Undecidable problem

In computability theory and computational complexity theory, an **undecidable problem** is a decision problem for which it is proved to be impossible to construct an algorithm that always leads to a correct yes-or-no answer.[1] The halting problem is an example: it can be proven that there is no algorithm that correctly determines whether arbitrary programs eventually halt when run.

Example: the halting problem in computability theory

In computability theory, the halting problem is a decision problem which can be stated as follows:

> Given the description of an arbitrary program and a finite input, decide whether the program finishes running or will run forever.

Alan Turing proved in 1936 that a general algorithm running on a Turing machine that solves the halting problem for *all* possible program-input pairs necessarily cannot exist. Hence, the halting problem is *undecidable* for Turing machines.

Halting Problem

The halting problem is a decision problem in computer science that asks whether, given a description of an arbitrary computer program and an input, the program will eventually halt when run with that input or continue to run forever without halting. It is named after the concept of “halting”, which means that the program on certain input will accept it and halt or reject it and halt and it would never go into an infinite loop. Basically halting means terminating.

The halting problem is undecidable, meaning that no general algorithm exists that solves the halting problem for all possible program–input pairs. This proof is significant to practical computing efforts, defining a class of applications which no programming invention can possibly perform perfectly. The proof shows that any algorithm that can solve the halting problem can be made to produce contradictory output and therefore cannot be correct.
In computability theory, the halting problem is a decision problem about properties of computer programs on a fixed Turing-complete model of computation, i.e., all programs that can be written in some given programming language that is general enough to be equivalent to a Turing machine. The question is simply whether the given program will ever halt on a particular input.

For example, in pseudocode, the program while (true) continue does not halt; rather, it goes on forever in an infinite loop. On the other hand, the program print "Hello, world!" does halt. While deciding whether these programs halt is simple, more complex programs prove problematic.

Post Correspondence Problem

Post Correspondence Problem is a popular undecidable problem that was introduced by Emil Leon Post in 1946. It is simpler than Halting Problem. In this problem we have N number of **Dominos** (tiles). The aim is to arrange tiles in such order that string made by Numerators is same as string made by Denominators. In simple words, lets assume we have two lists both containing N words, aim is to find out concatenation of these words in some sequence such that both lists yield same result. Let's try understanding this by taking two **lists A** and **B**

```
A=[aa, bb, abb] and B=[aab, ba, b]
```

Now for sequence 1, 2, 1, 3 first list will yield aabbaaabb and second list will yield same string aabbaaabb. So the solution to this PCP becomes 1, 2, 1, 3.

Unsolvable problem for context free languages

There are **several unsolvable problems for context-free languages**. Some examples of these problems include:

- Whether a context-free grammar (CFG) generates all the strings or not.
- Whether two CFGs are equal.
- Whether a CFG is ambiguous.
- Whether it is possible to convert a given ambiguous CFG into a corresponding non-ambiguous CFL.
- Whether a language learning which is a CFL is regular.

Measuring and Classifying Complexity

There are different ways to measure the complexity of a system or an algorithm:

- Time complexity: the amount of time taken by an algorithm to run as a function of the input size.
- Space complexity: the amount of memory required by the algorithm to solve a problem.
- Auxiliary space: the amount of extra space used by the algorithm, excluding the input and output.
- Difficulty of description: the number of bits needed to describe the system or the algorithm. This can be measured by various concepts such as information, entropy, algorithmic complexity, minimum description length, Fisher information, etc.

Measuring and classifying complexity is a topic that involves different aspects of computer science, mathematics and information theory. Depending on the context, complexity can refer to the difficulty of solving a problem, the amount of resources required to do so, or the structure and behavior of a system.

Some examples of measures and methods for complexity are:

- **Cyclomatic complexity**: The number of independent cycles in the flow graph of a program or module[1].
- **Purity and neighbourhood separability**: Measures of classification complexity based on the spatial distribution of data points and their expected classes[2].
- **Computational complexity theory**: A branch of theoretical computer science that classifies computational problems according to their resource usage and relates these classes to each other[3].

