What Is Theory of Computation and Why Do We Need It

What do computer scientists do? Computer science is an applied discipline. The main task of a computer scientist is:

- given a problem,
- to find (or to design) – and, if needed, to program – an algorithm for solving this problem.

The algorithm should be general. Usually, when we encounter a practical problem, we expect that many similar problems will appear in the future.

For example, when computer specialists designed a trajectory for the first flight to the Moon, they understood that similar trajectories will need to be computed for future flights as well, with different starting dates, different orientations of the Moon with respect to the Earth, etc.

You do not design a car just for one purpose: to go from El Paso to Las Cruces. You design a car that will allow you to go everywhere – at least everywhere where there are roads.

In general, computer scientists try to make their algorithms general, so that each resulting algorithm will be able to solve not only one specific problem, but all similar future problems as well. This will be much more efficient than solving each new problem "from scratch".

How general should our algorithms be? As we have just argued, the more general the solution, the more future efforts we save. In many cases – and the trajectory computation is one such example – by involving a relatively small amount of additional efforts, we can make the resulting algorithm (and the corresponding program) sufficiently general.

However, this easy-to-get generality has a limit. This limit is caused by the fact that the more general we make our current task, the more effort it will require right now. If we make the task too general, it will require too much extra efforts right now. Moreover, if we make it every general, maybe the resulting general problem will become algorithmically unsolvable – or at least not solvable in reasonable time.

For example, if we design a car to be able to travel between any two points in the Americas, we can do it. But if we want to make a machine that can

reach any point in the Solar system, this is not so easy – and at the present technological level, not yet possible.

This making the problem too general is not an abstract possibility – especially is this general task is formulated by a boss who is not a specialist in computer science. Sometimes, practitioners are asked to solve the problems in such a generality that no general algorithm is possible – and, of course, do not succeed. To avoid wasting time on such impossible efforts, it is desirable to know which problems can be algorithmically solved and which cannot. Understanding which problems can be algorithmically solved is one of the main objectives of theory of computation.

First objective: *Understand which problems can be algorithmically solved and which cannot be algorithmically solved.*

But even if an algorithm exists, is it realistic? In principle, there are situations when a general algorithm *is* possible, but the only possible algorithms require so much computation time – e.g., longer than the lifetime of the Universe – that they are not practically feasible. It is therefore desirable to know if the given problem can be feasibly solved. This is also one of the main objectives of theory of computation.

Second objective: *Understand which problems can be feasibly solved and which cannot.*

If the problem is, in principle, feasible solvable, what do we need to solve it? If the problem is, in principle, feasibly solvable, then the next natural question is: what equipment do we need to have to solve it? Some problems can be feasibly solved on a single computer, some require a highly parallel high performance computer for their solution, some require a quantum computer. This is the next objective of theory of computation.

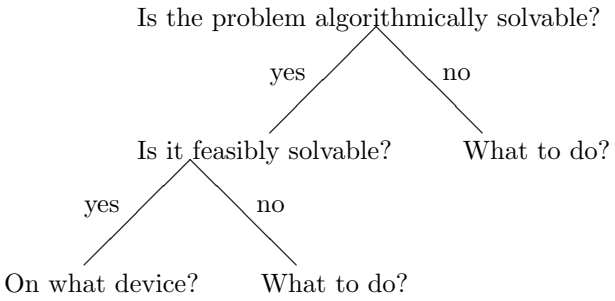
Third objective: *Understand what type of computational device is needed to solve a given problem.*

But what if a problem cannot be feasible solved? Suppose that a problem cannot be solved – or cannot be feasibly solved. What should we do? If this is a student defense, and this impossibility is a new result, everyone applauds, and the student gets his/her degree. But what if it is a practical problem?

Then, we need to modify the problem. Maybe we cannot compute the optimal solution, but we can compute a solution which is close to optimal? Maybe we cannot always come up with a solution, but we can provide solution is most cases – i.e., with probability close to 1? All this is another objective of theory of computation.

Fourth objective: *Decide how to modify the problem if the original problem does not have a feasible solution.*

All these objectives can be described by a simple graph:



How can we achieve these objectives? To prove that a problem is algorithmically solvable, a natural way is to provide an algorithm for solving this problem – and to show that this algorithm always works. But it is known that not all the problems are algorithmically solvable. How can we prove that some problem is not algorithmically solvable? To be able to do that, we need a formal definition of what is algorithmically solvable, i.e., of what is computable. This is what we will start the class with.

Similarly, to be able to prove that some problem is not feasibly solvable, we will need to have a formal definition of feasibility. This will be our next topic.