Time complexity

1 Introduction

Firstly, let's try to give some (informal) definition of **problem**. **Problem** can be described as a situation with an initial state of the system (input data) and a final state (output data). We need an algorithm to achieve the final state. Moreover, we will need a computational device to execute our algorithm. So, let's look at one.

2 Turing machine

A **Turing machine** (TM) is a mathematical model of computation. This model consists of several parts:

- Infinite memory **tape** divided into cells, $S = \{s\}$;
- A **head** g, that positioned on one cell at a time and can read from that cell, write into it, and move to one of its neighbors;
- A finite **alphabet** $A = \{a\}$, which symbols are stored in head and cells.

The state of TM is denoted as $x_i : s \cup \{g\} \to A$ (what symbols are in cells and head) and s_i (where is the head). For each time we consider two symbols: one in the head, another in cell s_i. This pair defines a set of actions that we will call an **instruction**:

- $\alpha : A \times A \to A$ – which symbol we'll write to cell;
- $\beta : A \times A \to A$ – which symbol we will store in head;
- $\gamma : A \times A \to \{-1, 0, 1\}$ – where we'll move the head.

So, the new state of TM can be calculated as:

- $x_{i+1}(s) = x_i(s), s \neq s_i$;
- $x_{i+1}(s_i) = \alpha(x_i(s_i), x_i(g))$;
- $x_{i+1}(g) = \beta(x_i(s_i), x_i(g))$;

- $s_{i+1} = \gamma(x_i(s_i), x_i(g))$.

Besides TM there are other machines like the von Neumann model, etc. But they can be emulated on TM and vice versa, so everything we discuss here can be applied to other models.

So, we have a computational machine. Now we can give some (informal) definition of an algorithm. **Algorithm** is a finite sequence of computational machine instructions, that translates it from the initial state to the final state (in other words, translates the input into the output).

3 Algorithm complexity and problem complexity

The algorithm α solves the problem R if it is being executed on a machine with input $x \in X$, and terminated after a finite amount of time in the final state, which contains the solution. $\mathrm{A}(R)$ – set of algorithms, that solve problem R.

Usually, we want to know how long some algorithms will work or how fast we can solve the problem. Let $r(x)$ – size of input, Then

$$t_R^\alpha(r) = \max_{r(x)=r} t_R^\alpha(x)$$

is an **algorithm complexity**. In other words, it's the worst-case for algorithm α from all possible inputs of the problem R. But for one problem R we have a set of algorithms, and we try to use the fastest one. With this we can define a **problem complexity** as

$$t_R(r) = \min_{\alpha \in \mathrm{A}(R)} t_R^\alpha(r).$$

Usually, when we estimate complexity, we use "$O()$" notation. Let's omit a strict mathematical definition. From here we denote size of input as n, $n = r(x)$. Phrase "algorithm complexity is $O(f(n))$" means, that with increase of n, computational time increases not faster than $af(n) + b$, where a and b – some constants.

There are some common time complexities:

- $O(1)$ – constant time;
- $O(\log n)$ – logarithmic;
- $O(n)$ – linear;
- $O(n \log^k n)$ – quasilinear (for $k = 1$ it can also be called linearithmic);
- $O(n^2)$ – quadratic;
- $O(2^{O(logn)} = poly(n)$ – polynomial;
- $O(2^{O(n)})$ – exponential (with linear exponent);
- $O(2^{poly(n)})$ – exponential;
- $O(n!)$ – factorial.

4 Classes P and NP

There are several types of problems. From here we will keep in mind on type – **decision problems**. A **decision problem** is a problem that can be posed as a yes-no question. For example: given two numbers x and y, does x evenly divide y?

The concept of polynomial time leads to several complexity classes. Here we will consider two of them. The first on is class **P** – the complexity class of decision problems that can be solved on a deterministic TM (usually referred just as TM) in polynomial time. We can say that this class contains "efficiently solvable" problems. Examples are decision versions of linear programming, GCD, check if the number is prime.

But there are problems, that have no "fast" solution, at least for now. And part of them is classified as **NP problems**. Class **NP** (nondeterministic polynomial) contains decision problems that can be solved on nondeterministic TM in polynomial time. Before we talk about nondeterministic TM, let's give **NP** another definition. It's the set of decision problems for which the problem instances, where the answer is "yes", have proofs verifiable in polynomial time by a deterministic TM.

Now we return to **nondeterministic TM**. As we said earlier, for each moment of time we have two symbols (in the head and in the cell). This pair defines instruction uniquely. But in **nondeterministic TM** one pair can define a set of instructions. And we can choose one path or another. So let's imagine one **nondeterministic TM** as multiple **deterministic** ones. So, if a pair of symbols defined two instructions, we get two TMs, where the first one uses first instruction, and the second TM uses the second instruction. All of our machines start with the same initial state (as we solve the problem for one input). Now, if one **nondeterministic TM** solved the problem looking the multiple possible solutions simultaneously, each **deterministic TM** looking one solution, and together they look all those possible solutions.

Problems in **NP** does not have "efficient algorithm". For example, "integer factorization" problem; its best-known algorithm has sub-exponential complexity. Even in **NP** there are algorithms, that harder than the others. They called **NP-complete**. **NP-complete** is a decision problem from **NP**, and any other problem from **NP** can be reduced to it in polynomial time. The most popular example is the decision version of the "traveling salesman problem". Others are "SAT", "Knapsack problem" etc. Their exact solution cannot be achieved at an adequate time. So, there is a set of common techniques to get an approximate one. But we will not discuss them today.

5 NP-hard

It has been said earlier, that **NP-complete** problems are the hardest in **NP**. **NP-complete** problems are contained in class, with problems that "at least as

hard as **NP-complete**". These are **NP-hard** problems. This class intersects with **NP**, and their intersection is **NP-complete** problems. The formal definition of **NP-hard**: a problem is **NP-hard** if every problem from **NP** can be reduced to it in polynomial time. Keep in mind that **NP-hard**, despite its name, doesn't contain the whole **NP**, nor is fully contained in it. The optimization version of the "traveling salesman problem" is proven to be **NP-hard**. Some of the problems are undecidable at all (all problems in **NP** are decidable). One of them is "halting problem":

Given the description of an arbitrary program and an input, whether the program will finish running, or continue to run forever.

Note: Here I didn't write its proof since it can be easily found in internet. Also, I'm quite lazy :-)