Title
Comparing algorithmic complexity of recursive and inductive algorithms

Journal
Theoretical Computer Science, 317(1-3)

ISSN
0304-3975

Author
Burgin, Mark

Publication Date
2004-06-01

Peer reviewed

COMPARING ALGORITHMIC COMPLEXITY OF RECURSIVE AND INDUCTIVE ALGORITHMS

Mark Burgin

Department of Computer Science
University of California, Los Angeles
405 Hilgard Ave.
Los Angeles, CA 90095

You can make your model
more complex and more faithful to reality,
or you can make it simpler and easier to handle.

James Gleick "*Chaos*"

Abstract: The main goal of this paper is to compare recursive algorithms such as Turing machines with such super-recursive algorithms as inductive Turing machines. This comparison is made in a general setting of dual complexity measures such as Kolmogorov or algorithmic complexity. To make adequate comparison, we reconsider the standard axiomatic approach to complexity of algorithms. The new approach allows us to achieve a more adequate representation of static system complexity in the axiomatic context. It is demonstrated that for solving many problems inductive Turing machines have much lower complexity than Turing machines and other recursive algorithms. Thus, inductive Turing machines are not only more powerful, but also more efficient than Turing machines.

Key words: efficiency, complexity, dual complexity measure, Kolmogorov complexity, recursive algorithm, Turing machine, super-recursive algorithm, inductive Turing machine

1. Introduction

Today complexity has become an important and, at the same time, one of the most popular notions in science and society. It is a frequent word in present days' scientific literature, in various fields and with diverse meanings, appearing in some contexts as a precise concept, while being a vague idea in other texts. The reason is that people study and create more and more complex systems. This is especially true for such fields as information technology and software development.

As it is written in [34], recently Bill Gates, in a Microsoft internal memorandum, implicitly admitted of past complexity sins and introduced plans to redirect development efforts toward providing better systems. Paul Horn, the IBM vice president of research, confessed the complexity sins of the computer industry and also proposed an ambitious program for cleansing the sins [28].

The situation is reflected in the ironic Seventh Law of Computer Programming:

Program complexity grows until it exceeds the capabilities of the programmer who must maintain it.

To cope with such situations, we need a developed theory of complexity, which explains why and how complexity emerges and how to solve problems that involve very complex systems. This is especially true for computers, networks, and their software.

At the same time, there is no generally accepted, formalized, and unique definition of complexity. Complexity has proved to be an elusive concept. Different researchers in different fields are bringing new philosophical and theoretical tools to deal with complex phenomena in complex systems. “What is complexity?” is a basic question of Gell-Mann [22].

Here we use the following informal definition of system complexity.

Definition 1. The *complexity* of a system R is the amount of resources necessary for (used by) a process P that involves R.

Remark 1. Some think that the complexity of the system is independent of any process, unless the system includes the process. Others justly assume that the complexity of a problem is also a subjective matter. For instance, we can consider a problem related to some system R: to build R, to test R, to increase the power of R and so on. Two people having different models or views of the system, different

algorithms for dealing with such systems, and even different will to solve the problem in question will have different ideas of the complexity of solving this problem. Waxman [47] gives the following example. A problem with a car not starting might be very complex for a high-qualified mathematician, but not for the corner mechanic. On the other hand, solving a system of five linear equations with five variables will be simple for the mathematician and very complex for the corner mechanic. In other words, with respect to the process of repairing, the car is a complex system for the high-qualified mathematician, a simple system for the mechanic. At the same time, with respect to the process of solving, the system of five linear equations with five variables is a complex system for the mechanic, a simple system for the high-qualified mathematician.

There are different kinds of system involvement in a process.

P may be a process in the system *R*. For example, *R* is a computer, *P* is an electrical process in *R*, and the resource is energy.

P may be a process that is realized by the system *R*. For example, *R* is a computer, *P* is a computational process in *R*, and the resource is memory.

P may be a process controlled by the system *R*. For example, *R* is a program, *P* is a computational process controlled by *R*, and the resource is time.

P may be a process that builds the system *R*. For example, *R* is a software system, *P* is the process of its design, and the resource is programmers.

P may be a process that transforms *R*. *P* may be a process that utilizes *R*. *P* may be a process that models the system *R*. *P* may be a process that predicts behavior of the system *R*.

In cognitive processes complexity is closely related to information, representing specific kind of information measures.

Processes use different kinds of resources:

Natural resources consumed by a process *P*: time, space, information, energy/power, minerals, etc.

Social resources consumed by a process *P*: people involved, their time, efforts, expertise, knowledge, etc.

Artificial resources consumed by a process *P*: system time, system space, data, knowledge, memory, system units, system actions, etc.

If it is impossible to solve a problem with given resources, we assume that it has infinite complexity with respect to this resource. The halting problem, being restricted to recursive algorithms, is an example of a problem with infinite complexity since we know that it has no solution.

In general, complexity is a relative characteristic, which depends on considered processes and related resources. For instance, there are systems that are simple for usage but complex for study. There are computations that demand little memory (one resource) but take a lot of time (another resource) to finish.

Definition 1 implies that complexity is always complexity of doing something and as a result, must be attributed as an essential characteristic to both the system and the process being performed. Thus, it is also an essential characteristic of an algorithm that can be used as the basis of a process. As here we study algorithms, only measures of algorithm complexity are considered. However, it is possible to extend the constructions of such measures to complexity of arbitrary processes and through processes to arbitrary systems. Different processes may demand different complexity measures. At the same time, even one process or system may be characterized by several complexity measures.

All these peculiarities of complexity measures show that to measure complexity, we need many different measures. This corresponds to the real situation, in which researchers utilize a variety of such measures. It is useful to separate this variety into three categories: *static*, *dynamic*, and *processual* complexity measures.

We consider two classes of algorithms. Recursive algorithms are algorithms equivalent with respect to their computing power to Turing machines. Super-recursive algorithms can do more than Turing machines. The main goal of this paper is a comparison of recursive algorithms such as Turing machines with such super-recursive algorithms as inductive Turing machines. This comparison is made in a general setting of dual complexity measures such as Kolmogorov or algorithmic complexity. To make adequate comparison, we reconsider the standard axiomatic approach suggested by Blum [3] and developed further in [4, 6, 7, 11, 14, 26]. The new approach allows us to achieve a more adequate representation of static system complexity in the axiomatic context. This provides means for demonstrating that for solving many problems inductive Turing machines have much lower complexity than

Turing machines and other recursive algorithms. Thus, inductive Turing machines are not only more powerful, but also more efficient than Turing machines.

2. Direct Complexity Measures: An Axiomatic Approach

Static complexity measures of algorithms are functions of the form $\mathbf{c}$: $\mathbf{A} \to \boldsymbol{N}$ where $\mathbf{A} = \{ A_i; i \in \mathrm{I}\}$ is a class of algorithms (programs or automata/machines) and $\boldsymbol{N}$ is the set of all natural numbers. Such measures are direct as they estimate algorithms, programs or machines. We introduce several axioms to distinguish definite classes of complexity measures and to characterize their properties.

Very often, one algorithm A can be a part/component of another algorithm B. This relation between algorithms is denoted by $A \subseteq B$.

Compositional Axiom. If $A \subseteq B$, then $\mathbf{c}(A) \leq \mathbf{c}(B)$.

Let $\mathbf{B} = \{ B_j ; j \in \mathrm{J}\}$ be a class of algorithms.

Computational Axiom. The function $\mathbf{c}(A)$ is total and computable in $\mathbf{B}$.

Recomputational Axiom. For any number n, it is possible to compute all indices i such that $\mathbf{c}(A_i) = n$.

Reconstructibility Axiom. For any number n, it is possible to build all algorithms A from $\mathbf{A}$ for which $\mathbf{c}(A) = n$.

The difference between Reconstructibility Axiom and its weak version, Recomputational Axiom, is that it is not always possible to build an algorithm having its index. For instance, it is possible to enumerate all Turing machines by a standard procedure and get the sequence $l = \{T_1 , T_2 , \dots , T_n , \dots\}$. Then we define the new enumeration by the following rule. Taking the sequence l, we form two sequences, preserving the same order. We put all machines that always give the result in the first sequence and all other machines in the second sequence. Then a Turing machine T has the index $2i + 1$ if it occupies the place number i in the first sequence and has the index $2i$ if it occupies the place number i in the second sequence. For this enumeration, it is possible to build a recomputable complexity, which is not a reconstructible one.

Cofiniteness Axiom. The set $\mathbf{c}^{-1}(n)$ is finite for all numbers n from $\boldsymbol{N}$.

It is usually assumed that any finite set is recursively computable and even decidable. When a finite set X is given by a list, then this is true. However, this assumption is not valid in a general case when a finite set can be defined by a description. For instance, let us take the set X of all indices of those Turing machines that have the length of their description less than 1000 and that do not terminate on some input with the length less than 1000. This set is finite, but it is not recursively computable (enumerable).

Let X^* be the set of all words in the alphabet X and $l(x)$denotes the length of a word x.

Definition 2. A partial function f: $X^* \to \boldsymbol{N}^+$ tends to infinity (we denote it by $f(x) \to \infty$, or $f(x) \to \infty$ when $x \to \infty$) if for any number m from $\boldsymbol{N}^+$ there is a number k such that $f(x) > m$ when $l(x) > k$.

Definition 3. A partial function f: $X^* \to X^*$ tends to infinity (we denote it by $f(x) \to \infty$) if the partial function $l(f(x))$ tends to infinity.

Lemma 1. If a function $\mathbf{c}(A)$ satisfies the Cofiniteness Axiom and the set $\mathbf{A}$ is infinite, then this function tends to infinity.

Definition 4. a) A function $\mathbf{Sc}$: $\mathbf{A} \to \boldsymbol{N}$ is called an *axiomatic static complexity* of algorithms from $\mathbf{A}$ if it satisfies the Compositional Axiom.

b) An axiomatic static complexity is called reconstructible (computable, recomputable, weakly reconstructible, cofinite) if it satisfies the Reconstructibility (Computational, Recomputational, Weak Reconstructibility or Cofiniteness, respectively) Axiom.

Remark 2. Such approach to complexity of algorithms/programs reflects the condition that static complexity depends on structural features of algorithms/programs.

Remark 3. When all algorithms from $\mathbf{A}$ have indices (are enumerated by natural numbers), it is possible to consider a function $\mathbf{Sc}$: I $\to \boldsymbol{N}$ (a function $\mathbf{Sc}$: $\boldsymbol{N} \to \boldsymbol{N}$) instead of the function $\mathbf{Sc}$: $\mathbf{A} \to \boldsymbol{N}$. If we have some complexity function $\mathbf{c}$: $\mathbf{A} \to \boldsymbol{N}$ and an enumeration of algorithms $\mathbf{q}$: $\mathbf{A} \to \boldsymbol{N}$, then we can build the function $\mathbf{Sc}$: $\boldsymbol{N} \to \boldsymbol{N}$ by taking $\mathbf{Sc} = \mathbf{c} \circ \mathbf{q}^{-1}$. We also call this function an *axiomatic static complexity* of algorithms from $\mathbf{A}$.

