Ordered Sets for Data Analysis

Introduction to Algorithmic Complexity

Sergei O. Kuznetsov

Master Program in Data Science

National Research University Higher School of Economics, Moscow, Russia,
September 27, 2022

Asymptotic notation Θ, O, Ω, o, and ω

- Algorithmic complexity theory studies asymptotical (the size of input tends to infinity) efficiency of algorithms, independent of software realization and hardware used.
- Note:
  Usually, an algorithm that is asymptotically more efficient than another one is more efficient for all data except for some amount of small data.

$\Theta-$notation

For a function $g(n) : N \to N$ notation $\Theta(g(n))$ denotes the set of functions

$$\Theta(g(n)) = \left\{ \begin{array}{l} f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \\ \text{such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

Possible notations:

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

$\Theta(g(n))$ requires that every element $f(n) \in \Theta(g(n))$ be asymptotically nonnegative.

Note:

With $\Theta-$notation a function is asymptotically bounded from above and below.

$O-$notation

$O-$notation determines **asymptotic upper bound**.

$$O(g(n)) = \left\{ \begin{array}{l} f(n) : \text{there exist positive constants } c \text{ and } n_0 \\ \text{such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

- $f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n))$
- In set-theoretic notation $\Theta(g(n)) \subset O(g(n))$

Examples for $O-$notation

- $an^2 + bn + c$, where $a > 0$, belongs to set $\Theta(n^2)$ denotes that any square function of this form belongs to set $O(n^2)$.
- Any linear function $an + b$ for $a > 0$ also belongs to set $O(n^2)$. Check it by choosing $c = a + |b|$ and $n_0 = max(1, -b/a)$.

$\Omega-$notation

$\Omega-$notation determines **asymptotic lower bound**.

$$\Omega(g(n)) = \left\{ \begin{array}{l} f(n) : \text{there exist positive constants } c \text{ and } n_0 \\ \text{such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

For any functions $f(n)$ and $g(n)$ the relation $f(n) = \Theta(g(n))$ holds iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

$o-$notation

$o-$notation expresses that **upper bound** is **not asymptotically exact**

$$o(g(n)) = \left\{ \begin{array}{l} f(n) : \text{for any positive constant } c \text{ there exists} \\ n_0 > 0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

Example: $5n^2 = o(n^3)$, but $5n^3 \neq o(n^3)$

Difference between $O-$notation and $o-$notation

- $f(n) = O(g(n)) \Rightarrow 0 \leq f(n) \leq cg(n)$ only for **some** constant $c > 0$.
- $f(n) = o(g(n)) \Rightarrow 0 \leq f(n) < cg(n)$ for **all** constants $c > 0$.
- If $f(n) = o(g(n))$, then $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

$\omega-$notation

$\omega-$notation defines a **lower bound** which is not exact. Possible definition: $f(n) \in \omega(g(n))$ iff $g(n) \in o(f(n))$

$$\omega(g(n)) = \left\{ \begin{array}{l} f(n) : \text{for any positive constant } c \text{ there exists} \\ n_0 > 0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

Example: $\frac{n^2}{2} = \omega(n)$, but $\frac{n^2}{2} \neq \omega(n^2)$

Note:

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

.

Problems, reducibility, complexity classes

Mass problem (generic instance) is a general question for which an answer should be found.

- general list of all parameters;
- formulation of the properties that the **answer** or **solution of the problem** should match.

An instance I is obtained from a mass problem Π if all parameters are assigned particular values.

- Theory of *NP*-completeness considers only **decision problems** (with only *"yes"* or*"no"* answers).

Problems

A decision problem Π consists of two sets:

- D_Π is a set of all possible *instances*;
- $Y_\Pi \subset D_\Pi$ is a *"yes-instances"*-set.

Standard form for problem description:

- Description of the problem.
- Formulation of the question which allows for two answers, "yes" and "no".

Formal languages for problems representation

Decision problems can be represented by formal languages. For any finite set of symbols Σ the set of all finite sequences (words) composed of symbols from Σ is denoted by Σ^*.

