What are Algorithms?

By Amr El Abbadi

When trying to solve a problem, a computer scientist needs to devise a systematic and procedural solution that the computer can then follow to solve the problem. This systematic solution is called an **algorithm**.

Consider the problem of finding the telephone number of an individual. Assume we start with a phone set (in mathematical terms, a set does NOT have order) of pairs of names and their corresponding phone numbers: <name, number>. Given a query, e.g., a name, we would like to write an algorithm to retrieve from the phone set of pairs the corresponding number. Since the data is NOT sorted, the best we can do is to check all pairs in the list and see if the first component of any pair matches the query name. For example, we might query for Laila. Here is a simple algorithm:

- Compare Laila with the name in the first pair.
- If it matches "Laila", we are done and we return the corresponding phone number.
- Otherwise, we check the next pair in the set.
- Repeat this until either a name matches "Laila" or reach the end of the list and announce that Laila does not have a phone number (at least in our phone set).

This is an algorithm! From a Computer Science point of view, it is not a great algorithm, since it has to check every pair in the list. We call it a **linear** algorithm, since it takes linear amount of time proportional to the data set to come up with an answer. Imagine if the list had all 7 billion people on the earth, and each check took about 10 nano-seconds (10^{-9}). This process could take in the worst case about 1 second. That is a very simple "algorithm" and this is an eternity in terms of computers. Imagine that this operation was being done in the cloud (by a search engine). You would need to add communication time. Furthermore, assume you had to store this data on disk (the data was not only phone numbers, but also pictures and videos of Laila). Disks are slow. This simple algorithm would easily take multiple seconds, which is not acceptable for online interactions. (Would you get upset if a search engine took several seconds to answer a query?).

The study of algorithms explores how to develop methods and techniques that solve problems in an efficient manner. For example, in the phone search problem, a Computer Scientist might start by deciding to **sort** the phone list by name **before** executing any queries. Sorting is an algorithm that takes a list and orders them in numerical or alphanumerical order. Once sorted, we can devise a simple search algorithm:

- Given a *name*, Laila, compare the name with the name in the middle of the list, say *midname*.
- If Laila is less than midname (in alphabet order), we know that the phone number for Laila cannot be in the second half of the list, so we can discard all those pairs and we know *Laila* (if in the list) is in the first half. Or vice-versa.
- Now repeat this process for half of the list until you either find *name* or not.

This is a more efficient algorithm than our previous linear algorithm. The method used here is called **divide-and conquer**. It is a paradigm for many other algorithms. It only needs to check a logarithmic number of values. So, for example to search for a phone number in a list of 7 billion entries, we only need to check 33 names only! This makes a computer scientist happy. Divide and conquer is an example of an algorithm where the time it takes to solve the problem is much less than the time it takes to linearly touch every data item. We call this type of algorithm **logarithmic** (nothing to do with algorithm!). Of course we

had to first sort the list of names, which is an expensive operation in itself, but worth it if we are going to answer lots of queries.

Divide and conquer is one example of several different paradigms to help solve problems. These include **greedy methods, dynamic programing**, etc. Most of these algorithms aim to have complexity that is logarithmic, linear, quadratic or cubic in the size of the data set. Even though we would love for all algorithms to be logarithmic, realistically, many problems need to touch each data item more than once, hence the need for more than linear algorithms. All these algorithms are called **polynomial** time algorithms.

Unfortunately, there are many interesting problems, which we would like to solve, where computer scientists have not been able to design polynomial algorithms. This is not due to lack of trying: many very clever people have been banging their heads at some of these problems for decades (at least explicitly since the 1950s) to develop polynomial time algorithms, or prove that no such algorithm exists. An example of such a problem is **the travelling salesman problem**. In this problem, we have a set of cities connected by roads. The road between any two cities has a given length in miles. A travelling salesman would like to visit *all* cities while *minimizing* the travel time, i.e., the sum of the miles traveled. Turns out that to date, the best algorithm we can design is to enumerate all possible paths, i.e., test all permutations of the nodes and pick the cheapest. This is quite expensive as given only 20 cities, we need to generate more than 2.4 * 10^18 (24 followed by 17 zeros!). Even on the fastest computers, this would take too much time to solve. In general, there are many problems like this, where the best algorithms we know involve enumerating all possibilities. This type of algorithm is called **exponential** (or **non-polynomial**). (One can actually do better than generating all paths by using dynamic programming. The algorithm is faster than the brute force method, but still exponential.) In general these problems can easily take months to solve even on the fastest computers for relatively small problem sizes. In general, we do not know for a big set of problems whether there is a polynomial solution or not. This is a famous problem in Computer Science, and is referred to as the **"P=NP?" problem**. Many computer scientists have been working on this problem for decades and nobody has been able to ascertain if the answer is *Yes or No*.

Since many popular problems are NP, and we need to solve them at any rate, computer scientists have come up with many **approximation** algorithms. These algorithms are not guaranteed to give the exact correct solution (for example, they might not give the minimum cost solution to the travelling salesman problem), but often they give guarantees on how close. They also devised **randomized** algorithms that use randomization as part of their logic (flip a coin) and provide solutions with **probabilistic guarantees**.