Remark 4. A function **c**(A) can satisfy the Cofiniteness Axiom, but still can be non-computable (as well as **Sc**(i)) even in such powerful class as the class **R** of recursive algorithms, in particular, the class of all Turing machines. However, there exist such functions **c**(A) that the corresponding function **Sc**(i) is inductively computable [5, 11].

Remark 5. It is possible to use Definition 4 for determining complexity of any system.

Not all of axiomatic static complexity measures in the sense of [4, 6, 7, 14] and not all sizes of machines introduced in [3] are axiomatic static complexities. However, differences between the new definition and the old ones are natural because as some experts argue (cf., for example, [19]), the traditional axiomatic approach is too general to describe only complexity, corresponding measures include many other functions.

At the same time, all existing examples of static complexity measures satisfy the Compositional Axiom and thus, represent axiomatic static complexities. Let us consider some of them.

Example 1. Let **A** consists of algorithms generated by a Turing machine W with two input tapes. One tape is used for data, while the content of the second tape is considered as a program for computation. Each program for the machine W is an algorithm from **A**. Then the length $l(p)$ of this program p is a computable, reconstructible, cofinite static complexity for algorithms from **A**.

Example 2. Let **A** consists of all programs written in some programming language (e.g., Java or FORTRAN). Then the length $l(p)$ of a program p as number of characters or as number of words in p is a very popular static complexity, which is computable, reconstructible, and cofinite.

Both these measures satisfy even a stronger form of the Compositional Axiom.

Additive Compositional Axiom. If $A, C \subseteq B$, then **c**(B) ≥ **c**(A) + **c**(C).

Remark 6. Additive Compositional Axiom is not necessarily satisfied when algorithms/programs allow nesting. The following example demonstrates how nesting violates this axiom.

Example 3. Let us consider the following program B:

```
B()
{   sol = 1;
```

```
    A: if sol > 0 then {
            C: sol = sol + sol;
    }
}
```

Thus, we have the program B and its parts A and C. Let us consider such static complexity as the length l of a program that is measured in the number of symbols in the program. Here B has 23 letters (36 non-whitespace characters), A has 19 letters (27 nonwhitespace characters), and C has 9 letters (12 non-whitespace characters), and $23 < 19 + 9$ ($36 < 27 + 12$). Consequently, $l(\mathrm{B}) < l(\mathrm{A}) + l(\mathrm{C})$. This violates Additive Compositional Axiom.

However, this axiom can be true with additional conditions on programs A and C.

Restricted Compositional Axiom. If A, $C \subseteq B$ and A and C are disjoint, then $\mathbf{c}(B) \geq \mathbf{c}(A) + \mathbf{c}(C)$.

Strong Compositional Axiom. If A, $C \subseteq B$, then $\mathbf{c}(B) > \max\{\mathbf{c}(A), \mathbf{c}(C)\}$.

Software metrics give different examples of axiomatic static measures of complexity.

Example 4. When the length of a source line of code is bounded (and this is true for all programming languages as compilers demand this restriction), then the software metric "number of source lines of code" (SLOC) (cf. [49]) is a finite computable reconstructible static complexity.

Example 5. Describing a program formally as a sequence of operators and operands, we see that the length of program N(P) [25] is also a static complexity measure, namely, the length l(P) of a program. For a programming language in which the numbers n_1 of the unique operators and n_2 of the unique operands are finite, N(P) is a cofinite reconstructible static complexity. However, some languages (at least, potentially) operate with infinite alphabets of operands, for example, with all real numbers. There are also theoretical models in which there are infinitely many unique operators. In such cases, N(P) is not a finite complexity measure. If the sets of operands and operators are computable, then this measure is also computable.

Example 6. When it is defined, the volume V(P) of the program P [25] is always a cofinite reconstructible static complexity.

Example 7. Representing a program P formally as a structure of operators and operands, we can demonstrate that the cyclomatic number C(P), i.e. the number of

cycles in P [41], is a computable reconstructible static complexity. However, this measure does not satisfy Cofiniteness Axiom as, for instance, it is possible to build infinitely many programs with only one cycle.

Example 8. Representing a program P formally as a structure of operators and operands, we can demonstrate that $N(P) + V(P)$ is a static direct complexity measure.

Example 9. Representing a program formally as a structure of operators and operands, we can demonstrate that $N(P) + C(P)$ is a static direct complexity measure.

It is necessary to remark that some nice properties of the traditional definition of axiomatic static complexity measures are lost in the suggested approach. For example, axiomatic static complexities are not closed with respect to enumerations. It means that when we enumerate algorithms and take the induced function on these numbers, the new function does not necessarily satisfy the Composition Axiom. The reason is that static complexity reflects structural properties of algorithms/programs, while enumerations, in general do not reflect structural features of algorithms/programs.

Example 10. Let the class $\mathbf{A} = \mathbf{T}$ consists of all Turing machines. Each Turing machine T has a description/coding $\mathbf{c}(T)$. Then the length $l(\mathbf{c}(T))$ of this description $\mathbf{c}(T)$ is a static complexity measure for Turing machines.

Definition 5. A class $\mathbf{A} = \{ A_i; i \in I\}$ of algorithms is called constructible if there is a set **B** of primitive algorithms and all algorithms from **A** are built by combining algorithms from **B**.

Definition 6. The set **B** is called a base of **A**.

For instance, all programs in procedural programming languages, such as ALGOL, FORTRAN, COBOL, C^{++}, and Java, are built from a system of operators or instructions.

It is usually assumed that the base **B** is finite and when a finite number of elements from **B** are used, it is possible to construct only a finite number of algorithms from **A**.

Proposition 1. For a constructible set **A** of algorithms with a finite base, Compositional Axiom implies Cofiniteness Axiom.

We prove this statement by induction.

Corollary 1. Any computable reconstructible axiomatic static complexity on a constructible set **A** of algorithms is an axiomatic static complexity measure in the sense of [4, 6, 8].

Corollary 2. Any computable reconstructible cofinite axiomatic static complexity on the set **R** of recursive algorithms is a size of machines in the sense of [3].

Let us assume that **A** = **R**.

Lemma 2. If $g(x)$ and $h(x)$ are total computable functions, $g^{-1}(x)$ and $h^{-1}(x)$ are recursively computable multifunctions (relations), and for any number n both sets $g^{-1}(n)$ and $h^{-1}(n)$ are finite, then there is a computable increasing function $f(x)$ such that it tends to infinity and $f(g(x)) \le h(x)$ for almost all x.

Proof. We informally describe an algorithm for computation of such a function $f(x)$. Then by the Church-Turing Thesis (cf., for example, [42]), $f(x)$ is a recursively computable function.

An algorithm for computation of $f(x)$:

1. Compute $h(\varepsilon)$ where ε is an empty word. It is possible to do this because $h(x)$ is a recursively computable function.

2. If $h(\varepsilon) = r$, find all elements $x_1, x_2, \dots, x_n$ for which $h(x_i) \le r$ and choose from these values $h(x_i)$ the least number $p = h(x_j)$ for some x_j. It is possible to do this because $h^{-1}(x)$ is a recursively computable multifunction and for any n, the set $h^{-1}(n)$ is finite.

3. Find the largest element x_j for which $p = h(x_j)$. It is possible to do this because $h(x)$ is a recursively computable function.

4. Find the largest value (say t) of all values $g(x)$ with $x \le x_j$. It is possible to do this because $g(x)$ is a recursively computable function.

5. Define $f(k) = p$ for all $k \le t$.

6. Find the least number $q > p$ such that $q = h(x_j)$ for some x_j. Then go to the step 3 with q instead of p, continuing this process with u instead of t in step 4 and the condition $t < k \le u$ instead of the condition $k \le t$ and with q instead of p in step 5.

In such a way, we build the necessary increasing function $f(x)$. By Lemma1, both functions $g(x)$ and $h(x)$ tend to infinity. So, by its construction, the function $f(x)$ also tends to infinity.

Corollary 3. If $g(x)$ and $h(x)$ are total recursively computable functions, $g^{-1}(x)$ and $h^{-1}(x)$ are computable multifunctions (relations) and for any number n, both sets $g^{-1}(n)$ and $h^{-1}(n)$ are finite, then there is a recursively computable increasing function $f(x)$ such that $f(g(x)) \leq h(x)$ and $f(h(x)) \leq g(x)$ for almost all x.

Lemma 2 and Corollary 3 imply the following result.

Proposition 2. For any axiomatic static complexities $\mathbf{c}(A)$ and $\mathbf{b}(A)$ that satisfy the Computational, Recomputational, and Cofinite axioms, there is a recursively computable total increasing function $f(x)$ such that $f(\mathbf{c}(A)) \leq \mathbf{b}(A)$ and $f(\mathbf{b}(A)) \leq \mathbf{c}(A)$ for almost all A from **R**.

Lemma 3. If $g(x)$ and $h(x)$ are total computable functions, $g^{-1}(x)$ and $h^{-1}(x)$ are recursively computable multifunctions (relations) and for any number n, both sets $g^{-1}(n)$ and $h^{-1}(n)$ are finite, then there is a recursively computable increasing function $r(x)$ such that $r(g(x)) > h(x)$ for all x.

Proof. Let us take the function $r(n) = \max \{ h(x) ; \exists z$ such that $g(z) = n$ and for all y, the equality $g(y) = g(z)$ implies $y < z$, and $x \leq z \} + 1$. This function $r(n)$ is an increasing function. In addition, using conditions from the lemma, we can build an algorithm of computation for this function $r(n)$.

At first, using computability and finiteness of $g^{-1}(x)$, we find all x_j such that $g(x_j) = n$. Then we take the largest of them z. Then we can compute the function $r(n) = \max \{ h(x) ; x \leq z \} + 1$.

By the Church-Turing Thesis (cf., for example, [42]), $r(x)$ is a computable function and by its construction $r(g(x)) > h(x)$ for all x.

Corollary 4. If $g(x)$ and $h(x)$ are total recursively computable functions, $g^{-1}(x)$ and $h^{-1}(x)$ are recursively computable multifunctions (relations) and for any n, both sets $g^{-1}(n)$ and $h^{-1}(n)$ are finite, then there is a recursively computable strictly increasing function $r(x)$ such that $r(g(x)) > h(x)$ and $r(h(x)) > g(x)$ for all x.

Lemma 3 and Corollary 4 imply the following result.

Proposition 3. For any axiomatic static complexities $\mathbf{c}(A)$ and $\mathbf{b}(A)$ that satisfy the Computational, Recomputational, and Cofinite axioms, there is a recursively computable total strictly increasing function $r(x)$ such that $r(\mathbf{c}(A)) > \mathbf{b}(A)$ and $r(\mathbf{b}(A)) > \mathbf{c}(A)$ for all A from **A**.