- An arbitrary subset $L \subseteq \Sigma^*$ is a **language** over alphabet Σ.
- **Encoding schemes** establish the correspondence between decision problems and language.

Encoding scheme

Π is a given problem, e is an encoding scheme.

- e describes an instance from Π by a word in a fixed alphabet Σ.
- Π and e partition set Σ^* into three sets:
  - the set of words that are not codes of instances of Π,
  - the set of words that are codes of instances of Π for which the answer is "no",
  - the set of words that are codes of instances of Π for which the answer is "yes".

The third set of words constitutes the language $L[\Pi, e]$, which corresponds to problem Π under encoding e.

Encoding scheme

Length: $D_\Pi \to Z^+$, is an associated encoding-independent function for decision problem, giving a formal measure of the **input length (size)**.

The function Length is **polynomially equivalent** to the length of encoding of an instance:

$$\text{there are two polynomials } p \text{ and } p' : \text{ if } I \in D_\Pi \text{ and word } x$$
$$\text{is the code of } I \text{ under encoding } e, \text{ then Length}[I] \leq p(|x|)$$
$$|x| \leq p'(\text{Length}[I]), \text{ where } |x| \text{ is the length of word } x.$$

Worst-case **complexity** of an algorithm

Time complexity

Time complexity of a program for deterministic Turing machine or DMT-program for solving instance $I \in \Pi$ (in particular, $I \in D_\Pi$) with input length (*size*) n is defined as the number of computation steps before the termination of the program.

$T_M(n)$

The **worst-case time complexity** of an algorithm for solution of problem Π is the maximal time complexity among all instances of size n.

$S_M(n)$

The **worst-case space complexity** of an algorithm for solution of problem Π is the maximal amount of memory needed for the algorithm among all instances with input size n.

Polynomial algorithm

An algorithm (a program for Turing machine) is called **polynomial** if there exists a polynomial p such that for all $n \in NT_M(n) \leq p(n)$. The class of **polynomially computable** functions P is defined as a set of languages

$$P = \{L : \text{ there exists a polynomial algorithm } M \text{ such that } L = L_M\}.$$

Example

Consider a list of objects with unique identifiers from a linearly ordered set $(K; \leq)$, e.g., the set of natural numbers. The problem of **sorting** consists of reordering the list in accordance with the linear order $(K; \leq)$, where $|K| = n$.

- **Sorting by choice** starts with finding the object with the least identifier, placing it in the beginning of the list, and iterating these actions until the set is reordered.
- $T_M(n) = O(n^2)$ time.
- **The advantage** is the small amount of data relocation.

Example

Consider set $A = \{a_1, \ldots, a_n\}$ and set F of some subsets of A. The problem is to compute **all possible intersections of sets from** F. If $F = \{A \setminus \{a_i\} \mid i \in 1, \ldots, n\}$, then any proper subset of A can be obtained as intersection of some sets from F.
The number of all proper subsets of A is $2^n - 1$.
In the worst case any algorithm would spend at least $O(2^n)$ time computing and outputting the intersections.

Problem complexity

- What is the fastest algorithm for solving a problem?
- What is the lowest worst-case complexity of an algorithm for solving a particular problem?

There is a famous class of **NP-complete** problems, for which no polynomial-time algorithms are known, but it is not yet proved mathematically that such algorithms do not exist.

Nondeterministic Turing machine (NDMT)

Two modules of NDMT:

- usual Turing machine used as **checking module**,
- a **guessing module**.

Two stages of computing with NDMT:

- **the stage of guessing**,
- **the stage of checking**.

Nondeterministic Turing machine (NDMT)

Computing with NDMT:

1. At the guessing stage, upon receiving instance T the guessing module guesses some structure $S \in \Sigma^*$;
2. I and S are input to the checking module, which works like a usual program for Turing machine, it either outputs "yes" (if S is a solution to problem I) or "no", or runs without termination.

A nondeterministic algorithm (NDA)

NDA is a pair
$\langle$guess object S; check S by deterministic algorithm $i\rangle$. NDA "solves" decision problem Π if for any $I \in D_\Pi$:

1. If $I \in Y_\Pi$, then $\exists S$, guessing which results in the checking stage with input (I, S) and termination with answer "yes".
2. If $I \notin Y_\Pi$, then for input I there is no S, guessing which would guarantee that the checking stage with input (I, S) terminates with answer "yes".

