is a cat or not.
- Transforming a soundwave into writing (text-to-speech algorithms).

There can be many different algorithms for one given problem. For example, the simple task of adding two numbers can be done in at least two different ways :

First method : we write the numbers in binary (or in decimal), align the two numbers, and add each pair of bits (or digits), while carrying the one.

Second method : we repeat the following : check if $b = 0$, if not replace a by $a + 1$ and b by $b - 1$ and repeat.

Those two methods are quite different and the second one is strikingly inefficient. As we wish to classify problems (and also algorithms), a good way to do so is to count the number of operations needed. However, the problem is always to choose which operation is relevant. For example, for humans, computing $a + b$ is easier than $a \times b$, but on small numbers (smaller than 2^{63}), computers can do either in a single operation. Also, we can easily agree that an addition is an elementary operation, but what happens when we try to find the shortest path between two points ? What should the elementary operation be then ? Multiple answers are possible, such as looking at a road and finding its cost, or looking at a city and checking whether it has a road to another city (or get a list of the roads from that city). There is another consideration we must have : finding the exact number of operation an algorithm takes depends on the input, and it could be very messy if we try to always be exact (not mentioning the fact that it would take a lot of space to describe all cases). We would like to have a function that gives us the time taken (in number of operations) by an algorithm at a lower level of precision. To this end, we introduce the following, called Landau notation:

- We say that $f(n) \in O(g(n))$ if there exists $k > 0$ such that $f(n) \leq kg(n)$ for all n
- We say that $f(n) \in \Omega(g(n))$ if there exists $k > 0$ such that $f(n) \geq kg(n)$ for all n
- We say that $f(n) \in \theta(g(n))$ if $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$.

For example, adding two numbers a and b (with $a \geq b$) using the first method takes a number of operations close to the number of digits, but depending on whether we count carrying the one as an operation, and depending on how often we do it, the total number of operations varies between $n - 1$ and $3n$, where $n = \lceil \log_{10} a \rceil$, that is, n is such that $10^n \geq a$ (or equivalently, n is the number of digits of a). However, using the previous notation, $n - 1$ and $3n$ are equivalent, and are both $\theta(n)$.

This notation is very useful because what matters the most to us most is generally the fastest growing part of the function and a factor 2 or 3 is often not that relevant. If an algorithm takes time $2n$ or $4n$, there is not much difference, as opposed to n^2 or n^3. The Landau notation also has two more symbols which are also useful in other fields (mostly in calculus):

- We say that $f(n) \in o(g(n))$ if for any $k > 0$ there exists n_0 such that $f(n) \leq kg(n)$ for all $n \geq n_0$.
- We say that $f(n) \in \omega(g(n))$ if for any $k > 0$ there exists n_0 such that $f(n) \geq kg(n)$ for all $n \geq n_0$.

Those symbols are used when we still want to know the constant factor instead of a $\theta(f(n))$, but don't care about the smaller terms. For example, $2n+1$ and $2n+\sqrt{n}$ take roughly the same time, as the important (fastest growing) part is $2n$. They are then both in $2n + o(n)$.

Remark. Sometimes we write $f(n) = O(n)$ instead of $f(n) \in O(n)$, and those two ways of writing are both quite frequent (some might consider $'='$ incorrect though).

Maximum and Minimum

As we've seen, computing the maximum of a table of integers can be done by comparing an element to another one by one, keeping at each point only the greatest. This uses $n-1$ comparisons. Computing the minimum can be done in the same fashion. Hence, we can compute minimum and maximum in total time $2n-3$. There is however a better algorithm : we get the maximum and minimum of the first two elements, then compare the elements number $2k$ and $2k+1$, and compare the biggest with the maximum so far, and the smallest with the minimum so far. This means that we do a total of 3 comparisons for 2 elements, hence a total of $\frac{3}{2}n$ comparisons. This can be shown to be optimal via hard arguments (using games), but the interesting part is that it is actually slower when coded on computers, because of optimisations not working anymore (something called a prediction miss).

Remark. Another point worth noting is that we spoke of integers, but these algorithms can be used to sort any data where two elements can be compared.

Sorting

Sorting is an interesting algorithmic problem, and one of the classics. We are given a table (a sequence) of n different integers, and the goal is to output a table of n integers in increasing order. There are multiple naive algorithms to do that :

- Find the smallest, put it in first place, then find the second smallest and so on. Or equivalently, do this with the greatest number.
- Take the number, find out where it belongs (count the number of elements greater and smaller than it), and insert it there.
- Look at the first two numbers and exchange them if they are in the wrong order. Then do it on the 2nd and 3rd, 3rd and 4th and so on till the end. If they were all already in the write order stop, if not restart from the 1st and 2nd numbers.

Those algorithms all take $\theta(n^2)$ operations. The first one because finding the smallest takes $n-1$ comparisons, and the second smallest takes $n-2$ and so on. For the second, finding where a number belongs also takes $O(n)$ comparison. For the last one, each run takes $n-1$ comparisons, and if the smallest number is at the end we will need to make n total runs, hence also $\theta(n^2)$. There is however a faster method (multiple in fact). Consider the following algorithm, called fusion sort:

1. If you have 2 elements, compare them and output them in the right order.
2. Else :
   (a) Cut your table in two equal parts (or with one part having at most 1 more element).
   (b) Sort each table independently.
   (c) Compare the smallest element of each table and move it into a new table, and repeat until one of the two tables until one of the tables are empty, then add the rest of the other table. Output the new table.