Proposition 3 implies corresponding results from [3, 6, 26].

3. Dual Complexity Measures: An Axiomatic Approach

While direct complexity measures characterize algorithms/machines/programs, dual complexity measures are related to problems solved by these algorithms/machines/programs and to results of their functioning. As a rule, the problem that is considered for algorithms is building (computing) some word or making a decision whether a given element belongs to a given set.

The complexity of a problem often differs from the complexity of its solution. Simple problems, i.e., problems that have short descriptions, may have only complex solutions, i.e., they demand long proofs or a lot of computations. Moreover, as proved by Juedes and Lutz [30] many important problems that have hard solutions (those that are P-complete for ESPACE) have low problem complexity, that is, their Kolmogorov complexity or algorithmic information is rather low.

There was an attempt to build a universal dual complexity measure, which does not depend on a specific class of algorithms. However, this goal has not been achieved. One reason was that it turned out that the original definition was not sufficient for solving some mathematical and practical problems. For example, such universal measure was not appropriate for formalization of the concept of randomness and for the development of algorithmic probability theory and information theory. The second reason for impossibility to achieve this goal (and for necessity of constructing relative dual measures) was the discovery of super-recursive algorithms. Before it happened, all believed that Turing machines or other universal models of recursive algorithms give an absolute class for algorithms and computation. This discovery changed the existing situation. The third reason for impossibility to build a universal dual complexity measure was that actually computer scientists have already used several distinct dual measures. As a result, the universal approach was discarded and dual measures have been introduced and studied for some specific classes of algorithms. Later an axiomatic approach to dual complexity measures has been elaborated.

Dual complexity measures are properties of objects that are constructed and processed by algorithms, as well as of problems that are solved by these algorithms. On the other hand, it is possible to interpret these measures as properties of classes of algorithms. Here we consider only static dual complexity measures for algorithms.

Let $\mathbf{A} = \{ A_i ; i \in \mathrm{I}\}$ be a class of algorithms, A be an algorithm that works with elements from I as inputs and $\mathbf{Sc}: \mathrm{I} \to \boldsymbol{N}$ be a static complexity measure of algorithms from a class **A**. Elements of I are usually interpreted as programs for the algorithm A. In addition, developing the theory of Kolmogorov complexity, researchers assume for simplicity that I consists of natural numbers in a form of binary sequences. These numbers can be indices enumerating algorithms from **A** or codes of these algorithms (cf., for example, [27]). In what follows, we consider only computable recomputable cofinite complexities.

Definition 7. Given a complexity measure $\mathbf{Sc}: \mathrm{I} \to N$, an algorithm A from **A**, and that the codomain (set of all outputs) Y is a subset of the domain (set of all inputs) X of algorithms from **A**, the dual to **Sc** with respect to A complexity measure is denoted by $\mathbf{Sc}_A^{\mathrm{o}}: Y \to N$ and is defined as

$$\mathbf{Sc}_A^{\mathrm{o}}(x) = \min \{ \mathbf{Sc}(p); p \in \mathrm{I} \text{ and } A(p) = x\}.$$

Naturally when there is no such p that $A(p) = x$, the value of $\mathbf{Sc}_A^{\mathrm{o}}$ at x is undefined.

When $\mathbf{Sc}(x)$ measures information in the word/text x, the dual complexity measure $\mathbf{Sc}_A^{\mathrm{o}}(x)$ estimates minimal information necessary to compute/build x by the algorithm A.

If $\mathrm{L}^{\mathrm{o}}_A(x)$ is the dual to the length $l(p)$ of program/algorithm description p complexity measure, i.e., $\mathbf{Sc}(p) = l(p)$, with respect to a algorithm A, then

$$\mathrm{L}^{\mathrm{o}}_A(x) = \min \{ l(p); p \in \mathrm{I} \text{ and } A(p) = x\}.$$

Let M and T be some algorithms.

Proposition 4. If $M(x) > T(x)$ for almost all x and $M(x)$ is an increasing function, then $\mathrm{L}^{\mathrm{o}}_T(x) \geq \mathrm{L}^{\mathrm{o}}_M(x)$ for almost all x for which both $\mathrm{L}^{\mathrm{o}}_M(x)$ and $\mathrm{L}^{\mathrm{o}}_T(x)$ are defined.

The most interesting case is when A is a universal algorithm V for the class **A**. Let $\mathbf{c}: \mathbf{A} \to X^*$ be some coding of algorithms from **A**.

Definition 8. An algorithm W is called *universal* for the class **A** if for any $A \in \mathbf{A}$ and any x given the pair $p = (\mathbf{c}(A), x)$ as its input, the result of W is equal to the result of A applied to x.

Examples of universal algorithms are a universal Turing machine and a universal inductive Turing machine [11].

The dual complexity measure that corresponds to a universal algorithm gives an invariant characteristic of the whole class **A**.

Definition 9. Given complexity measure **Sc**: $I \to N$, an algorithm A from **A**, and that the codomain (set of all outputs) Y is a subset of the domain (set of all inputs) X of algorithms from **A**, the dual to **Sc** with respect to the class **A** is denoted by $\mathbf{Sc}^{o}_{\mathbf{A}}$: $Y \to N$ and is defined as

$$\mathbf{Sc}^{o}_{\mathbf{A}}(x) = \min \{ \mathbf{Sc}(p); p \in I \text{ and } W(p) = x\}.$$

Naturally when there is no such p that $W(p) = x$, the value of $\mathbf{Sc}_{\mathbf{A}}^{o}$ at x is undefined.

Because algorithm W is universal for the class **A**, this condition is equivalent to the condition that there is no such algorithm A from **A** and such p that $A(p) = x$.

In other words, $\mathbf{Sc}^{o}_{\mathbf{A}}(x) = \mathbf{Sc}_{V}^{o}(x)$ for a universal algorithm W for the class **A**. However, it is possible that **A** has several universal algorithms. In such a case, the function of $\mathbf{Sc}^{o}_{\mathbf{A}}(x)$ is not defined uniquely. Nevertheless, as Theorem 3 shows, the definition of $\mathbf{Sc}^{o}_{\mathbf{A}}(x)$ is invariant with respect to certain transformations.

When $\mathbf{Sc}(x)$ measures information in the word/text x, the dual complexity measure $\mathbf{Sc}^{o}_{\mathbf{A}}(x)$ estimates minimal information necessary to compute/build x by algorithms from the class **A**.

Proposition 5. For any algorithm A, any axiomatic static complexities $\mathbf{Sc}(x)$ and $\mathbf{Sb}(x)$, and an increasing function $f(z)$, the condition $f(\mathbf{Sc}(p)) \leq \mathbf{Sb}(p)$ for almost all p ($f(\mathbf{Sb}(p)) > \mathbf{Sc}(p)$ for all p) implies the condition $f(\mathbf{Sc}^{o}_{A}(x)) \leq \mathbf{Sb}^{o}_{A}(x)$ for almost all x ($f(\mathbf{Sb}^{o}_{A}(x)) > \mathbf{Sc}^{o}_{A}(x)$ for all x).

Indeed, if $\mathbf{Sb}^{o}_{A}(x) = s$, then there is $q \in I$ such that $\mathbf{Sb}(q) = s$ and $A(q) = x$. By assumption, for almost all such q, we have $f(\mathbf{Sc}(q)) \leq \mathbf{Sb}(q)$. By the definition, if $\mathbf{Sc}^{o}_{A}(x) = \mathbf{Sc}(r)$ and $A(r) = x$, then $\mathbf{Sc}(r) \leq \mathbf{Sc}(p)$ for all $p \in I$ such that $A(p) = x$. In particular, we have $\mathbf{Sc}(r) \leq \mathbf{Sc}(q)$. Consequently, $f(\mathbf{Sc}^{o}_{A}(x)) = f(\mathbf{Sc}(r)) \leq f(\mathbf{Sc}(q)) \leq \mathbf{Sb}(q) = \mathbf{Sb}^{o}_{A}(x)$ because $f(z)$ is an increasing function.

Inequality $f(\mathbf{Sb}^{o}_{A}(x)) > \mathbf{Sc}^{o}_{A}(x)$ is proved in a similar way.

Proposition 5 is proved.

Remark 7. If we can choose different algorithms from **A** to build the element x, the dual measure with respect to the class **A** is defined in a different way.

Definition 10. Given complexity measure **Sc**: I $\to N$, an algorithm A from **A**, and that the codomain (set of all outputs) Y is a subset of the domain (set of all inputs) X of algorithms from **A**, the dual to **Sc** with respect to the class **A** with selection is denoted by $\mathbf{Sc}^{o}_{\mathbf{A}}$: $Y \to N$ and is defined as

$$\mathbf{Sc}^{o}_{\mathbf{SA}}(x) = \min \{ \mathbf{Sc}(p); p \in \mathrm{I}, A \in \mathbf{A}, \text{ and } A(p) = x\}.$$

Naturally when there is no such algorithm A from **A** and such p that $A(p) = x$, the value of $\mathbf{Sc}_A^{o}$ at x is undefined.

Lemma 4. $\mathbf{Sc}^{o}_{\mathbf{SA}}(x) \leq \mathbf{Sc}^{o}_{\mathbf{A}}(x) \preccurlyeq \mathbf{Sc}(x)$.

In general, both functions $\mathbf{Sc}^{o}_{\mathbf{SA}}(x)$ and $\mathbf{Sc}^{o}_{\mathbf{A}}(x)$ are defined for all elements x from the domain $\cup_{A \in \mathbf{A}} \mathrm{C}(A)$. In particular, when all algorithms from **A** have a common codomain X, then both functions $\mathbf{Sc}^{o}_{\mathbf{SA}}(x)$ and $\mathbf{Sc}^{o}_{\mathbf{A}}(x)$ are defined for all elements x from X. For example, when **A** is the set of all partial recursive functions, both functions $\mathbf{Sc}^{o}_{\mathbf{SA}}(x)$ and $\mathbf{Sc}^{o}_{\mathbf{A}}(x)$ are defined for all natural numbers. This is a consequence of the following more general result.

Let the class **A** contains an identity algorithm E that computes the function $e(x) = x$.

Theorem 1. $\mathbf{Sc}^{o}_{\mathbf{A}}(x)$ is a total function on $\boldsymbol{N}^{+}$ (on the set of all words in some alphabet).

Dual complexity measures are usually interpreted as complexity of problem solution with the help of algorithms from **A**. More exactly, the problem under consideration is construction or computation of a word x by means of algorithms from **A**.

The first developed form of dual complexity measures was the, so-called, *Kolmogorov* or *constructive* or *algorithmic complexity*.

