1.2 Problem-types

Problem-types provide primitive motivations for computing. To clarify the setup, let us start with an easy question,

Is 10 a multiple of 3?

Needless to say, this is a fairly easy problem and one may choose many different approaches to solve it. But let us consider the following scenarios:

- One lists multiples of 3 in an ascending order and reaching 12 without seeing 10 is a proof for an answer "No" to this question.
- Subtracting 3 iteratively one finds that the answer is "Yes" if and only if 1 is a multiple of 3 which is false. Hence, by *contradiction* the answer to this question is "No".
- One may be aware of more advanced techniques such as the fact that a number is a multiple of 3 if and only if the sum of its digits is a multiple of 3, and consequently, provides the answer "No".
- One may not be good enough in mathematics and provide an answer "Yes" although it is a wrong answer.
- Some other person may not be good enough in mathematics but she may know a bit about computer programing using the Java compiler and writes a program to verify the answer, however, since she is not careful enough her program enters an infinite loop and she does not receive any answer from the computer, with no strategy to stay for it or leaving the room!

Also, note that, even as an expert, although the answer to this question may be of some interest to you but as soon as the question is answered and the answer is verified for correctness somehow, then the solution is not of any general interest or importance anymore. In other words, the importance of this question lies in thinking about whether a specific integer 10 is a multiple of 3, and the method we use may not be useful for verifying the same question for some other integers. But, what if one is interested in methods that can answer questions about divisibility by 3 in general. In that case one may concentrate on the following question:

Is a given integer n a multiple of 3?

Clearly, this is not a question to think about until n is given and fixed to a specific integer, however, it is still meaningful to try to provide *methods* (not answers) that may provide answers to this question for different given integers n. Let us call such a parametric question, a *problem-type*. It is instructive to note that in this new setup what we are interested in are general methods that may answer the question *uniformly* in terms of a parameter n, meaning that by varying the integer n the method itself is fixed and just uses n as some given input or parameter. From another different point of view, one may think of such solutions as a family of proofs indexed by n which are identical in procedure except that n is used as a variable within the reasoning. Evidently, one can not explicitly go through such a procedure unless the variable n is fixed to some given integer.

Consequently, we may conclude that problem-types are primitive questions when one is not interested in a specific answer but when one is interested in *parametric procedures* that may be used to solve a *parametric family of questions*. Such procedures with *inputs* and *outputs* are usually called *algorithms* [4] while defining such concepts in a precise way is one of the most basic achievements of *theory of computation*.

Hence, before we proceed any further let us define a problem-type in a more explicit way as follows.

Definition 1.1.2 Problem-types

A *problem-type* $P(C, G, R)$ is a triplet consisting of a *list* (i.e. an ordered tuple) of *constants*, C, and a list of *inputs* or *given data*, I, along with a *request*, R, asking for some output. By

[4] Good reference to the roots of the name coming from Al Kharazmi

definition, when the request of a problem-type is a question whose answer is either "Yes" or "No", the problem-type is said to be a *Yes/No problem-type* and instead of a "request", R, we will talk about a *"query"* which is usually denoted by Q. Also, an *instance* of a problem-type is an explicit problem obtained by fixing the input of the problem-type to some given data. It is always assumed that a problem-type is well-defined in the sense that the request or the query is a legitimate and meaningful sentence. ▶

→ Assumptions

One of the main objectives of theory of computation is to provide a sound framework in which one may analyze and compare uniform solutions of problem-types, as well as the problem-types themselves. To begin, let us consider the following examples.

