Wstęp do informatyki

Elements of computational complexity theory

Piotr Fulmański

Wydział Matematyki i Informatyki,
Uniwersytet Łódzki, Polska

January 17, 2019

Table of contents

Theory of computation

The *theory of computation* is the branch of *theoretical computer science* that deals with how efficiently problems can be solved on a model of computation, using an algorithm. The field is divided into three major branches:

- automata theory and languages,
- computability theory,
- and computational complexity theory.

All of them allow us to search an answer for the following question

What are the fundamental capabilities and limitations of computers?

Automata theory is the study of abstract machines and automata, as well as the computational problems that can be solved using them. The word automata (the plural of automaton) comes from the Greek word which means *self-acting*.

The field of the theory of computation that deals with researching which problems are solvable using computers.
One of the fundamental question of computer science is to determine the power of computers by understanding the problems that can be solved using them. Modern computers allow to compute so many things that it is tempting to think that solving each problem by them is only a matter of time. However, it turns out that we can find problems that computers will never be able to solve, regardless of the resources available.

In computability theory, the halting problem is the problem of determining, from a description of an arbitrary computer program and an input, whether the program will finish running (i.e., halt) or continue to run forever.
Alan Turing proved in 1936 that a general algorithm to solve the halting problem for all possible program-input pairs cannot exist. A key part of the proof was a mathematical definition of a computer and program, which became known as a Turing machine; the halting problem is undecidable over Turing machines.

The field of the theory of computation that deals with determining the amount of resources needed to solve computational problems (which is known to be computable).
The resources considered are such as time, memory or the number of processors.

Computational complexity

Computational complexity determine the amount of resources needed to solve a given computational problem. Computational complexity, in most cases the amount of memory or time, is expressed as a function of input data size. It is common for this function to expressed the worst-case complexity, that is the maximum of the amount of resources that are needed for all inputs of a given size.

The space complexity of an algorithm or a computer program is the amount of **memory space, expressed in bytes or in number of basic data types variables like `int` or `float`**, required to solve an instance of the computational problem as a function of the size of the input.

The time complexity describes the amount of time it takes to run an algorithm. We do not express time complexity in standard units of time. Providing time complexity in units of time is inconvenient, because the result depends on the speed of the computer on which the measurements were made and it is difficult to refer such results to other computers, equipped with other/different hardware resources, where the time of performing similar operations may vary significantly. Therefore, **we express the computational complexity in the number of elementary (dominating) operations**. The dominant operation is an operation whose execution directly affects the overall execution time of the entire algorithm. We treat the other operations as irrelevant – that is, their execution time is negligibly small compared to the time of execution of all dominant operations. Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, supposing that each elementary operation takes a fixed amount of time to perform. Thus, the amount of time taken and the number of elementary operations performed by the algorithm are taken to differ by at most a constant factor.

```
sumOfNumbers(numbers[]) {
    sum = 0;
    for number in numbers {
        sum += number
    }
    return sum
}
```

time complexity

$$f(n) = 1 + n + 1 = n + 2,$$

where n is a length of `numbers` array.

Computational complexity

The time complexity. Example

```
sumOfNumbers(numbers[]) {
    sum = 0;
    for number in numbers {
        sum += number
    }
    return sum
}
```

time complexity

$$f(n) = 1 + n + 1 = n + 2,$$

where n is a length of `numbers` array.

In practice, finding the exact form of the function f can be very difficult and at the same time completely pointless, because it can be enough to estimate it.
These estimates form *computational complexity classes* that specify the f function order. One way to write an order is to use universal notation: O, Ω, Θ, o and ω.

Computational complexity classes

Let f be a real or complex valued function and g a real valued function, both defined on some unbounded subset of the real positive numbers, such that $g(n)$ is strictly positive for all large enough values of n.
One writes

$$f(n) = O(g(n)) \text{ as } n \to \infty$$

if and only if there exists a positive real number M and a real number n_0 such that

$$|f(n)| \leq Mg(n) \text{ for all } n \geq n_0.$$

Other words

- We use big O notation to determine **upper asymptotic limits**.
- $|f|$ is bounded above by g (up to constant factor) asymptotically.
- It limits the increase in execution time for large input data.

Computational complexity classes

Big O notation

Let f be a real or complex valued function and g a real valued function, both defined on some unbounded subset of the real positive numbers, such that $g(n)$ is strictly positive for all large enough values of n.
One writes

$$f(n) = O(g(n)) \text{ as } n \to \infty$$

if and only if there exists a positive real number M and a real number n_0 such that

$$|f(n)| \leq Mg(n) \text{ for all } n \geq n_0.$$

Other words

- We use big O notation to determine **upper asymptotic limits**.
- $|f|$ is bounded above by g (up to constant factor) asymptotically.
- It limits the increase in execution time for large input data.

Plot

plot

Computational complexity classes

Big Ω notation

f and g as for big O notation.
One writes

$$f(n) = \Omega(g(n)) \text{ as } n \to \infty$$

if and only if there exists a positive real number M and a real number n_0 such that

$$f(n) \geq Mg(n) \text{ for all } n \geq n_0.$$

Other words

- We use big O notation to determine **lower asymptotic limits**.
- f is bounded below by g asymptotically.
- The algorithm takes at least some time without giving the upper limit.