Traditionally the theory of Kolmogorov complexity has been developed top down: from larger classes to smaller classes of algorithms that were more relevant to computational problems. At first, as the history of the subject tells us, Kolmogorov complexity C(x) was defined and studied independently for the class of all recursive algorithms by three mathematicians: Solomonoff [45], Kolmogorov [32], and Chaitin [16]. Later another name *algorithmic complexity* has been also used for this measure.

The original Kolmogorov complexity of a word x is taken equal to the size of the shortest program (in number of symbols) for a universal Turing machine U that without additional data, computes the string and terminates. To formalize this, we define Kolmogorov complexity for a class **R** of recursive algorithms such that **R** has a universal algorithm. For example, in the class of all Turing machines, a universal Turing machine is a universal algorithm.

Definition 11. The *Kolmogorov complexity* $C(x)$ of an object/word x is defined as

$$C(x) = \min \{ l(p);\ U(p) = x\}$$

where $l(p)$ is the length of the word p and U is a universal algorithm in the class **R**.

This measure is called *absolute* Kolmogorov complexity because Kolmogorov complexity has another form, which is called relative.

Definition 12. The *Kolmogorov complexity* $C(x \mid y)$ of an object/word x *relative* to a given word y is defined as

$$C(x \mid y) = \min \{l(p);\ U(p, y) = x \}$$

where $l(p)$ is the length of the word p and U is a universal algorithm in the class **R**.

Kolmogorov complexity $C(x)$ of a word x in some sources is denoted by $K(x)$, while $C(x \mid y)$ is denoted by $K(x \mid y)$.

Absolute Kolmogorov complexity is a particular case of relative Kolmogorov complexity, namely:

$$K(x) = K(x \mid \Lambda) = \min \{l(p);\ U(p,\Lambda) = x \}$$

The aim of introducing Kolmogorov complexity was to ground probability theory and information theory, creating the new approach based on algorithms. This goal was achieved. The new theories became very popular, although they did not substitute either the classical probability theory, which was grounded before by the same Kolmogorov [31] on the base of measure theory, or Shannon's information theory.

However, an attempt to define in this setting an appropriate concept of randomness was unsuccessful. It turned out that the original definition of Kolmogorov complexity was not relevant for that goal. To get a correct definition of a random infinite sequence, it was necessary to restrict the class of utilized algorithms. That is why Kolmogorov complexity under different names was defined and studied for

various classes of subrecursive algorithms. For example, researchers discussed different reasons for restricting the power of the device used for computation when the minimal complexity is estimated.

When Kolmogorov complexity is defined for the class of Turing machines that compute symbols of a word x, we obtain uniform complexity $KR(x)$ studied by Loveland [39].

When Kolmogorov complexity is defined for the class of prefix functions, we obtain prefix complexity $K(x)$ studied by Gasc [20], Levin [36], and Chaitin [17].

When Kolmogorov complexity is defined for the class of monotonous Turing machines, we obtain monotone complexity $Km(x)$ studied by Levin [35].

When Kolmogorov complexity is defined for the class of Turing machines that have some extra initial information, we obtain conditional Kolmogorov complexity $CD(x)$ studied by Sipser [44].

Let $t(n)$ and $s(n)$ be some functions of natural number variables.

When Kolmogorov complexity is defined for the class of recursive automata that perform computations with time bounded by some function of a natural variable $t(n)$, we obtain time-bounded Kolmogorov complexity $C^t(x)$ studied by Kolmogorov [32] and Barzdin [2].

When Kolmogorov complexity is defined for the class of recursive automata that perform computations with space (i.e., the number of used tape cells) bounded by some functions of a natural variable $s(n)$, we obtain space-bounded Kolmogorov complexity $C^s(x)$ studied by Hartmanis and Hopcroft [26].

When Kolmogorov complexity is defined for the class of multitape Turing machines that perform computations with time bounded by some function $t(n)$ and space bounded by some function $s(n)$, we obtain resource-bounded Kolmogorov complexity $C^{t,s}(x)$ studied by Daley [16].

All of these kinds of complexity are dual complexity measures. The generalized Kolmogorov complexity introduced and studied in [4, 6, 7] gives a general setting for all of them.

However besides different kinds of the generalized Kolmogorov complexity, there are other dual complexity measures. As an example of another kind of a dual

complexity measure, we can take Boolean circuit complexity, which is also a nonuniform complexity measure [1].

There are two direct and two dual complexity measures for such automata as Boolean circuits.

Definition 13. The *cost* or *size* $c(A)$ of a Boolean circuit A is the number of gates it has.

This is a direct static complexity measure of Boolean circuits.

Let f be a Boolean function.

Definition 14. The Boolean *cost* $c(f)$ of f is the size of the smallest circuit computing f:

$$c(f) = \min \{ c(A) ; A \text{ defines the function equal to } f \}$$

This is a dual complexity measure.

Definition 15. The *depth* of a Boolean circuit is the length of the longest path in the graph of this circuit.

This is a direct static reconstructible complexity measure of Boolean circuits.

Definition 16. The Boolean depth $d(f)$ of f is the depth of the minimal depth circuit computing f:

$$d(f) = \min \{ d(A) ; A \text{ defines the function equal to } f \}$$

This is a dual complexity measure. Thus, we see that not all dual complexity measures are the Kolmogorov complexity or some its kind. There are other examples of dual complexity measures.

Due to its applications to problems of cryptography and network security, communication complexity has become one of the most popular types of complexity measures (cf. [29, 33]). Usually communication complexity is considered for the following situation. Two computers (persons) C_1 and C_2 are working together and solving the same problem. The problem taken for this purpose is computation of some finite function f: $X_1 \times X_2 \to Y$. As a rule, f is a Boolean function with m variables. At the beginning of the process, the input from X_1 is given to C_1 and the input from X_2 is given to C_2.

These computations, which include communication, are performed by (according to) algorithms P_i that are called communication protocols and describe a distributed computational processes of two computers C_1 and C_2 . The goal is for one of the

computers to compute $f(x_1, x_2)$ with the least amount of communication between them. In contrast to computational complexity, here we are not concerned about the number of computational steps or the size of the computer memory used. Communication complexity tries to quantify the amount of communication required for distributed computation.

It is supposed that both computers have unlimited computational resources. As a result, they can, for example, always succeed by having C_1 send its whole n-bit string to C_2, allowing C_2 to compute the function, but we are interested in finding better ways of calculating f with less than n bits of communication.

Definition 17. The *communication complexity* $\mathbf{cc}(P)$ of a communication protocol P is defined as the length of communicated word or, in other words, the maximal number of bits exchanged during the computational processes defined by P_i for all pairs of inputs. Inputs are taken from some finite sets X_1 and X_2 .

This is a direct static complexity measure

Definition 18. The *communication complexity* $\mathbf{cc}(f)$ of a function/problem f is defined as:

$$\mathbf{cc}(f) = \min \{ \mathbf{cc}(P) ; P \text{ computes the function } f \}$$

It is possible to represent any finite function by a table and then to represent this table as a word. In this context, the communication complexity $\mathbf{cc}(f)$ is a dual complexity measure on the set **A** of all protocols.

There are also other approaches leading to dual complexity measures. For example, Gell-Mann [21] introduced the concept of crude complexity of a system. It is possible to find many other examples of direct and dual complexity measures in [6, 10 - 14].

Let $\mathbf{Sc_A}^{o}(x)$ and $\mathbf{Sc_B}^{o}(x)$ be dual to **Sc** complexity measures with respect to classes **A** and **B**, respectively. If $\mathbf{A} \subseteq \mathbf{B}$, then any algorithm universal for **B** is also universal for **A**. We assume that such an algorithm is taken for building dual complexity measures with respect to **A** and **B**. This implies the following results.

Theorem 2. If $\mathbf{A} \subseteq \mathbf{B}$ and $\mathbf{Sc_A}^{o}(x)$ is defined for x, then $\mathbf{Sc_B}^{o}(x)$ is defined for x and $\mathbf{Sc_B}^{o}(x) \leq \mathbf{Sc_A}^{o}(x)$.

Corollary 5. If $\mathbf{A} \subseteq \mathbf{B}$ and $\mathbf{Sc}_{\mathbf{A}}^{o}(x)$ is defined for all x, then $\mathbf{Sc}_{\mathbf{B}}^{o}(x)$ is defined for all x and $\mathbf{Sc}_{\mathbf{B}}^{o}(x) \leq \mathbf{Sc}_{\mathbf{A}}^{o}(x)$ for all x.

Dual complexity measures with respect to the class **A**, i.e., determined by a universal algorithm, have invariance properties, defining minimal resources that are necessary in **A** to build/compute objects from Y. The set Y contains such objects that can be computed by algorithms from **A**.

Let **H** and **G** be two sets of functions.

Definition 19. A function $f(n)$ is called (asymptotically) **H**-optimal in **H** if there is such $h \in \mathbf{H}$ that $f(n) \leq h(g(n))$ for any $g \in \mathbf{G}$ and (almost) all $n \in \mathbf{N}$.

If there is such $h \in \mathbf{H}$ that $f(n) \leq h(g(n))$ for almost all $n \in \mathbf{N}$, we denote this relation by $f(x) \preccurlyeq_{\mathbf{H}} g(x)$. In the case, when **H** consists of such functions that add some constant to the argument, for example, $f(n) = g(n) + c$, we write simply $f(x) \preccurlyeq g(x)$ or $g(x) \succcurlyeq f(x)$. This relation is basic for the theory of Kolmogorov complexity [38].

Lemma 5. Relations $g(x) \succcurlyeq f(x)$ and $f(x) \preccurlyeq g(x)$ mean that there is a constant number c such that $f(x) \leq g(x) + c$ for all x.

Let $\mathbf{H} = \mathbf{H}(h) = \{ h_k(n) = h(h(n)+k), \ k \in \mathbf{N} \}$ and **A** be a class of algorithms with a universal algorithm U.

Theorem 3 [4]**.** For any axiomatic static complexity measure $\mathbf{Sc}(p)$ on **A** and for some recursively computable function $h(n)$, there is a $\mathbf{H}(h)$-optimal dual measure $\mathbf{Sc}^{o}(x)$.

Proof. Let A be some algorithm from the class **A**. At first, we consider dual complexity measures $L^{o}_{A}(x)$ and $L^{o}(x)$ dual to the length $l(p)$ of program/algorithm description p with respect to A and to general class **A** of algorithms that work with words or natural numbers, respectively. These measures are defined by the formulas: $L^{o}_{A}(x) = \min \{ l(p); p \in \mathrm{I} \text{ and } A(p) = x\}$ and $L^{o}(x) = \min \{ l(p); p \in \mathrm{I} \text{ and } U(p) = x\}$ where U is a universal algorithm in **A**.

If $L^o_A(x) = l(p)$ for some x and $A(p) = x$, then $U(\mathbf{c}(A), p)) = x$. By the definition, $l((\mathbf{c}(A), p)) = l(p) + k_A$ where k_A is some natural number. Then by the definition of $L^o(x)$, we have $L^o(x) \leq l((\mathbf{c}(A), p)) = l(p) + k_A = L^o_A(x) + k_A$ for any x.