Example 2.1.2 A couple of Yes/No problem-types

The question "Does 3 divides n?" as a problem-type can be formulated as follows,

→ Sec. 2.1

$$P_1 \stackrel{\text{def}}{=} \begin{cases} C: & 3 \\ I: & n \text{ an integer} \\ Q: & \text{"Is } \frac{n}{3} \text{ in } \mathbb{Z}?" \end{cases}$$

Also, the question "Does n divides m?" as a problem-type has no constant, and may be formulated as follows,

$$P_2 \stackrel{\text{def}}{=} \begin{cases} C: & \emptyset \\ I: & n, m \text{ two integers} \\ Q: & \text{"Is } \frac{n}{m} \text{ in } \mathbb{Z}?" \end{cases}$$

Note that, in general, P_1 is a special case of P_2, and consequently, we expect that

"a typical solution of P_2 be more complex than a typical solution of P_1, while it may be used to solve P_1, anyhow."

but is this a true statement in general? One may think of very absurd and complex pathological solutions of P_1 while one may also think of very efficient solutions for P_2. How one may compare solutions and how one may compare problems? Does there exist best (i.e. optimum) solutions? Even to be more fundamental, one may ask,

"what is a *solution*"?

These are a couple of questions that motivate our investigations in Part II of this book, where we will introduce some fundamental concepts as *acceptors* and *deciders* and *reductions*. ◇

One of the main challenges one faces in analyzing problem-types and their solutions is the vast variety of these species. To get a feeling about this variety consider the following examples.

Example 3.1.2 A counting problem-type

The request part of a problem-type may ask for a variety of concepts. In particular, it may ask for the number of some species in which case we call it a *counting problem-type*.

$$P_3 \stackrel{\text{def}}{=} \begin{cases} C: & \emptyset \\ I: & (H, u, v) \text{ a simple graph along with two of its vertices .} \\ R: & \text{"What is the number of paths from } u \text{ to } v \text{ in } H?" \end{cases}$$

◇

Example 4.1.2 An optimization problem-type

Also, one may think of problem-types whose requests ask for some more general concepts than numbers. For instance, one may think of the following problem-type asking for a shortest path.

$$P_4 = \begin{cases} C: & \emptyset \\ G: & (H, u, v) \text{ a simple graph along with two of its vertices .} \\ R: & \text{"Provide a shortest path from } u \text{ to } v \text{ in } H." \end{cases}$$

◇

We will see that the *coding trick* helps to handle the variety of problem-types by reducing their solutions to the solution of some special problem-types with nice and clear structures. To begin. in what follows, first, we concentrate on one of the most important special cases, i.e. the case of Yes/No problems, and it turns out that in this case the coding trick reduces the whole case to the study of the membership problem-type which is defined as follows,

Definition 5.1.2 The membership problem-type

The *membership problem-type* for a given set $L \subseteq U$ (where U is a universal set) is defined as, → Sec. 2.1

$$P_{_L} \stackrel{\text{def}}{=} \begin{cases} C: & L \subseteq U \\ G: & w \in U \\ Q: & \text{“Is } w \text{ a member of } L\text{?”} \end{cases} .$$

▶

Moreover, one may use Yes/No problem-types to, somehow, approximate some other problem-types. For instance, consider the following Yes/No problem-type

$$P_3^k \stackrel{\text{def}}{=} \begin{cases} C: & \mathbb{N} \ni k \geq 1 \\ I: & (H, u, v) \text{ a simple graph along with two of its vertices} \\ R: & \text{“Is the number of paths from } u \text{ to } v \text{ in } H\text{, less than } k\text{?”} \end{cases} .$$

that provides some information about the problem-type P_3 mentioned before. Hence, solving Yes/No problems turns out to be a central aspect of theory of computation.

On the other hand, to solve a given problem-type, one of the main issues one faces at the very beginning is,

“How the problem is communicated to the computer (i.e. the solver)?”

whatever a *computer* is. In particular, for the case of the membership problem, $P_{_L}$, one has to first communicate a set L that may not have a finite size. This leads us to the most important assumption in the whole theory of computation, based on the finiteness of the communication time, indicating that, → Assumptions

“Only a finite number of symbols can be communicated between two parties.”

Consequently, one understands that, based on this assumption,

“Anything which is going to be communicated between two parties must have a *finite description* (i.e. as a finite string of symbols).”