Computational complexity classes

Big Ω notation

f and g as for big O notation.
One writes

$$f(n) = \Omega(g(n)) \text{ as } n \to \infty$$

if and only if there exists a positive real number M and a real number n_0 such that

$$f(n) \geq Mg(n) \text{ for all } n \geq n_0.$$

Other words

- We use big O notation to determine **lower asymptotic limits**.
- f is bounded below by g asymptotically.
- The algorithm takes at least some time without giving the upper limit.

Plot

plot

Computational complexity classes
Big Θ notation

f and g as for big O notation.
One writes

$$f(n) = \Theta(g(n)) \text{ as } n \to \infty$$

if and only if there exist a positive real number M and N and a real number n_0 such that

$$Mg(n) \leq f(n) \leq Ng(n) \text{ for all } n \geq n_0.$$

Other words

- We use big Θ notation to determine **lower and upper asymptotic limits**.
- f is bounded both above and below by g asymptotically.
- The algorithm takes at least but no more than some time.

Computational complexity classes

Big Θ notation

f and g as for big O notation.
One writes

$$f(n) = \Theta(g(n)) \text{ as } n \to \infty$$

if and only if there exist a positive real number M and N and a real number n_0 such that

$$Mg(n) \leq f(n) \leq Ng(n) \text{ for all } n \geq n_0.$$

Other words

- We use big Θ notation to determine **lower and upper asymptotic limits**.
- f is bounded both above and below by g asymptotically.
- The algorithm takes at least but no more than some time.

Plot

plot

Computational complexity classes
Little o notation

f and g as for big O notation.
One writes

$$f(n) = o(g(n)) \text{ as } n \to \infty$$

if for every positive constant ε there exists a constant N such that

$$|f(n)| \leq \varepsilon g(n) \text{ for all } n \geq N.$$

Plot

plot

Computational complexity classes

Little ω notation

f and g as for big O notation.
One writes

$$f(n) = \omega(g(n)) \text{ as } n \to \infty$$

if for every positive constant ε there exists a constant N such that

$$|f(n)| \geq \varepsilon |g(n)| \text{ for all } n \geq N.$$

Plot all at once

plot all at once

Most offten used computational complexity classes

$O(1)$ – the complexity is constant, independent of the number of input data.
An example of a problem for which there exists an algorithm of $O(1)$ complexity.
As the input there is an array of numbers with N elements. The numbers are sorted in ascending order. The difference between every two subsequent numbers is constant. Find the sum of the numbers in the array.

Most offten used computational complexity classes

$O(n)$ – linear complexity. This is a specific case of polynomial complexity. The problem solving time is directly proportional to the size of the input data.
An example of a problem for which there exists an algorithm of $O(n)$ complexity.
As the input there is an array of numbers with N elements. Find the sum of all numbers in the input array.

Most offten used computational complexity classes

$O(log(n))$ – logarithmic complexity
An example of a problem for which there exists an algorithm of $O(log(n))$ complexity.
As the input there is an array of numbers with N elements. The numbers are sorted in ascending order. Check if the number x exists in the input array.

Most offten used computational complexity classes

$O(n \log(n))$ – linear-logarithmic complexity.
An example of a problem for which there exists an algorithm of $O(n \log(n))$ complexity.
As the input there is an array of numbers with N elements. Sort the input array.
We can use for example *merge sort* which complexity is of $O(n \log(n))$ order.

Most offten used computational complexity classes

$O(n^2)$ – square complexity. This is a specific case of polynomial complexity.
An example of a problem for which there exists an algorithm of $O(n^2)$ complexity.
As the input there is an array of numbers with N elements. Sort the input array.
We can use for example *selection sort* which complexity is of $O(n^2)$ order.

Most offten used computational complexity classes

$O(n^3)$ – cubic complexity. This is a specific case of polynomial complexity.
An example of a problem for which there exists an algorithm of $O(n^3)$ complexity.
Multiplication of square matrix of size n.

Most offten used computational complexity classes

$O(n^x)$ – polynomial complexity.

Most offten used computational complexity classes

$O(2^n)$ – exponential complexity
An example of a problem for which there exists an algorithm of $O(2^n)$ complexity.
As the input there is an array of numbers with N elements. All elements are different. Return an array that will contain all possible subsets of the elements of the input array.

Most offten used computational complexity classes

$O(n!)$ – factorial complexity
An example of a problem for which there exists an algorithm of $O(n!)$ complexity.
The travelling salesman problem (TSP) asks the following question:
"*Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?*" It is an NP-hard

Plot of orders

plot of orders

General complexity classes

Complexity classes introduced so far can be grupped into more general complexity classes.

- P (polynomial time) complexity class – problem solving during polynomial time.
  It contains all decision problems that can be solved by a deterministic Turing machine using a polynomial amount of computation time, or polynomial time.
- NP (nondeterministic polynomial time) complexity class – the problem is not known at polynomial time, but the solution can be checked in polynomial time.
  It contains all decision problems solvable in polynomial time by a non-deterministic Turing machine.
- NP-hard – even solution to a problem, no matter how we get it, cannot be verified "quickly", in polynomial time.