Notes —#1

- This course is about the **inherent limitations** of computing: **The things we cannot do by writing a program**!

- At the **intuitive level**, any practicing mathematician or computer scientist—indeed any student of these two fields of study—**will have no difficulty at all in recognizing** a *computation* or an *algorithm* as soon as they see one

- But how about:

  "is there an algorithm that solves such and such a problem for all possible inputs?"—a question with potentially a **"no"-answer**—and also

  "is there an algorithm that solves such and such a problem via computations **that take no more *steps* than some (fixed) polynomial function of the input length?**"—this, too, being a question with a, potentially, "no" answer.

- **Example**:
  - "is there an algorithm which can determine whether or not a given computer program (the latter written in, say, the C-language) is **correct**?"[1]
    and
  - "is there an algorithm that will determine whether or not any given Boolean formula is a tautology, doing so via computations that take no more *steps* than some (fixed) polynomial function of the input length?"

[1]A "correct" program produces, *for every input*, precisely the output that is expected by an *a priori* specification.

- But **what do we mean** by

  "**there is *no algorithm* that solves a given problem"—with or without restrictions on the algorithm's efficiency**?

  This **appears** to be **a much harder statement to validate** than "there **is** an algorithm that solves such and such a problem"

  ▶ for the latter, all we have to do is *to **produce** such an algorithm* and a proof that it works as claimed.

  By contrast, the former statement implies, mathematically speaking, a***provably failed search* over the entire set of *all algorithms***, while we were looking for one that solves our problem.

- One evidently needs a **mathematically precise definition of the concept of algorithm** that is *neither experiential nor technology-dependent* in order to assert that we encountered such a failed "search".

  This directly calls for a *mathematical theory* whose objects of study include *algorithms* (and, correspondingly, *computations*) **in order to construct such sets of (all) algorithms within the theory and to be able to reason about the membership problem of such sets**.

- The “theory of computation” is the *metatheory* of computing.

  In the field of computing one **computes**: that is, develops programs and large scale software that are well-documented, correct, efficient, reliable and easily maintainable.

  In the (meta)theory of computing one **tackles the fundamental questions** of the *limitations of computing*, limitations that are intrinsic rather than technology-dependent.[2] These limitations may rule out outright the existence of algorithmic solutions for some problems, while for others they rule out efficient solutions.

[2]However, this metatheory is called by most people “theory”. Hence the title of this volume.

- Our approach is anchored on the **concrete (and assumed) practical knowledge** about general computer programming attained by the reader in a first year programming course, as well as the knowledge of discrete mathematics at the same level.

- Our chapter on computability is **the most general** *meta*theory of computing.

  We develop this metatheory via the programming formalism known as Shepherdson-Sturgis *Unbounded Register "Machines"* (URM)—which is a straightforward abstraction of modern high level programming languages.

  ► Contrast with TMs.

  Within that chapter we will also explore a restriction of the URM programming language, that of the *loop programs* of A. Meyer and D. Ritchie.

  We will learn that while these loop programs can only compute a very small subset of "all the computable functions", nevertheless they are *significantly more than adequate* for programming solutions of any "practical", computationally solvable, problem.

  For example, even restricting the nesting of loop instructions to *as low as two*, we can compute—in principle—enormously large functions, which with input x can produce outputs such as

$$\left.2^{\cdot^{\cdot^{\cdot^{2^x}}}}\right\} 10^{350000} \text{ 2's} \tag{1}$$

  The qualification above, "in principle", stems from the enormity of the output displayed in (1)—even for the input $x = 0$—that renders the above function way beyond "practical".

- The chapter on Computability—after spending due care in developing the technique of *reductions*—concludes by demonstrating the intimate connection between the *unsolvability phenomenon* of computing on one hand, and the *unprovability phenomenon* of proving within first-order logic (cf. [Göd31]) on the other, when the latter is called upon to reason about "rich" theories such as (Peano's) arithmetic—that is, the theory of natural numbers, equipped with: the standard operations (plus, times); relations (less than); as well as with the principle of mathematical induction.

- **Restricted Models**. FA and NFA and their Languages.

References

[Dav65] M. Davis, *The undecidable*, Raven Press, Hewlett, N. Y., 1965.

[Göd31] K. Gödel, *Über formal unentsceidbare sätze der pricipia mathematica und verwandter systeme i*, Monatshefte für Math. und Physic **38** (1931), 173–198, (Also in English in Davis [Dav65, 5–38]).