Therefore, to have a well-defined theory of computation, at least for the case of Yes/No problem-types, first, one must clarify and precisely define the concept of “*a description of a set*” and then somehow study the class of sets with a finite description (compare to the solutions given at the beginning of this section for the divisibility by 3 where L is the set of numbers that are multiples of 3). Again, we will see that from this point of view, theory of computation for the case of Yes/No problems is essentially a study of sets having finite descriptions and the study of transforming these descriptions to each other.

One finds out that essentially there are many different and legitimate methods to describe a set, leading to one of the most important challenges of the theory of computation asking,

“Which one of the methods of description ought to be considered as the basis for the definition of *computation*?”

Let us consider a couple of kids, each having a pack of LEGO containing different kinds of pieces. Also, assume that the LEGO factory has provided a catalog containing many different kinds of shapes with the guarantee that each kid is capable of constructing these shapes using her set of pieces. Needless to say, this is a privilege of LOGO factory to be able to provide such a catalog, since the set of shapes available there will make all kids quite happy, although they

may be of different ages with different LEGO sets whose pieces may be quite different in shape and size. This sets forward a very interesting scenario, saying that, in spite of the difference in the size and shapes of the pieces in each LEGO package,

"there exists a set of shapes which are constructable by all LEGO packages."

The message of this setup is the important fact that, for any shape provided by the LEGO factory catalog, each LEGO package provide a new description of the shape in terms of its pieces.

The same phenomenon may happen when one is dealing with sets and different machineries to describe them. We will see that there are *abstract computers* (i.e. computing machines), *grammars*, *enumerators* and *computable functions* along with many other legitimates methods of descriptions for sets, giving rise to the following important question,

"What is the largest collection of sets whose elements can be described by all these methods of description?"

Actually, strictly speaking, the Church-Turing thesis was proposed to set an answer to the above mentioned question, since it is essentially not possible to logically distinguish and single out one of these possible methods of description as the best and most intuitive one. Hence, the Church-Turing thesis states that it is quite logical to consider a set of valid descriptions whose powers to describe sets are *equal* and *maximal*. Clarifying and formulating these intuitive concepts in a sound and precise way covers a large part of the theory of computation called the theory of *formal languages*.

On the other hand, one may try to generalize the above mentioned ideas for problem-types in general, while this approach gives rise to *function problem-types* and the theory of *recursive* (i.e. *computable*) functions providing and building the main pillars of the theory.

It is quite interesting to note that this generalization turns out to be quite fruitful, both in theory and scope, in the sense that it not only broadens the field of applicability of the theory from Yes/No problem-types to general ones, but also it provides the most fundamental facts of the theory which are not quite clear in the previous case. It is also quite remarkable that this general setting is actually the starting scenario when the theory was born (i.e. in 1930's) and there was noting similar to the notion of a *conventional computer* we are using today! → Sec. 1

Within the new setup, after generalizing the *many-to-one reductions* as our basic machinery to compare problems, to the *s.m.n. Theorem*, one will be able to prove the *recursion theorem* that will prove to be of fundamental importance, in such a way that most of the other basic results of the theory will follow as consequences of it. This, from another point of view, proves the importance of *fixpoint theorems* (i.e. *self-reference*) in theory of computation. → Sec. 10.4 → Sec. 17

Computing machines (hereafter called *computers*) and computational models are among the most basic concepts in theory of computation, in which one tries to answer the *design* and *analysis* problems for different variants and models of such machines, as well as to provide *methods of comparison* and *optimization* when one is dealing with them. In this sense, we will see that, essentially any such computer can be described as an *string processor* that admits a *finite description* in a well-defined setup. This property of having a finite description will turn out to be a consequence of the *local property* of such machines that distinguishes these species as dynamical systems. Hence, one of our main objectives throughout the book will be to convince the reader that → Sec. 3.2

"*computers* are *local discrete dynamical systems* on a set of strings and *computation* is the phenomenon of *evolution* of such a machine in time."

Therefore, one may claim that *computers* are local string processors, while a *computation* is a trajectory of such a machine showing its evolution in time. In what follows, we will try to construct the necessary theoretical setup to explain and precisely formulate such statements which will clarify the answer to the question, asking

"What is *computation*?"