How many operations does that method use ? Well, checking the first IF takes one operation. Cutting the table in two also takes 1 operation. Building the new table takes $O(n)$ operations, and sorting each table takes twice the time needed for a table twice as small. Hence if the cost is $f(n)$, then $f(n) \leq O(n) + 2 \times f(n/2)$. We can show that this is in fact $\theta(n \times \log_2 n)$. This is because we sort one table of size n, two of size $n/2$, four of size $n/4$ and so on. At each level the cost will be $\theta(n)$, and the number of levels is equal to $\log_2 n$. This method is called divide and conquer, because we first split the problem into smaller problems, which we solve by induction before pasting the solutions together.

There are multiple methods that use $\theta(n \times \log_2 n)$ albeit with different constant in front. The best one is actually probabilistic in nature (it is expected to run in $\theta(n \times \log_2 n)$ with a very small constant), but has a very small probability of running in time $\theta(n^2)$ (probability that goes down exponentially with n). The interesting part is that we can actually show that this is optimal. To do this we'll introduce lower bounds on problems. A lower bound on a problem is a function $f(n)$ such that there can be no algorithm that solves the problem in time $\leq f(n)$ for all inputs. A trivial lower bound for sorting is $f(n) = n$, because we need to read the input which takes time n.

As seen in the combinatorics course, there are $n!$ ways of having a sequence of n different numbers (because each sequence corresponds to a permutation). If we know which permutation the input corresponds to, we can simply apply the inverse permutation to it and output that. Hence the task of sorting is equivalent to the one of finding how the input is permuted from an already sorted sequence. Each time we compare two numbers, we have two possible answers. Hence we can distinguish between two possibilities. If we do k comparisons, that means we can distinguish between 2^k possibilities. To be able to distinguish between different permutations we must then be able to distinguish $n!$ possibilities, so we need k comparisons where k is such that $2^k \geq n!$. This actually means that $k \in \Omega(n \times \log_2 n)$. Hence those sorting algorithms are optimal (up to a constant). This method to find lower bounds, where you count a number of possibilities you must distinguish from, and then see how many cases you can distinguish in k operations, is quite useful for simple result, but seldom gives good bounds (because it assumes that each time we make a comparison we really divide the number of possibilities by two, which is rarely the case).

Remark. There is an other sorting algorithm called bozosort, where one shuffles randomly the table and checks whether it is sorted. Checking whether it is sorted takes time $O(n)$, but on average we expect to do $O(n!)$ runs before we end up with a sorted table. It is potentially the fastest algorithm though.

Problems and Classes

So far the problems we have seen are simple, and can be solved quite quickly. An interesting thing is that many situations in real life can be modelized as such problems. However we must distinguish two types of problems : function and decision problems.

Decision problems are quite easy in appearance : we are asked a question and must respond yes or no. This is indeed very easy when the question is "is this number even?" (we only need to check the last bit), but much harder when the question is "is there a winning stategy from this position in chess?".

Function problems on the other hand, ask us to give a real output, such as a sorted table, or the length of a shortest path between two points, or the multiplication of two numbers. There is a link between the two classes though, because if I can ask questions like is there a path of length at most k, I can easily find the length of the shortest path (I just need to do it for k, $k-1$ and so on - although there are faster ways).

The algorithms we have seen so far (and most algorithms you'll ever see) all run in $\theta(n)$, $\theta(n^2)$ or $\theta(n^3)$. There is a reason for that : we like algorithms working in polynomial time ($O(n^k)$ for a given k). Those "behave nicely", in the sense that if you take an input twice as big they might run two times as slow, or four times, but in any case you get a constant factor slowdown. An algorithm taking $O(2^n)$ on the other hand, might be solved quickly on an instance of size 10, but be unsolvable on an instance of size 20 (which takes 1024 times more operations). Polynomial time algorithms are hence "nice", but that's not all. Suppose that you have a polynomial time algorithm solving a problem A, and another transforming in polynomial time an instance of a problem B into an instance of A. Then you combine both and you end up with a polynomial time algorithm solving the problem B. Hence, by composing (appropriately) polynomial time algorithms, or by doing a WHILE loop on one of them, we still work in polynomial time. This is why we call this class **P**, and why it is so well studied.

There is another class about as famous though, called **NP** (hence the problem **P** vs **NP**). As we have seen, the problems in **P** are all those that can be solved in polynomial time. The problems in **NP** on the other hand are those where, if we already have a solution, we can check its correctness in polynomial time. For example, we currently do not know how to quickly find a route of length less than k going through n cities (today our best algorithms take exponential time). However, given such a route, it is very easy to check that it is indeed of correct length and goes through all the cities. There are many problems in **NP** and we continue finding new ones, and they generally seem way more difficult than the ones in **P**. Indeed, finding a solution should probably be harder than checking that a solution is correct. However, so far we have no proof that the two classes are distinct, so there might be a quick algorithm for each problem in **NP**. It turns out that we would only need one : most of the problems studied in **NP** are in fact **NP − complete**. This means that if we can solve one of those quickly, we can solve all the others in **NP** quickly by something called a reduction. This is because there is a problem − called − SAT, such that finding a quick algorithm for it would give a quick algorithm for any problem in **NP**. SAT is defined as follows : you have a logical formula "(a AND b) OR (c IMPLIES (a AND (NOT d)))", and you must say yes if there is an assignment of each variable (a,b,c,d) to true or false that makes the formula true (SAT means satisfiable). For many problems, it has been showned that being able to solve them quickly would allow us to quicly solve SAT.

For more information I strongly recommend Introduction to Algorithms by Cormen, Leiserson, Rivest, Stein.