By Proposition 3, we have $\mathbf{Sc}^o(x) \leq h(L^o(x)) \leq h(L^o_A(x) + k_A)$ as f is an increasing function. At the same time, $L^o_A(x) \leq h(\mathbf{Sc}^o_A(x))$. Consequently, $\mathbf{Sc}^o(x) \leq h(h(\mathbf{Sc}^o_A(x)) + k_A)$. This inequality means that $\mathbf{Sc}^o_A(x)$ is a $\mathbf{H}(h)$-optimal dual measure.

Theorem 3 is proved.

Corollary 6 [16, 32]**.** Algorithmic complexity $C(x)$ is an optimal dual complexity measure.

The result of Theorem 3 spares a researcher and a student:

1) to prove optimality for different versions of Kolmogorov complexity;

2) to prove optimality for other specific dual complexity measures.

Optimality for Kolmogorov complexity and its versions is additive. However, there are other kinds of optimality, which depend on the measure $\mathbf{Sc}^o(x)$. It is possible to find an example of such measures in [39]. In this book, instead of the length $l(x)$ of the word x representing some number n, this number n is taken as a direct static measure $\mathbf{Sc}(x)$ of x, i.e., $\mathbf{Sc}(n) = n$. As a result, for the corresponding dual complexity measure $\mathbf{Sc}^o(x)$, we have a different type of optimality. Namely, $\mathbf{Sc}^o(x) \leq k_A \cdot \mathbf{Sc}^o_A(x)$ for any Turing machine A.

Definition 20. $f(n) \preccurlyeq_{\mathbf{H}(h)} g(n)$ ($f(n) \preccurlyeq^a_{\mathbf{H}(h)} g(n)$) if there is a function $h \in \mathbf{H}$ such that $f(n) \leq h(g(n))$ for all $n \in \boldsymbol{N}$ (almost all $n \in \boldsymbol{N}$).

Definition 21. Functions $f(n)$ and $g(n)$ are called (asymptotically) $\mathbf{H}(h)$-equivalent if $f(n) \preccurlyeq_{\mathbf{H}(h)} g(n)$ and $g(n) \preccurlyeq_{\mathbf{H}(h)} f(n)$ ($f(n) \preccurlyeq^a_{\mathbf{H}(h)} g(n)$ and ($g(n) \preccurlyeq^a_{\mathbf{H}(h)} f(n)$)).

Theorem 4 [4]. Any two (asymptotically) $\mathbf{H}(h)$-optimal functions are (asymptotically) $\mathbf{H}(h)$-equivalent.

This means that optimal dual measures are in some sense invariant.

Theorems 3 and 4 imply existence and uniqueness of optimal/invariant measures for many dual complexity measures: Kolmogorov complexity, uniform complexity, prefix complexity, monotone complexity, process complexity, conditional Kolmogorov complexity, time-bounded Kolmogorov complexity, space-bounded Kolmogorov complexity, conditional resource-bounded Kolmogorov complexity,

time-bounded prefix complexity, resource-bounded Kolmogorov complexity, etc. We do not need to prove these theorems for each case separately because it is sufficient only to check conditions from theorems 3 and 4 and then to apply these theorems.

However, not all properties of optimal dual measures are good. For example, it has been proved that Kolmogorov complexity, which is an optimal dual measure for all recursive algorithms, is not itself a recursive function [50], although it can be computed by an inductive Turing machine [5].

4. Algorithmic and Communication Complexity of Recursive Algorithms

In the study of dual complexity measures, it is possible to make the following reductions. Algorithms work with words in some alphabet X. We can codify all symbols from X by finite strings consisting of two symbols 1 and 0. This allows us to consider only algorithms that work with words in the alphabet {1, 0}. In addition, it is practical in some cases to interpret such binary words as representations of nonnegative integer numbers and assume that the algorithms work with such numbers.

At first, we find some properties of complexity measures $L^{o}_{\mathbf{A}}(x)$ dual to the length $l(p)$ of program/algorithm description p with respect to a general class **A** of algorithms that work with words or natural numbers. We assume that **A** has a universal algorithm V. Then we have:

$$L^{o}_{\mathbf{A}}(x) = \min \{ l(p); p \in I \text{ and } V(p) = x\}.$$

Let the class **A** contains an identity algorithm E that computes the function $e(x) = x$.

Corollary 7. $L^{o}_{\mathbf{A}}(x)$ is a total function on $\boldsymbol{N}^{+}$ (on the set of all words in some alphabet).

The dual to the length of program/algorithm description complexity measure $C_{\mathbf{R}}(x)$ with respect to a class **R** of recursive algorithms (Turing machines, partial recursive functions, etc.) is Kolmogorov or algorithmic complexity [31]. For simplicity, we consider only such class **R** as the class **T** of all Turing machines and denote $C_{\mathbf{R}}(x)$ by $C(x)$.

Corollary 8 [32]. $C(x)$ is a total function on $\boldsymbol{N}^+$ (on the set of all words in some alphabet).

Let us suppose that the class **A** is infinite and contains only such algorithms that give as the result only one word or one number. In addition, we assume, without loss of generality, that all algorithms from **A** are working with natural numbers that are represented by words in the alphabet {1, 0}.

Lemma 6. For any number n there is some number z such that for all elements x that are larger than z, the values $L^{o}_{\mathbf{A}}(x)$ are larger than n.

Proof. The number of those elements x for which $L^{o}_{\mathbf{A}}(x)$ is less than or equal to a given number n is less than or equal to 2^n because there are at most 2^n programs having the length less than or equal to n and the universal inductive Turing machine W computes only one word with one program. Consequently, for all elements y that are larger than some element z, the values $L^{o}_{\mathbf{A}}(y)$ are larger than n.

Lemma 6 implies the following result.

Theorem 5. $L^{o}_{\mathbf{A}}(x) \to \infty$ when $l(x) \to \infty$.

Proof. Since the number of elements x for which $L^{o}_{\mathbf{A}}(x)$ is less than or equal to a given number n is finite by Lemma 3, so as n tends to infinity, the function $L^{o}_{\mathbf{A}}(x)$ does the same.

Corollary 9 (Kolmogorov). $C(x) \to \infty$ when $l(x) \to \infty$.

Theorem 6. For any dual complexity $\mathbf{Sc}^{o}_{\mathbf{A}}(x)$ for which the direct measure $\mathbf{Sc}_{\mathbf{A}}(x)$ satisfies Computational, Recomputational, and Cofinite axioms, we have $\mathbf{Sc}^{o}_{\mathbf{A}}(x) \to \infty$ when $l(x) \to \infty$.

Proof. By Proposition 5, there is a computable total strictly increasing function $f(x)$ such that $f(\mathbf{Sc}^{o}_{\mathbf{A}}(x)) > L^{o}_{\mathbf{A}}(x)$ for all x. If $\mathbf{Sc}^{o}_{\mathbf{A}}(x)$ does not tend to infinity, it must be bounded. Then the function $f(\mathbf{Sc}^{o}_{\mathbf{A}}(x))$ is bounded. So, it cannot be larger than the function $L^{o}_{\mathbf{A}}(x)$ that tends to infinity. This contradiction concludes the proof of Theorem 6.

Let **A** be an enumerable class of recursive or subrecursive algorithms that contains a universal algorithm.

Theorem 7. $L^{o}_{\mathbf{A}}(x)$ is an inductively computable function, namely, it is computable by some inductive Turing machine of the first order.

It is known (cf. [50]) that the function $C(x)$ is not recursively computable. At the same time, we have the following result implied by Theorem 7 that shows one more time the advantages of inductive Turing machines.

Corollary 10 [4]**.** $C(x)$ is an inductively computable function, namely, it is computable by some inductive Turing machine of the first order.

This result also follows from the theorem of Kolmogorov that states that $C(x)$ is a limiting recursive function (cf. [50]).

Traditionally (cf., for example, [49]), researchers in Kolmogorov complexity also consider the function $mC(x) = \min \{C(y); y \geq x\}$, which bounds $C(x)$ from below.

Taking some class **R** of recursive algorithms, we consider the function $m\mathbf{Sc}^{o}_{\mathbf{A}}(x) = \min \{ \mathbf{Sc}^{o}_{\mathbf{A}}(y); y \geq x\}$, which bounds $\mathbf{Sc}^{o}_{\mathbf{A}}(x)$ from below. Let us find some properties of this function.

Lemma 7. $m\mathbf{Sc}^{o}_{\mathbf{A}}(x)$ is a total increasing function.

Corollary 11 (Kolmogorov). $mC(x)$ is a total increasing function.

Lemma 8. $mC(x) \to \infty$ when $l(x) \to \infty$.

Let us assume that $m\mathbf{Sc}^{o}_{\mathbf{A}}(x)$ satisfies Recomputational and Cofiniteness axioms.

Lemma 9. If $\mathbf{Sc}^{o}_{\mathbf{A}}(x)$ is recursively computable, then $m\mathbf{Sc}^{o}_{\mathbf{A}}(x)$ is recursively computable.

Proof. We suppose that $\mathbf{Sc}^{o}_{\mathbf{A}}(x)$ is recursively computable and informally describe an algorithm of computation of such a function $f(x)$ that is equal to $m\mathbf{Sc}^{o}_{\mathbf{A}}(x)$. By the Church-Turing Thesis (cf., for example, [42]), $f(x)$ is a computable function.

An algorithm for computation of the function $f(x) = m\mathbf{Sc}^{o}_{\mathbf{A}}(x)$:

1. Compute $\mathbf{Sc}^{o}_{\mathbf{A}}(x)$. It is possible to do this because $\mathbf{Sc}^{o}_{\mathbf{A}}(x)$ is a recursively computable function.

2. Let $\mathbf{Sc}^{o}_{\mathbf{A}}(x) = p$ for some number p. Compute the set $X = \cup_{t \leq p} \mathbf{Sc}^{-1}(t)$. It is possible to do this because $\mathbf{Sc}(x)$ is a recomputable and cofinite function.

3. Let $X = \{ x_1, x_2, \dots , x_n \}$. Take the subset Y of all elements from X that are larger than or equal to x.

4. Let $Y = \{ y_1, y_2, \dots , y_m \}$. Compute all values $\mathbf{Sc}^{o}_{\mathbf{A}}(y_1)$, $\mathbf{Sc}^{o}_{\mathbf{A}}(y_2)$, … , $\mathbf{Sc}^{o}_{\mathbf{A}}(y_m)$. It is possible to do this because $\mathbf{Sc}^{o}_{\mathbf{A}}(x)$ is a recursively computable function.

5. Choose the least of these values, say, $\mathbf{Sc}^{o}_{\mathbf{A}}(y_i)$. Then by the definition, $m\mathbf{Sc}^{o}_{\mathbf{A}}(x) = \mathbf{Sc}^{o}_{\mathbf{A}}(y_i)$.

