TIME COMPLEXITY

This material is in Chapter 7 in the textbook.

- Our previous discussion on computability gives absolute limits concerning which questions can, even in principle, be solved algorithmically. As we have seen, many important problems related to program correctness are undecidable.
- However, from a practical point of view the situation is even worse: decidability (as discussed in computability theory) does not give any consideration to the question *how long* it takes to find a solution to a problem.

Example. Traveling salesman problem, TSP:
You are given a map with 25 cities and distances between each two cities in miles. The task is to find an itinerary that minimizes the total distance traveled.

From a computability point of view the problem is trivially decidable. A straightforward algorithm to solve the TSP-problem is the following: on a map with n cities try all $n!$ possible itineraries and choose the shortest.

However, if checking one route (itinerary) takes a microsecond, using the above algorithm the present age of the universe would not be sufficient to solve TSP with 25 cities.

Important question: Is it possible to design an algorithm for TSP that is essentially more efficient than checking all possible solutions?

In particular, is there an algorithm where the required time is bounded by some polynomial function in n, where n is the number of cities?

Computational complexity considers the structure of computationally difficult problems (like TSP) and tries to determine whether there exist efficient algorithms or, in some cases, approximation algorithms for them.

The most important resources used to measure computational complexity are time and space. The Turing machine model has a very natural and precise definition of

- atomic computation step, (time = number of computation steps);
- amount of memory (= space) used by the computation.

In principle, it would be possible to consider the running time of a TM for each specific input, but this would be extremely cumbersome. In practice, as a time bound we consider the amount of time the machine uses in the worst case on inputs of given length.

The general results of computational complexity (for instance, whether some problem is "polynomial time" or "exponential time") do not depend on the Turing machine model, and we could get similar results using other realistic models of computation. However, the exact complexity of a given problem does depend on the model of computation used.

Definition. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ be a (deterministic, single-tape) Turing machine. The length of a halting computation of M on input w is the number of computation steps (one configuration yielding another) in the computation

$$q_0 w \vdash \ldots \vdash u_1 q u_2, \quad q \in \{q_{accept}, q_{reject}\}.$$

Here "$\vdash$" denotes the "yields" relation between configurations.

The time complexity of M with input w is

$$time_M(w) = \begin{cases} \text{length of the computation "}q_0 w \vdash \ldots\text{" if it halts;} \\ \infty \text{ if } M \text{ does not halt on } w. \end{cases}$$

Formally the time complexity of a Turing machine M is a function

$$time_M : \Sigma^* \longrightarrow \mathbb{N} \cup \{\infty\}$$

Determining the precise time complexity as defined above would be very messy even for simple Turing machines. Thus it is useful to consider only the

- worst-case, or
- average-case

time complexity *on inputs of given length.*

In worst-case analysis we define the time complexity for inputs of length n:

$$time_M^{max}(n) = \max_{|x|=n} time_M(x)$$

Since we will concentrate on worst-case complexity, we denote the function $time_M^{max}$ simply by $time_M$. In particular, if M halts on all inputs, then the running time or time complexity of M is a function

$$time_M : \mathbb{N} \longrightarrow \mathbb{N}$$

- Also the worst-case time complexity, if determined precisely, would usually have a very "messy" expression.
- For all practical purposes it is sufficient to determine the order of the function $time_M$.
- We use the asymptotic "big-O" and "small-o" notation.

Definition. *Asymptotic upper bound* (Definition 7.2)

Let $f, g : \mathbb{N} \longrightarrow \mathbb{R}_+$ (the positive reals). We say that f is of order $O(g)$, denoted

$$f(n) = O(g(n))$$

if there exist constants $c, n_0 \in \mathbb{N}$ such that

$$f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0.$$

If $f(n) = O(g(n))$ and $g(n) = O(f(n))$ we say that f and g are of same order and denote

$$f(n) = \Theta(g(n))$$

Definition. (*Asymptotically less than*)

We say that f is of lower order than g, denoted

$$f(n) = o(g(n))$$

if for any $c > 0$ there exists $n_c \in \mathbb{N}$ such that

$$f(n) < c \cdot g(n) \text{ for all } n \geq n_c.$$

The following lemma often helps us to determine the order of a function:

Lemma. Let $f, g : \mathbb{N} \longrightarrow R_+$ be functions. If the limit

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$

exists, is finite and non-zero, then $f(n) = \Theta(g(n))$.

If the limit is 0, then $f(n) = o(g(n))$.

If the limit is ∞, then $g(n) = o(f(n))$.

Examples.

1. If $p(n)$ is a polynomial of rank r with positive coefficients, then $p(n) = \Theta(n^r)$.
2. $\log_a n = \Theta(\log_b n)$ for all $a, b > 0$.
3. $n^a = o(n^b)$ if $a < b$
4. $c \cdot f(n) = \Theta(f(n))$ for any constant $c > 0$
5. $f(n) + g(n) = \Theta(\max(f(n), g(n))$ for all functions f, g.