Nondeterministic polynomial time

A **Nondeterministic algorithm** (NDA) for Π is **polynomial time** if there exists a polynomial p : for any $I_\Pi \in Y_\Pi$ there is guess S : for input (I, S) the checking module outputs "yes" in time $p(Length|I|)$.

Informally:

Class NP is the class of all decision problems Π, which (under reasonable encoding) can be solved by a nondeterministic algorithm in polynomial time.

Example. Graph Isomorphism

INSTANCE. Two graphs $G = (V, E)$ and $G' = (V', E')$.
QUESTION. Is it true that G and G' are isomorphic, i.e., there exists bijection $f : V \to V'$ such that $\{u, v\} \in E$ iff $\{f(u), f(v)\} \in E'$?

Polynomial transformations by Carp

Decision problem Π_1 is **polynomially transformed** to decision problem Π_2 (denoted by $\Pi_1 \propto \Pi_2$) if there exists $f : D_{\Pi_1} \to D_{\Pi_2}$ satisfying two conditions:

1. there exists a polynomial DTM program (algorithm) that computes f;
2. for any $I \in D_{\Pi_1}$ one has $I \in Y_{\Pi_1}$ iff $f(I) \in Y_{\Pi_2}$.

Polynomial equivalence of decision problems

Decision problem Π_1 and decision problem Π_2 are **polynomially equivalent** if $\Pi_1 \propto \Pi_2$ and $\Pi_2 \propto \Pi_1$.

NP-**hard**

A decision problem is **NP-hard** if any decision problem from NP can be transformed to it.

NP-**complete**

A decision problem is **NP-complete** if it belongs to NP and is NP-hard.

Properties of decision problems

1. If $\Pi \in$ NP, then $\exists$ a polynomial $p : \Pi$ can be solved by a deterministic algorithm of the time complexity $O(2^{p(n)})$.
2. If $\Pi_1 \propto \Pi_2$ then $\Pi_2 \in P \Rightarrow \Pi_1 \in P$.
3. If $\Pi_1 \propto \Pi_2$ and $\Pi_2 \propto \Pi_3$, then $\Pi_1 \propto \Pi_3$.
4. If Π_1 and Π_2 belong to NP, Π_1 is NP-complete and $\Pi_1 \propto \Pi_2$, then Π_2 is NP-complete.

Some classical NP-complete problems

1. **3-SATISFIABILITY (3-SAT)**

   INSTANCE. 3-CNF, conjunction $C = c_1 \,\&\, c_2 \,\&\, \ldots, \,\&c_m$ of disjunctions (clauses) $c_i = (x_i^i \vee y_i^i \vee z_i^i)$ over a finite set U of variables is given so that $|c| = 3$, x_i^i means variable x_i, either negated or nonnegated, and $1 \leq i \leq m$.
   QUESTION. Is there a satisfying truth assignment for variables from U so that C becomes a tautology, i.e., ALL clauses of C become tautologies?

Some classical NP-complete problems

2. **SUBGRAPH ISOMORPHISM**

   INSTANCE. Two graphs $G = (V_1, E_1)$ and $H = (V_2, E_2)$.
   QUESTION. Does graph G contain a subgraph isomorphic to graph H?
   Formally, are there subsets V, E, and function $f : V_2 \to V$: $V \subseteq V_1, E \subseteq E_1$ such that $|V| = |V_2|, |E| = |E_2|$, and $\{u, v\} \in E_1$ iff $\{f(u), f(v)\} \in E_2$?

Some classical NP-complete problems

3. **CLIQUE**

   INSTANCE. A graph $G = (V, E)$ and a natural number $J \leq |V|$.
   QUESTION. Is there a subgraph of G that is a clique of size not less than J, i.e., a subset $V' \subseteq V$ such that $|V'| \geq J$ and any two vertices from V' are connected by an edge from E?

