Computational Complexity 1: Decision Problems and the Polynomial Class

Yufei Tao

Department of Computer Science and Engineering
Chinese University of Hong Kong

So far our study has focused on showing that a problem **can** be solved efficiently, namely, in **polynomial** time. There are, however, many other problems that

- either have been proved to be **unsolvable** by our computers today (i.e., no algorithms can possibly exist);
- or are solvable but are widely conjectured to **admit no polynomial-time algorithms** on our computers.

This lecture marks the beginning of a discussion on **computational complexity**, more specifically, on the **NP-hardness theory**, which is a powerful tool for arguing that a problem belongs to the class described in the second bullet.

The theory on computational hardness is built on **Turing machines**, which are taught in CSCI3130 (Formal Languages and Automata Theory). Unsolvable problems — formally known as **undecidable problems** — are discussed in that course.

The NP-hardness theory is, in essence, about **decision problems**. Intuitively, these are problems whose outputs are either "yes" or "no".

Examples of non-decision problems:

- Sort an array of n integers.
- Count the number of inversions in an array of n integers.
- Find the shortest path from a source vertex s to a destination vertex t.

Examples of decision problems:

- Given an array of n integers, are there two identical integers?
- Given an array of n distinct integers, are there at least $n/100$ inversions?
- Is there a path from a source vertex s to a destination vertex t that has a length of at most ℓ, where ℓ is an integer.

We will now define formally what is a “decision problem”.

Encoding an Input

An **input sequence** σ is a non-empty sequence of integers.

For any problem that can be solved by our computers, every possible input can be encoded as a sequence of integers, a.k.a. an input sequence (**think:** why?).

Not All Input Sequences are Meaningful

On the other hand, **not** every input sequence encodes a legal input for the underlying problem.

Example: Suppose that a problem takes as the input an array of n **distinct** integers, and that each input should be encoded as an input sequence of length n. The following input sequences are not legal inputs:

"325, 325"
"1, 32522, 1".

Decision Problem

A **decision problem** is defined by a pair of (L, Π) where

- L is a set of input sequences;
- Π is a function that maps L to $\{0, 1\}$.

This is actually very intuitive. Each bit string $\sigma \in L$ is a "legal" input to the problem, while $\Pi(\sigma)$ is the output of the problem (recall that a decision problem should return yes or no).

As mentioned, a main objective of the NP-hardness theory is to study which decision problems can be solved in "polynomial time" on a **Turing machine** which, however, is excessively complicated for our purposes.

We will take a different approach by resorting to the **RAM model** — the model we have been using throughout the course — to explain the NP-hardness theory with the same mathematical rigor. Towards that purpose, we must refine the the RAM model a bit so that it has the same computing power as a Turing machine, as far as polynomial time is concerned. As we will see, the refinement lies in specifying an appropriate value for the **word length**.

Refining the RAM model w.r.t. an input sequence

Storing an Input in the RAM Model

Fix a decision problem (L, Π).

Let σ be an input sequence in L, i.e., an input to the problem.
Define $m = |\sigma|$.
Denote by $\sigma[i]$ $(1 \leq i \leq m)$ the i-th integer in σ.

At the beginning:

- m is stored in a register;
- $\sigma[1], \sigma[2], ..., \sigma[m]$ are stored in the first m cells of the memory.

Recall that each register/cell is defined as a **word**, namely, a sequence of w bits, where w is the **word length**. So far we have never needed to worry about the value of w. But to discuss the NP-hardness theory, we must limit w to avoid an "unreasonably powerful" RAM model.

Choosing the Word Length

In general, we can represent a **signed** integer x as a sequence that

- Starts with a bit to represent the sign (e.g., 0 for negative and 1 for positive);
- and continues with the binary representation of $|x|$.

Denote by $len(x)$ the number of bits required to represent x.

For example, "-3" is represented as 0011 and "24" as 111000. Thus, $len(-3) = 4$ and $len(24) = 6$.

Choosing the Word Length (cont.)

We define

$$N = \sum_{i=1}^{m} len(\sigma[i])$$

namely, N is the number of bits required to represent σ. We will refer to N as the **bit length** of σ.

We will set the word length as

$$w = N.$$

Algorithms

An algorithm $\mathcal{A}$ is said to **solve** the decision problem (L, Π) if:

> given any input sequence $\sigma \in L$, the algorithm $\mathcal{A}$ correctly outputs $\Pi(\sigma)$, after the registers and memory have been initialized according to σ in the way described earlier.

The **cost** of $\mathcal{A}$ is defined in the same way as before, namely, the number of atomic operations performed.

Equivalence to Turing Machines Up to a Polynomial Factor

This slide is **not required** in the syllabus, and assumes the knowledge of Turing machines.

Theorem: A decision problem can be solved by a Turing machine in polynomial steps **if and only if** there is a RAM algorithm solving the problem in time polynomial to N, where N is the bit length of the input sequence.

The proof is beyond the scope of this course and omitted.

It is important to note that the theorem does not hold if w is allowed to be arbitrarily large (which equivalently says that a memory cell can represent arbitrarily large integers).

Be Careful with “Overflows”

Remember that a word of length w can represent integers in $[-2^{w-1}, 2^w - 1]$. Therefore, when you do arithmetic operations on two integers (e.g., $+, -, \cdot, /$), the result may fall out of the range, and therefore, cannot be stored in a single word.

The issue can be dealt with in numerous ways. We will follow a very simple approach: simply **disallow** such arithmetic operations, or equivalently, when the result of an operation falls out of the aforementioned range, a special register will be set to 1 so that the algorithm knows that an overflow has occurred.

The P class

A decision problem (L, Π) is said to be **polynomial-time solvable** if there is an algorithm $\mathcal{A}$ such that, for any input sequence $\sigma \in L$, $\mathcal{A}$ solves the instance represented by σ in time polynomial to N, where N is the bit-length of σ.

P is the set of decision problems that can be solved in polynomial time.

Example

Is the following problem in $\mathtt{P}$?
"Given an array of n integers, are there two identical integers?"

The answer is Yes.

An input sequence σ has n integers.
The problem can be easily solved in $O(n \log n)$ time with sorting in RAM.
If N is the bit-length of an input sequence σ, it clearly holds that $n \leq N$.
Hence, the problem can be settled in $O(N \log N)$ time, which is a polynomial of N.

Example

Similarly, it is easy to show that all the following problems are in P:

- Given an array of n distinct integers, are there at least $n/100$ inversions?
- Is there a path from a source vertex s to a destination vertex t that has a length of ℓ, where ℓ is an integer.
- Given an undirected weighted graph and an integer t, decide whether there is a spanning tree whose edges have a total weight of at most t.
- ...

Rule of Thumb: If you can find an algorithm whose running time is polynomial in all the parameters of the problem, then the problem is (almost for sure) in P.

Ideally, we would like to have all the decision problems in P — this would suggest that our computers are powerful enough to settle all those problems efficiently (i.e., in polynomial time). Unfortunately, it is commonly believed that this is not true. We will delve into this further in the subsequent lectures.