Lemma 9 is proved.

Theorem 8 [50]. If h is an increasing computable function that is defined in a decidable set V and tends to infinity when $l(x) \to \infty$, then for infinitely many elements x from V, we have $h(x) > C(x)$.

Theorem 9. For any cofinite computable recomputable static complexity $\mathbf{Sc}(x)$, its dual measure $\mathbf{Sc}^{o}(x)$ with respect to $\mathbf{R}$ is not recursively computable.

Proof. Let $\mathbf{Sc}^{o}(x)$ be a recursively computable function. Then by Lemma 9, $m\mathbf{Sc}^{o}(x)$ is also a recursively computable function.

By Propositions 2 and 5, there is a recursively computable function $f(x)$ such that $f(\mathbf{Sc}^{o}(x)) \le l^{o}(x) = C(x)$ for almost all x. Then $f(m\mathbf{Sc}^{o}(x)) \le f(\mathbf{Sc}^{o}(x)) \le C(x)$ for almost all x and $f(m\mathbf{Sc}^{o}(x))$ is an increasing computable function of x as the composition of two recursively computable functions is a recursively computable function.

At the same time, by Theorem 8, there are infinitely many elements x for which $f(m\mathbf{Sc}^{o}(x)) > C(x)$. This contradiction completes the proof.

Corollary 12 [30]. $C(x)$ is not a recursively computable function.

Corollary 13. The function $m\mathbf{Sc}^{o}(x)$ is not recursively computable.

Corollary 14 (Kolmogorov). The function $mC(x)$ is not recursively computable.

Noncomputability of Kolmogorov complexity allows one to prove noncomputability of communication complexity $\mathbf{cc}(f)$.

Theorem 10 [24]. Communication complexity $\mathbf{cc}(f)$ is not a recursively computable function in a general case.

To prove this result, we consider two computers (persons) C_1 and C_2 that are solving a problem f and are represented by universal Turing machines. The problem taken for this purpose is computation of x. Thus, we denote the problem by x.

At the beginning of the process, x is given as input to C_1 and nothing is given to C_2 , while it is C_2 , which has to compute x. That is why the value $\mathbf{cc}(x)$ determines the minimal number of bits that allow C_2 to compute x. By the definition, this number is equal to $C(x)$.

As C(x) is not a recursively computable function, **cc**(f), which in this case coincides with C(x), is also not a recursively computable function. Theorem 10 is proved.

However, inductive computations realized by inductive Turing machines allow one to compute different dual complexity measures in many interesting cases.

Let **A** be an enumerable class of recursive (subrecursive) algorithms that contains a universal algorithm and **Sc**(x) be a computable recomputable cofinite static complexity.

Theorem 11. $\mathbf{Sc}^{o}_{\mathbf{A}}(x)$ is an inductively computable function, namely, it is computable by some inductive Turing machine of the first order.

Corollary 15. $\mathbf{mSc}^{o}_{\mathbf{A}}(x)$ is an inductively computable function, namely, it is computable by some inductive Turing machine of the first order.

Corollary 16 (Kolmogorov). mC(x) is inductively computable;

The same is true for communication complexity. Let us assume that any problem f under consideration can be solved by some recursive algorithm (Turing machine) A.

Theorem 12. Communication complexity **cc**(f) is an inductively computable function, namely, it is computable by some inductive Turing machine of the first order.

Proof utilizes the Church-Turing Thesis and is based on the assumption, which is usually made in studies of communication complexity, that all protocols are recursive algorithms.

Remark 8. For some classes of distributed computation problems, **cc**(f) is a recursively computable function. For example, let us consider two computers (persons) C_1 and C_2 that are solving the problem f and are represented by universal Turing machines. The problem taken for this purpose is computation of x. However, in contrast to the situation in theorem 10, x is given as input to C_2 and C_2 has to compute x. In this case, **cc**(x) is identically equal to 0.

Remark 9. It is interesting to study computability of the communication complexity **cc**(f) in the case when protocols are inductive or other super-recursive algorithms.

5. Algorithmic Complexity of Inductive Turing Machines

The dual to the length of program/algorithm description complexity measure C(x) with respect to a class **SR** of super-recursive algorithms (inductive Turing machines, limiting partial recursive functions, grid automata, etc.) is called super-recursive Kolmogorov complexity. Here is its explicit definition.

Definition 22. The super-recursive Kolmogorov complexity SRC(x) of an object/word x is defined as

$$\mathrm{SRC}(x) = \min \{ l(p);\ U(p) = x\}$$

where $l(p)$ is the length of the word p and U is a universal algorithm in the class **SR**.

As any class of super-recursive algorithms contains some class of recursive algorithms, Theorem 1 implies the following result.

Proposition 2. SRC(x) is a total function on $\boldsymbol{N}$ (on the set of all words in some alphabet).

Here we limit ourselves to such super-recursive algorithms as the class **IT** of all inductive Turing machines of the first order.

Let **Sc** be a static complexity measure.

Definition 23. The dual to **Sc** inductive complexity $\mathbf{ISc}^{o}(x)$ of an object/word x is defined as

$$\mathbf{ISc}^{o}(x) = \min \{ \mathbf{Sc}(p);\ U(p) = x\}$$

where U is a universal inductive Turing machine of the first order.

A particular case of dual complexity measures is inductive Kolmogorov complexity, while $\mathrm{IL}^{o}(x)$ is an inductive counterpart of the measure $\mathrm{L}^{o}(x)$.

Definition 24. The *inductive Kolmogorov complexity* IC(x) of an object/word x is defined as

$$\mathrm{IC}(x) = \min \{ l(p);\ U(p) = x\}$$

where $l(p)$ is the length of the word p and U is a universal inductive Turing machine of the first order.

Corollary 17 [9]. IC(x) is a total function on $\boldsymbol{N}$ (on the set of all words in some alphabet).

In what follows, we assume, without loss of generality, that all considered inductive Turing machines are working with natural numbers that are represented by words in the alphabet {1, 0}.

As we will see the function IC(x) is essentially smaller than the function C(x). However, IC(x) also tends to infinity as Lemma 6 implies the following result.

Proposition 3. IC(x) $\to \infty$ when $l(x) \to \infty$.

However, IC(x) grows slower than any total increasing inductively computable by inductive Turing machines of the first order function.

Theorem 13. If f is a total strictly increasing inductively computable by inductive Turing machines of the first order function, then for infinitely many elements x, we have $f(x) >$ IC(x).

Proof. Let us assume that there is some element z such that for all elements y that are larger than z, we have $f(x) \leq$ IC(x). Because $f(x)$ an inductively computable function, there is an inductive Turing machine T of the first order that computes $f(x)$. It is done in the following way. Given a number x, the machine T makes the first step, producing $f_1(x)$ on its output tape. Making the second step, the machine T producing $f_2(x)$ on its output tape. After n steps, T has $f_n(x)$ on its output tape. Since the function is inductively computable by inductive Turing machines of the first order, this process stabilizes on some value $f_n(x) = f(x)$, which is the result of computation with the input x. Taking the function $h(m) = \min \{ x ; f(x) \geq m \}$, we construct an inductive Turing machine M of the first order that computes the function $h(x)$.

The inductive Turing machine M contains a copy of the machine T. Utilizing this copy, M finds one after another the values $f_1(1), f_1(2), \ldots, f_1(m + 1)$ and compares these values to m. Then M writes into the output tape the least x for which the value $f_1(x)$ is larger than or equal to m. Then M finds one after another the values $f_2(1), f_2(2), \ldots, f_2(m + 1)$ and compares these values to m. Then M writes into the output tape the least x for which the value $f_2(x)$ is larger than or equal to m. This process continues until the output value of M stabilizes. It happens for any number m due to the following reasons. First, $f(x)$ is a total function, so all values $f_i(1), f_i(2), \ldots, f_i(m + 1)$ after some step $i = t$ become equal to $f(1), f(2), \ldots, f(m + 1)$. Second, $f(x)$ is a strictly increasing function. This implies that $f_i(m + 1) > m$. In such a way, the machine M computes $h(m)$. Since m is an arbitrary number, the machine M computes the function $h(x)$.

Since for all elements y that are larger than z, we have $f(y) \leq \mathrm{IC}(y)$, there is an element m such that $\mathrm{IC}(h(m)) \geq f(h(m))$ and $f(h(m)) \geq m$ as $f(x)$ is a strictly increasing function and $h(m) = \min \{ x ; f(x) \geq m \}$. By the definition, $\mathrm{IC}_T(h(m)) = \min \{ l(x) ; T(x) = h(m) \}$. As $T(m) = h(m)$, we have $\mathrm{IC}_T(h(m)) \leq l(m)$. Thus, $l(m) \geq \mathrm{IC}_T(h(m)) \succcurlyeq \mathrm{IC}(h(m)) \geq m$. However, it is impossible that $l(m) \succcurlyeq m$. This contradiction concludes the proof of Theorem 13. □

Theorem 14. The function $\mathrm{IC}(x)$ is not inductively computable by inductive Turing machines of the first order.

Proof. Let us suppose that the function $\mathrm{IC}(x)$ is inductively computable by inductive Turing machines of the first order. It means that there is an inductive Turing machine M of the first order such that $M(x) = \mathrm{IC}(x)$ for all x. We define the function $\mathrm{mIC}(x) = \min \{ \mathrm{IC}(y) ; l(y) \geq l(x) \}$. This function has the following properties:

1. $\mathrm{mIC}(x)$ is a total increasing function;

2. $\mathrm{mIC}(x) \to \infty$ when $l(x) \to \infty$.

Indeed, since $\mathrm{IC}(x)$ is a total function, $\mathrm{mIC}(x)$ is also a total function. By its definition, $\mathrm{mC}(x)$ is increasing. In addition, as $\mathrm{IC}(x) \to \infty$ when $l(x) \to \infty$, the same is true for the function $\mathrm{mIC}(x)$.

Inductive computability of $\mathrm{IC}(x)$ allows us to build an inductive Turing machine H of the first order such that it computes $\mathrm{mIC}(x)$.

By the properties of inductive Turing machines, they can include Turing machines as submachines/subprograms. The machine H includes three such submachines: G, M_0 , and D. They perform the following functions.

The machine G, given a word x and a number k, generates all words z for which $l(x) + k \geq l(z) \geq l(x)$ is true. The machine M_0 has infinite number of input and output tapes. Given n words $x_1, x_2, \ldots , x_n$, the machine M_0 simulates n steps of computation of the machine M with these words $x_1, x_2, \ldots , x_n$ as its inputs. The results of these computations, i.e., the content of the output tape of M for each $x_1, x_2, \ldots , x_n$, are written in the output tapes of M_0 . The functioning of the machine D is described as a part of the functioning of H. General methods of the theory of Turing machines (cf., for example, [42]) allows us to build this Turing machine.