Relation between complexity classes

It is generally believed that the most plausible relation between complexity classes is $P \neq NP$, $P \subset NP$.

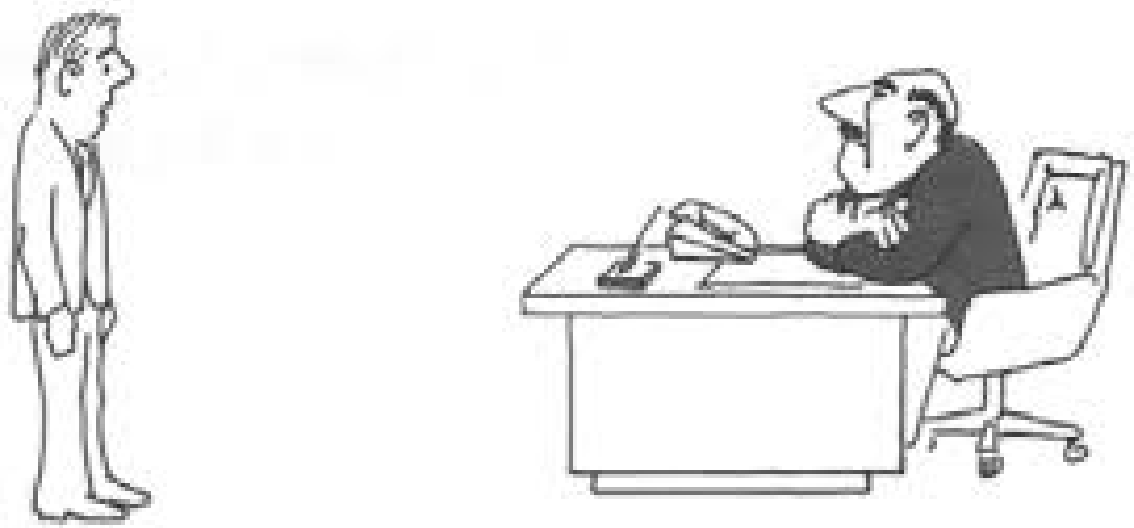
Practical use of NP-completeness (NP-hardness)

[M. Garey, D. Johnson]

Suppose your boss gives you a task to design a fast algorithm for computing bandersnatch. You are thinking over for two weeks and...

Practical use of NP-completeness (NP-hardness)