Now we can define the classes of languages corresponding to various time bounds. Let $t : \mathbb{N} \longrightarrow \mathbb{N}$ be a function. The time complexity class

$$TIME(t(n))$$

is defined to consist of all languages that can be decided by a deterministic Turing machine in time $O(t(n))$.

It should be noted that, differing from computability theory, now the class $TIME(t(n))$ can depend on the exact Turing machine model that we are using, in particular, whether or not we allow multiple work tapes. See the examples in section 7.1 (the subsection "Analyzing algorithms").

Theorem. (Th. 7.8) Let $t(n)$ be a function where $t(n) \geq n$. Then every $t(n)$-time multitape Turing machine has an equivalent $O(t^2(n))$-time single-tape Turing machine.

Proof. The construction showing how a single-tape TM can simulate a multitape TM was presented in Chapter 3. Now we need to analyse the time needed for the simulation, this is done in the proof of Theorem 7.8.

Nondeterministic time complexity

Different branches of a nondeterministic computation may use very different amounts of time. To keep the definitions simple, our textbook defines time complexity only for *nondeterministic deciders*, that is, NTMs where all branches of the computation halt. The time used by an NTM is defined in terms of the *worst* (the longest) possible computation.[1]

[1]Some textbooks use a more general (and somewhat different) definition, where we allow non-halting branches in the computation and the time used by the computation is defined to be the length of the *shortest* accepting computation. Our definition and the more general definition result in the same general time complexity classes, like the class NP. Differences would occur only when considering time bounds that are not "well behaved".

Nondeterminism represents unlimited parallelism and hence the running time of an NTM is not intended to correspond to any real-world computing device.

Definition. (Def. 7.9) Let M be a nondeterministic Turing machine that is a decider. The running time of M is the function

$$time_M : \mathbb{N} \longrightarrow \mathbb{N}$$

where $time_M(n)$ is the maximum number of computation steps M uses on any branch of its computation on any input of length n.

Theorem. (Th. 7.11) Let $t(n) \geq n$. Then every $t(n)$ time nondeterministic single-tape TM has an equivalent $2^{O(t(n))}$ time deterministic single-tape TM.

- Observe that there is a polynomial difference in time complexity when we go from the deterministic multitape model to the single-tape model. However, the *upper bound* for simulating a nondeterministic Turing machine by a deterministic one is exponential.
- More generally, all "reasonable" deterministic computational models are (believed to be) polynomially equivalent, that is, they can simulate each other with only a polynomial increase in running time. This extension of the Church-Turing thesis is sometimes called the *Invariance Thesis.*

We define P to be the class

$$P = \bigcup_k TIME(n^k)$$

The class P has a similar role in complexity theory as the class of decidable languages has in computability theory: a problem is considered to be solvable "in practice" if it is in the class P. However, the classification P as the class of algorithmic problems that can be solved in practice has its own problems. A few of them are mentioned below.

Why does P not necessarily consist of problems that can be solved in practice:

1. P is "too large": a problem that with input n needs n^{1000} computation steps should not be considered solvable in practice.

   Comment: Although in theory it is possible to show that such problems exist, it seems that most natural problems in P can be solved in time bounded by a low degree polynomial.

2. P is "too small": What about algorithms that asymptotically require super-polynomial time, but for all inputs of realistic size work reasonably fast.

   Comment: This is a valid point if realistic inputs have a very small upperbound for the size. Consider, for instance, some problem where we have to select a subset of days of the week according to some criteria (number of different possibilities is only 2^7, although exponential).

3. Worst-case analysis can be misleading: a superpolynomial time algorithm can be "practically usable" if the worst-case inputs occur very infrequently.

   Comment: This is a valid objection. There are known examples (simplex method in linear programming) but usually worst-case exponential algorithms behave very "badly" also with average inputs.

Since P is invariant for all reasonable deterministic computational models, in order to show that a problem is in P it is usually sufficient to give a high-level description of a polynomial time algorithm to solve the problem. When implemented on a Turing machine each stage of the algorithm could require many steps: this is OK as long as each stage needs only polynomially many TM steps (as a function of input length).

As before, the notation $< \cdot, \ldots, \cdot >$ is used to denote an encoding of input objects as strings. The exact encoding is not important, as long as the used encodings are polynomially related (see section 7.2 in the textbook).

Note: When considering problems involving numbers (for example, determining whether two given numbers are relatively prime) the running time of an algorithm is measured as a function of the length of the representations of the numbers.

Example. Consider the encoding of numbers in unary notation:

$n \in \mathbb{N}$ is encoded as a string of n symbols 0.

Unary encoding is exponentially longer than the encoding of numbers in any base that is at least 2. Unary encoding is not "reasonable" in the above sense. For example, the integer factorization problem for an input given in unary is trivially in P. (Why?) The time complexity of integer factorization (with numbers written in binary) is a well known open problem.

We have seen that context-free languages are decidable. The usefulness of context-free grammars is based on the following result.

Theorem. (Th. 7.16) Every context-free language is in P.

The algorithm used in our earlier decidability proof for context-free languages was quite inefficient (obviously not a polynomial time algorithm). Using a dynamic programming technique we can parse context-free grammars essentially faster (to be discussed in class).