Having these subprograms, the machine H works in the following manner:

1. The variable t is made equal to 1 and 0 is written in the output tape of H.

2. The machine M_0 simulates 1 step of computation of the machine M with the word x as its input, eventually changing the content of the output tape of H.

3. The variable t is made equal to $k + 1$ where k is the previous value of t, in other words $t := t + 1$.

4. The machine G, given a word x and the number t, generates all words $x_1, x_2, \dots, x_n$ for which $l(x) + t \geq l(z) \geq l(x)$ is true.

5. Given n words $x_1, x_2, \dots, x_n$, the machine M_0 simulates n steps of computation of the machine M with these words $x_1, x_2, \dots, x_n$ as its inputs. The results of these computations, i.e., the content of the output tape of M for each $x_1, x_2, \dots, x_n$, are written in the output tapes of M_0.

6. The machine D takes all current outputs of the machine M_0, i.e., the natural numbers $m_1, m_2, \dots, m_n$ that are written in the output tapes of M_0 and which are the results of $M(x_1), M(x_2), \dots, M(x_n)$ after n steps of computation.

7. The machine D selects the least of them (say m_i) in the lexicographical order and compares it to the previous output of the machine H, writing in the output tape of H the least of these two numbers, and goes to the step 3.

The machine M_0 computes values of the function $\mathrm{IC}(x)$. So, after some step of computation the output of M_0 will be equal to $\mathrm{IC}(x)$. After another step it will be equal to $\mathrm{IC}(x_1)$ and so on. By the definition of the measure $\mathrm{IC}(x)$ and Proposition 3, there is only a finite number (say d) of such words z that $\mathrm{IC}(z) \leq \mathrm{IC}(x)$ for the given element x. For each word z, the process of computation of $M_0(z)$ stabilizes after some step. So, the output of the machine H with the input x also stabilizes after some step. It means that H computes the function $\mathrm{mIC}(x)$.

As it is demonstrated, $\mathrm{mIC}(x)$ is a monotone function that tends to infinity when $l(x)$ tends to infinity. Besides by the definition $\mathrm{mIC}(x) \leq \mathrm{IC}(x)$ for all x. This contradicts Theorem 13 and thus, concludes the proof of Theorem 14. □

However, inductive Turing machines of the second order have greater computing power.

Theorem 15. The function $\mathrm{IC}(x)$ is computable in the class of inductive Turing machines of the second order.

Proof. Let us consider an inductive Turing machine T of the first order. For simplicity, we assume that its working alphabet consists of two symbols 1 and 0. As it is possible to codify any system of symbols as words in such an alphabet, this assumption does influence the generality of the proof.

Then it is possible to find an inductive Turing machine M of the second order such that it computes the function $IC_T(x)$.

Indeed, we can build the memory E of the machine M in the following way. It is structured as three linear tapes: the input, output, and working tape. In addition, the working tape has connections between its cells of the type r. Namely, the cell with the number x is connected by r to the cell with the number y if and only if $T(x) = y$. As T is an inductive Turing machine of the first order, memory E and M is the machine of the second order.

Assuming that words in the alphabet {1, 0} denote natural numbers, we define functioning of M by the following rules.

1. M reads the number x written in its input tape and the head of M goes to the first cell of the working tape.

2. If there is a connection of the type r from this cell to another cell c, M compares the number of c with the number x.

3. If the number of c is equal to the number x, then M writes 1 on its output tape and does not change it forever. By the definition, $IC_T(x) = 1$.

4. If the number of c is not equal to the number x, then M writes 1 on its output tape and its head goes to the second cell of the working tape.

5. If there is a connection of the type r from this cell to another cell c, M compares the number of c with the number x.

6. If the number of c is equal to the number x, then M writes 2 on its output tape and does not change it forever. By the definition, $IC_T(x) = 2$.

Continuing this process, the machine M computes the function $IC_T(x)$.

Taking a universal inductive Turing machine U as T, we obtain the statement of Theorem 15.

Theorem 16. For any cofinite computable recomputable static complexity $\mathbf{Sc}(x)$, its dual measure $I\mathbf{Sc}^{o}(x)$ is not inductively computable by inductive Turing machines of the first order.

Proof. Let I**Sc**$^{o}(x)$ be an inductively computable by inductive Turing machines of the first order function. Then as it is demonstrated in the proof of Theorem 14, mI**Sc**$^{o}(x)$ is also an inductively computable by inductive Turing machines of the first order function.

By Propositions 2 and 5, there is a recursively computable function $f(x)$ such that $f(\mathrm{I}\mathbf{Sc}^{o}(x)) \leq \mathrm{IL}^{o}(x) = \mathrm{IC}(x)$ for almost all x. Then $f(\mathrm{mI}\mathbf{Sc}^{o}(x)) \leq f(\mathrm{I}\mathbf{Sc}^{o}(x)) \leq \mathrm{IC}(x)$ for almost all x and $f(\mathrm{mI}\mathbf{Sc}^{o}(x))$ is an increasing inductively computable function of x as the composition of a total recursively computable function and an inductively computable by inductive Turing machines of the first order function is an inductively computable by inductive Turing machines of the first order function.

At the same time, by Theorem 13, there are infinitely many elements x for which $f(\mathrm{m}\mathbf{Sc}^{o}(x)) > \mathrm{IC}(x)$. This contradiction completes the proof of Theorem 16. □

We can prove a stronger statement than Theorem 13 that allows us to get more exact comparison of complexity of recursive and inductive algorithms and computations. To do this, we assume for simplicity that inductive Turing machines are working with words in some finite alphabet and that all these words are well ordered, that is, any set of words contains the least element. It is possible to find such orderings, for example, in [42].

Theorem 17. If h is an increasing inductively computable by inductive Turing machines of the first order function that is defined in an infinite inductively decidable by inductive Turing machines of the first order set V and tends to infinity when $l(x) \to \infty$, then for infinitely many elements x from V, we have $h(x) > \mathrm{IC}(x)$.

Proof. Let us assume that there is some element z such that for all elements x that are larger than z, we have $h(x) \leq \mathrm{IC}(x)$. Because $h(x)$ an inductively computable by inductive Turing machines of the first order function, there is an inductive Turing machine T of the first order that computes $h(x)$. Taking the function $g(m) = \min \{ x ; h(x) \geq m \text{ and } x \in V \}$, we construct an inductive Turing machine M of the first order that computes the function $g(x)$.

As V is an inductively decidable by inductive Turing machines of the first order set, there is an inductive Turing machine H of the first order that given an input x,

produces 1 when $x \in V$, and produces 0 when $x \notin V$. It means that H computes the characteristic function $c_V(x)$ of the set V.

The inductive Turing machine M contains a copy of the machine H and a copy of the machine T. Utilizing the copy of T, the machine M computes the value $h_1(1)$ and compares it to m. Utilizing the copy of H, the machine M computes the value $c_{V1}(1)$. If $h_1(1)$ is larger than m and $c_{V1}(1) = 1$, then M writes 1 into the output tape. Otherwise, M writes nothing into the output tape. After this, M finds the values $h_2(1)$ and $h_2(2)$ and compares these values to m. Concurrently, M finds the values $c_{V2}(1)$ and $c_{V2}(2)$. Then M writes into the output tape the least x for which the value $h_1(x)$ is larger than or equal to m and at the same time, $c_{V2}(x) = 1$. This process continues. Making cycle i of the computation, M computes the values $h_i(1)$, $h_i(2)$, ... , $h_i(i)$ and compares these values to m. We remind here that $h_i(j)$ is the result of i steps of computation of T with the input j. Concurrently, M computes the values $c_{V\,i}(1)$, $c_{Vi}(2)$, … , $c_{V\,i}(i)$. Then M writes into the output tape the least x for which the value $h_i(x)$ is larger than or equal to m and at the same time, $c_{Vi}(x) = 1$. Such cycle is repeated until the output value of M stabilizes.

Each value $c_{V\,i}(x)$ stabilizes at some step t because $c_V(x)$ is a total inductively computable function. In a similar way, each value $h_i(x)$ stabilizes at some step q because $h(x)$ is an inductively computable function defined for all $x \in V$. Thus, after this step $p = \max\{q, t\}$, the value $h_i(x)$ becomes equal to the value $h(x)$. In addition, there is such a step t when a number n is found for which $h(n) \geq m$. After this step, only such numbers x can go to the output tape of M that belong to V and are less than or equal to n.

This happens for any given number m due to the following reasons. First, $h(x)$ is defined for all elements from V total function, so those values $h_i(1)$, $h_i(2)$, ... , $h_i(m+1)$ for which the argument of h_i belongs to V after some step $i = r$ become equal to $h(1)$, $h(2)$, ... , $h(m)$. Second, $h(x)$ is an increasing function that tends to infinity.

This shows that the whole process stabilizes and by the definition of inductive computability, the machine M computes $g(m)$. Since m is an arbitrary number, the machine M computes the function $g(x)$.

To conclude the proof, we repeat the reasoning from the proof of Theorem 13. Since for all elements y that are larger than z, we have $f(x) \leq \mathrm{IC}(x)$, there is an element

m such that IC($g(m)$) $\geq h(g(m))$ and $h(g(m)) \geq m$ as $h(x)$ is an increasing function and $g(m) = \min \{ x ; h(x) \geq m \}$. By the definition, $IC_T(g(m)) = \min \{ l(x) ; T(x) = g(m) \}$. As $T(m) = g(m)$, we have $IC_T(g(m)) \leq l(m)$. Thus, $l(m) \geq IC_T(h(m)) \succcurlyeq IC(h(m)) \geq m$. However, it is impossible that $l(m) \succcurlyeq m$. This contradiction concludes the proof of the theorem.

Remark 10. Although Theorem 13 can be deduced from Theorem 17, we give here an independent proof because it demonstrates another technique, which displays essential features of inductive Turing machines.

Corollary 18. If h is a total increasing inductively computable by inductive Turing machines of the first order function that tends to infinity when $l(x) \to \infty$, then for infinitely many elements x, we have $h(x) > IC(x)$.

Corollary 19. If h is an increasing inductively computable by inductive Turing machines of the first order function that is defined on an infinite recursive set V and tends to infinity when $l(x) \to \infty$, then for infinitely many elements x from V, we have $h(x) > IC(x)$.

Since the composition of two increasing functions is an increasing function and the composition of a recursive function and an inductively computable function is an inductively computable function, we have the following result.

Corollary 20. If $h(x)$ and $g(x)$ are increasing functions, $h(x)$ is inductively computable by inductive Turing machines of the first order and defined on an infinite inductively decidable by inductive Turing machines of the first order set V, $g(x)$ is a recursive function, and they both tend to infinity when $l(x) \to \infty$, then for infinitely many elements x from V, we have $g(h(x)) > IC(x)$.

Corollary 21. The function IC(x) is not inductively computable by inductive Turing machines of the first order. Moreover, no inductively computable by inductive Turing machines of the first order function $f(x)$ defined for an infinite inductively decidable by inductive Turing machines of the first order set of numbers can coincide with IC(x) in the whole of its domain of definition.

