1. INTRODUCTION

We treat a computational problem, i.e. a problem of evaluating some function, to be hard, if we do not know whether it can be actually solved on real computers. From the point of view of classical theory of algorithms or set-theoretic mathematics (but hardly from the point of view of Al-Khwarizmi) such a problem may be solvable: there is a trivial way of solving it, which consists (for a given argument) in more or less exhaustive search in a finite set of candidates for a solution of the problem and in rather a simple test for any such candidate whether it is a solution. (By Kleene normal form theorem [1], §1.10, every computable function can be evaluated in such a way in its domain.) However, one could hardly implement such a method: the sun will die out and computer still will be at the beginning of the computation by the program defining it.

We define the time complexity of a program \( \tau \) as a function mapping an argument, i.e. input datum, into the number of computer operations done by the program for this argument. The value of this function for \( X \) will be denoted by \( \tau^*(X) \). It is not simple to get a detailed description of the behaviour of \( \tau^*_n \), and the information, we obtain from it, often does not justify expenses. It is more convenient and enough informative to get upper bounds on \( \tau^*_n(X) \) for a given class of arguments. This way we come to the notion of the worst-case complexity: the worst-case time complexity is a function mapping a natural number \( n \) into the maximum of \( \tau^*_n(X) \) for \( X \) with the length not greater than \( n \). The worst-case time complexity will be denoted by \( \tau^*_n(n) \) and referred to as complexity.

Besides the time of processing we have other complexity characteristics of computation which are worth being estimated - they are the size of the memory (space complexity), the length of registers, etc. However, without estimation of time, estimations of other complexity characteristics are of little value, as a rule. On the other hand, if we reasonably choose our computational model, e.g.
see [2], then upper bound on the time may give rather a nontrivial,
sometimes even good, bounds on the other complexity characteristics.

The complexity of a problem is defined by 2 bounds: upper and
lower. Unary function $\varphi$ is an upper bound on the complexity of a
given problem $f$ if one can construct a program $\Pi$ solving this
problem (i.e. evaluating $f$) and such that $t_{\Pi}(n) \geq \varphi(n)$ for all $n$.
Binary function $\psi$ is a lower bound on the complexity of $f$ if for
every $\Pi$, evaluating $f$, we have $t_{\Pi}(n) \geq \psi(|\Pi|,n)$ for all $n$, where
$|\Pi|$ is the length of $\Pi$. These definitions show that we can speak
about coincidence of upper and lower bounds within some type of
proximity. "Ideal" proximity of these bounds is given by the equa-
liity $\psi(k,n)=\psi_1(k)\varphi(n)$ with some $\psi_1$, decreasing "enough slowly".
But such a proximity can not be always achieved even in theory.

We can prove some problems to have no practically effective
algorithms for all inputs. We shall call them genuinely hard. And
if we cannot prove such a property, and do not know an effective
algorithm solving the problem, then we shall call it non-effectiviz-
ed or vaguely hard.

Problems of evaluating functions with large values are genuine-
ly hard, e.g. problem of enumerating elements of a finite set with
very large number of elements. Once computer designers begged the
first author to develop a program enumerating all the directed cir-
cuits of a digraph. Its number may be not large but it can reach
$\sum_{k=1}^{n} k!$, where $n$ is the number of vertices of the graph. We are
not aware of what would do the designers if the amount of circuits
had been only $10^6$ times less than this value, when $n \geq 90$.

One can get genuinely hard problems using various diagonal
constructions. In particular, for a given bound $\varphi$ one can const-
struct a two-value function (i.e. a set recognition problem) which
has both complexity bounds very close to $\varphi$ (we assume $\varphi$ being
computable enough simply, e.g. $\varphi$ being a polynomial, exponential
function and so on). Though examples of genuinely hard problems,
built by this method, are artificial, they can be sometimes rather
simply reduced to natural problems, and so a high lower bound on
the complexity of the latter problems can be gained. This way of
reasoning is rather a faithful copy of proofs of algorithmic unsol-
vability of concrete problems. Results of this kind are of negative
flavour; as a rule, they mean the problem under consideration to be
practically unsolvable, and we are to modify its formulation. A
proof of genuine hardness of a concrete problem may be of a definite