Tractable and Intractable Problems

Tractable and intractable problems are two classes of computational problems based on their solvability in polynomial time. **Tractable problems** can be solved by a polynomial-time algorithm, which means the time required to solve them grows slowly with the size of the input **Intractable problems** cannot be solved by a polynomial-time algorithm, which means the time required to solve them grows exponentially or faster with the size of the input.

Tractable and Intractable Problems So far, almost all of the problems that we have studied have had complexities that are polynomial, i.e. whose running time T(n) has been O(n k) for some fixed value of k. Typically k has been small, 3 or less. We will let P denote the class of all problems whose solution can be computed in polynomial time, i.e. O(n k) for some fixed k, whether it is 3, 100, or something else. We consider all such problems efficiently solvable, or tractable. Notice that this is a very relaxed definition of tractability, but our goal in this lecture and the next few ones is to understand which problems are intractable, a notion that we formalize as not being solvable in polynomial time. Notice how not being in P is certainly a strong way of being intractable. We will focus on a class of problems, called the NP-complete problems, which is a class of very diverse problems, that share the following properties: we only know how to solve those problems in time much larger than polynomial, namely exponential time, that is 2O(n k) for some k; and if we could solve one NP-complete problem in polynomial time, then there is a way to solve every NP-complete problem in polynomial time. There are two reasons to study NP-complete problems. The practical one is that if you recognize that your problem is NP-complete, then you have three choices: 1. you can use a known algorithm for it, and accept that it will take a long time to solve if n is large; 2. you can settle for approximating the solution, e.g. finding a nearly best solution rather than the optimum; or 3. you can change your problem formulation so that it is in P rather than being NPcomplete, for example by restricting to work only on a subset of simpler problems. Most of this material will concentrate on recognizing NP-complete problems (of which there are a large number, and which are often only slightly different from other, familiar, problems in P) and on some basic techniques that allow to solve some NP-complete problems in an approximate way in polynomial time (whereas an exact solution seems to require exponential time). The other reason to study NP-completeness is that one of the most famous open problem in computer science concerns it. We stated above that “we only know how to solve NPcomplete problems in time much larger than polynomial” not that we have proven that NPcomplete problems require exponential time. Indeed, this is the million dollar question,1 one of the most famous open problems in computer science, the question whether “P = NP?”, or whether the class of NP-complete problems have polynomial time solutions. everyone believes that P6=NP, i.e. that no polynomial-time solutions for these very hard problems exist. But no one has proven it. If you do, you will be very famous, and moderately wealthy. So far we have not actually defined what NP-complete problems are. This will take some time to do carefully, but we can sketch it here. First we define the larger class of problems called NP: these are the problems where, if someone hands you a potential solution, then you can check whether it is a solution in polynomial time. For example, suppose the problem is to answer the question “Does a graph have a simple path of length |V |?”. If someone hands you a

path, i.e. a sequence of vertices, and you can check whether this sequence of vertices is indeed a path and that it contains all vertices in polynomial time, then the problem is in NP. It should be intuitive that any problem in P is also in NP, because are all familiar with the fact that checking the validity of a solution is easier than coming up with a solution. For example, it is easier to get jokes than to be a comedian, it is easier to have average taste in books than to write a best-seller, it is easier to read a textbook in a math or theory course than to come up with the proofs of all the theorems by yourself. For all this reasons (and more technical ones) people believe that P=NP, although nobody has any clue how to prove it. (But once it will be proved, it will probably not be too hard to understand the proof.) The NP-complete problems have the interesting property that if you can solve any one of them in polynomial time, then you can solve every problem in NP in polynomial time. In other words, they are at least as hard as any other problem in NP; this is why they are called complete. Thus, if you could show that any one of the NP-complete problems that we will study cannot be solved in polynomial time, then you will have not only shown that P=NP, but also that none of the NP-compete problems can be solved in polynomial time. Conversely, if you find a polynomial-time algorithm for just one NP-complete problem, you will have shown that $P=NP^2$.