In addition to the function IC(x), we also consider the function mIC(x) = min { IC(y); $l(y) \geq l(x)$ }, which was introduced in the proof of Theorem 14. This function has the following properties.

Theorem 17 and Corollary 10 imply the following result.

Theorem 18. For any increasing recursive function $h(x)$ that tends to infinity when $l(x) \to \infty$ and any infinite inductively decidable by inductive Turing machines of the first order set V, there are infinitely many elements x from V for which $h(\mathrm{C}(x)) > \mathrm{IC}(x)$.

Corollary 22. In any infinite inductively decidable by inductive Turing machines of the first order set V, there are infinitely many elements x for which $\mathrm{C}(x) > \mathrm{IC}(x)$.

Corollary 23. In any infinite recursive set V, there are infinitely many elements x for which $\mathrm{C}(x) > \mathrm{IC}(x)$.

Corollary 24. In any inductively decidable by inductive Turing machines of the first order (recursive) set V, there are infinitely many elements x for which $\ln_2(\mathrm{C}(x)) > \mathrm{IC}(x)$.

If $\ln_2(\mathrm{C}(x)) > \mathrm{IC}(x)$, then $\mathrm{C}(x) > 2^{\mathrm{IC}(x)}$. At the same time, for any natural number k, the inequality $2^n > k \cdot n$ is true almost everywhere. This and Corollary 24 imply the following result.

Corollary 25. For any natural number k and in any inductively decidable by inductive Turing machines of the first order (recursive) set V, there are infinitely many elements x for which $\mathrm{C}(x) > k \cdot \mathrm{IC}(x)$.

These inequalities allow one to obtain corresponding results on complexity of super-recursive algorithms from [10] and [46].

Corollary 26. There are infinitely many elements x for which $\mathrm{C}(x) > \mathrm{IC}(x)$.

Corollary 27. For any natural number a, there are infinitely many elements x for which $\ln_a(\mathrm{C}(x)) > \mathrm{IC}(x)$.

Corollary 28. There are infinitely many elements x for which $\ln_2(\mathrm{C}(x)) > \mathrm{IC}(x)$.

6. Conclusion

All these results show that inductive Turing machines are much more efficient than any kind of recursive algorithms with respect to Kolmogorov/algorithmic complexity and many other dual complexities of algorithms. Informally, it means that in comparison with recursive algorithms, super-recursive programs for solving the same problem are shorter, have lower branching (i.e., less instructions of the form IF *A* THEN *B* ELSE *C*), make less reversions and unrestricted transitions (i.e., less instructions of the form GO TO *X*) for infinitely many problems solvable by recursive algorithms.

In addition, connections between communication complexity and Kolmogorov complexity imply that communication complexity becomes much less for many problems if we use inductive computations instead of recursive computations.

It is also demonstrated that inductive Turing machines can compute Kolmogorov complexity for recursive algorithms. This greater power of inductive Turing machines has many implications for practical problems of programming. For instance, Lewis [35], using boundaries that are set by the theory of algorithmic complexity, demonstrates limits of software estimation. Inductive Turing machines are able to make estimations that are inaccessible for conventional Turing machines. It is possible because inductive Turing machines are more powerful and have lower algorithmic complexity in comparison with conventional Turing machines.

All this is true for inductive Turing machines of the first order. An interesting problem is to compare efficiency and complexity of inductive Turing machines of different orders.

Acknowledgments

The author is grateful to Paul Stelling for helpful remarks and suggestions.

References

[1] J.L. Balcazar, J. Diaz, J. Gabarro, *Structural Complexity*, Springer-Verlag, Berlin/Heidelberg/New York, 1988

[2] Ja. M. Barzdin, Complexity of programs which recognize whether natural numbers not exceeding *n* belong to a recursively enumerable set, *Dokl. Akad. Nauk SSSR* 182 (1968) 1249-1252

[3] M. Blum On the Size of Machines, *Information and Control* 11 (1967) 257-265

[4] M. Burgin, Generalized Kolmogorov Complexity and Duality in Theory of Computations, *Notices of the Academy of Sciences of the USSR* 264 (1982) 19-23 (translated from Russian, v.25, No. 3)

[5] M. Burgin, Inductive Turing Machines, *Notices of the Academy of Sciences of the USSR* 270 (1983) 1289-1293 (translated from Russian, v. 27, No. 3)

[6] M. Burgin, Generalized Kolmogorov Complexity and other Dual Complexity Measures, *Cybernetics*, No. 4 (1990) 21-29 (translated from Russian)

[7] M. Burgin, Universal limit Turing machines, *Notices of the Russian Academy of Sciences* 325 (1992) 654-658 (translated from Russian)

[8] M. Burgin, Complexity measures in the axiomatic theory of algorithms, in *Methods of design of applied intellectual program systems*, Kiev (1992a) 60-67 (in Russian)

[9] M. Burgin, Algorithmic Approach in the Dynamic Information Theory, *Notices of the Russian Academy of Sciences* 342 (1995) 7-10 (translated from Russian)

[10] Burgin, M. (1999) Super-recursive Algorithms as a Tool for High Performance Computing, *Proceedings of the High Performance Computing Symposium*, San Diego, pp. 224-228

[11] M. Burgin, *Super-recursive Algorithms*, Springer, New York/Berlin/Heidelberg, 2004

[12] M. Burgin, Yu. A. Chelovsky, Program Quality Estimation Based on Structural Characteristics, International Conference "*Software*", Kalinin (1984) 79-81 (in Russian)

[13] M. Burgin, Yu. A. Chelovsky, Quality of the Automated Control System Software, in "*Perspectives of the Automated Control System Development*," Kiev (1984) 58-66 (in Russian)

[14] M. Burgin, N.C. Debnath, Complexity of Algorithms and Software Metrics, in Proceedings of the ISCA 18th International Conference "*Computers and their Applications*", International Society for Computers and their Applications, Honolulu, Hawaii (2003) 259-262

[15] A. I. Cardoso, R.G. Crespo, P. Kokol, Two different views about software complexity, in *Escom 2000*, Munich, Germany (2000) 433-438

[16] G.J. Chaitin, On the Length of Programs for Computing Finite Binary Sequences, *J. Association for Computing Machinery* 13 (1966) 547-569

[17] G.J. Chaitin, A Theory of Program Size Formally Identical to Information Theory, *J. Association for Computing Machinery* 22 (1975) 329-340

[18] R.P. Daley, Minimal-program complexity of sequences with restricted resources, *Information and Control* 23 (1973) 301-312

[19] B. Edmonds, *Syntactic Measures of Complexity***,** CPM Report No.: 99-55**,** University of Manchester, Manchester, UK, 1999

[20] P. Gasc, On a Symmetry of Algorithmic Information, *Soviet Math. Dokl*. 218 (1974) 1265-1267

[21] M. Gell-Mann, *The Quark and the Jaguar*, W. H. Freeman, 1994

[22] M. Gell-Mann, Remarks on Simplicity and Complexity, *Complexity* 1 (1995) 16-19

[23] E.M. Gold, Language Identification in the Limit*, Information and Control* 10 (1967) 447-474

[24] V.S. Gupta, Communication complexity for file synchronization is undecidable, *ACM SIGACT News* 33 (2002) 110 - 112

[25] M.H. Halstead, *Elements of Software Science*, Elsevier, New York, 1977

[26] J. Hartmanis, J.E. Hopcroft, An Overview of the Theory of Computational Complexity, *J. Association for Computing Machinery* 18 (1971) 444-475

[27] J.E. Hopcroft, R. Motwani, J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison Wesley, Boston/San Francisco/New York, 2001

[28] P. Horn, *Announcement of the Autonomous Computing Program*, (2001) www.research.ibm.com/autonomic/

[29] J. Hromkovic, *Communication Complexity and Parallel Computing*, Springer, New York, 1997

[30] D.W. Juedes, J.H. Lutz, Kolmogorov Complexity, Complexity Cores and the Distribution of Hardness, in *Kolmogorov Complexity and Computational Complexity*, Springer-Verlag, Berlin/Heidelberg/New York, 1992

[31] A.N. Kolmogorov, *Foundations of the Theory of Probability*, Chelsea, 1950

[32] A.N. Kolmogorov, A.N. Three approaches to the definition of the quantity of information, *Problems of Information Transmission* 1 (1965) 3-11

[33] E. Kushilevitz, N. Nisan, *Communication Complexity*, Cambridge University Press, Cambridge, 1997

[34] H.W. Lawson, Rebirth of Computer Industry, *Communications of the ACM* 45 (2002) 25-29

[35] L. A. Levin, On the notion of a random sequence. *Soviet Math. Dokl*. 14 (1973) 1413-1416

[36] L. A. Levin, Laws of information (nongrowth) and aspects of the foundation of probability theory, *Problems of Information Transmission* 10 (1974) 206-210

[37] J.P. Lewis, Limits to Software Estimation, *Software Engineering Notes*, 26 (2001) 54-59

[38] M. Li, P. Vitanyi, *An Introduction to Kolmogorov Complexity and its Applications,* Springer-Verlag, New York,1997

[39] D. W. Loveland, A variant of the Kolmogorov concept of complexity, *Information and Control* 15 (1969) 510—526

[40] Yu I. Manin, *Course in Mathematical Logic*, Springer-Verlag, New York, 1991

[41] T. J. McCabe, A *Complexity Measure, IEEE Transaction on Software Engineering*, SE-2 (1976) 308-320

[42] H. Rogers, *Theory of Recursive Functions and Effective Computability*, MIT Press, Cambridge Massachusetts, 1987

[43] C.-P. Schnorr, Process complexity and effective random tests, Fourth Annual ACM Symposium on the Theory of Computing, *J. Comput. System Sci*. 7 (1973) 376-388

[44] M. Sipser, A topological view of some problems in complexity theory, Theory of algorithms (Pécs, 1984), *Colloq. Math. Soc. János Bolyai* 44 (1985) 387-391

[45] R.J. Solomonoff, A Formal Theory of Inductive Inference, *Information and Control* 7 (1964) 224-254

[46] Schmidhuber, J. Hierarchies of Generalized Kolmogorov Complexities and Nonenumerable Universal Measures Computable in the Limit, *International Journal of Foundations of Computer Science* 3:4 (2002) 587-612

[47] M.J. Waxman, M. J. *On Problem Complexity*, 1996 (unpublished work)

[48] R.W. Wolverton, The Cost of Developing Large-Scale Software. *IEEE Transactions on Computer*, C-23 (1974) 615-636

[49] H. Zuse, *History of Software Measurement*, Berlin, 1998

[50] A.K. Zvonkin, L.A. Levin, The Complexity of Finite Objects and the Development of the Concepts of Information and Randomness by Means of the Theory of Algorithms, *Russian Mathematics Surveys* 256 (1970) 83-124