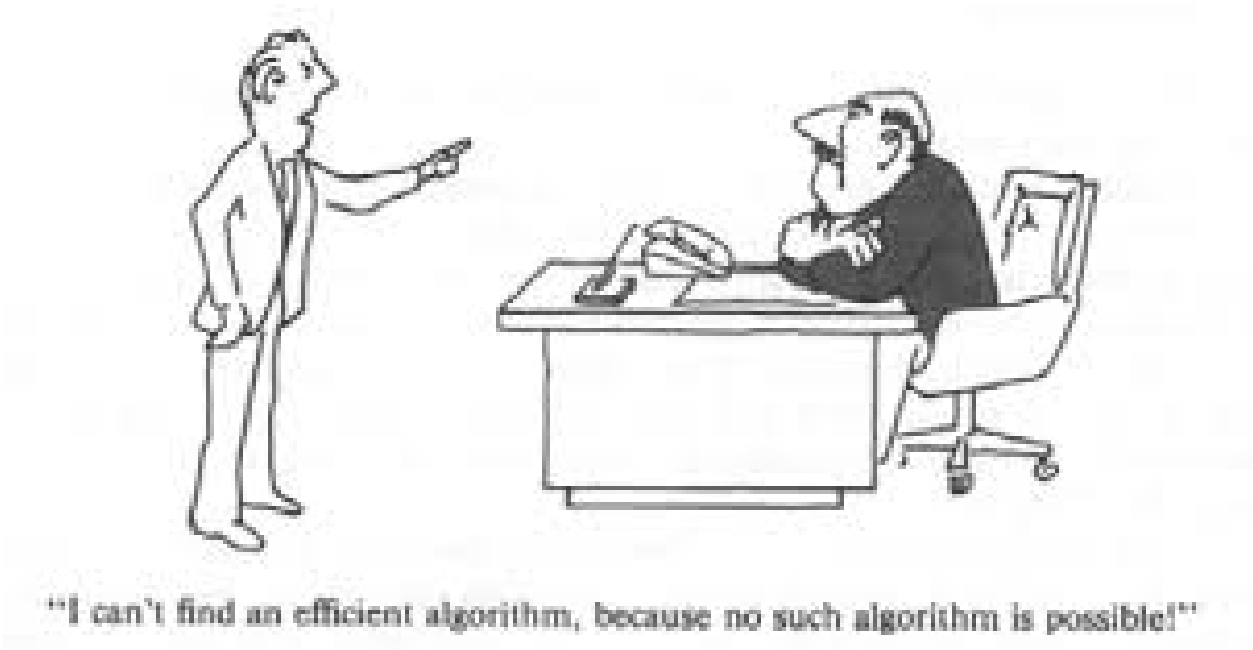
...you can take the looser's way



"I can't find an efficient algorithm, I guess I'm just too dumb."

Practical use of NP-completeness (NP-hardness)

... or, if you are extremely smart and strong in math ...



"I can't find an efficient algorithm, because no such algorithm is possible!"

Practical use of NP-completeness (NP-hardness)

... or, if you are just smart enough...

"I can't find an efficient algorithm, but neither can all these famous people."

From decision problems to enumeration problems

General question

"Does there exist?" $\rightarrow$ "How many exist?".

A search problem Π consists of:

- a set D_Π of finite objects (**instances**),
- a set (possibly empty) $S_\Pi[I]$ of finite objects (**solutions**), $\forall I_\Pi \in D_\Pi$.

An algorithm solves a search problem Π if for an instance $I_\Pi \in D_\Pi$ as input it outputs "no" if set $S_\Pi[I]$ is empty or outputs a solution $s \in S_\Pi[I]$.

#P and #P-completeness

Definition: #P is the class of counting problems associated with the decision problems in NP. More formally, a problem is in #P if there is a non-deterministic, polynomial time Turing machine that, for each instance I of the problem, has a number of accepting computations that is exactly equal to the number of distinct solutions for instance I.

A problem is #P-**complete** if it is in #P and it is #P-**hard**, i.e., any problem in #P can be reduced by Turing to it (or a #P-complete problem can be reduced to it). Obviously, #P = P $\Longrightarrow$ NP = P.

Examples of #P-complete problems:

- Given a matrix, output its permanent
- Given a graph, output the number of its perfect matchings
- Given a CNF, output the number of its satisfying assignments
- Given a graph, output the number of its vertex covers
- Given a context, output the number of its concepts

Exercises

1. What is the worst-case complexity of computing the inverse relation in case where the relation is represented a) by the set of pairs of the relation b) by the matrix of the relation?
2. What is the worst-case complexity of computing the product of relations in case where the relation is represented a) by the set of pairs of the relation b) by the matrix of the relation?
3. What is the worst-case complexity of computing the transitive closure of a relation in case where the relation is represented a) by the set of pairs of the relation b) by the matrix of the relation?
4. What is the worst-case complexity of checking properties from Definition 1.6 in case where the relation is represented a) by the set of pairs of the relation b) by the matrix of the relation?
5. What is the worst-case complexity of topological sorting? Compare with the complexity of sorting a subset of natural numbers.
6. Show that the transformability by Cook and by Turing define quasiorders on the set of generic problems.
7. Show that the following problem of computing dimension of a partial order is NP-complete:
   INSTANCE Partially ordered set $(P; \leq)$, natural number K.
   QUESTION Is K order dimension of $(P; \leq)$?
8. What is the delay of the algorithm computing elements of a relation on set A, $|A| = n$, which is the product of two relations given by binary matrices of size $n \times n$?

Literature on Algorithmic Complexity

- T.H. Cormen, C.E Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, MIT Press (2009).
- M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman (1979).
- L.G. Valiant, The complexity of enumeration and reliability problems, SIAM Journal on Computing (1979), vol. 8, no. 3, pp. 410-421.